An investigation of the XMOS XS1 architecture as a platform for development of audio control standards

James Dibley

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

Master of Science

in Computer Science

October 2013

Rhodes University

Abstract

This thesis investigates the feasiblity of using a new microcontroller architecture, the XMOS XS1, in the research and development of control standards for audio distribution networks.

This investigation is conducted in the context of an emerging audio distribution network standard, Ethernet Audio/Video Bridging ('Ethernet AVB'), and an emerging audio control standard, AES-64. The thesis describes these emerging standards, the XMOS XS1 architecture (including its associated programming language, XC), and the open-source implementation of an Ethernet AVB streaming audio device based on the XMOS XS1 architecture.

It is shown how the XMOS XS1 architecture and its associated features, focusing on the XC language's mechanisms for concurrency, event–driven programming, and integration of C software modules, enable a powerful implementation of the AES–64 control standard.

Feasibility is demonstrated by the implementation of an AES-64 protocol stack and its integration into an XMOS XS1-based Ethernet AVB streaming audio device, providing control of Ethernet AVB features and audio hardware, as well as implementations of advanced AES–64 control mechanisms.

It is demonstrated that the XMOS XS1 architecture is a compelling platform for the development of audio control standards, and has enabled the implementation of AES–64 connection management and control over standards–compliant Ethernet AVB streaming audio devices where no such implementation previously existed.

The research additionally describes a linear design method for applications based on the XMOS XS1 architecture, and provides a baseline implementation reference for the AES–64 control standard where none previously existed.

Contents

1 Introduction **2**
- 1.1 Research question . . . 4
- 1.2 Research objectives . . . 4
- 1.3 Chapter layout . . . 5

2 Audio networks and Ethernet AVB **8**
- 2.1 Introduction . . . 8
  - 2.1.1 Audio distribution applications and requirements . . . 8
  - 2.1.2 Audio distribution networks . . . 9
  - 2.1.3 Providing quality of service for audio transport . . . 10
  - 2.1.4 Audio network technologies . . . 11
- 2.2 Ethernet AVB: an overview . . . 12
  - 2.2.1 The Ethernet AVB network . . . 13
  - 2.2.2 Audio distribution over Ethernet AVB . . . 16
  - 2.2.3 Audio encapsulation . . . 18
  - 2.2.4 Ethernet AVB use case: live sound . . . 18
  - 2.2.5 Quality of service in Ethernet AVB . . . 20
- 2.3 The Ethernet AVB component standards . . . 24
  - 2.3.1 IEEE 802.1Qav: Forwarding and queueing enhancements for time-sensitive streams . . . 25
  - 2.3.2 IEEE 802.1AS: Timing and synchronisation for time-sensitive applications in bridged local area networks . . . 27
  - 2.3.3 IEEE 802.1Qat: Stream reservation protocol (SRP) . . . 29
  - 2.3.4 IEEE 1722: Layer 2 transport protocol for time-sensitive applications in bridged local area networks . . . 35
  - 2.3.5 IEEE 1722.1 (AVDECC): Draft standard for device discovery, connection management and control protocol for IEEE 1722 based devices . . . 37
  - 2.3.6 Ethernet AVB components: summary . . . 47
- 2.4 The XMOS Ethernet AVB reference design . . . 47
  - 2.4.1 Hardware . . . 47
  - 2.4.2 Software . . . 48
  - 2.4.3 The XMOS Ethernet AVB API . . . 53
- 2.5 Conclusion . . . 54

3 Connection management and control in audio networks, and the AES-64 standard **55**
3.1 Introduction . . . 55
3.2 Connection management and control in audio networks . . . 55
3.2.1 Control standards and autoconfiguration . . . 57
3.3 Introduction to AES-64 . . . 57
3.3.1 AES-64 device requirements . . . 58
3.4 Device representation in AES-64 . . . 59
3.4.1 The AES-64 parameter . . . 59
3.4.2 The level hierarchy . . . 61
3.5 Messaging in AES-64 . . . 63
3.5.1 Message targets . . . 63
3.5.2 Wildcards . . . 64
3.5.3 Message types . . . 65
3.5.4 Message sources . . . 65
3.5.5 Message commands . . . 66
3.6 Control features in AES-64 . . . 66
3.6.1 Network discovery . . . 67
3.6.2 Parameter grouping . . . 67
3.6.3 Connection management . . . 68
3.7 An implementation of AES-64: the UNOS suite . . . 75
3.8 Conclusion . . . 75

4 The XMOS XS1 microcontroller architecture **78**
4.1 Introduction . . . 78
4.2 Processes and concurrency . . . 79
4.3 Events and event-driven execution . . . 81
4.3.1 The select statement . . . 82
4.3.2 Parameterised select functions . . . 82
4.3.3 Replicated select functions . . . 83
4.3.4 Events and determinism . . . 83
4.4 Channels . . . 84
4.4.1 Channel input-output in XC . . . 84
4.4.2 Channel transactions . . . 87
4.5 XC and C . . . 89
4.5.1 Reinterpretation . . . 89
4.5.2 Pointers and passing by reference . . . 89
4.6 Integration of C modules in XC programs . . . 90
4.7 Conclusion . . . 94

5 Design **95**
5.1 Introduction . . . 95
5.2 Requirements capture and analysis . . . 96
5.2.1 Forensic capture of existing AES-64 implementations . . . 97

5.2.2 Review of existing AES-64 applications . . . 99
5.2.3 Close reading of the AES-64 standard . . . 101
5.2.4 The AES-64 requirements document . . . 103
5.3 Design . . . 105
5.3.1 Selection of a design method . . . 105
5.3.2 Real-Time Structured Analysis and Design (RTSAD) . . . 108
5.3.3 The essential model . . . 109
5.3.4 Building the context diagram and event list . . . 111
5.3.5 Building the data specifications . . . 113
5.3.6 Building the transformation schemas . . . 115
5.3.7 Building the process specifications . . . 119
5.3.8 Process specifications for handling AES-64 commands . . . 119
5.4 The implementation model . . . 123
5.4.1 Task interfaces . . . 126
5.4.2 The device representation task . . . 126
5.4.3 The message processing task . . . 128
5.4.4 The stack control task . . . 130
5.5 Conclusion . . . 131

6 Implementation **132**
6.1 Introduction . . . 132
6.2 Implementation overview . . . 135
6.3 Prototyping . . . 135
6.3.1 Message formats and serialisation . . . 136
6.3.2 The device node and level hierarchy . . . 139
6.3.3 The parameter . . . 142
6.3.4 Prototyping summary . . . 143
6.3.5 Porting the prototype to the XMOS platform . . . 145
6.4 Integration with the Ethernet AVB reference design . . . 147
6.5 Task structuring within the AES-64 implementation . . . 154
6.5.1 Event flows within the AES-64 implementation . . . 154
6.5.2 Task communications over XC channels . . . 158
6.5.3 Processing example: a ‘GET VAL’ request . . . 164
6.6 Value functions . . . 167
6.6.1 Value functions and parameters . . . 168
6.6.2 Value functions and the stack control task . . . 169
6.7 The stack control task . . . 173
6.7.1 Coordinating input-output of AES-64 messages . . . 173
6.8 The device representation task . . . 174
6.8.1 The device node . . . 174
6.8.2 The parameter and parameter table . . . 177
6.8.3 The level items table . . . 177
6.9 The message processing task . . . 180
6.9.1 Parsing of AES-64 request messages . . . 180

6.9.2 Accessing the device representation . . . 184
6.9.3 Resolution of full address blocks . . . 184
6.9.4 Support for messaging patterns . . . 185
6.10 AES-64 device control . . . 187
6.10.1 AES-64 control of the Ethernet AVB Talker and Listener . . . 187
6.10.2 AES-64 control of the Ethernet AVB media clock server . . . 195
6.10.3 AES-64 control of the CS4272 audio codec . . . 196
6.11 Configuration . . . 204
6.11.1 Protocol stack configuration . . . 204
6.11.2 Configuration of the device representation task . . . 206
6.11.3 Configuration file for the AES-64-enabled Ethernet AVB audio device . . . 207
6.12 The AES-64 connection management and control utilities . . . 208
6.13 Conclusion . . . 209

7 Evaluation **210**
7.1 Introduction . . . 210
7.2 Qualitative evaluation: the AES-64 implementation for Ethernet AVB device control . . . 210
7.2.1 Network discovery . . . 212
7.2.2 Proof of concept: device enumeration . . . 213
7.2.3 Stream connection management . . . 214
7.2.4 Internal connection management . . . 218
7.2.5 Media clock server control . . . 222
7.2.6 Audio hardware control . . . 222
7.2.7 Proof of concept: parameter grouping . . . 223
7.2.8 Limitations of the AES-64 implementation . . . 227
7.3 Quantitative evaluation: the AES-64 implementation . . . 231
7.3.1 Timing measurements methodology . . . 231
7.3.2 Findings . . . 232
7.4 Qualitative assessment: the XMOS XS1 architecture . . . 234
7.4.1 The XC language . . . 235
7.4.2 The XMOS XS1 microcontroller . . . 238
7.4.3 Reference designs . . . 239
7.4.4 Limitations . . . 239
7.5 Conclusion . . . 241

8 Conclusion **242**
8.1 Introduction . . . 242
8.2 Summary of the previous chapters . . . 242
8.3 Review of the research question . . . 243
8.4 Review of the research objectives . . . 244
8.5 Achievements and limitations . . . 245
8.5.1 The AES-64 implementation . . . 245

8.6 Possibilities for further research . . . 247

A AES-64 requirements specification 248
A.1 Introduction . . . 248
A.2 Prerequisites . . . 248
A.3 The AES-64 protocol stack . . . 249
A.4 The device primary control software . . . 249
A.5 Fundamental data structures . . . 249
A.5.1 The Parameter . . . 250
A.5.2 The Parameter Group List . . . 251
A.5.3 The Level . . . 253
A.5.4 The Device Node . . . 257
A.6 Messaging patterns . . . 257
A.7 Message formats . . . 260
A.7.1 Message Header formats . . . 261
A.7.2 AES-64 commands . . . 266
A.8 AES-64 commands and responses . . . 269
A.8.1 ACT SNP . . . 269
A.8.2 CLEAR FLAG . . . 270
A.8.3 GET CLA . . . 270
A.8.4 GET FLAG . . . 271
A.8.5 GET ID . . . 272
A.8.6 GET MSTGRP . . . 272
A.8.7 GET NAME . . . 273
A.8.8 GET PTPGRP . . . 273
A.8.9 GET SLVGRP . . . 274
A.8.10 GET VAL . . . 275
A.8.11 GET VTBL . . . 275
A.8.12 IS ALIVE . . . 276
A.8.13 JOIN MSTSLV . . . 276
A.8.14 JOIN PTP . . . 277
A.8.15 SAVE SNP . . . 278
A.8.16 SET FLAG . . . 278
A.8.17 SET GRPVAL . . . 278
A.8.18 SET MASTER . . . 279
A.8.19 SET MASTER OFF . . . 279
A.8.20 SET MSTGRP . . . 280
A.8.21 SET PTPGRP . . . 280
A.8.22 SET SLAVE . . . 282
A.8.23 SET VAL . . . 282
A.8.24 UNJOIN MSTSLV . . . 283
A.8.25 UNJOIN PTP . . . 283
A.9 Messaging interactions . . . 285
A.9.1 Network discovery . . . 285

A.9.2 Device enumeration . . . 286
A.9.3 Parameter grouping - overview . . . 287
A.9.4 Parameter grouping - setup . . . 288
A.9.5 Parameter grouping - maintenance . . . 294
A.9.6 Parameter grouping - value changes . . . 294
A.9.7 Parameter grouping - teardown . . . 297

B AES-64 protocol stack essential model 301
B.1 Introduction . . . 301
B.2 Context diagram . . . 302
B.3 Event list . . . 302
B.4 Transformation schema . . . 303
B.5 Process specifications . . . 309
B.5.1 Device command handler specifications . . . 313
B.5.2 Level command handler specifications . . . 313
B.5.3 Parameter command handler specifications . . . 315
B.6 Data specification . . . 319

C AES-64 task interfaces 333
C.1 Introduction . . . 333
C.2 Stack control task client interface . . . 333
C.3 Message processing task client interface . . . 335
C.4 Device representation task client interface . . . 336

D Device representation configuration for the XMOS Ethernet AVB streaming audio device 341
D.1 Introduction . . . 341
D.2 Approach . . . 341
D.3 Configuration . . . 342

E The AES-64 utilities 353
E.1 Introduction . . . 353
E.2 Utility documentation . . . 354
E.2.1 a64_discover . . . 354
E.2.2 a64_connect . . . 354
E.2.3 a64_disconnect . . . 355
E.2.4 a64_query . . . 356
E.2.5 a64_channel . . . 358
E.2.6 a64_join_setup_1 . . . 358
E.2.7 a64_join_setup_2 . . . 359
E.2.8 a64_get_mstgrp . . . 359
E.2.9 a64_get_slvgrp . . . 360
E.2.10 a64_mute . . . 360
E.2.11 a64_volume . . . 361

E.2.12 a64_level_check . . . 361
E.2.13 a64_set_clock_type . . . 361
E.2.14 a64_set_clock_rate . . . 361
E.2.15 a64_set_clock_state . . . 362
E.2.16 a64_set_clock_source . . . 362

F Latency timings for the AES-64 implementation 363
F.1 Introduction . . . 363
F.2 Timing measurements . . . 363

List of Figures

1.1 Outline diagram of this thesis. 6

2.1 An AVB network using the star-of-stars topology. Some of the end stations are simultaneously operating as Talkers and Listeners. 14
2.2 Three AVB domains connected as parts of a larger general-purpose network. Slashed ports (/) indicate boundary ports for each AVB domain. 14
2.3 An AVB end station delivering audio sourced from the network to a powered loudspeaker. 15
2.4 The rear panel of a Yamaha O3D mixing console, showing several classes of audio inputs and outputs. [Yamaha Corporation Pro Audio and Digital Musical Instrument Division 1997, 16] 15
2.5 An AVB end station illustrating concurrent bidirectional audio transport. 16
2.6 Talker advertisement propagation in 'avb domain 2' from Figure 2.2. . . 17
2.7 An Ethernet AVB 'multicore' stream, showing the multiplex/demultiplex of multiple audio channels. The Listener in this example is passing channels 1, 2, and 5 to its DAC, but is silently discarding the content from channels 3 and 4. 18
2.8 Example deployment of multichannel audio distribution using Ethernet AVB. 19
2.9 Transport of an audio stream over an AVB network, maintaining the original signal's continuity. 21
2.10 Transport of an audio stream over an AVB network, disrupting the original signal's continuity. 21
2.11 Simultaneous presentation of an audio signal by multiple endpoints, delivering the intended relation. 22
2.12 Simultaneous presentation of an audio signal by multiple endpoints, with failure to deliver the intended content. The third endstation is out of synchronisation with the first two by one beat. 23
2.13 Initiating stream reservation in 'avb domain 2' from Figure 2.2. 32
2.14 Listener responses to stream reservation as initiated in Figure 2.13. 33
2.15 The IEEE 1722 Audio/Video Transport Delivery Unit [IEEE 1722 2012, 8]. 35
2.16 Encapsulation of an IEEE 1722 Audio/Video Transport Delivery Unit within an IEEE 802.3 frame [IEEE 1722 2012, 43]. 36
2.17 IEEE 1722.1 Entity Model of a simple Ethernet AVB endpoint [IEEE P1722.1/D21 2012, 42]. 39

2.18 A controller performs network discovery by issuing an IEEE 1722.1 ADP 'Entity Discover' request. 41
2.19 Devices respond to the controller by sending IEEE 1722.1 ADP 'Entity Available' messages in response to the 'Entity Discover' request. 41
2.20 WireShark capture of an IEEE 1722.1 ADP 'Entity Discover' message as produced by the Universal Media Access Networks 'UNOS Vision' desktop application. 42
2.21 WireShark capture of an IEEE 1722.1 ADP 'Entity Available' message as produced by the XMOS Ethernet AVB reference design IEEE 1722.1 implementation. 43
2.22 The common AVDECC command header format [IEEE 1722 2012, 10] [IEEE P1722.1/D21 2012, 282-5] 43
2.23 IEEE 1722.1 Connection management between a Talker and a Listener. . 44
2.24 A 1722.1 Controller enumerates an XMOS/Attero Tech Ethernet AVB end station. 45
2.25 Specimen response to the IEEE 1722.1 'READ DESCRIPTOR [ENTITY]' request shown in Fig. 2.24. 46
2.26 A simple AVB network, consisting of two XMOS/Attero Tech end stations and a LabX Titanium 411 AVB-capable bridge. 48
2.27 An AVB end station consisting of a DSP4YOU AVBStreamer module (upper board) and a DSP4YOU E-mic module (lower board). The XMOS XS1-G4 processor is beneath the heatsink on the rightmost corner of the upper board. 49
2.28 Mapping of AVB end station software tasks on the XMOS 'low-cost AVB endpoint'. 50

3.1 An Ethernet AVB audio network with a network controller 56
3.2 An Ethernet AVB audio network with multiple controllers, one a mobile device 57
3.3 Schematic of an AES64 device node with configuration parameters, showing the parameter array (R) and the level hierarchy's structuring of the parameter array (L). 60
3.4 Parameter types. The signal meter level (L) can be represented by a read-only parameter; the volume fader control (R) by a read-write parameter. . 60
3.5 The AES-64 device representation provides an interface between a device's control software and AES-64 requests from the network. 61
3.6 An AES64 request message which does not demand a response. 63
3.7 An AES64 request which demands a response, and the corresponding response. 64
3.8 An AES64 request that expects a response, addressing n parameters through the use of a wildcarded address block. Each parameter returns a response message. 65

3.9 Value changes in a parameter group that enforces an 'absolute' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group. 69
3.10 Value changes in a parameter group that enforces a 'relative' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group. 70
3.11 An AES64 request that initiates the target parameter, which is a member of a parameter group, to issue group value update requests. 71
3.12 An AES64 request that initiates the target parameter to set up a parameter group, which involves the target parameter issuing follow-on requests, receiving and processing responses, and issuing further follow-on requests before finally acknowledging success or failure to the source of the original request. (Details of Param C are made available to Param B in the value field of the initiating request sent by Param A.) 72
3.13 A multicore audio distribution cable, which conveys many independent audio signals within a single insulator. AES-64 uses the 'multicore' metaphor to describe the concept of an audio stream, agnostic of the specific audio network in use. 73
3.14 AVB stream connection parameters. 74
3.15 The 'UNOS Vision' AES-64 controller application. 76
3.16 An audio network that implements AES-64 control and monitoring. A central controller is running the 'UNOS Vision' application and 'UNOS Core' protocol stack, while four devices run the 'UNOS Core' protocol stack. 76

4.1 Schematic of the XS1-L2 microcontroller, showing two XS1 processing tiles and their software-configurable switches within a single package. Derived from [XMOS 2013]. 80
4.2 Channel communications between the message processing task and the device representation task to verify the existence of a device node. 85

5.1 The 'receive one request, issue many responses' messaging pattern. 102
5.2 The 'receive one request, issue further requests' messaging pattern used to perform group value updates. 103
5.3 An example of the most complex AES-64 messaging pattern, outlining setup of a master-slave parameter group. 104
5.4 An example transformation schema, showing a data transformation and its two inputs, a control message and a data flow, and its output to a data store. Each type of input-output is denoted by a specific arrowhead. . . . 111
5.5 The context diagram for the AES-64 protocol stack. 112
5.6 The top-level transformation schema for the AES-64 protocol stack. . . . 116

5.7 The 'Receive Message' transformation schema. 117
5.8 The transformation schema for '1.4 Handle Request'. 120
5.9 The transformation schema for '1.4.3 Handle Level Request'. 121
5.10 Task structuring of the essential model's top-level transformation schema (excluding the 'IP Stack' and 'Device Primary Control Software' terminators) resulting in the specification of three communicating tasks: 1, the 'Stack control task'; 2, the 'Message processing task'; 3, the 'Device representation task'. 124
5.11 The concurrent tasks of the AES-64 protocol stack implementation model. 125
5.12 The device representation task and its communications with [L] the stack control task and [R] the message processing task. 127
5.13 The message processing task and its communications with [L] the stack control task and [R] the device representation task. 129
5.14 The stack control task and its communications with [L] the XMOS IP stack and device primary control software, and [R] the message processing task. 130

6.1 The AES-64 implementation tasks and their channels of communication. . 133
6.2 Chapter map for discussion of the AES-64 implementation. 134
6.3 Implementation of the AES-64 standard on the XMOS Ethernet AVB reference design. 135
6.4 Prototyping of core AES-64 functionality. 136
6.5 The AES-64 response message header. 137
6.6 Structure of the prototype device node, illustrating the link between the level hierarchy and the parameter array. 140
6.7 AVB stream connection parameters. 144
6.8 Memory overheads associated with multiple instances ('1' and '2') of a single task ('A') in two different configurations. 153
6.9 Channel communications between the message processing task and the device representation task to process and resolve a full address block supplied by an AES-64 request (see Listing 6.14). 159
6.10 Task communications in the AES-64 protocol stack during the successful processing of a 'GET VAL' request. 165
6.11 Selection and elimination of level item table entries as successive identifiers in a request full address block are parsed, resulting in one level item table entry (and parameter), #10, being identified as the target of the request. Each identifier is checked against the range of level item table entries that matched the previous identifier (in the case of the Section Block identifier, that matched the device node ID), so that: Section Block: checked against entry #1 — #12 Section Type: checked against entry #5 — #10 Section Number: checked against entry #7 — #10 Parameter Block: checked against entry #8 — #10 all subsequent identifiers: checked against entry #10 186

6.12 The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB Talker unit. Multiple Talker units are represented by multiple sets of these parameters, indexed at the 'Parameter Block Index' level. 188
6.13 The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB Listener unit. Multiple Listener units are represented by multiple sets of these parameters, indexed at the 'Parameter Block Index' level. 189
6.14 The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB media clock server. Multiple media clock server units may be represented by multiple sets of these parameters indexed at the 'Parameter Block Index' level. 195
6.15 The level hierarchy for the AES-64 'audio' params. 197
6.16 An expanded protocol stack task diagram showing interactions between the stack control task and the `ptp_gpio_i2c_server()` task to control the CS4272 audio codec. 199

7.1 The level hierarchy for the AES-64 'configuration device node'. 212
7.2 The `a64_discover` utility displays discovered devices and rudimentary enumeration, including the value of the 'Device Name' parameter. 213
7.3 'GET CLA' data returned from the AES-64 capable Ethernet AVB device in response to the insert 'GET CLA' request. 214
7.4 Connection management between AES-64-capable Ethernet AVB devices, via the `a64_connect` utility. 216
7.5 AES-64 messaging for the a64_connect utility. 217
7.6 Teardown of a stream at the Talker, via the `a64_disconnect` utility. 218
7.7 Teardown of a stream at the Listener, via the `a64_disconnect` utility. 218
7.8 Mapping between the media input buffers and the output stream channels on a Talker. The dot-dashed lines indicate dynamic mappings from a media input buffer to a channel in the Talker's multicore stream. In this diagram, the default mappings have been changed: by default, input buffer 1 is mapped to output channel 1, input buffer 3 is mapped to output channel 3, and so on. 219
7.9 Mapping between the input stream channels and media output buffers of a Listener. The dot-dashed lines indicate dynamic mappings from the multicore stream received by the Listener to one of its media output buffers. On the DSP4YOU AVBStreamer board, only the first two media output buffers are connected to actual audio outputs - in this case, the output channels of the CS4272 audio codec. 220
7.10 Multicore channel assignment at an Ethernet AVB device's Listener component using the `a64_channel` utility. The '`l`' argument indicates that the channel assignment is to be performed on a Listener component; '`t`' would specify a Talker component. 221

7.11 The `a64_mute` utility mutes and un-mutes output channels on the CS4272 audio codec. . . . 222
7.12 The `a64_volume` utility sets volume level on output channels of the CS4272 audio codec. . . . 222
7.13 Set-up, verification and teardown of a master-slave group between two parameters on the same device (see Footnote 2 for clarification of '255.255.255.255' address). . . . 224
7.14 Set-up, verification and teardown of a master-slave group between two parameters on different devices. . . . 224
7.15 Group value updates in effect between a master and slave parameter on the same device. . . . 225
7.16 Group value updates in effect between a master and slave parameter on separate devices. . . . 226
7.17 Task mapping of the AES-64-capable Ethernet AVB device on an XS1-G4 microcontroller. The 'demo application' task incorporates the AES-64 stack control task. . . . 228
7.18 Mapping of AVB end station software tasks on the XMOS 'low-cost AVB endpoint'. . . . 229
7.19 AES-64 requests are passed between a Stack Control Task and a Message Processing Task in a shared-memory system, requiring the use of semaphores and mutexes. . . . 236

A.1 Location and identification of parameters within a device node through the level hierarchy. . . . 255
A.2 An AES64 request message which does not demand a response. . . . 258
A.3 An AES64 request which demands a response, and the corresponding response. . . . 258
A.4 An AES64 request that expects a response, addressing n parameters through the use of a wildcarded address block. Each parameter returns a response message. . . . 259
A.5 An AES64 request that initiates the target parameter, which is a member of a parameter group, to issue group value update requests. . . . 259
A.6 An AES64 request that initiates the target parameter to set up a parameter group, which involves the target parameter issuing follow-on requests, receiving and processing responses, and issuing further follow-on requests before finally acknowledging success or failure to the source of the original request. (Details of Param C are made available to Param B in the value field of the initiating request sent by Param A.) . . . 260
A.7 Generic AES-64 message structure. . . . 262
A.8 AES-64 header for a parameter request providing a full address block target and expecting a response. . . . 263
A.9 AES-64 header for a parameter request providing a full address block target and not expecting a response. . . . 263

A.10 AES-64 header for a parameter request providing a parameter index target and expecting a response. 264
A.11 AES-64 header for a parameter request providing a parameter index target and not expecting a response. 264
A.12 AES-64 header for a response message (note mandatory 'Sequence ID' field) 265
A.13 AES-64 header for a device node request expecting a response. 265
A.14 AES-64 header for a device node request not expecting a response. 266
A.15 Example command / response message diagram 269
A.16 Example GET CLA full address block 270
A.17 Example GET NAME full address block 273
A.18 Setup of a master-slave parameter group by JOIN MSTSLV 290
A.19 Setup of a peer-to-peer parameter group by JOIN PTP (1) 292
A.20 Setup of a peer-to-peer parameter group by JOIN PTP (2) 293
A.21 Value changes in a parameter group that enforces an 'absolute' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group. 295
A.22 Value changes in a parameter group that enforces a 'relative' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group. 296
A.23 Teardown of a master-slave parameter group by UNJOIN MSTSLV 298
A.24 Teardown of a master-slave parameter group by UNJOIN PTP 300

B.1 Context diagram for the AES-64 protocol stack 302
B.2 Top-level transformation schema for the AES-64 protocol stack. 304
B.3 The 'Handle Request' transformation schema. 305
B.4 The 'Handle Device Request' transformation schema. 306
B.5 The 'Handle Level Request' transformation schema. 306
B.6 The 'Handle Parameter Request' transformation schema. 307
B.7 The 'Receive Message' transformation schema. 308

E.1 Generic AES-64 messaging patterns: (a) 'Get Val' request; (b) 'Set Val' request requiring a response; (c) 'Set Val' request not requiring a response 355
E.2 AES-64 messaging for the a64_discover utility. 356
E.3 AES-64 messaging for the a64_connect utility. 357
E.4 AES-64 messaging for set up of a master-slave join. This diagram applies to both join setup utilities, although when a join is established by a64_join_setup_1, the slave parameter will be on the same device as the master parameter. 360

List of Tables

5.1 Data specification notation 114
5.2 Required operations to enable processing of AES-64 requests 128

6.1 Example parameters and their value functions 169
6.2 Six forms of AES-64 request 181

7.1 AES-64 'full address block' request processing times in microseconds . . . 233

A.1 Parameter data specification 252
A.2 Parameter group list data specification 253
A.3 Level data specification 256
A.4 Device node data specification 258
A.5 Message Type field values 261
A.6 Command Executive values 267
A.7 Command Qualifier values 267
A.8 Parameter Requests 268
A.9 Device Requests 268

Listings

2.1 Execution threads in the XMOS AVB application (part 1) 51
2.2 Execution threads in the XMOS AVB application (part 2) 52
2.3 Event handling in the demo() thread 53

4.1 Task allocation 81
4.2 Event registration using the select statement 82
4.3 Example of XC channel communications to perform an AES-64 query and receive the result 86
4.4 Example of XC channel communications to receive an AES-64 query . . . 86
4.5 Use of an XC transaction to pass an AES-64 full address block over a channel 88
4.6 Reinterpretation 89
4.7 Passing arguments by reference in XC 90
4.8 Passing arguments by reference in C 90
4.9 The device representation thread waits for events 91
4.10 The device representation task receives a request from the message processing task and validates the request's command code 92
4.11 The device representation thread locates the correct command handler . . 93
4.12 The command handler for validating a device node ID 93
4.13 The function prototype for dnm_get_dnode_id_data 94

5.1 A Wireshark capture of an example 'GET CLA' request 98
5.2 A Wireshark capture of the response to the 'GET CLA' request shown in Listing 5.1 99
5.3 Example process specification for Fig. 5.4 111
5.4 Example of data specification for an AES-64 Device Node 114
5.5 Specification of '1.1 Control AES-64 Protocol Stack' 118
5.6 Specification of '1.5.1 Control Handle Receive Message' 118
5.7 Specification of '1.3 Transmit AES-64 Message' 119
5.8 Specification of '1.4.2 Validate Request' 121
5.9 Specification of '1.4.3.2 Process Command' 122
5.10 Specification of the AES-64 'GET CLA' command handler 122
5.11 A message processing function employs an XC channel-based interface to query the existence of an identified device node on the device representation task. 126

6.1 Data structure and serialisation macro for the AES-64 response message header (deserialisation macro not shown) . . . 138
6.2 Data structure and de-serialisation routine for JOIN MSTSLV value field 139
6.3 Prototype device node . . . 140
6.4 Creation of two parameters and their level hierarchy entries using the prototype device node . . . 141
6.5 The first basic parser for AES-64 full address block requests . . . 142
6.6 The prototype parameter structure . . . 143
6.7 Some of the XMOS Ethernet AVB API functions for AVB stream connection 144
6.8 Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 1) . . . 149
6.9 Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 2) . . . 150
6.10 Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 3) . . . 151
6.11 The top-level function for the device representation task . . . 155
6.12 The select statement for the demo function (including the stack control task) 156
6.13 Handler for the stack control task's client API . . . 157
6.14 Resolution of a full address block performed over an XC channel . . . 160
6.15 The device representation task's handler for resolving full address blocks . 161
6.16 The stack control task's client API . . . 163
6.17 Value function classes . . . 169
6.18 The stack control task's handler for 'get' value function requests from the message processing task . . . 171
6.19 The value function handler for an Ethernet AVB Listener's 'Stream ID' parameter . . . 172
6.20 The AES-64 device node (part 1) . . . 175
6.21 The AES-64 device node (part 2) . . . 176
6.22 The AES-64 parameter table and parameter implementation . . . 178
6.23 The AES-64 level items table . . . 179
6.24 The message processing structure and functions . . . 182
6.25 Basic message parsing in the AES-64 stack . . . 183
6.26 Excerpt of device representation task API showing operation targets . . . 184
6.27 Implementation of transmit request via the stack control task . . . 187
6.28 An AES-64 value function for the Talker 'VLAN ID' parameter . . . 192
6.29 The AES-64 value function for Ethernet AVB Talker channel map parameters 193
6.30 The I^2C interface instance . . . 198
6.31 The audio hardware control commands . . . 200
6.32 Event selection in the `ptp_gpio_i2c_server()` task . . . 201
6.33 The `ptp_gpio_i2c_handle_set_output_level()` function . . . 202
6.34 Setting output channel level via the I^2C interface . . . 202
6.35 The AES-64 value function for CS4272 output channel level . . . 203
6.36 Configuration of multiple message processing tasks . . . 205
6.37 Configuration of the device representation task . . . 206

6.38 Creating a level items table entry and a device parameter 207

tspecs/TSpec11 309
tspecs/TSpec12 309
tspecs/TSpec13 309
tspecs/TSpec1411 309
tspecs/TSpec1412 310
tspecs/TSpec142 310
tspecs/TSpec1431 310
tspecs/TSpec1432 310
tspecs/TSpec1441 311
tspecs/TSpec1442 311
tspecs/TSpec151 311
tspecs/TSpec152 312
tspecs/TSpec153 312
tspecs/TSpec154 312
tspecs/TSpec16 312
cspecs/IsAlive 313
cspecs/GetCLA 313
cspecs/GetNAME 314
cspecs/GetID 315
cspecs/GetMSTGRP 315
cspecs/GetPTPGRP 315
cspecs/GetSLVGRP 315
cspecs/GetVAL 316
cspecs/JoinMSTSLV 316
cspecs/SetGRPVAL 316
cspecs/SetMASTER 316
cspecs/SetMASTEROFF 317
cspecs/SetMSTGRP 317
cspecs/SetPTPGRP 317
cspecs/SetSLAVE 317
cspecs/SetVAL 318
cspecs/UnjoinMSTSLV 319

./a64_control_api.h 333
./a64_msg_proc_client_api.h 335
./a64_dnm_client_api.h 336

D.1 Important configuration function prototypes 343
./a64devrep.c 344

Acknowledgments

I wish to thank Professor Richard Foss for his dedicated supervision of this research project and thesis. I am very grateful to have worked with Richard on this project: his patience and rigorous attention to detail have been invaluable.

I would like to acknowledge the support of the Distributed Multimedia Centre of Excellence, which receives financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107).

I wish to thank my lab-mates and alumni of the Audio Research Group for the prior research and discussions that have contributed to this work: Philip Foulkes, Osedum Igumbor, Nyasha Chigwamba and Bradley Klinkradt.

I would like to thank the staff and students of the Department of Computer Science, particularly the students of the 2012 and 2013 Audio Networks Honours courses.

I would like to thank Ali Dixon and Andy Lucas of XMOS for the discussions that contributed to this work.

I wish to thank my family for their love and faith in my commitment to this work. From my father I received a passion for engineering, from my mother I received a love of the written word, and from both my parents I received the love of learning. From my sister and brother I received all the encouragement and support I could have asked for, and then some. Thank you very much.

Finally, I wish to thank my partner Nishlyn for his love, succour and forgiveness, and for making the possible so much more possible. Thank you.

1 Introduction

Audio distribution occurs in many professional domains: production facilities, broadcasting, the creative arts, and public address systems [N. Bouillot, et al 2009] [AES Standards Committee 2008] [Gross 2006]. Current audio distribution systems are typically configured around point-to-point connections between system components, meaning that signal routing is largely determined by physical cabling. This approach presents problems for system implementers and operators, including issues of scalability, fault tolerance, management and dynamic reconfiguration [Rumsey 2009].

Audio distribution over a specialised form of data network has the potential to address these problems by decoupling audio transport from physical cabling. A network-based approach to audio distribution enables audio transport between any two (or more) devices on the network. Consequently, audio networks can be designed and implemented for redundancy and scalability with far more ease than conventional systems [N. Bouillot, et al 2009]. Audio networks show great potential for enhancing audio distribution in all of its operating contexts, promising improved control and monitoring, opportunity for more redundant and resilient distribution, and greater scalability [AES Standards Committee 2008].

Early forms of audio network were typically proprietary systems, requiring expert configuration and offering limited scope for interoperability. Recent developments in audio network technology include Ethernet Audio/Video Bridging ('Ethernet AVB'), a set of open standards for audio transport, control and quality of service networking that adapt and extend conventional Ethernet to meet the demanding requirements of real-time audio distribution [IEEE 802.1BA/D2.1 2010] [Boatright 2009].

The opportunity for greater control, instrumentation and connection management of audio distribution is considered to be a central benefit of audio network technology [Gross 2006]. Software control enables dynamic routing of audio, increasing resilience to equipment or link failure [Rumsey 2009]; it also provides a sound basis for performance monitoring and proactive system maintenance [Gross 2006]. To achieve this, software controllers and network devices must implement some form of standard that comfortably encapsulates high-level system operations, insulating the consumer or professional end-

user from unfamiliar concepts, terminology and practices. One such standard is the Audio Engineering Society's AES-64 standard [Audio Engineering Society 2012].

An audio control standard provides a standardised distributed control system for an audio distribution network. It is expected to operate in three related areas:

1. to represent the connection management, control and monitoring functionality of arbitrarily complex audio devices;

2. to specify protocols by which those devices may communicate control and monitoring information between each other and to / from controller systems (such as PC or tablet applications);

3. to encapsulate commonplace application tasks and present the real-time audio distribution network using familiar metaphors.

Any implementation of an audio control standard must interface with the primary control software of an audio device. This means that research and development of audio control standards depends in part on the availability of audio devices that provide programming interfaces for their primary control software.

A number of Ethernet AVB streaming audio devices based around the XMOS XS1 microcontroller [XMOS 2011c] [DSP4YOU 2012] satisfy this requirement, where the devices include a JTAG programming interface. A 'reference design' providing basic streaming functions is available as open-source software [XMOS 2011]. The result is a standards-compliant and actively-maintained implementation of an Ethernet AVB streaming audio device that (in addition to providing a programming interface for its primary control software) may be reconfigured and adapted freely [XMOS 2012a]. This is enabled by the architecture of the XMOS XS1 microcontroller, which provides hardware and language support for concurrency, event-driven software design and high-speed interfacing to external hardware [XMOS 2012b] [XMOS 2013] [May 2009a]: as a result, even the lowest-level behaviour of the Ethernet AVB streaming audio device is defined in software rather than digital design.

This research explores the viability of the XMOS XS1 architecture as a basis for the research and development of audio control standards. This is evaluated through the design and development of an implementation of an AES-64 protocol stack and its integration with an XMOS XS1-based Ethernet AVB streaming audio device to provide connection management and control services.

1.1 Research question

This research investigates whether the XMOS XS1 architecture and its supporting features offer a compelling platform for research and development of audio control standards, focusing on the XMOS XS1's associated 'XC' programming language and its features [XMOS 2011b] [XMOS 2011a]. In other words, this research is not focused on investigating the implementation of the XMOS XS1 microcontroller itself, but the programming language and approach it facilitates. In particular, the research discusses the XC language's support for concurrency, event-based programming and integration with ANSI C in the context of implementing a demanding audio control standard and integrating that implementation into an existing concurrent application.

The research presents a qualitative evaluation of the AES-64 implementation, considered as an effective implementation of the AES-64 standard and as an effective control system for the Ethernet AVB streaming audio device. It also presents a quantitative evaluation of the implementation's latency (or reaction time) for common connection management and control operations. The research also discusses in detail the design and implementation methods used in the development of the AES-64 implementation, evaluating the capabilities and limitations of the XS1 architecture to support efficient development and an understandable, well-structured software implementation of a demanding set of requirements.

1.2 Research objectives

The research described in this thesis was constructed to achieve the following objectives:

- Investigate the capabilities and limitations of the XMOS XS1 architecture by implementing an AES-64 protocol stack and integrating it with the XMOS implementation of an Ethernet AVB streaming audio device to provide:
  - Ethernet AVB connection management and control.
  - Proof-of-concept implementations of advanced AES-64 control mechanisms.
- Evaluate the effectiveness of the resulting AES-64-controlled Ethernet AVB device as an implementation of the AES-64 standard, as a form of control over the Ethernet AVB streaming audio device, and as a low-latency control system for Ethernet AVB audio networks.
- Evaluate the capabilities and limitations of the XMOS XS1 architecture as a platform for research and development of audio control protocols on the strength

of the produced software product and of the efficiency of the software design and development process.

- Identify and understand the requirements of a typical audio control standard by carrying out requirements analysis and constructing a design document for the published AES-64 audio control standard.

1.3 Chapter layout

Fig. 1.1 shows the outline structure of this thesis, including the relation of the appendices to thesis chapters.

Chapter 2 introduces audio networks, discusses the Ethernet AVB standards, and describes the XMOS 'reference design' implementation of an Ethernet AVB streaming audio device.

Chapter 3 discusses the significance of control and monitoring services to audio networks and provides a detailed overview of the Audio Engineering Society's AES-64 audio control standard.

Chapter 4 provides a description of the XMOS XS1 microcontroller, its resources and conventions, including an overview of the XC programming language and basic techniques for programming concurrent applications.

Chapter 5 describes the requirements analysis and design processes used to develop the XMOS XS1-based implementation of the AES-64 standard, the application of the Real-Time Structured Analysis & Design (RTSAD) methodology, and the mapping of the final RTSAD design onto the XMOS XS1 architecture.

Chapter 6 discusses the development of an AES-64 implementation for the XMOS Ethernet AVB reference design, including the prototyping and test procedures used, the translation of the design mapping into a set of concurrent tasks, the interface between the AES-64 implementation and the Ethernet AVB control subsystems, and the configuration of the AES-64 device representation. It also introduces the command-line tools developed to perform connection management and control over the AES-64-enabled Ethernet AVB device, which are described in detail in Appendix E.

Chapter 7 presents qualitative and quantitative evaluations of the AES-64 implementation for the XMOS Ethernet AVB reference design and of the XMOS XS1 architecture as a platform for the research and development of audio control standards.

Chapter 8 presents the conclusions of the research.

Appendix A is a requirements specification for the AES-64 standard, the development of which is described in Chapter 5.

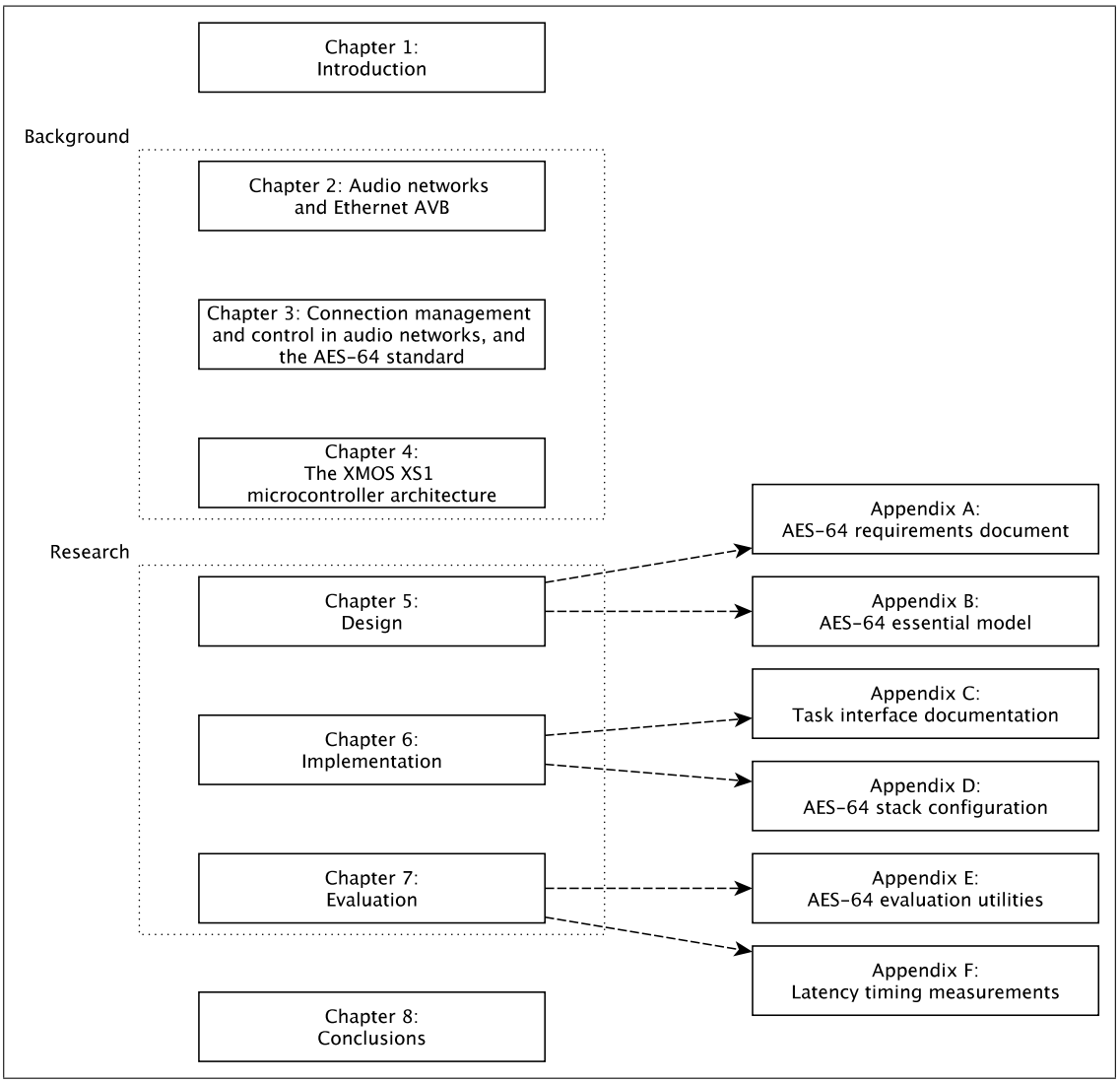


Figure 1.1: Outline diagram of this thesis.

Appendix B documents the essential and implementation models prepared in the software design process.

Appendix C documents the interfaces of the device representation task, message processing task and stack control task within the AES-64 implementation.

Appendix D presents a specimen configuration file for the XMOS XS1-based AES-64 implementation, as described in Chapter 6, which configures the AES-64 representation of an XMOS Ethernet AVB device.

Appendix E describes the command-line tools developed to perform connection management and control over the AES-64-enabled Ethernet AVB device.

Appendix F presents the compiled timing measurements used in the quantitative evaluation of the AES-64 implementation (Section 7.3).

2 Audio networks and Ethernet AVB

2.1 Introduction

This chapter discusses the Ethernet Audio Video Bridging ('Ethernet AVB') technical standards as a platform for audio distribution networks ('audio networks'), illustrated by examples of audio distribution applications. The chapter provides a summary account of the Ethernet AVB standards and how they meet the demanding requirements of audio distribution.

The chapter also introduces the XMOS Ethernet AVB 'reference design' [XMOS 2011], which formed the practical foundation of the research project.

2.1.1 Audio distribution applications and requirements

Audio distribution applications are diverse and present many requirements [AES Standards Committee 2008], of which some generalisations can be made:

1. there is a requirement to transport independent channels of audio and control data;
2. there is a requirement to synchronise transport of some or all of the channels;
3. audio and control data is typically real-time, in which case:
    a) continuity of transport must be guarded;
    b) buffering of transport must be minimised.

In conventional audio distribution environments, these requirements are met through complex configurations of 'point-to-point connections, switching systems and processors that are centred on specific media formats' [Foulkes 2011, 1]. Independent audio channels may be distributed on cables between devices, while control data (e.g., MIDI signals) governing the operation of those devices may be communicated over a different set of cables and interfaces.

Some devices within a configuration may not be capable of direct communication with all other devices, particularly where these devices implement proprietary control and/or

transport interfaces. In this case, conversion processors may be required to 'bridge' incompatible sections of the configuration.

Preparing a workable configuration may require expert knowledge and careful testing, in part due to the real possibility of incompatibility between devices. Once a configuration has been prepared, or is in active service, changing or adding to it may be difficult. In particular, the ability to re-route an audio channel within a live configuration may depend on whether the transport cable for that channel can be physically disconnected from the old destination and re-connected to the new destination. If the source is not directly connected to either destination, but reaches them by routing through other devices, the complexity of this operation increases. In any case, the re-routing is not instantaneous.

These and other factors can mean that conventional distribution systems do not fully meet the requirements of audio distribution applications. In a 2005 survey of audio engineers conducted by the Audio Engineering Society Technical Committee on Network Audio Systems (TC-NAS),

> Among the interesting results from the survey was a reported required level of reliably [*sic*: 'reliability'] well in excess of the capabilities of incumbent non-networked distribution systems [Gross 2006, 66]:

2.1.2 Audio distribution networks

Audio distribution network technologies aim to rationalise audio distribution applications. Exact approaches vary, but typically involve the following general goals:

1. achieving reductions in physical cabling through multiplexed signals;
2. making a given audio signal available throughout the full network;
3. audio transport using digital encoding;
4. committing to real-time transport, control, monitoring and connection management;
5. providing high-level control and management applications.

Audio networks typically provide a unified real-time transport for all application data: for example, each of the audio network technologies surveyed by [N. Bouillot, et al 2009] uses a common transport medium for audio and control data [Ibid., 735]. Audio networks typically also enable the transport of multiple independent channels over a single cable; this is provided by many non-network digital audio transports (e.g., AES-3

[Audio Engineering Society 2003]) and represents substantial cost savings in cabling and cartage, but is impossible to achieve in analogue audio transport.

Network topologies vary [N. Bouillot, et al 2009, 735], but audio networks typically make the source of an audio channel available for connection to any destination available on the network; connections may be one-to-one or one-to-many.

By definition, audio signals distributed by an audio network are *digitally encoded*; an audio network may support the transport of one or more digital audio encoding formats. In transporting audio, an audio network also commits to transporting the *clock information* that a destination device must receive in order to accurately regenerate audio from the digitally encoded signal. Real-time transport of audio across a network infrastructure presents a number of challenges, which shall be discussed shortly.

Audio networks must also provide or support the provision of high-level control and management tools. In some use cases – for example, the 'installed sound' category of audio applications, which concerns public address systems for institutions, mass transit, airports, etc. – these are particularly important, although the use cases described in [AES Standards Committee 2008] identify automation and management tools as significant requirements in every category of professional audio application.

In addition to this rationalisation effort, audio networks have been identified as showing potential for significant advances in audiovisual presentation:

> The AES Technical Committee on Network Audio Systems (TC-NAS) was chartered in 1998 when it was recognised that the increase in network bandwidth and quality of service made possible brand new classes of audiovisual applications, as well as unconventional and high-performance implementations of existing ones [Gross 2006, 62].

To provide these and other benefits, however, audio networks must first be capable of audio transport over a network medium. As indicated earlier, this presents a number of difficulties.

2.1.3 Providing quality of service for audio transport

The distribution of professional-quality audio is a high bandwidth application: Bouillot [2009, 736] gives a figure of 32 audio channels as a maximum load for a 100Mbps Ethernet network. Combined with the fact that audio data streams are time-sensitive to a degree that many conventional data transmissions are not, this means that the quality-of-service ('QoS') requirements for an audio network are extensive and challenging:

> This is due to several interrelated characteristics distinct to audio: the constraint of low-latency, the demand for synchronization, and the desire for high fidelity, especially in the production environment, which discourages the use of lossy encoding and decoding (codec) processes [N. Bouillot, et al 2009, 730].

The quality-of-service requirements for audio distribution may be summarised as:

1. Latency incurred in distribution must be minimal and deterministic.
2. Buffering and error correction increase latency, so both are to be minimised. As a result, the bandwidth required to distribute a stream must be reserved for the full duration of the stream transmission.
3. Digital audio transport mandates the simultaneous transport of the corresponding *clock signal*.
4. Audio transport to multiple destinations requires a mechanism allowing the multiple destinations to synchronise (i.e., make simultaneous) their presentation of the audio.

An audio network must be capable of satisfying these requirements to provide effective audio transport. The Audio Engineering Society has quantified maximum acceptable latency in audio transport on an application-by-application basis: tolerances given in [AES Standards Committee 2008] range from '10ms or less' for a portable recording studio [6] to '[less than] 3ms' for live sound in a concert hall [15]. The most commonly-cited figure, however, is reported by[Gross 2006, 65]:

> In all application areas, the most common absolute latency requirement cited was "less than 5 ms."

2.1.4 Audio network technologies

As indicated previously and as surveyed in [N. Bouillot, et al 2009], a number of technologies implement audio distribution networks with QoS control. These include CobraNet and EtherSound, each of which provide proprietary QoS audio distribution networks above Layer 2 Ethernet [N. Bouillot, et al 2009, 735]. However, [Gross 2006, 66] identifies a 'desire to see open and multi-sourced technology standards for audio networking'.

2.2 Ethernet AVB: an overview

The IEEE's Ethernet Audio/Video Bridging ('Ethernet AVB') project was tasked with providing a standardised technology that could provide network transport services for streaming audio, video, and other time-sensitive data. Ethernet AVB includes provisions that:

1. transport digital audio, with support for clock recovery, diverse encoding formats and selective multiplexing/demultiplexing
2. guarantee quality of service for audio transport;
3. permit the use of converged networks;
4. distribute time information, enabling
   a) communication of clock signals;
   b) latency compensation;
   c) synchronised presentation of audio streams.
5. support high-level control and monitoring systems

Ethernet AVB provides the following components:

1. IEEE 1722, a standard for audio/video transport across QoS networks;
2. IEEE 1722.1, a standard for the discovery, enumeration, connection management and control of IEEE 1722-compatible devices;
3. IEEE 802.1AS, a targeted profile of the IEEE 1588 Precision Time Protocol standard, which enables IEEE 1722 devices to share a distributed reference clock;
4. IEEE 802.1Qat and IEEE 802.1Qav, amendments and extensions to the IEEE 802.1Q VLAN standard that jointly provide quality of service (QoS) networking for real-time audio/video services[1];
5. IEEE 802.1BA, which provides descriptive profiles of Ethernet AVB devices and infrastructure.

[1] Video transport is beyond the scope of this project, and throughout the rest of this document Ethernet AVB technology is discussed solely as a medium for audio distribution networking.

Ethernet AVB aims to enable the simultaneous transport of streaming audio/video data and conventional best-effort data on a common infrastructure. Since streaming audio/video data is a high-bandwidth network application and conventional data bandwidth consumption is less predictable, Ethernet AVB manages the allocation of network resources at a low level. For example, the transport of an audio stream on an Ethernet AVB network takes place within a bridged virtual network, where the only hosts within the virtual network are those that have formally connected to the stream: this conserves bandwidth in 'uninterested' sections of the network. IEEE 802.1Qav provides priority transmission mechanisms that can flexibly allocate up to 75% of transmission opportunity on a given port to streaming audio traffic (unused transmission opportunity is released to conventional traffic). IEEE 802.1Qat provides an arbitration protocol that advertises the bandwidth requirements of a given stream ahead of transmission and either secures the required transmit allocations between the source and its destination(s), or returns a failure condition; if the route(s) to the destination(s) cannot allocate the required bandwidth, the stream connection cannot be made.

2.2.1 The Ethernet AVB network

An Ethernet AVB network ('AVB network' hereafter) comprises two or more *end stations* connected via *bridges*. Each end station in an AVB network is connected to precisely one bridge. Bridges filter and forward AVB and non-AVB traffic between end stations and other bridges.

AVB networks employ a star or star-of-stars topology (Figure 2.1, adapted from [IEEE 802.1BA/D2.1 2010, 6]). The IEEE standards anticipate that an AVB network may form part of a larger general-purpose network (Figure 2.2, adapted from [IEEE 802.1BA/D2.1 2010, 7]), and support this by:

1. automatically discovering the boundaries of an AVB-capable network
2. constraining AVB traffic within those boundaries
3. permitting general-purpose traffic to be distributed on the AVB-capable network, albeit at a lower priority than audio distribution traffic.

End stations are devices which can interface between the AVB network and conventional audio devices. A very simple example is shown in Figure 2.3. This end station delivers an audio stream from the AVB network to an amplified speaker system: any data that is transmitted to the end station via the AVB transport stream will be presented to a digital-analogue converter (DAC) and output as analog audio over the speaker system.

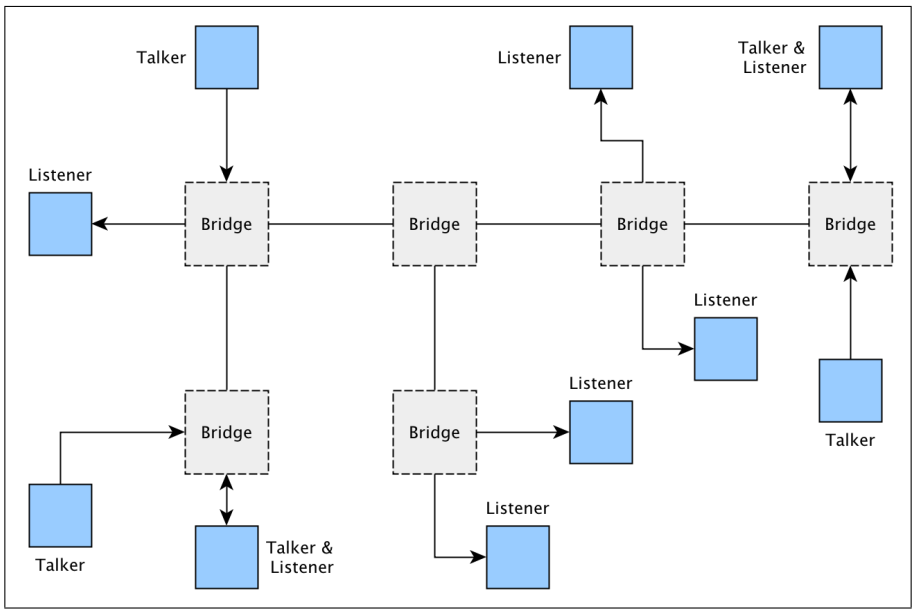


Figure 2.1: An AVB network using the star-of-stars topology. Some of the end stations are simultaneously operating as Talkers and Listeners.

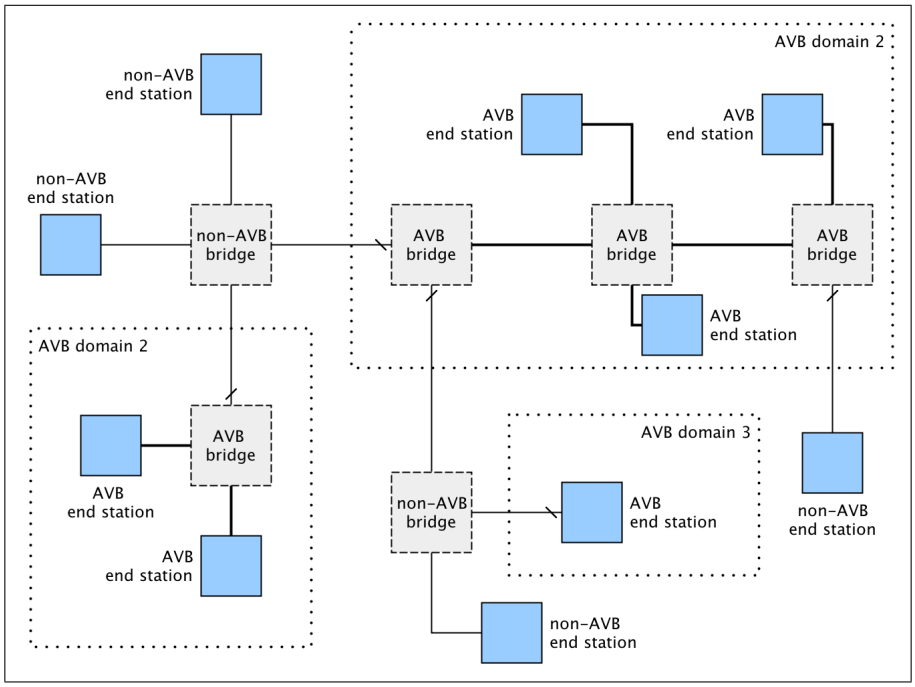


Figure 2.2: Three AVB domains connected as parts of a larger general-purpose network. Slashed ports (/) indicate boundary ports for each AVB domain.

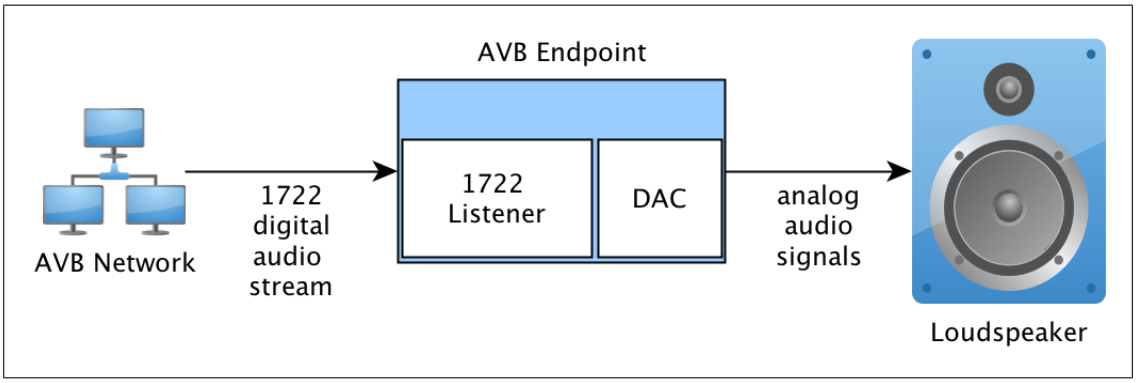


Figure 2.3: An AVB end station delivering audio sourced from the network to a powered loudspeaker.

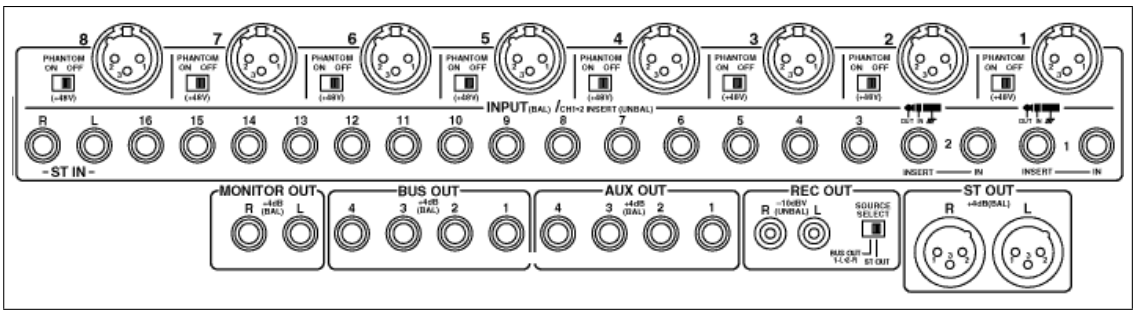


Figure 2.4: The rear panel of a Yamaha O3D mixing console, showing several classes of audio inputs and outputs. [Yamaha Corporation Pro Audio and Digital Musical Instrument Division 1997, 16]

Since the AVB network links are bidirectional, and many classes of audio device input and output signals simultaneously (the canonical example being the audio mixing console, or 'mixer' (Figure 2.4)), end stations are capable of simultaneous input and output to the AVB network (Figure 2.5).

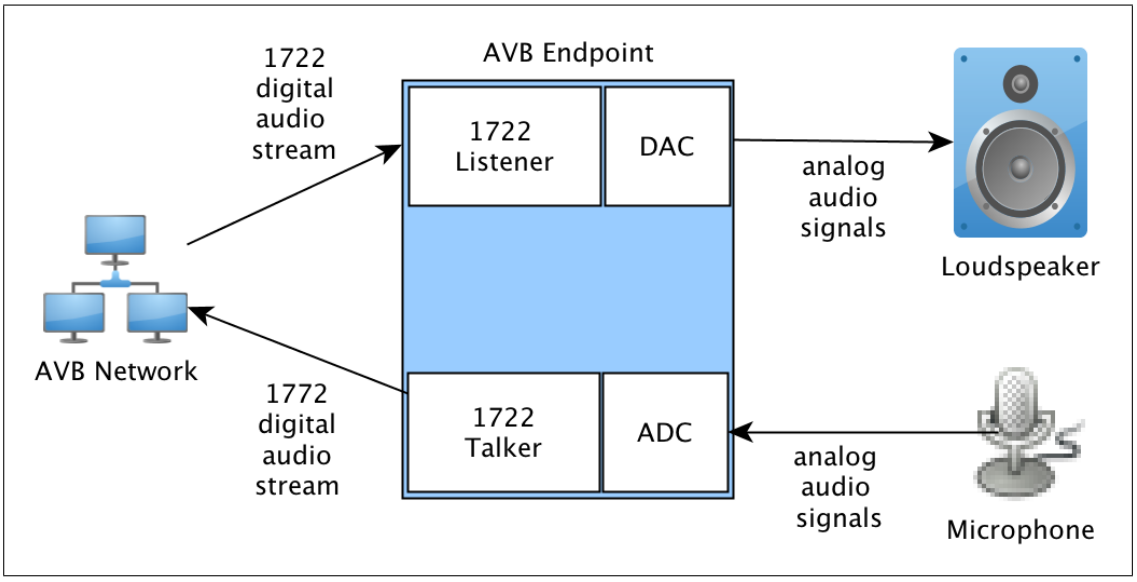


Figure 2.5: An AVB end station illustrating concurrent bidirectional audio transport.

2.2.2 Audio distribution over Ethernet AVB

An audio signal (or set of related signals) is transported across an AVB network as an IEEE 1722 *stream*. An end station that delivers audio onto the network as a stream is termed a *Talker*. An end station that receives an audio stream from the network and presents it elsewhere — through a speaker system, for instance, or over a point-to-point link to some other audio device — is termed a *Listener* [IEEE 802.1BA/D2.1 2010, 4]. Notably, a single end station may simultaneously act as both a Talker and a Listener[2].

Before a Talker can deliver audio onto the AVB network, it must register a stream. It registers the stream by declaring an IEEE 802.1Qat *advertisement* that propagates to every bridge and end station within the AVB network. The advertisement specifies several important details about the prospective stream: the priority of the stream, often expressed in terms of a 'traffic class'; its estimated bandwidth consumption (based on

[2]In theory, a network device may present multiple Talker and Listener interfaces to the network, meaning that the device can simultaneously transmit multiple streams and/or simultaneously receive multiple streams. This order of configuration is outside the scope of this study.

metrics including the sampling rate and word size of the audio format conveyed by the stream); and the maximum permissible transport latency.

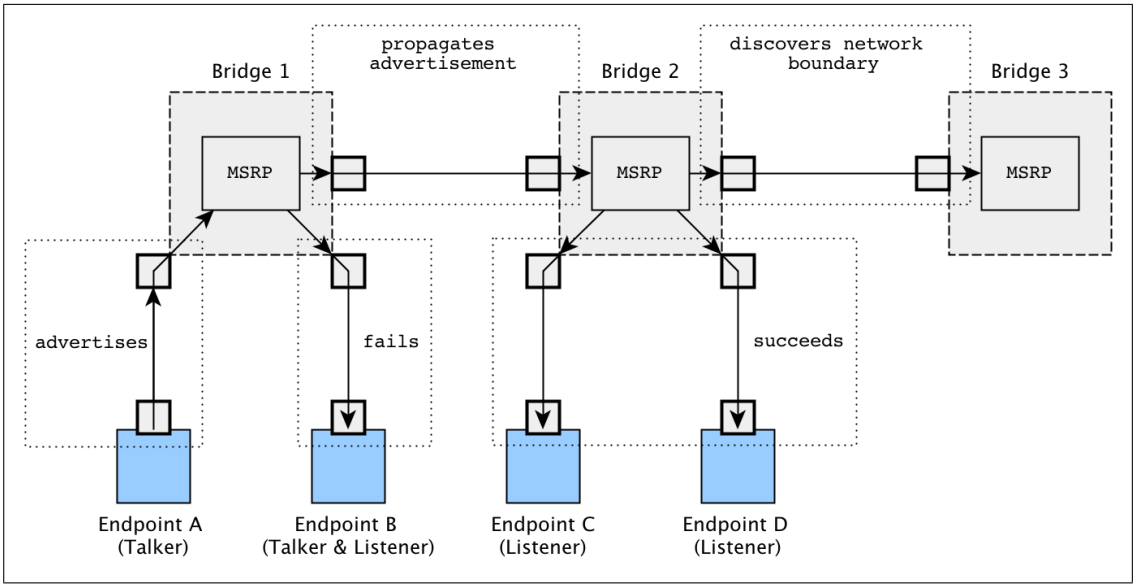


Figure 2.6: Talker advertisement propagation in 'avb domain 2' from Figure 2.2.

As Endpoint A's Talker advertisement propagates across the network (Figure 2.6), it accumulates quality-of-service information. Each bridge on the AVB network will assess the advertisement's stated resource requirements against the stated available resources on each of the bridge's ports. If a port on the bridge is *not* capable of supporting the advertised resource requirements, the advertisement will be updated to indicate a failure state before being forwarded out of the port, as shown on the link between Bridge 1 and Endpoint B. If a port on the bridge has no problem meeting the resource requirements, the advertisement will be forwarded out of the port as-is, as shown on the links between Bridge 2 and Endpoints C and D.

When the advertisement ultimately reaches another end station on the AVB network, the end station may choose to register an interest in receiving the stream. (Whether the end station *is* interested in receiving the stream is typically determined by connection management messages sent to that end station from a control system. IEEE 1722.1 provides one such connection management protocol.) If the end station has no interest in receiving the stream, it makes no response. If it does have an interest, the end station will make one of two initial responses:

1. "I am a prospective Listener. There are sufficient resources for you to transmit your stream to this end station."

2. "I am a prospective Listener. Do not attempt to deliver your stream, as there are insufficient resources to reach this end station."

Responses from prospective Listeners are concatenated by the intermediary bridges into a single response to the Talker's initial advertisement. As long as at least one Listener on the AVB network is both willing and capable of receiving the Talker's stream, the Talker will start to stream audio.

2.2.3 Audio encapsulation

An IEEE 1722 stream may convey data for many separate audio channels (Figure 2.7). These channels may be multiplexed or demultiplexed into / from a single stream by the transmitting end station or receiving end station(s), but not by a bridge.

IEEE 1722 streams are capable of encapsulating a variety of digital audio encoding formats through the provisions of the IEC 61883-6 audio and music data transmission protocol [IEEE 802.1AS 2011, 17] [IEC 2005].

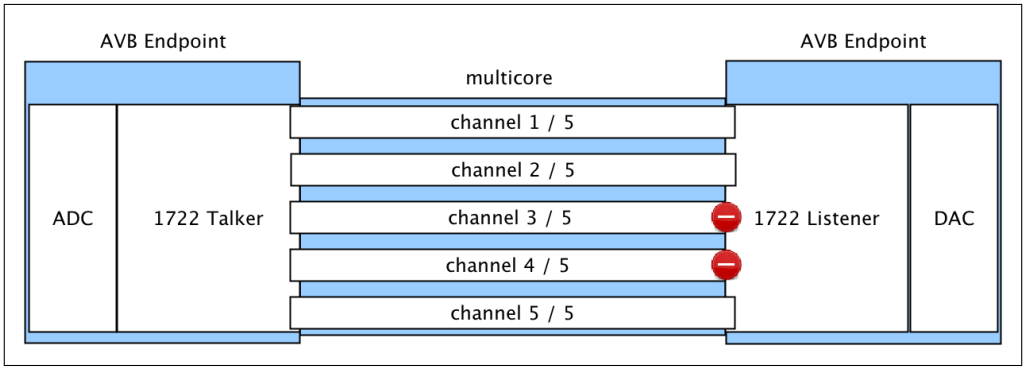


Figure 2.7: An Ethernet AVB 'multicore' stream, showing the multiplex/demultiplex of multiple audio channels. The Listener in this example is passing channels 1, 2, and 5 to its DAC, but is silently discarding the content from channels 3 and 4.

2.2.4 Ethernet AVB use case: live sound

Figure 2.8 depicts a simple Ethernet AVB system designed for live sound. In this example, four choral singers (soprano, alto, tenor and bass) use microphones connected to the analogue audio interface of an Ethernet AVB endpoint. The endpoint 'sees' four discrete analogue audio signals, which it accordingly encodes as four discrete channels of digital audio. The Talker component of the endpoint has been configured to multiplex these four channels into a single Ethernet AVB stream.

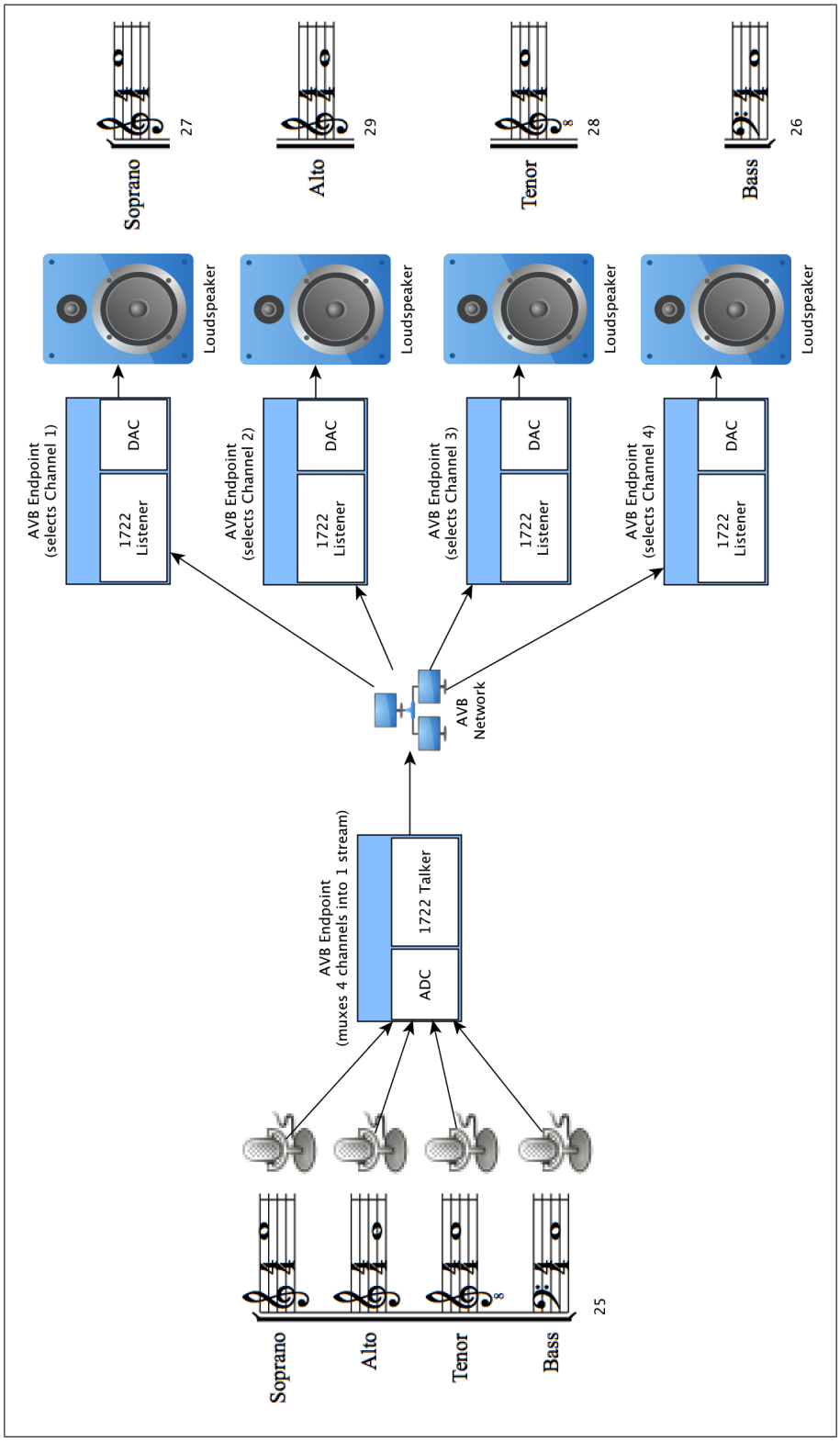


Figure 2.8: Example deployment of multichannel audio distribution using Ethernet AVB.

This stream is transmitted onto the AVB network, where four Listener endpoints receive the stream. Each Listener endpoint has been configured to select one and only one channel from the stream, with the result that each of the original four signals is presented through a dedicated speaker system.

This type of deployment allows for a variety of useful musical considerations: for example, a signal containing a certain type of musical content can be directed to a speaker system with optimal design and placement to reproduce that type of content. Alternatively, speaker systems may be placed around an auditorium for spatialization or other creative effects [Zvonar 2006].

2.2.5 Quality of service in Ethernet AVB

The Ethernet AVB network has been designed to provide guaranteed quality of service for real-time audio distribution. 'Quality of service' in this context highlights a number of issues:

1. Latency — the time lag between transmission and reception of the audio stream. Latency incurred by transport over the network must be minimal, and deterministic (where possible). Where latency is *not* deterministic (for example, within a bridge), it must be *bounded*.

2. Continuity — the continuous quality of individual signals. The continuity of the original audio transmission must be preserved. In the case of an AVB network, streaming audio is a packetised representation of an effectively continuous signal (compare Figures 2.9 and 2.10). In other applications this is often achieved through the use of buffering; however, any significant amount of buffering will conflict with the system's realtime commitment.

3. Synchronization — the temporal consistency of multiple signals. Where audio signals are independent but deliver related content (e.g., the stems of an audio mix), a transport network must be capable of distributing the set of related independent signals to their respective destinations in a way that preserves the original synchronisation between the audio signals (for a simplified example, compare Figures 2.11 and 2.12).

4. Clock recovery — the temporal consistency of individual signals. Digital audio transport is contingent upon accurate clock recovery. Digital audio is a sampled representation of a continuous signal, and the representation conveyed by any given streaming channel is generated against a clock signal (a 'media clock'). A receiver

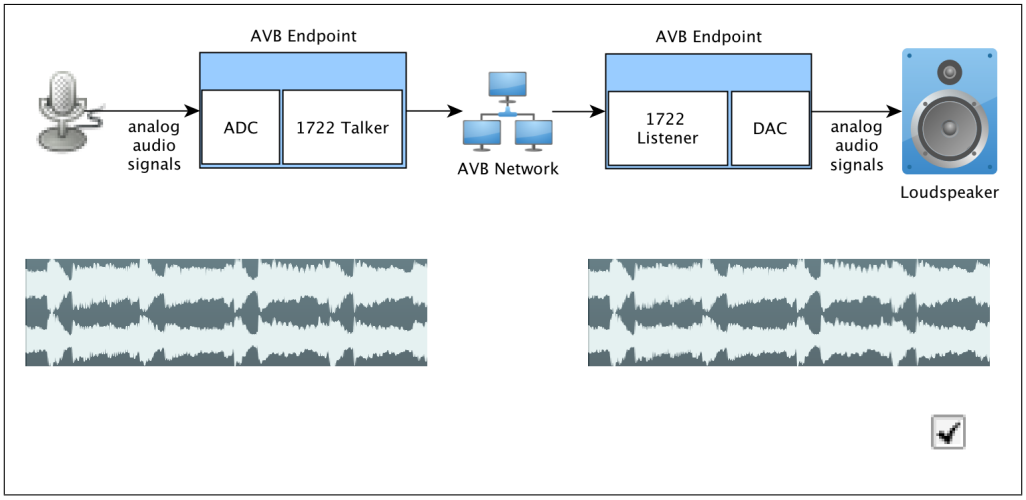


Figure 2.9: Transport of an audio stream over an AVB network, maintaining the original signal's continuity.

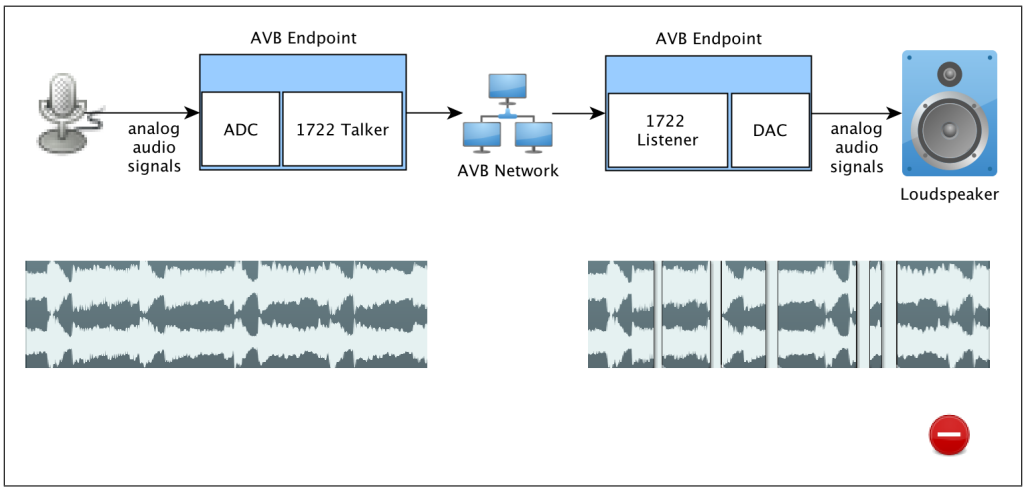


Figure 2.10: Transport of an audio stream over an AVB network, disrupting the original signal's continuity.

must accurately recover that channel's clock signal in order to accurately reproduce the digital audio: inaccuracies (or 'jitter') in clock recovery affect the conversion of digital audio to an audible analogue signal, and can lead to various undesirable audible effects [Watkinson 1994, 8]. This means that the transport network must provide a mechanism for end stations to accurately recover the media clock of streams they are expected to output.

This issue is reasonably easy to achieve over point-to-point connections or in bus networks, where the timing of data frames can be used as a reliable indicator of the speed at which the streaming source is generating the digital audio. In a bridged network such as Ethernet AVB, however, this no longer applies, as indeterminate forwarding and queuing delays within bridges will disrupt frame timing.

5. Prioritization — the privileging of audio data transport over conventional data. Network bridges must conveniently identify and prioritise audio transport traffic over conventional data traffic. A network must distribute both forms of traffic while guaranteeing that conventional data traffic will not compromise audio transport or introduce unpredictable latency.

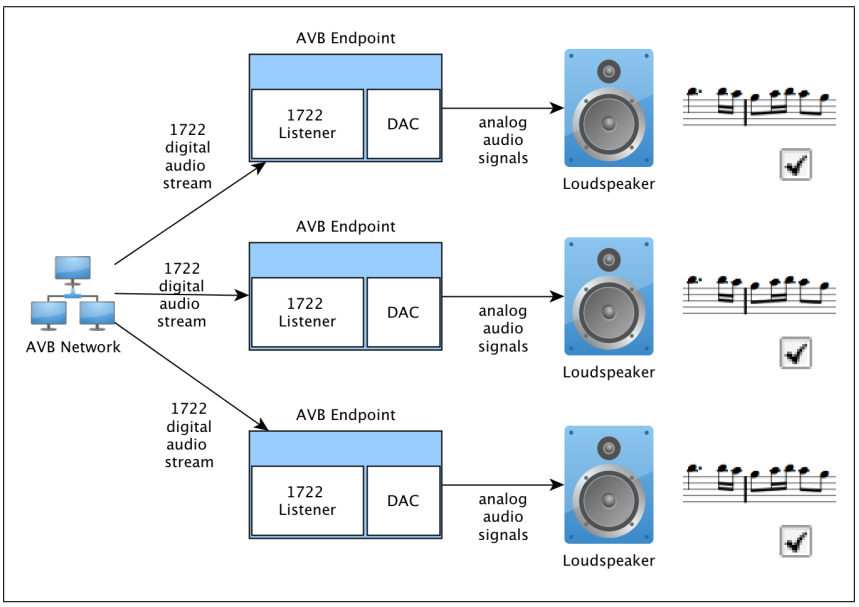


Figure 2.11: Simultaneous presentation of an audio signal by multiple endpoints, delivering the intended relation.

In addition to the quality of service issues, Ethernet AVB introduces the complexity of networked systems into application areas that have hitherto largely relied upon

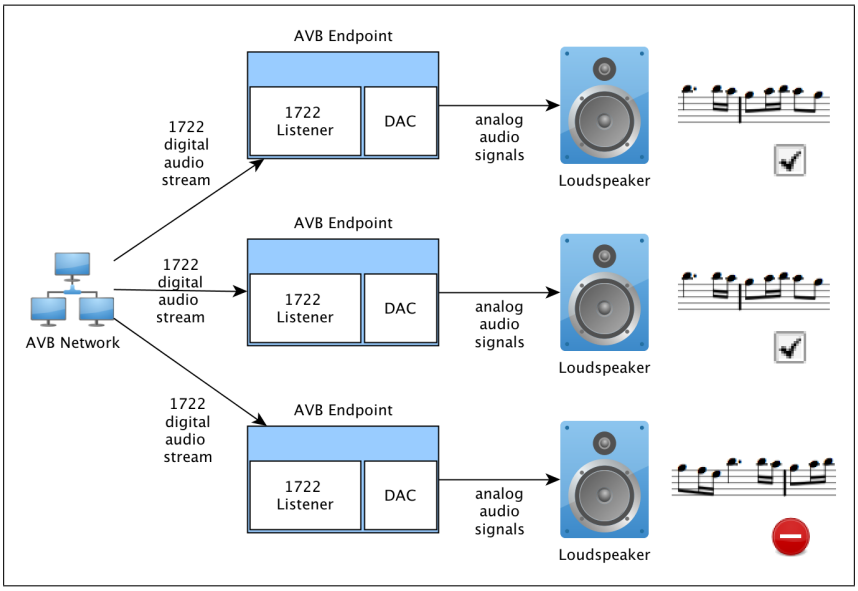


Figure 2.12: Simultaneous presentation of an audio signal by multiple endpoints, with failure to deliver the intended content. The third endstation is out of synchronisation with the first two by one beat.

(conceptually, if not practically) simple point-to-point distribution. As such, there is a clear expectation for Ethernet AVB to provide a foundation for connection management and control software tools to assist with AVB network set up, operation and maintenance.

Ethernet AVB engages the issues described above in the following ways:

1. Prioritization, latency, synchronization and clock recovery — IEEE 1722 defines an audio transport protocol that employs VLAN tagging to clearly identify and segregate AVB data traffic from conventional data traffic, while encapsulating popular audio formats and providing mechanisms for synchronised presentation and media clock recovery.

2. Priorization and latency — IEEE 802.1Qav defines extensions to the specifications of MAC bridges and VLAN-aware bridges that enable prioritised queueing and transmission of AVB data traffic, as well as deterministic reasoning about worst-case forwarding and transport latency.

3. Synchronization and clock recovery — IEEE 802.1AS defines a generalised time protocol for AVB networks that automatically elects a device on the network as a grandmaster time source. This grandmaster then provides a network clock for the synchronisation of control and stream delivery operations. This protocol also

provides syntonisation (timebase correction) network devices, enabling devices across the network to accurately communicate time measurements (including IEEE 1722 presentation times).

4. Prioritization, continuity and latency — IEEE defines a distributed resource reservation protocol that guarantees and reserves network bandwidth for a prospective transmission. If network bandwidth cannot be guaranteed, the transmission does not proceed.

5. Connection management and control software — IEEE 1722.1 defines a standard to enable software network set up and operation.

2.3 The Ethernet AVB component standards

This chapter has described the operation of Ethernet AVB networks at a high level. It is important to discuss the component protocols which enable Ethernet AVB stream delivery to take place with guaranteed quality of service.

The fundamental premise of Ethernet AVB is that conventional Ethernet is a high-quality platform for data networking, providing features that audio distribution applications can make good use of: redundant connections, resiliency, convenient set up and tear down. Ethernet AVB aims to make conventional Ethernet suitable for audio distribution. To do so it provides the following component standards:

1. [IEEE 802.1Qav 2010] defines enhancements to forwarding and queueing logic for bridges and end stations, and transmission classes and algorithms that enable self-configuring traffic shaping. This provides a basis for AVB and non-AVB traffic to share network infrastructure and enforces the most efficient possible use of available bandwidth.

2. [IEEE 802.1AS 2011] defines mechanisms and procedures that provide a self-configuring time domain for the AVB network. This enables several QoS requirements, including transport synchronisation, media clock regeneration, and calibration of link latency.

3. [IEEE 802.1Qat 2010] defines mechanisms and procedures to advertise and satisfy QoS requirements for an audio transport stream. This determines whether there is sufficient bandwidth available between a Talker and a prospective Listener for stream transmission to take place. If there is insufficient available bandwidth then transmission does not take place.

4. [IEEE 1722 2012] defines a standard for transport of audio data across a QoS network as defined by IEEE 802.1Qav, IEEE 802.1Qat and IEEE 802.1AS. IEEE 1722 provides mechanisms for transport synchronisation and media clock recovery, and capably encapsulates popular audio data formats.

5. [IEEE P1722.1/D21 2012] provides a Layer 2 standard for discovery, enumeration, connection management and control of Ethernet AVB-compatible end stations, including a comprehensive scheme for the structured representation of end stations and their interfaces.

The following sections describe each of these standards in turn.

2.3.1 IEEE 802.1Qav: Forwarding and queueing enhancements for time-sensitive streams

IEEE 802.1Qav defines mechanisms for end stations and bridges to classify and prioritise network traffic, enabling the *guaranteed* allocation of bandwidth resources. Alongside IEEE 802.1AS ('generalised Precision Time Protocol'), IEEE 802.1Qav provides the foundation for Ethernet AVB's QoS and transport services.

Annex C of the IEEE 1722 standard [2012, 42] describes the encapsulation of AVB stream and control frames within the IEEE 802.3 frame format[3]. Stream data frames are VLAN-tagged[4], where a 'priority code point' field within the VLAN tag indicates the *stream reservation class* of the frame: an AVB bridge uses this field to process frames with the appropriate prioritisation.

Stream reservation classes

Stream reservation ('SR') classes are defined by IEEE 802.1Qav [2010, 5] to indicate successively higher priorities of stream traffic. (The standard use case is that standard-priority stream traffic could provide ambient music, while higher-priority stream traffic could be reserved for emergency messages. In an emergency, the AVB network would drop delivery of the ambient music in preference to delivering the emergency message.) At the time of writing, two SR classes ('A' and 'B') have been defined. A maximum of seven ('A' – 'G') are proposed.

[3] A 'stream frame' conveys data for audio transport. A 'control frame' encapsulates a Layer 2 control message as defined in IEEE 1722.1

[4] The *VLAN Identifier* field of the VLAN tag indicates to a bridge that the frame is a stream frame; the use of VLAN tagging to identify audio streaming frames does not affect the bridge's handling of non-AVB VLAN-tagged traffic.

Each stream reservation class has certain known attributes — for instance, a traffic class defines the greatest possible size of a traffic frame. This becomes an important part of how IEEE 802.1Qat estimates the accumulated latency on a network route[5].

Frames for conventional data traffic are received and forwarded by IEEE 802.1Qav bridges as non-prioritised traffic.

Transmission queues

The forwarding process defined by IEEE 802.1Qav provides two or more queues for each transmission port on an AVB network device. Each queue stores pending transmit frames of a particular traffic class. At least one queue for each port on the device must support an SR class[6].

It is expected that one queue on each transmission port will be assigned to non-AVB traffic, and that this queue will be treated as a lower priority than those assigned to AVB traffic. Each queue on a transmission port must be treated as a distinct priority from any other queue on that port. Each queue of a transmission port may be assigned a *transmission selection* ('TS') algorithm, of which IEEE 802.1Qav defines two: a 'strict priority algorithm' [2010, 17] and a 'credit-based shaper algorithm' [2010, 55]. Further algorithms may be vendor-defined.

High-level responsibilities

IEEE 802.1Qav satisfies the following quality of service requirements:

1. Guaranteed bandwidth allocations for streaming traffic. SR-class traffic is forwarded to high-priority transmission queues. Each transmission queue can be allocated a deterministic proportion of the port's total bandwidth, in compliance with the maximum tolerances for latency defined for each SR class.

2. Bounding of indeterminate forwarding and queuing latency. Pending frames of SR-class traffic will (almost always[7]) be selected for transmission ahead of pending frames from lower-priority traffic classes.

[5] These attributes may also inform decisions about the PHY layer used in a bridge: for instance, a given 802.11 (wireless) PHY layer may be capable of serving low-speed traffic classes but incapable of serving others. [IEEE 802.1Qat 2010, 101]

[6] A bridge does not have to support *every* SR class — but it must support at least *one* SR class.

[7] The credit-shaping algorithm prevents the port's available transmission bandwidth from being entirely exhausted by SR-class traffic. The result is that if an SR-class queue's credit goes negative with frames still pending, a lower-class queue may be selected to transmit frames until the SR-class queue's credit is once again positive.

3. Convergence. The definition, separation and prioritisation of traffic classes allows non-streaming traffic to be delivered on an AVB network without disrupting stream traffic.

Additionally, the metrics and mechanisms provided by IEEE 802.1Qav are critical to both IEEE 802.1Qat and IEEE 1722.

However, IEEE 802.1Qav also presents substantial challenges, as it mandates substantial changes to the behaviour of the conventional IEEE 802.3 Media Access Control (MAC) component. As shown later in this chapter, the XMOS XS1 architecture provides (among other benefits) a software-based MAC implementation that has enabled the development of a software-defined Ethernet AVB end station.

2.3.2 IEEE 802.1AS: Timing and synchronisation for time-sensitive applications in bridged local area networks

IEEE 802.1AS defines mechanisms to provide timing and synchronisation services across bridged and virtual bridged local area networks, including auto-configuration and re-configuration (for instance, during the addition or removal of stations from the bridged network) of a distributed network timing signal.

IEEE 802.1AS is derived (and greatly simplified) from the IEEE 1588-2008 Precision Timing Protocol, and defines timing and synchronisation services for a number of MAC networks, including MAC networks that do not support the wider Ethernet AVB standard.

Time-aware systems and IEEE 802.1AS domains

The end stations and bridges within an Ethernet AVB network are said to be *time-aware*: in other words, they conform to the behaviours defined by IEEE 802.1AS. End stations and bridges that are not time-aware cannot participate as part of an AVB network: for example, a bridge that is not time-aware cannot forward IEEE 802.1AS traffic between two end stations that are. Accordingly, an AVB network constitutes an IEEE 802.1AS *domain*.

All time-aware systems within an IEEE 802.1AS domain are synchronised to a single network clock signal. The source of the network clock signal is termed the *grandmaster*. Any suitably-equipped time-aware system is a potential grandmaster: for example, an end station or bridge with access to a GPS clock would be capable.

The grandmaster and BMCA elections

The grandmaster of an IEEE 802.1AS domain is elected through the *best master clock algorithm* (BMCA), which has been simplified from that used in IEEE 1588-2008. The elected grandmaster issues a periodic announcement message that asserts its status as the grandmaster. If a timeout period expires where no announcement is issued (indicating that the grandmaster has been removed from the network), a BMCA election is triggered.

During the election, each potential grandmaster issues an announcement that it is the grandmaster. Bridges within the IEEE 802.1AS domain arbitrate the competing announcements: the 'best' announcement received by a bridge is forwarded to all other bridges and end stations. The exception to this rule is if a bridge itself is a potential grandmaster and none of the announcements that it receives are superior to its own: in this case, the bridge will announce that it is the grandmaster.

If a time-aware system is joined to the network, it will eventually receive one of the grandmaster's periodic announcements. If the time-aware system is *not* a potential grandmaster, it will accept the grandmaster's status and adjust its time information accordingly. If the time-aware system *is* a potential grandmaster, it will respond to the grandmaster's status by issuing an announcement that it is the grandmaster. This contestation will be handled by the nearest bridge, which will determine whether the new announcement describes a superior clock source to the current grandmaster, and will forward the best potential grandmaster's announcement accordingly.

This process also constructs a spanning tree of the IEEE 802.1AS domain that is used thereafter to propagate time information from the grandmaster.

Time synchronisation and link calibration

The elected grandmaster sends the current time, as derived from its time source, to all of its directly attached time-aware systems. Each time-aware system maintains the ability to convert between grandmaster time and its own local clock.

IEEE 802.1AS proposes auto-calibration mechanisms that take account of:

- link latency between directly connected time-aware systems, which must be accounted for in processing grandmaster time updates;
- differences in clock frequency (syntonisation error) between directly connected time-aware systems, which must also be accounted for.

A detailed account of these mechanisms is beyond the scope of this study: see [IEEE 802.1AS 2011, 21-23] and [Foulkes 2011, 128-136].

High-level responsibilities

IEEE 802.1AS satisfies the following low-level requirements:

1. Election of the best available clock as the source of AVB network time;
2. Calibration of link latency and other delays affecting propagation of AVB network time;
3. Logical syntonisation of AVB device clocks.

These low-level requirements enable devices within the AVB network to communicate meaningful time measurements, supporting the following transport priorities:

1. Accurate media clock recovery
2. Precompensation for known transport latency in the network
3. Elimination of jitter accumulation throughout the network
4. Synchronised presentation of audio traffic at multiple destinations

Within the context of Ethernet AVB, IEEE 802.1AS provides services that are essential to the effective operation of IEEE 1722 audio transport services and the control and instrumentation services provided by IEEE 1722.1.

2.3.3 IEEE 802.1Qat: Stream reservation protocol (SRP)

IEEE 802.1Qat defines mechanisms that enable Ethernet AVB end stations and bridges to negotiate and reserve network bandwidth to meet the quality of service requirements of audio streams. It provides a lightweight framework to set up and identify network routes for specific transport streams, and describes behaviour to handle reservation failures. An outline example of this process was provided in Subsection 2.2.2.

Stream resource reservation

The process of stream resource reservation in Ethernet AVB networks is defined by the Stream Reservation Protocol (SRP). SRP employs three signalling protocols, defined as Multiple Registration Protocol (MRP) applications:

1. Multiple VLAN Registration Protocol (MVRP). SRP employs MVRP to assign a unique VLAN ID to each potential stream, and to register bridge ports and end stations that are interested in the stream as members of that VLAN.

This enables the use of VLAN tagging in the SRP process and in IEEE 1722 transport packets. SRP's use of MVRP conserves network bandwidth by prohibiting distribution of stream traffic over bridge transmission ports that are *not* registered to the corresponding VLAN ID.

MVRP will not be discussed further in this document.

2. Multiple MAC Registration Protocol (MMRP), which may optionally be used to control the propagation of some MSRP operations over the bridged LAN [IEEE 802.1Qat 2010, 72].

   MMRP will not be discussed further in this document.

3. Multiple Stream Registration Protocol (MSRP), which advertises the resource requirements of a prospective Talker's stream across the AVB network and registers:

   a) which bridged routes across the network, if any, can support the stream's advertised requirements

   b) which prospective Listeners, if any, are interested in receiving the stream

   c) whether bridged routes exist between the prospective Talker and the interested Listeners that can support the stream's requirements

   This registration process incorporates rudimentary error reporting for when a stream's requirements cannot be satisfied.

   MSRP also discovers the boundary of the AVB network by a defined process for bridges and end stations to exchange details of what SR classes they can support.

Each Ethernet AVB end station or bridge implements *participants* for each of these MRP applications. There is one participant per MRP application per port, and each participant performs signalling and registration tasks as defined by its application and by the behaviour defined in MRP. Within a bridge, participants can communicate registrations through a Multiple Attribute Propagation (MAP) component.

The multiple stream reservation protocol (MSRP)

A detailed account of the operation of MSRP is beyond the scope of this document (see [Foulkes 2011, 252-278]), but the following overview illustrates the MSRP actions that enable the connection of an IEEE 1722 stream.

Initiating stream reservation. A Talker may announce that it has a stream to deliver to the network by declaring a *Talker Advertise* (TA) attribute. This declaration specifies the quality of service requirements that must be guaranteed to convey the Talker's stream.

The Talker will periodically declare this attribute throughout the lifespan of the stream. The Talker may withdraw the TA declaration at any time, which indicates that the stream identified is no longer available.

When the Talker makes the TA declaration, it will be registered by the MSRP participant on the bridge port that the Talker is directly attached to. The TA registration will be forwarded to the bridge's MAP component.

The bridge's MAP component will review the advertised resource requirements against the bandwidth available on each of its transmitting ports[8]. If a transmission port is capable of meeting the stream's requirements, the MAP component will declare a TA attribute on the port. However, if a transmission port is *not* capable of meeting the stream's requirements, the MAP component will instead declare a *Talker Failed* (TF) attribute, indicating that insufficient resources exist for the stream to be delivered to any stations downstream of the transmission port.

An MSRP participant on a bridge that registers a TF attribute (typically from another bridge) is being notified that there are insufficient resources upstream for the stream to reach the bridge *at all*. Accordingly, the MSRP participant will propagate the TF attribute to all of its transmission ports.

In Figure 2.13, Endpoint A is the prospective Talker. Endpoint A declares ('D: TA') a Talker Advertise attribute which is registered ('R: TA') and/or declared in turn ('D: TA') by Bridge 1, Bridge 2, Endpoints C and D, and Bridge 3.

The TX port on Bridge 1 connected to Endpoint B is unable to support the stream requirements of Endpoint A's Talker Advertise declaration, and Bridge 1's MAP component declares a Talker Failed attribute ('D: TF') on that port's MSRP participant, which is duly registered by Endpoint B ('R: TF').

An MSRP participant on a port on Bridge 2 registers Endpoint A's Talker Advertise attribute and, through its MAP component, declares a Talker Advertise ('D: TA') on its links to Endpoint C and Endpoint D, as well as its link to Bridge 3.

Finally, the MAP component on Bridge 3 has no live ports (see Figure 2.2) to propagate the Talker Advertise attribute onto.

[8] A given transmitting port may already have reservations in place to enable the delivery of other streams, which might leave the port incapable of supporting the new stream's requested resources.

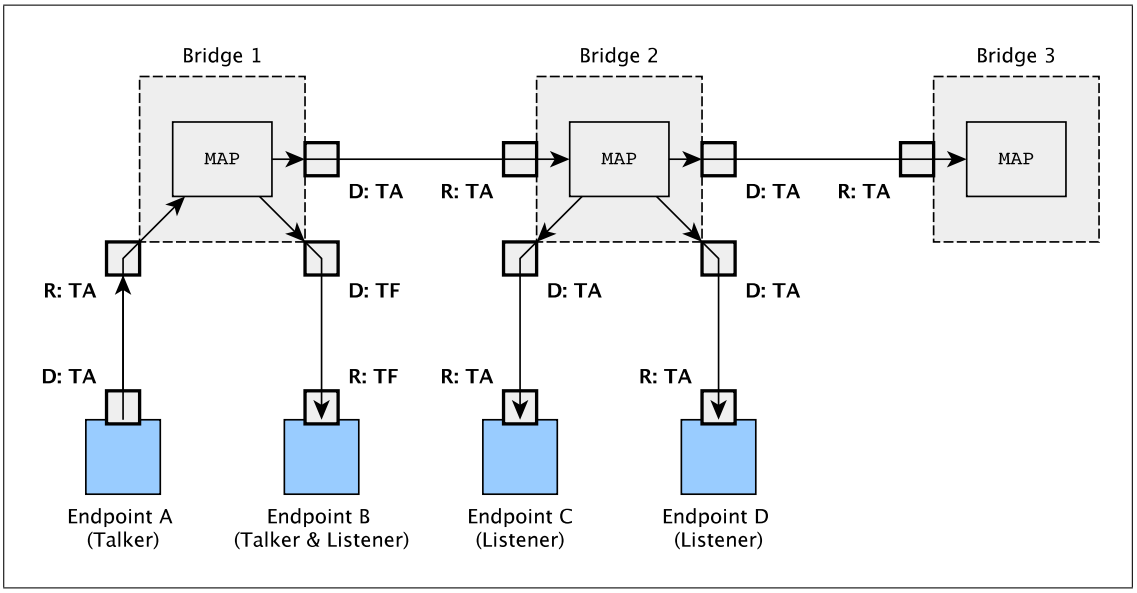


Figure 2.13: Initiating stream reservation in 'avb domain 2' from Figure 2.2.

Listener responses. If a Listener registers a TA attribute, it knows that sufficient resources can be guaranteed on a route between the Listener and the Talker for stream delivery to occur. Registering a TF attribute indicates that there are no routes with sufficient resources for stream delivery between the Listener and the Talker.

Whether or not the Listener is interested in the stream is typically determined by a higher-level connection management protocol or a local software application. If the Listener is *not* interested in receiving the stream, it will not declare any attribute.

If the Listener registered a *Talker Advertise* attribute and it is interested in receiving the stream it describes, it first declares an MVRP attribute, citing the VLAN identifier specified in the registered TA attribute. The bridge directly attached to the Listener will then register the connection port as a member of the stream's assigned VLAN. The Listener then declares an MSRP *Listener Ready* (LR) attribute, which indicates that a Listener is interested in receiving the stream and that sufficient resources exist to allow the Listener to do so.

If the Listener received a *Talker Failed* registration and it is interested in receiving the stream, it issues an MSRP *Listener Asking Failed* (LAF) declaration, which indicates that the Listener is interested in receiving the stream but that there are insufficient resources for it to do so.

When Listener attributes are registered by a bridge, the bridge's MAP component will *merge* them into a single Listener attribute [IEEE 802.1Qat 2010, 74]. Alongside

the two Listener attributes already described, this introduces the possibility of a third type of Listener attribute: *Listener Ready Failed* (LRF), which indicates that at least two Listeners are requesting the advertised stream, although at least one of them is incapable of receiving it so. In other words, the declaration of an LRF attribute indicates partial success in stream reservation, with the expectation that the Talker may want to communicate this status over a higher-level connection management protocol.

Figure 2.14 illustrates this process. Here, Endpoint B and D are interested in receiving Endpoint A's stream; Endpoint C is uninterested and makes no declaration. Endpoint D is interested in receiving Endpoint A's stream and declares a Listener Ready ('D: LR') attribute. The MAP component on Bridge 2 registers this attribute and declares it on the port that connects with Bridge 1.

Endpoint B previously registered a Talker Failed attribute (see Figure 2.13) and declares a Listener Asking Failed ('D: LAF') attribute, which is registered by an MSRP participant on Bridge 1. This registration is propagated to Bridge 1's MAP component, which has also received a registration of the Listener Ready attribute declared by Bridge 2. Bridge 1's MAP component then merges these attributes, consequently declaring a Listener Ready Failed ('D: LRF') attribute to Endpoint A.

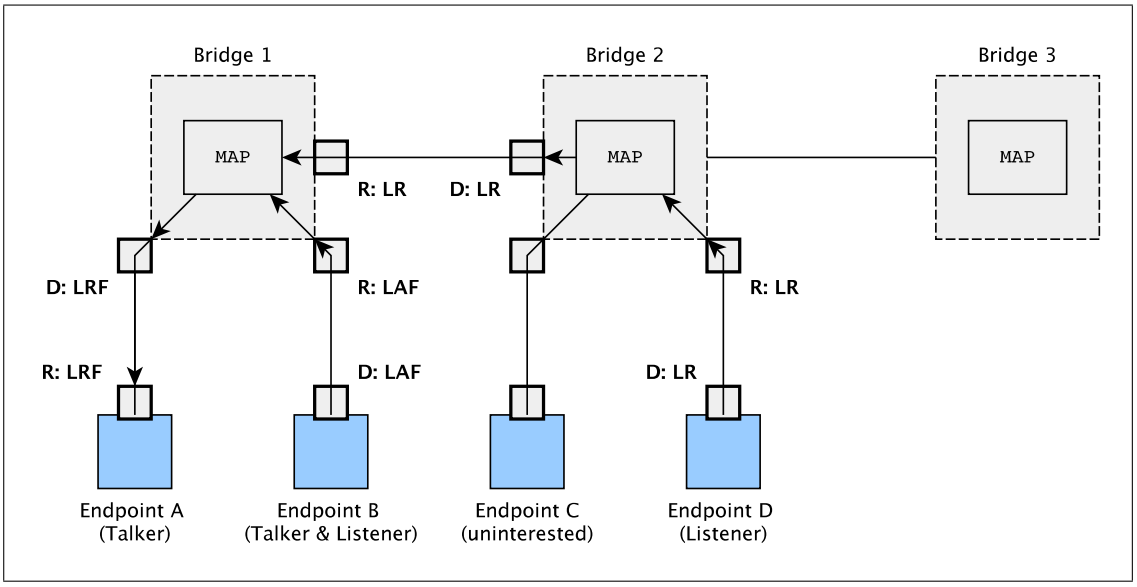


Figure 2.14: Listener responses to stream reservation as initiated in Figure 2.13.

Completing stream reservation. The Talker will expect to register Listener Declarations within a timeout period. The Listener Declarations that have been registered

by the Talker will then be merged along the lines described above, and the final merged Declaration will determine what happens next:

1. If it is a *Listener Ready* declaration, the Talker will begin transmitting the stream to the associated VLAN immediately.
2. If it is a *Listener Ready Failed* declaration, the Talker will begin transmitting the stream to the associated VLAN immediately, but should be aware that not all Listeners will receive it.
3. If it is a *Listener Asking Failed* attribute, there are no Listeners capable of receiving the stream and the Talker will not transmit.

Stream transmission. Once (if) the Talker begins to transmit the IEEE 1722 audio stream, bridges that forward the stream will associate its ID with the preceding MSRP declarations, and allocate internal resources accordingly [IEEE 802.1Qat 2010, 49].

High-level responsibilities

IEEE 802.1Qat provides logical standards and protocols that:

1. allow Talkers to declare quality of service requirements for a stream.
2. enable bridges to assess capability to satisfy declared requirements.
3. guarantee that streams can be transported within stated QoS tolerances.
4. reserve transmission capacity on network bridges such that transport latency becomes effectively deterministic.
5. prohibit transport of a stream if resources cannot be guaranteed.
6. support all of the above for multiple concurrent streams.
7. integrate effectively with existing VLAN technology to conserve bandwidth and support secure audio transport.

IEEE 802.1Qat is particularly effective at abstracting these requirements from the end-user experience of an AVB network operator, while making it possible for a higher-level control / connection management protocol to provide useful error reporting of the stream reservation process.

2.3.4 IEEE 1722: Layer 2 transport protocol for time-sensitive applications in bridged local area networks

IEEE 1722 defines high-level transport mechanisms for Ethernet AVB: the tagging of stream traffic to enable SR class and stream identification, synchronised stream presentation and media clock recovery, and the encapsulation of popular audio encoding schemes.

IEEE 1722 is intended to provide a transport layer for time-sensitive audio/video traffic that can operate over any IEEE 802 network, including IEEE 802.11 wireless networks. It specifies:

> the protocol, data encapsulations, and presentation time procedures used to ensure interoperability between audio- and video-based end stations that use standard networking services provided by all IEEE 802 networks meeting quality-of-service requirements for time-sensitive applications [IEEE 1722 2012, 1].

As such, IEEE 1722 is dependent on the standards already reviewed in this chapter.

IEEE 1722 defines a generalised message format, the Audio/Video Transport Protocol Delivery Unit (AVTPDU), which is used for transmission of control and stream data packets (Figure 2.15). Control data packets convey traffic for the specialised protocols defined by IEEE 1722.1 (see Subsection 2.3.5).

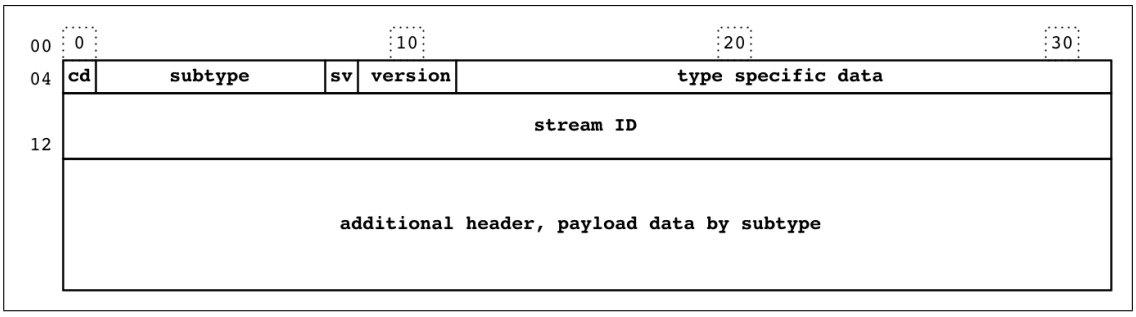


Figure 2.15: The IEEE 1722 Audio/Video Transport Delivery Unit [IEEE 1722 2012, 8].

Here, the `cd` field indicates whether an AVTPDU packet carries control or stream data; the `subtype` field indicates the subtype of the packet's payload data - each subtype defines the composition of the `type specific data` field and the payload (from byte 12 onwards).

IEEE 1722 streams are uniquely identified by a 64-bit stream identifier. Stream data packets and control data packets concerning a single stream will both identify the stream

within the `stream ID` field. Control data packets concerning multiple streams will set the `stream ID` field to zero and the `sv` ('stream ID valid') field to zero ('invalid').

Transport protocol

The IEEE 1722 Layer 2 transport protocol provides mechanisms for the time-sensitive delivery of audio and video streams between Ethernet AVB-compliant Talker and Listener end stations.

Devices that intend to use IEEE 1722 for delivery of stream data must support IEEE 802.1Qav, IEEE 802.1AS and IEEE 802.1Qat. The uses that IEEE 1722 makes of these supporting standards has been summarised in Subsections 2.3.1, 2.3.2 and 2.3.3 as follows:

- IEEE 1722 employs IEEE 802.1AS to supply a robust network time, necessary to provide synchronised and syntonised presentation of transported audio.
- IEEE 1722 employs IEEE 802.1Qav and IEEE 802.1Qat to provide a reasonable level of control and monitoring of network bandwidth, and consequently transport latency.

IEEE 1722 mandates minor revisions to the IEEE 802.3 Ethernet standard. In particular, IEEE 1722 makes use of the IEEE 802.1Q header format to tag all stream data frames; use of this format for IEEE 1722 control data frames is optional. Figure 2.16 shows the encapsulation of an IEEE 1722 stream or control data frame within an IEEE 802.3 frame.

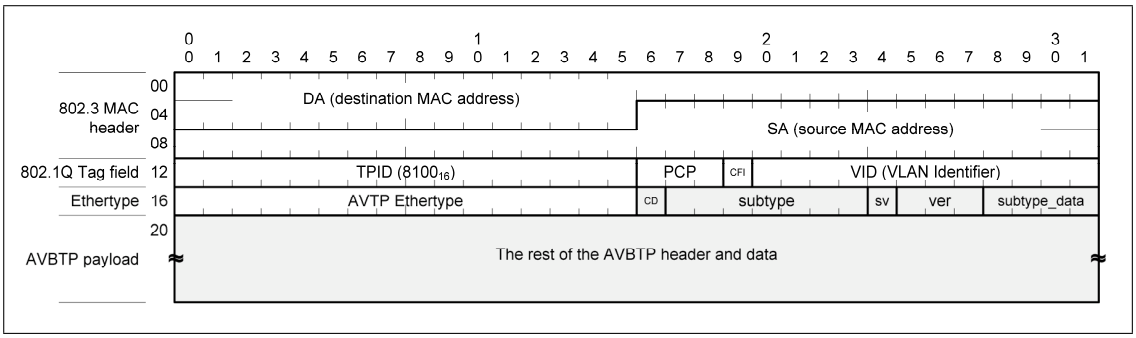


Figure 2.16: Encapsulation of an IEEE 1722 Audio/Video Transport Delivery Unit within an IEEE 802.3 frame [IEEE 1722 2012, 43].

2.3.5 IEEE 1722.1 (AVDECC): Draft standard for device discovery, connection management and control protocol for IEEE 1722 based devices

IEEE 1722.1 (AVDECC[9]) defines a standard for representation and messaging between Ethernet AVB end stations. A sophisticated entity model allows the representation of devices in terms of *operating configurations* and, within those configurations, *descriptors* that represent components of each configuration. Three specialised communication protocols coordinate the independent processes of network discovery, connection management for IEEE 1722 transport streams, and device enumeration and control.

An Ethernet AVB endpoint does not necessarily have to implement an IEEE 1722.1 service, although it is expected that it implements some form of control and monitoring protocol. The XMOS Ethernet AVB reference design (see Section 2.4) has provided several software projects that build device firmware: some of these projects implement 1722.1 representation and control of an AVB device; other projects have implemented proprietary control protocols.

Device representation through the 1722.1 Entity Model

The IEEE 1722.1 Entity Model defines the representation of a device's features and logical structure. Compliant devices are to be modelled as a hierarchy of media-related objects (interfaces, clock sources, etc.) which can be composed into sets of operating modes:

> An operating mode of an Entity describes limitations when certain settings are applied. For example, an audio Entity which supports 10 channels at 48kHz and 96kHz sample rates but only 6 channels at 192kHz sample rate would have two configurations, one describing the Entity at 48kHz and 96kHz and a second describing the Entity at 192kHz. [IEEE P1722.1/D21 2012, 41]

A configuration contains functional objects. These may include basic input/output objects (ports, interfaces, jacks) and processing or control objects (including various kinds of mixer, selector, signal multiplexer/demultiplexer and general-purpose control objects). Objects are encoded as descriptors. Each instance of a descriptor of a given type within a device configuration is addressed by a 16-bit index. The IEEE 1722.1 Entity Model specification defines the valid objects and the structure of each corresponding descriptor [2012, 41-262].

[9] Audio/Video Discovery, Enumeration, Control and Connection Management

Figure 2.17 shows an IEEE 1722.1 Entity Model representation of a simple Ethernet AVB endpoint with bidirectional audio capability, as illustrated previously in Figure 2.5.

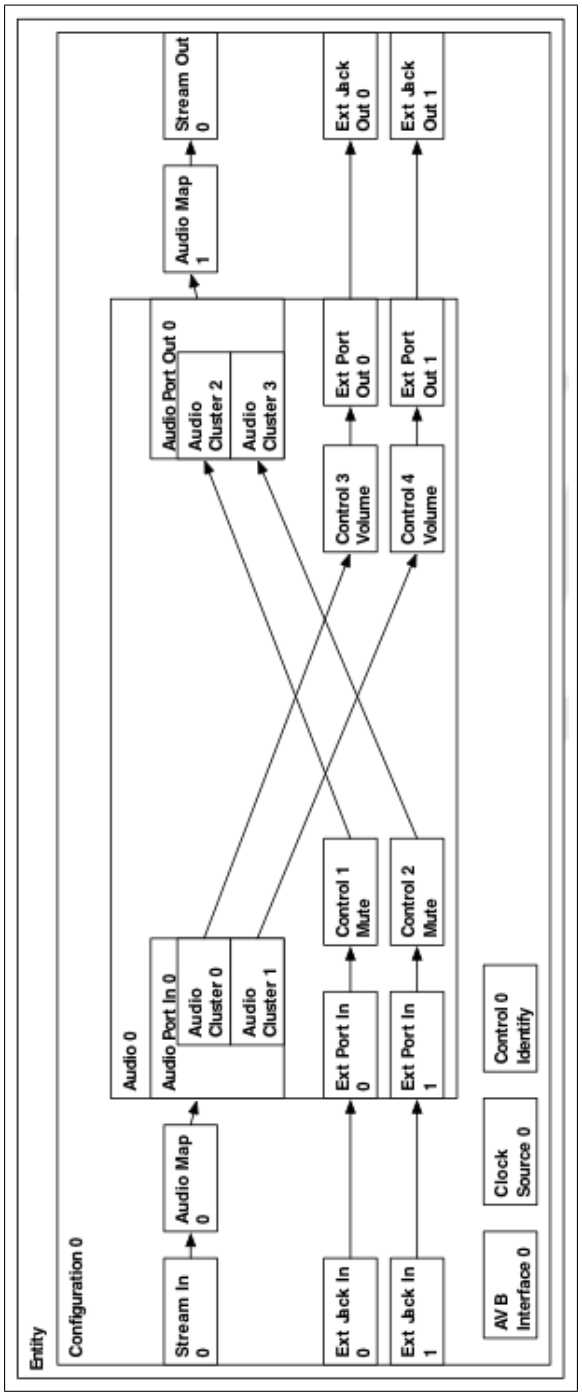


Figure 2.17: IEEE 1722.1 Entity Model of a simple Ethernet AVB endpoint [IEEE P1722.1/D21 2012, 42].

Network discovery

IEEE 1722.1 defines the AVDECC discovery protocol (ADP) to allow 1722.1 Entities to interact. ADP employs the control form of IEEE 1722 AVTPDUs as a messaging format. Three message types are defined:

1. Entity Available: 'The Entity sending this message has arrived on the network'.
2. Entity Departing: 'The Entity sending this message is about to leave the network'.
3. Entity Discover: 'The Entity sending this message would like any Entity that is addressed by it and receives it to respond with its Entity Available message'.

These tightly-constrained message formats allow the efficient representation of a wide range of device capabilities; ADP-compliant devices can report many of the capabilities of their Entity Model through a brief sequence of bitfields.

ADP is the most straightforward protocol defined by IEEE 1722.1, providing three basic messages (shown below). Each AVDECC device must implement an Entity Available and Entity Departing message; AVDECC devices implementing a Controller role must also implement the Entity Discover message.

1. Entity Discover: an AVDECC Entity (typically an AVDECC Controller) requests other AVDECC Entities to issue their Entity Available messages.
2. Entity Available: an AVDECC Entity reports its presence on the network and its basic capabilities (e.g., number of Talker sources, number of Listener sources, whether it implements a Controller role [IEEE P1722.1/D21 2012, 29-31]).
3. Entity Departing: an AVDECC Entity reports that it will imminently leave the network.

Fig. 2.18 and 2.19 illustrate the ADP network discovery process. Fig. 2.20 and 2.21 show captures of an 'Entity Discover' request and an 'Entity Available' response.

Connection management for IEEE 1722 streams

IEEE 1722.1 defines the AVDECC connection management protocol (ACMP) to allow 1722.1 Entities to perform connection management. As with other components of IEEE 1722.1, the control form of IEEE 1722 AVTPDU is employed. A common message format is used for initiating command requests and responses (Fig. 2.22). A wide variety of error conditions are efficiently represented.

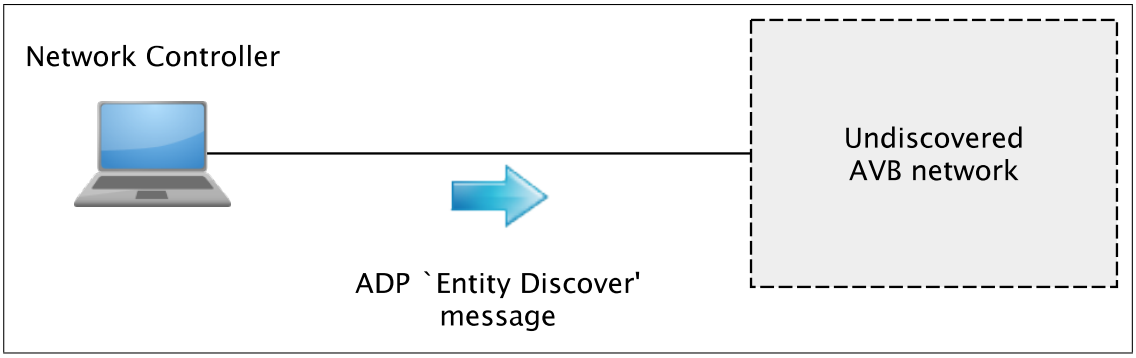


Figure 2.18: A controller performs network discovery by issuing an IEEE 1722.1 ADP 'Entity Discover' request.

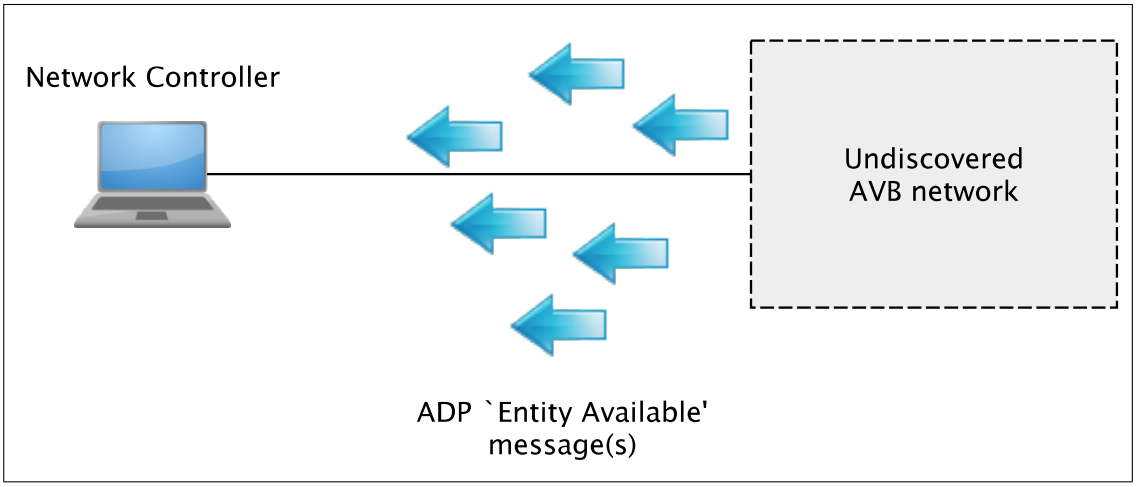


Figure 2.19: Devices respond to the controller by sending IEEE 1722.1 ADP 'Entity Available' messages in response to the 'Entity Discover' request.

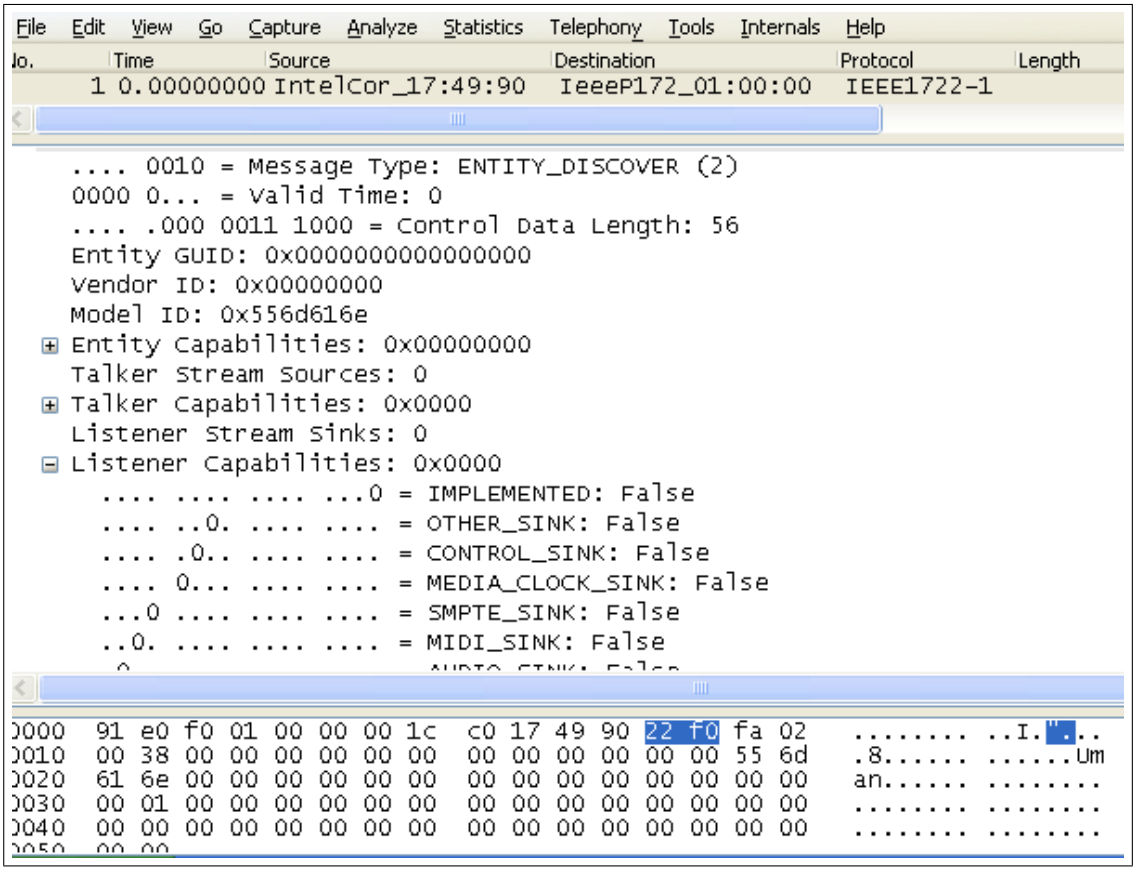


Figure 2.20: WireShark capture of an IEEE 1722.1 ADP 'Entity Discover' message as produced by the Universal Media Access Networks 'UNOS Vision' desktop application.

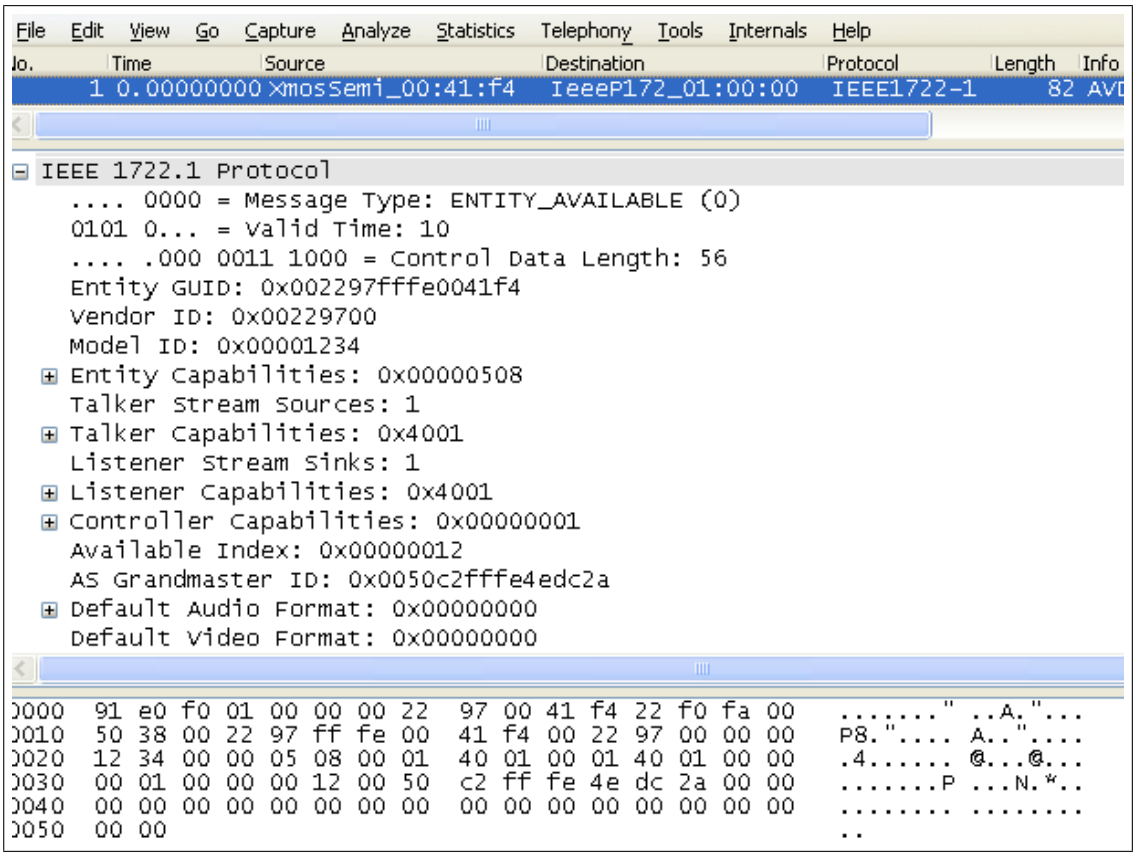


Figure 2.21: WireShark capture of an IEEE 1722.1 ADP 'Entity Available' message as produced by the XMOS Ethernet AVB reference design IEEE 1722.1 implementation.

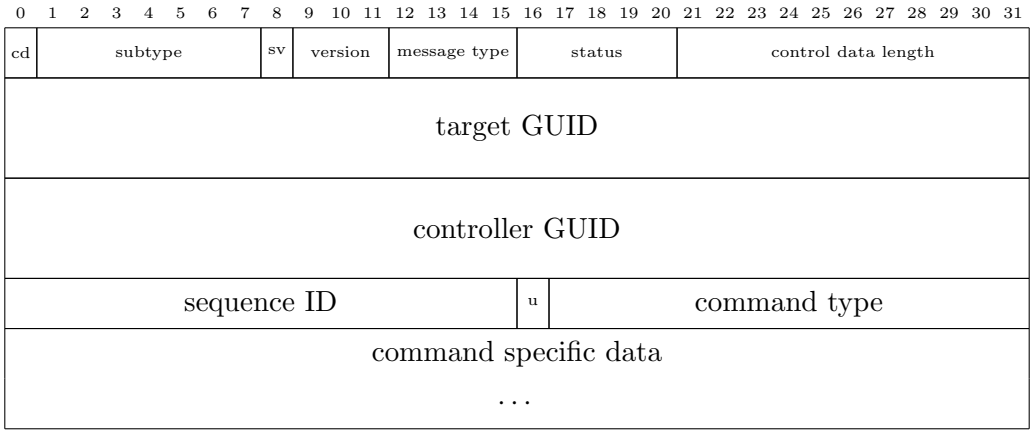


Figure 2.22: The common AVDECC command header format [IEEE 1722 2012, 10] [IEEE P1722.1/D21 2012, 282-5]

Fig. 2.23 shows a basic ACMP sequence to establish a connection between a Talker and a Listener:

1. The sequence is initiated by a Controller, which sends the `CONNECT RX` command to the Listener, nominating the Talker;
2. The Listener reacts by logging the Talker nomination and sending the `CONNECT TX` command to the Talker;
3. The Talker reacts by initiating the SRP process:
   a) The Listener registers a 'Talker Advertise' that matches the logged Talker nomination, and declares 'Listener Ready';
   b) The Talker registers a 'Listener Ready' and begins streaming;
4. The Talker returns a 'success' status to the Listener;
5. The Listener returns a 'success' status to the Controller;

If the SRP process fails (e.g., because of insufficient bandwidth), the Talker returns a 'failed' status to the Listener, where the status message provides a rudimentary diagnostic, and the Listener forwards this status to the Controller.

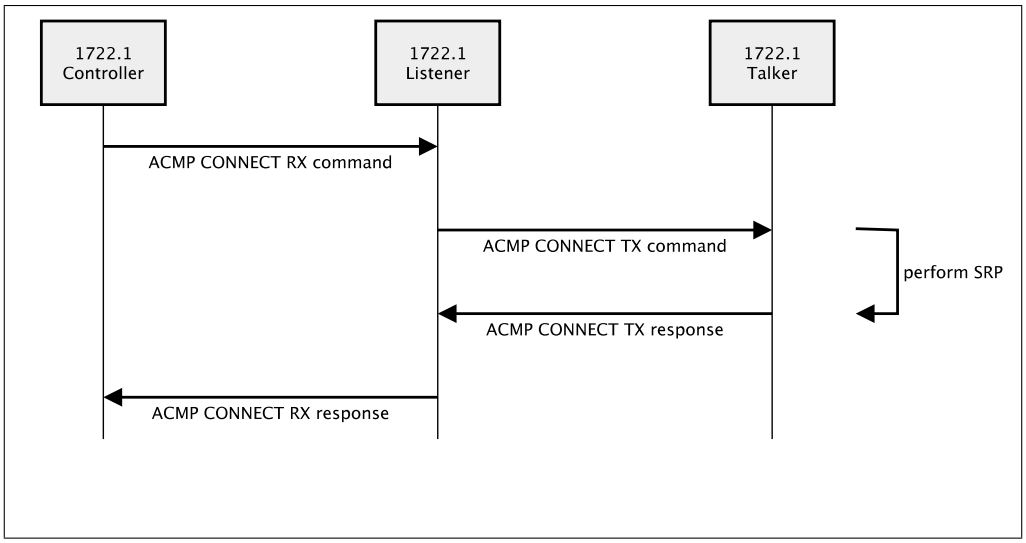


Figure 2.23: IEEE 1722.1 Connection management between a Talker and a Listener.

Device enumeration and control

IEEE 1722.1 defines the AVDECC enumeration and control protocol (AECP) to enable 1722.1 Controllers to enumerate 1722.1 Entities. It employs the control form of the IEEE 1722 AVTPDU, with a number of sub-formats defined in accordance with the requirements of various applications.

A 1722.1 Controller addresses AECP messages to known 1722.1 Entities on a unicast basis. A 1722.1 Entity in receipt of a Controller's request is expected to respond with the requested details within 250ms [IEEE P1722.1/D21 2012, 285].

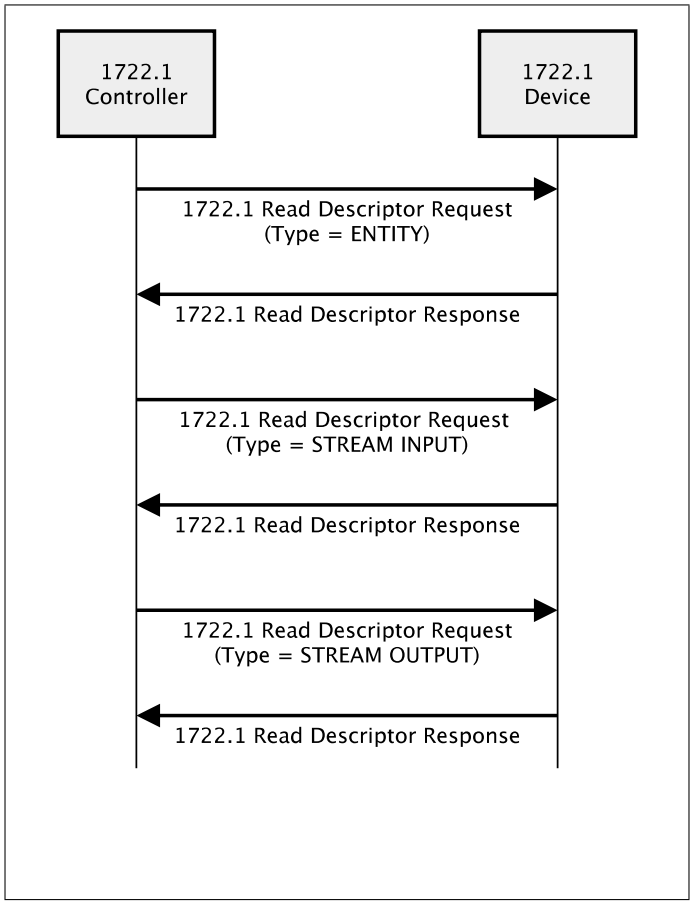


Figure 2.24: A 1722.1 Controller enumerates an XMOS/Attero Tech Ethernet AVB end station.

Fig. 2.24 shows a basic AECP enumeration sequence. The 'READ DESCRIPTOR' requests are addressed to successive descriptors on the device under enumeration: 'ENTITY', 'STREAM IN', 'STREAM OUT', each of which can be seen in Fig. 2.17. Fig. 2.25 shows a specimen response to the first request, which conforms to the outline given in [IEEE P1722.1/D21 2012, 45-6]. From this first response, the controller can learn how

many configurations the device supports, the number of stream inputs and outputs it has, what features each stream input and output support, and device metadata: its manufacturer, its name, firmware version, and serial number. It may then follow up with 'READ DESCRIPTOR' requests targeting the 'STREAM IN' and 'STREAM OUT' descriptors, as many times as necessary.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type == ENTITY	Index == 0
Entity GUID == 00 22 97 FF FE 00 41 C2	
Vendor ID == 00 22 97 00	
Entity Model ID == 00 00 12 34	
Entity Capabilities == 1722.1 AEM, 802.1AS, Class A stream transport	
Talker Stream Sources == 00 01	Talker Capabilities == AUDIO SRC
Listener Stream Sinks == 00 01	Listener Capabilities == AUDIO SINK
Controller Capabilities == CONTROLLER	
Available Index == 00 00 00 00	
Association ID == 00 00 00 00	
Entity Name == "XMOS AVB Endpoint" ...	
Vendor Name String == 00 00	Model Name String == 00 01
Firmware Version == "5.2.0beta2" ...	
Group Name == "XMOS AVB Group" ...	
Serial Number == 0123456789 ...	
Configurations Count == 1	Current Configuration == 0

Figure 2.25: Specimen response to the IEEE 1722.1 'READ DESCRIPTOR [ENTITY]' request shown in Fig. 2.24.

2.3.6 Ethernet AVB components: summary

This section has shown the interactions and dependencies of Ethernet AVB components to meet the rigorous quality of service requirements that apply to audio distribution applications. The following section introduces an implementation of this technology, which has formed a practical basis for the research project.

2.4 The XMOS Ethernet AVB reference design

The XMOS XS1 architecture defines a general-purpose multicore microcontroller that combines hardware support for concurrency, an event-driven approach focused on providing bounded execution of indeterminate processes, programmable input-output (I/O), and the ability to execute languages like C. It is described in detail in Chapter 4.

The motivation of the architecture has been described as 'to make it practical to use software to perform many functions which would normally be done by hardware' [May 2009a, 1]. In the context of Ethernet AVB, that statement specifically applies to the Ethernet MAC component, which is implemented wholly in software on the XMOS XS1 microcontroller: the only external components required are an 8P8C header and a standard Ethernet transceiver [XMOS 2011d, 3]. This has allowed XMOS to revise the operation of the MAC component in line with the demands of IEEE 802.1Qav.

Accordingly, XMOS has developed and released an Ethernet AVB 'reference design' [XMOS 2012a] that implements the major features of an Ethernet AVB streaming audio end station. The reference design has been released with a royalty-free license and was used in this project to provide a development baseline for research and development of the AES-64 implementation.

2.4.1 Hardware

Several development boards are available from XMOS that can execute the reference design software to run as Ethernet AVB end stations: these include the XMOS/Attero Tech 'low-cost AVB endpoint', which is built around an XS1-L2 processor[10] [Attero Tech 2010]. Fig. 2.26 shows two of these end stations connected in a simple network configuration.

Developers may also design and implement their own hardware based around one or more XS1 processors. In the latter case, the hardware design must provide a 100Mbit

[10]Since the start of this research, the XS1-L2 microcontroller has subsequently been rebranded as the XS1-L16A-128: for the purposes of this thesis, the older nomenclature is used.

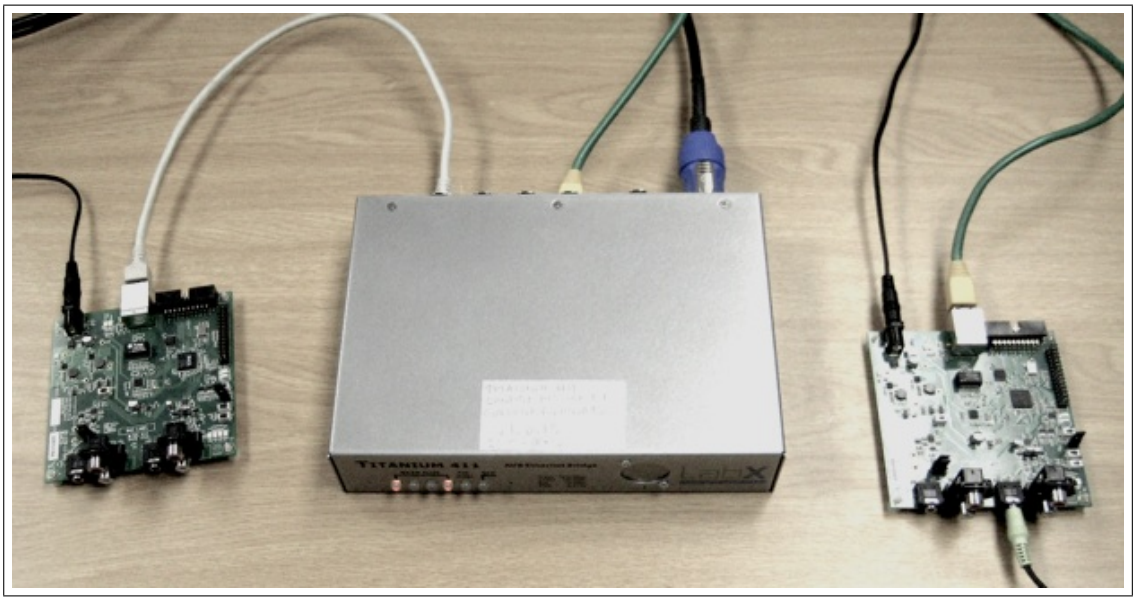


Figure 2.26: A simple AVB network, consisting of two XMOS/Attero Tech end stations and a LabX Titanium 411 AVB-capable bridge.

Ethernet PHY layer, an audio codec with input-output interfaces, and a clock source for the audio device [XMOS 2011, 6].

In this project, the primary hardware platform was the DSP4YOU AVBStreamer module, which employs an XS1-G4 processor [DSP4YOU 2012]. This device was interfaced with the DSP4YOU E-mic module, which provides a Cirrus Logic CS4272 audio codec and two-channel audio input-output. Fig. 2.27 shows a single DSP4YOU end station.

2.4.2 Software

The XMOS Ethernet AVB reference design has provided a number of applications. The most economical application can be executed on a XS1-L2 dual-tile processor, providing:

1. an IEEE 802.1Qav-compliant Ethernet stack
2. a TCP/IP stack,
3. an IEEE 802.1AS server,
4. a media clock server,
5. an I2S digital audio server,

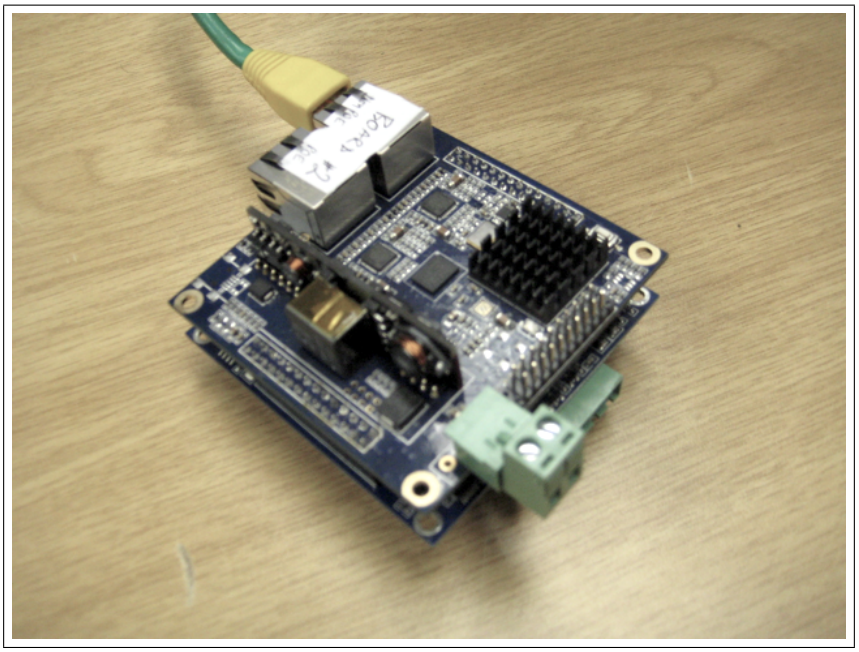


Figure 2.27: An AVB end station consisting of a DSP4YOU AVBStreamer module (upper board) and a DSP4YOU E-mic module (lower board). The XMOS XS1-G4 processor is beneath the heatsink on the rightmost corner of the upper board.

6. an IEEE 1722 Talker, interfaced with the input channels of the external audio device,
7. an IEEE 1722 Listener, interfaced with the output channels of the external audio device, and
8. a high level control application, `demo()`, including a proprietary remote control service providing diagnostics and connection management.

Figure 2.28 shows the mapping of these tasks on the two tiles of the XS1-L2 processor on the XMOS/Attero Tech end station. Listing 2.1 and Listing 2.2 show the mapping of these tasks to hardware threads in the application source code. These listings are in the XC language (Section 4.5) and some elements may be unfamiliar:

1. the `chan` statement declares a channel for communication between concurrent tasks (arrays of channels may also be declared);
2. the `par` statement defines a set of tasks that are to execute in parallel;
3. the `on` statement defines which tile (in the case of the XS1-L2 package, this is either `stdcore[0]` or `stdcore[1]`) a task should be mapped to: what follows can either be a

single task that executes indefinitely, as with `uip_server()`, or a sequence of tasks, as with the brace-enclosed block of tasks that initialise and execute the Ethernet stack [`l. 31 - 37`];

4. the call to `ethernet_getmac_otp()` includes the expression `(mac_address, char[])`. This is an XC 'reinterpretation' [XMOS 2011a, 14], analogous to a C 'cast' but capable of dealing with arrays, in which the `mac_address` array of `int` is interpreted (without conversion) as an array of `char`.

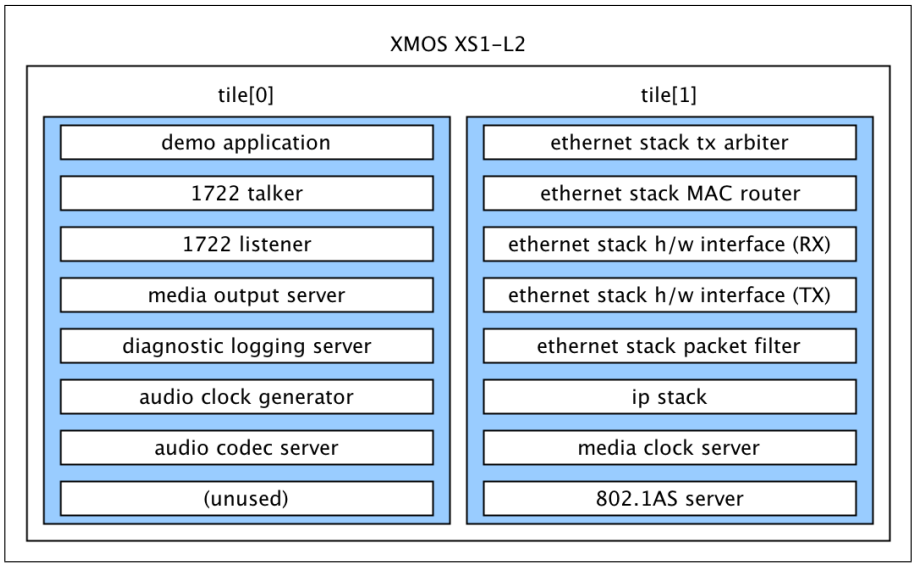


Figure 2.28: Mapping of AVB end station software tasks on the XMOS 'low-cost AVB endpoint'.

Listing 2.1: Execution threads in the XMOS AVB application (part 1)

```
void demo(chanend tcp_svr, chanend c_rx, chanend c_tx,
  chanend c_gpio_ctl);

// global declarations omitted
int main(void) {
  // chanend declarations for inter-thread communication
  // ethernet server chanends
  chan tx_link[5];
  chan rx_link[4];
  chan connect_status;

  // 802.1AS server chanends
  chan ptp_link[3];

  // 1722 thread chanends
  chan listener_ctl[AVB_NUM_LISTENER_UNITS];
  chan buf_ctl[AVB_NUM_LISTENER_UNITS];
  chan talker_ctl[AVB_NUM_TALKER_UNITS];

  // media control chanends
  chan media_ctl[AVB_NUM_MEDIA_UNITS];
  chan clk_ctl[AVB_NUM_MEDIA_CLOCKS];
  chan media_clock_ctl;

  streaming chan c_samples_to_codec; // streaming channel for audio
      samples

  chan xtcp[1]; // IP stack chanends

  chan c_gpio_ctl; // general-purpose I/O (buttons)

  par
  {
  // Ethernet stack
    on stdcore[1]: {
      int mac_address[2];
      ethernet_getmac_otp(otp_data, otp_addr, otp_ctrl, (mac_address,
          char[]));
      phy_init(clk_smi, p_mii_resetn, smi[0], mii0);
      ethernet_server(mii0, mac_address, rx_link, 4, tx_link, 5, smi[0],
          connect_status);
    }

  // IP stack
    on stdcore[1]: uip_server(rx_link[1], tx_link[2], xtcp, 1, null,
        connect_status);
```

Listing 2.2: Execution threads in the XMOS AVB application (part 2)

```
  // 802.1AS server
    on stdcore[1]: {
      DSP_RST <: 0;
      audio_clock_CS4272_init(r_i2c, MASTER_TO_WORDCLOCK_RATIO);
      p_codec_gain <: 0x0F;
      audio_codec_CS4272_init(p_codec_ctl, r_i2c, 0);
      ptp_server_and_gpio(rx_link[0], tx_link[0], ptp_link, 3,
          PTP_GRANDMASTER_CAPABLE, c_gpio_ctl);
    }
    on stdcore[1]: media_clock_server(media_clock_ctl, ptp_link[1],
        buf_ctl, AVB_NUM_LISTENER_UNITS, clk_ctl, AVB_NUM_MEDIA_CLOCKS);

  // Audio codec server
    on stdcore[0]: {
      init_media_input_fifos(ififos, ififo_data, AVB_NUM_MEDIA_INPUTS);
      configure_clock_src(b_mclk, p_aud_mclk);
      start_clock(b_mclk);
  // Clock signal generation and audio codec run as parallel threads
      par
      {
        audio_gen_CS4272_clock(p_fs, clk_ctl[0]);
        i2s_master (b_mclk, b_bclk, p_aud_bclk, p_aud_lrclk, p_aud_dout,
            AVB_NUM_MEDIA_OUTPUTS, p_aud_din, AVB_NUM_MEDIA_INPUTS,
            MASTER_TO_WORDCLOCK_RATIO, c_samples_to_codec, ififos,
            media_ctl[0], 0);
      }
    }

    on stdcore[0]: avb_1722_talker(ptp_link[0], tx_link[1], talker_ctl
        [0], AVB_NUM_SOURCES);

    on stdcore[0]: avb_1722_listener(rx_link[3], tx_link[4], buf_ctl[0],
        listener_ctl[0], AVB_NUM_SINKS);

  // Media output server
    on stdcore[0]: {
      init_media_output_fifos(ofifos, ofifo_data, AVB_NUM_MEDIA_OUTPUTS);
      media_output_fifo_to_xc_channel_split_lr(media_ctl[1],
          c_samples_to_codec, 0, ofifos, AVB_NUM_MEDIA_OUTPUTS);
    }

    on stdcore[0]: xlog_server_uart(p_uart_tx);

  // Application threads
    on stdcore[0]: {
  // First initialize avb higher level protocols
      avb_init(media_ctl, listener_ctl, talker_ctl, media_clock_ctl,
          rx_link[2], tx_link[3], ptp_link[2]);
  // Now run primary event handler application
      demo(xtcp[0], rx_link[2], tx_link[3], c_gpio_ctl);
    }
  }
}
```

The `demo()` thread is the primary control and event handler for the Ethernet AVB end station application. Listing 2.3 is a (very abridged) representation of this thread, demonstrating the use of the `select()` statement to wait on and handle different kinds of event.

Listing 2.3: Event handling in the demo() thread

```
void demo(chanend tcp_svr, chanend c_rx, chanend c_tx, chanend c_gpio_ctl
    ) {
  // variable declarations go here

  // initialise and configure control subsystems, etc.
  // initialise periodic timeout
  // main loop
  while(1) {
    // further variable declarations go here
    select
    {
      case avb_get_control_packet(c_rx, buf, nbytes):
      // Process Layer 2 AVB control packets, including MSRP, MAAP etc
        avb_status = avb_process_control_packets(buf, nbytes, c_tx);
        switch(avb_status)
        {
          // Handling various cases
        }
        break;

      case xtcp_event(tcp_svr, conn):
      // Process Layer 3 IP events, including MDNS
      // and proprietary control protocol messages
        break;

      case c_gpio_ctl :> int cmd;
      // Handle button presses
        break;

      case tmr when timerafter(timeout) :> void:
      // AVB periodic processing - check for new streams
        break;
    }
  }
}
```

2.4.3 The XMOS Ethernet AVB API

The XMOS Ethernet AVB reference design provides an extensive API that provides control over the IEEE 1722 Talker and Listener subsystems, audio interfacing and the media clock server.

As shown in Listing 2.1 and 2.2, the XMOS XS1 architecture and the XC language enable clear functional separation between communicating concurrent tasks. Chapter 4 examines in more detail how the XS1 architecture and XC language facilitate software development at this level, with specific attention to the development of audio control protocols.

2.5 Conclusion

Many of the requirements of audio distribution applications can be expressed in terms of 'quality of service'. Providing quality of service networks suitable for realtime audio streaming presents significant challenges. Additionally, there is a requirement to encapsulate and present quality of service in an accessible way for end users, who may not and should not be expected to gain specialist networking expertise. A detailed overview of the Ethernet AVB component standards has shown how they interoperate to provide quality of service networking and other important requirements, including accessibility.

Issues of connection management and control of an audio distribution network were briefly discussed in the overview of IEEE 1722.1. These issues are discussed in more depth in Chapter 3, which also provides a detailed account of the Audio Engineering Society's AES-64 standard.

The XMOS Ethernet AVB 'reference design' provides a software-defined implementation of an Ethernet AVB end station and the practical foundation for the research project. The XMOS XS1 architecture and its associated development tools are discussed in Chapter 4. Chapter 5 and 6 provide an account of how the research project's software output, an XMOS-based implementation of an AES-64 protocol stack and control application, was integrated with the XMOS Ethernet AVB reference design.

3 Connection management and control in audio networks, and the AES-64 standard

3.1 Introduction

Effective connection management and control services are critical to the operation of an audio distribution network. The Audio Engineering Society's AES-64 standard provides an IP-based approach to connection management and control of professional audio devices. The AES-64 standard predates the Ethernet AVB standards, but has been shown to provide connection management services for Ethernet AVB and IEEE 1394-based audio networks [P. Foulkes et al. 2011].

This chapter discusses the significance and application of connection management and control in audio networks, followed by an introduction to the AES-64 standard.

3.2 Connection management and control in audio networks

As discussed in Chapter 2, audio networks aim to extend or supplant conventional forms of audio distribution in professional application areas: live sound, installed sound, and so on. Although these application areas have many disparate requirements and operational priorities, a requirement for in-service control and instrumentation of the network is accepted as universal. Gross reports:

> Sixty-four percent of respondents considered monitoring or remote control mandatory. Among installed-sound respondents the call was even louder at 84 percent ... this may turn out to be the most important factor driving adoption of audio networks. Especially in installed sound, the primary draw of an audio network may be in the control and monitoring capabilities. [Gross 2006, 66]

This introduces the concept of a *network controller* (Fig. 3.1), which must represent the live network configuration (in terms of devices, interfaces, and audio stream connections) to an audio application engineer and allow the engineer to make choices about how the network operates, including the set up and tear down of stream connections, assignment of stream channels to/from audio interfaces, and other tasks. To support this role, the audio network may implement a protocol or protocols to:

1. discover the live network configuration;
2. for each discovered device, enumerate its interfaces and capabilities;
3. manage stream connections between the discovered devices;
4. provide further control over devices.

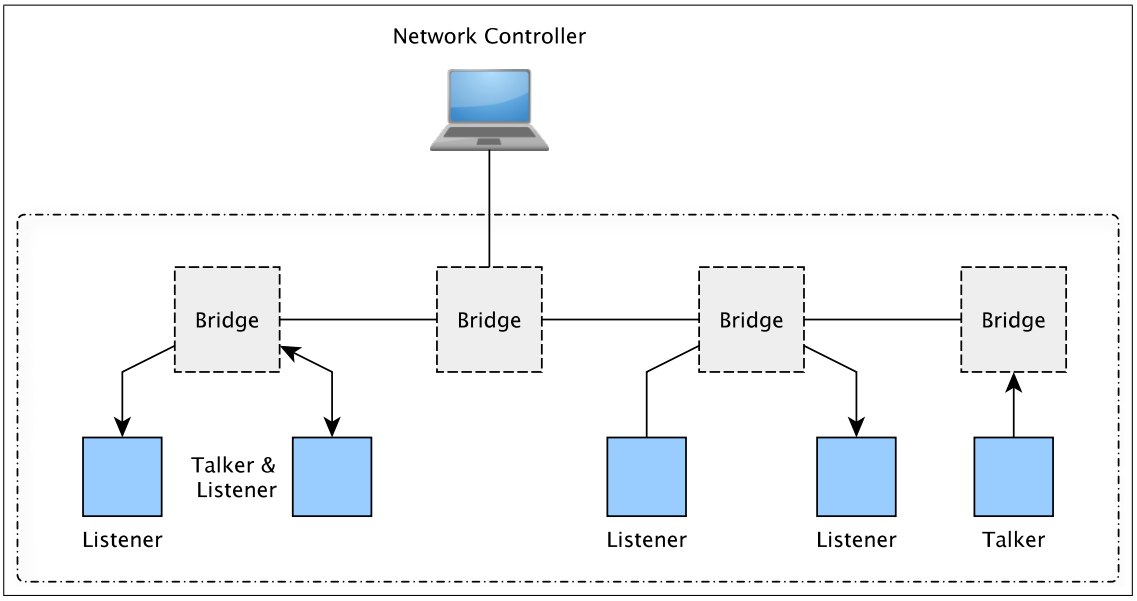


Figure 3.1: An Ethernet AVB audio network with a network controller

A network may be operated through multiple controllers (Fig. 3.2), and as long as a controller implements the basic tasks indicated above, it may present them to the user via any effective graphical interface.

As shown in Chapter 2, Ethernet AVB presents one approach to the set of protocol tasks in the specification of the IEEE 1722.1 standard.

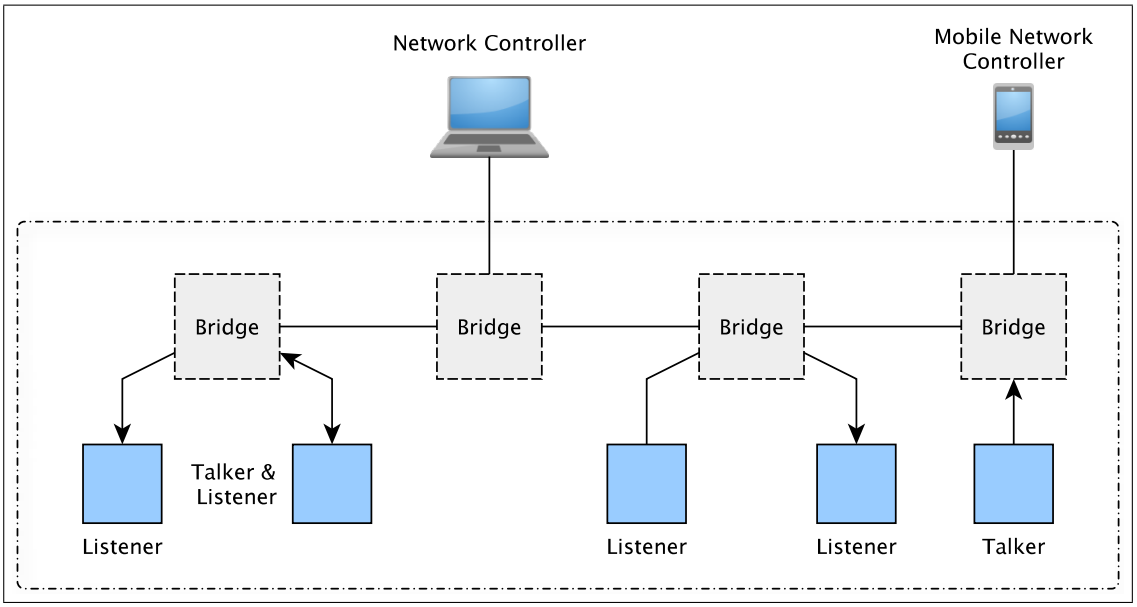


Figure 3.2: An Ethernet AVB audio network with multiple controllers, one a mobile device

3.2.1 Control standards and autoconfiguration

Chapter 2 briefly discussed conventional (i.e., non-network) audio distribution systems and the complexity of initial configuration, maintenance, and operation. While audio networks aim to rationalise and simplify these user processes, the transition from 'many point-to-point connections' to 'a freely re-routable network of connection nodes' obviously involves concepts that may be unfamiliar to a user of conventional audio systems, e.g., 'audio streams' or even device (IPv4, IPv6, MAC) addresses.

As such, users will clearly benefit from a control standard or standards that encapsulate the processes involving these concepts. In the context of connection management, this may be achieved by providing a graphical representation of a software patchbay which the user can use to patch connections between network devices; in the context of network setup and configuration, this may be achieved by a simple discovery protocol that identifies any device on the network infrastructure and launches an enumeration procedure for each.

3.3 Introduction to AES-64

The Audio Engineering Society's AES-64 standard for networked command, control and connection management for integrated media 'specifies an approach to the control and monitoring of professional audio devices within any IP-based audio and other

media networks, and furthermore specifies how such an approach can be integrated with connection management.' [Audio Engineering Society 2012, 5]

As such, the AES-64 standard describes:

- a scheme for representing networked devices of arbitrary complexity,
- a IP-based peer-to-peer protocol for networked device discovery and control,
- a set of high-level command mechanisms for realtime control of multimedia systems.

The representation of networked audio devices presents some interesting challenges. Audio devices tend to have inherently complex structures: multiple instances of functional components (such as input or output channels) are common, and each instance of a functional component often comprises a set of instances of *smaller* functional components. The representation of an individual state or control on an audio device must be uniquely and clearly *located* within the representation of the device.

The messaging specification of a standard must define a vocabulary of commands with associated actions and responses – the fundamental commands being 'get control value' and 'set control value' – as well as providing an addressing scheme capable of locating instances of a functional component within the representation of a device. As implied, this scheme should be capable of locating a specific instance of a device component, as well as some or all of the device components within an instance of a higher-level device component.

Control mechanisms expand the basic command vocabulary to streamline common or mundane operations: enumerating the structure of a device, for example, or simultaneously setting the same value to multiple controls on one or more devices.

The following sections describe these aspects of the standard in turn.

3.3.1 AES-64 device requirements

An AES-64 compatible device must implement an AES-64 protocol stack between the device's primary control software and the AES-64 network. The high-level structure of such a device is shown in Fig. 3.5. Appendix A Section A.2 provides detailed prerequisites for an AES-64-compatible device.

Since AES-64 is an IP-based protocol, the device must also implement an IP stack. By comparison, this is not necessary for a device that implements IEEE 1722.1 (AVDECC); on the other hand, only Ethernet AVB devices can implement IEEE 1722.1.

Every AES-64 device *must* present a set of 'configuration' parameters (Fig 3.3) that provide its management address, the name of the device, and what *type* of device it is in

the most general terms (e.g., 'an Ethernet AVB device', 'an AES-64 network controller'). These parameters are always located on device node '0', at the following positions in the level hierarchy:

```
DEVICE INFO CONFIG  IP      1   CONFIG  1   IP ADDRESS   1   (parameter)
DEVICE INFO CONFIG  DEVICE  1   CONFIG  1   DEVICE NAME  1   (parameter)
DEVICE INFO CONFIG  DEVICE  1   CONFIG  1   DEVICE TYPE  1   (parameter)
```

These parameters are queried by any AES-64 controller performing network discovery (discussed in Subsection 3.6.1).

3.4 Device representation in AES-64

An AES-64 device representation contains at least one *device node*. Each device node contains an array of *parameters* and a *level hierarchy* that positions each parameter within a representation of the device's functional organisation (Fig. 3.3). Parameters can represent read-only states, like signal meter levels, or read-write controls, like volume controls (Fig. 3.4). The first category can be thought of as *monitor* parameters while the second can be thought of as *control* parameters. Monitor parameters indicate observable states, e.g., metering the amplitude of an audio signal. Control parameters represent controllable states or variables.

The AES-64 device representation provides an interface between the control system of the device itself and AES-64 messages (Fig. 3.5). If the control system of the device changes a state or the value of a control (for example, if a user moves the volume fader shown in Fig. 3.4), the corresponding AES-64 parameter's value will be updated. Conversely, if an AES-64 request message from a network controller alters the value of an AES-64 parameter, the value change will produce a change to the control system of the device.

3.4.1 The AES-64 parameter

The majority of command operations provided by the AES-64 protocol are based on the manipulation of parameters and their attributes. The primary attribute of an AES-64 parameter is its value, which represents and may control device behaviour as described in the preceding section. Additional parameter attributes support AES-64 control operations, such as parameter grouping.

Parameters are registered on a device node by the control system of the device itself, which must also register some form of link between the parameter's value attribute and

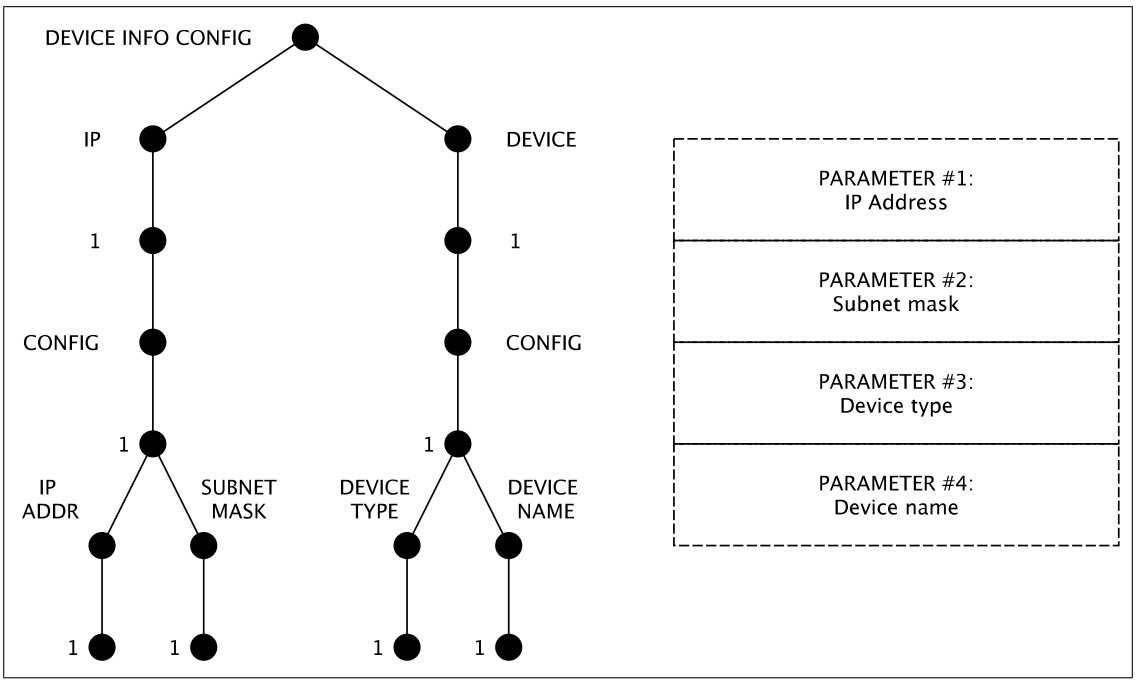


Figure 3.3: Schematic of an AES64 device node with configuration parameters, showing the parameter array (R) and the level hierarchy's structuring of the parameter array (L).

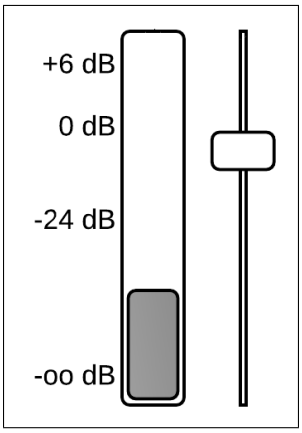


Figure 3.4: Parameter types. The signal meter level (L) can be represented by a read-only parameter; the volume fader control (R) by a read-write parameter.

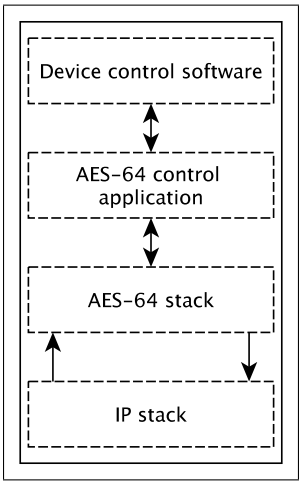


Figure 3.5: The AES-64 device representation provides an interface between a device's control software and AES-64 requests from the network.

the corresponding feature of the control system that the parameter represents [Audio Engineering Society 2011, 7-10]. Typically this link will need to support read and write access to the control system feature. For the purposes of this thesis, this link will be termed a parameter 'value function'.

3.4.2 The level hierarchy

Functionally, the device node and parameter are fairly straightforward: a device node contains a set of parameters and a level hierarchy that organises them; a parameter contains a value along with other attributes, and its value is linked to some aspect of the host device's operations.

The level hierarchy is more complex, in that it presents a logical representation of their functionality to the network. (AES-64 controllers are not exempt from this.) Representing audio devices presents particular challenges:

1. audio devices may be very complex;
2. audio devices frequently contain, within that degree of complexity, a great deal of 'self-similarity'

The Yamaha 01X digital mixing console [White 2004] provides convenient examples of both these issues. The 4-band frequency equalisation (EQ) block for a single input channel on this mixing console is represented in device firmware by twenty-five (25) parameters [Yamaha Corporation Pro Audio and Digital Musical Instrument Division

2003, 96-99]. This block of parameters is divided further into four control blocks, each for distinct frequency bands, and a set of storage and recall parameters. A satisfactory device representation of the 01X has to be capable of expressing these degrees of granularity.

Each AES-64 parameter has a unique location within its parent device node's level hierarchy. For example, the 'IP ADDRESS' parameter shown in Fig. 3.3 is both located and described by the graph of levels connecting it to the device node:

```
DEVICE INFO CONFIG  IP   1   CONFIG  1   IP ADDRESS  1   (parameter)
```

The 'full address block' that locates and describes a parameter consists of precisely *seven* entries, each more specific than the last:

1. 'Section Block' (`DEVICE INFO CONFIG`)
2. 'Section Type' (`IP`)
3. 'Section Number' (`1`)
4. 'Parameter Block' (`CONFIG`)
5. 'Parameter Block Index' (`1`)
6. 'Parameter Type' (`IP ADDRESS`)
7. 'Parameter Index' (`1`)

The AES-64 specification defines a set of valid identifiers for each type of entry: for example, *DEVICE INFO CONFIG* is a valid identifier for the Section Block entry, but not for any other. A level hierarchy may contain multiple instances of an identifier as long as each instance has a different parent: for example, in Fig. 3.3, there are two 'Parameter Block' instances which both have the identifier 'CONFIG', but each has a different parent.

In the case of the Yamaha 01X mixer's input channel equalisation block, the following full address blocks can identify a 'frequency' control for each of the four frequency bands:

```
INPUT SIGNAL  AUDIO   1   PARAMETRIC EQ  1   LO PASS FREQ       1   (parameter)
INPUT SIGNAL  AUDIO   1   PARAMETRIC EQ  1   BAND PASS LO FREQ  1   (parameter)
INPUT SIGNAL  AUDIO   1   PARAMETRIC EQ  1   BAND PASS HI FREQ  1   (parameter)
INPUT SIGNAL  AUDIO   1   PARAMETRIC EQ  1   HI PASS FREQ       1   (parameter)
```

The second challenge in providing a representation of this sort of device is that the representation must distinguish between strongly similar sections of a device. A representation of the 01X must be able to express that there is one instance of the EQ block for each of the mixer's twenty-four (24) input channels and a further two (2) instances for each channel of the mixer's stereo output [Yamaha Corporation Pro Audio and Digital Musical Instrument Division 2003, 28-29].

The AES-64 level hierarchy provides this capability through the three 'index' levels ('Section Number', 'Parameter Block Index', and 'Parameter Index'), which solely specify an *instance* of the control section identified by the preceding level entries. In the case of the mixer's 24 input channels, this looks like:

```
INPUT SIGNAL AUDIO   1   PARA EQ  1   LO PASS FREQ  1   (parameter)
...
INPUT SIGNAL AUDIO  24   PARA EQ  1   LO PASS FREQ  1   (parameter)
```

3.5 Messaging in AES-64

Messaging in AES-64 conforms to the request-response model:

> Every AES-64 message will either be a request sent from one device to another, or will be a response from a device after a request has been made. [Audio Engineering Society 2012, 16]

A request message may or may not demand a response (depending largely on the command the request provides).

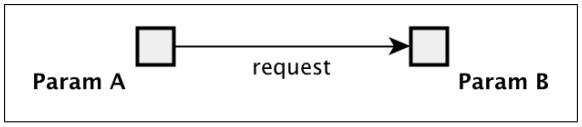


Figure 3.6: An AES64 request message which does not demand a response.

As will be shown later in this section, there are a few variations on this basic theme: some types of request provoke the recipient to make further requests, and it is also possible for a request to provoke more than one response.

3.5.1 Message targets

The AES-64 standard permits request messages to address a device in several ways. A request message may target:

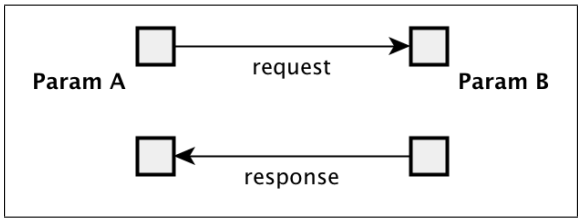


Figure 3.7: An AES64 request which demands a response, and the corresponding response.

1. an entire device node;
2. a group of related parameters within the level hierarchy of a device node; (see Subsection 3.5.2)
3. an individual parameter within a device node.

When a request has a parameter target, it may use one of two addressing schemes: it may either supply a 'full address block', locating the parameter within a device node's level hierarchy, or it may supply the parameter's unique index within the device node's parameter array. (The parameter's unique index can be retrieved through device enumeration or by sending a specific type of query request.) Addressing a request using the parameter's unique index conserves bandwidth (32 bits, as opposed to 104 bits for a full address block).

3.5.2 Wildcards

One important feature of the 'full address block' is that it enables a controller to use *wildcards* to specify multiple — or unknown — parameter targets.

In the case of the Yamaha 01X mixer, a single request can retrieve the current setting of the volume control on every one of the mixer's 24 input channels by supplying the following address block:

```
INPUT SIGNAL  AUDIO  [WILDCARD]  DIGITAL GAIN  1  LEVEL  1  (parameter)
```

Note that if a request that requires a response (e.g., a query request) is directed at multiple parameters, each parameter will send an *individual* response; so the request shown above would provoke the Yamaha 01X to send 24 response messages, one from each input channel's 'level' parameter (Fig. A.4).

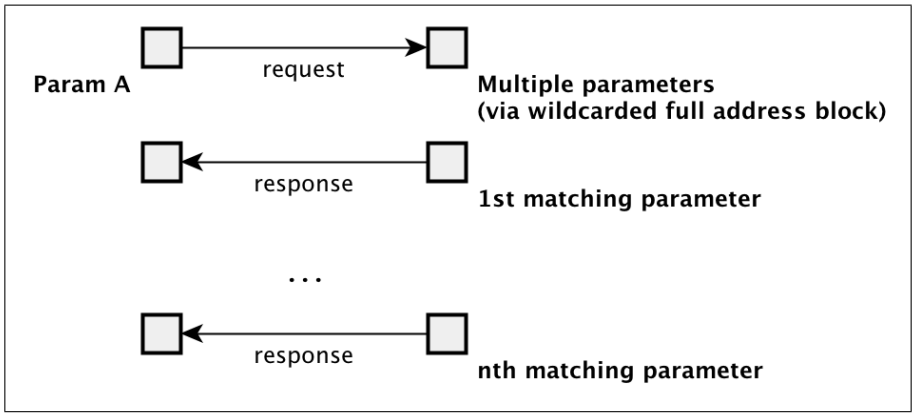


Figure 3.8: An AES64 request that expects a response, addressing n parameters through the use of a wildcarded address block. Each parameter returns a response message.

3.5.3 Message types

Requests that require a response have message types that are distinct from requests that do not.

The result of these distinctions is that AES-64 defines six formats of request message and one universal response format:

1. Parameter request providing a full address block target, expecting a response
2. Parameter request providing a full address block target, not expecting a response
3. Parameter request providing a parameter index target, expecting a response
4. Parameter request providing a parameter index target, not expecting a response
5. Response message
6. Device request expecting a response
7. Device request, not expecting a response

These message formats are defined and discussed in Appendix A Subsection A.7.1.

3.5.4 Message sources

All AES-64 request messages, regardless of their type, specify a parameter index *source*. For example, if an XFN request expects a response, the response will be addressed to the source parameter of the originating request. Requests that require a response must

provide a unique 'sequence ID', a 32-bit tag used to reconcile the expected response with its originating request. (When a single request provokes multiple responses, all the responses will use the original sequence ID.)

3.5.5 Message commands

AES-64 request messages express queries and imperative commands. The meaning of a command is expressed by the 'Command Executive' and 'Command Qualifier' fields of the request header, in combination with the Value field of the request message.

The value of the Command Executive field expresses a basic *verb*: e.g., 'GET' and 'SET'. The value of the Command Qualifier field typically refines the target of the command to an *attribute* of the object: for example, in a 'GET VAL' request, 'VAL' indicates a parameter's value attribute, whereas in a 'GET ID' request, 'ID' indicates a parameter's *unique ID* attribute (i.e., its index within the device node parameter array).

In some cases, the Command Qualifier provides an *adverb*: in the case of a 'JOIN PTP' request (Subsection 3.6.2), 'PTP' indicates the type of 'join' that should be performed.

There are a finite number of valid combinations of Command Executive and Command Qualifier: 'GET VAL', 'SET VAL', 'JOIN PTP' are all valid, whereas 'SET PTP' or 'JOIN VAL' are not. Each valid combination uniquely identifies one command.

Appendix A Subsection A.7.2 defines the possible values of the Command Executive and Command Qualifier fields. Appendix A Section A.8 provides a dictionary of AES-64 commands, including the format of their requests and responses.

3.6 Control features in AES-64

Many control tasks can be conveniently executed by issuing a single XFN request. However, certain control tasks involve extended interactions, including:

1. Network discovery, where an AES-64 controller discovers what AES-64 devices are present on a network;

2. Device enumeration, where an AES-64 controller retrieves the parameters (and active configuration) of an AES-64 device as efficiently and rapidly as possible;

3. Push notification, where an AES-64 device transmits regular updates of the values of interesting parameters;

4. Parameter grouping, which establishes relationships between changes in value between parameters on the same device or on different devices;

5. Connection management, where one or more devices are configured to connect to an audio stream.

A thorough account of all these mechanisms is beyond the scope of this review, and at the time of writing only parameter grouping has been described within the published standard. However, a brief overview of network discovery, parameter grouping and connection management follows.

3.6.1 Network discovery

This process allows an AES-64 software controller to poll the network for AES-64 devices to enumerate and interact with, analogous to the operation of the IEEE 1722.1 AVDECC Discovery Protocol described at the start of this chapter.

As described in Subsection 3.3.1, every AES-64 device must present a set of 'configuration parameters' on device node '0'[1]. In the simplest version of network discovery, the AES-64 network controller may simply send a 'GET VAL' request specifying device node ID '0' and the following full address block:

```
DEVICE INFO CONFIG  IP   1   CONFIG  1   IP ADDRESS  1
```

to the IPv4 broadcast address[2] (255.255.255.255) [Audio Engineering Society 2012, 24]. Following this broadcast, the AES-64 network controller can wait to receive responses.

3.6.2 Parameter grouping

A *parameter group* establishes a relationship between the *value attributes* of a set of related parameters. This mechanism allows the linking of AES-64 parameters within devices, between devices, or between a controller and a device.

The basic purpose of parameter grouping is to enable (effectively) simultaneous changes to values on multiple parameters, which may or may not be on the same device. Once a parameter group is established, any change to a parameter's value will be selectively relayed to other parameters in the group[3]. A parameter group may enforce one of two kinds of value relationship: 'absolute', in which all members of the group are locked to exactly the same value (Fig. 3.9); and 'relative', in which all members of the group

[1] Typically, all other device parameters are presented on device node '1', although this behaviour is not yet standardised.
[2] The broadcast address is viable for this purpose since conventional audio distribution networks do not involve routers.
[3] although some restrictions apply.

maintain the offsets between their values that existed when the group was created (Fig. 3.10).

AES-64's implementation of this mechanism is rich and subtle, and a full account of it is beyond the scope of this study. [N. Chigwamba, et al 2012] provides a detailed overview of the mechanism, its origins in conventional audio systems, and its varied behaviours. It is chiefly of interest here because it presents two final messaging patterns to the behaviour of an AES-64 stack (Fig. A.5 and A.6).

In the first messaging pattern, the request from Param A is a 'SET VAL' request targeting Param B. Param B is a member of a parameter group with Param C and Param D, and as such issues 'follow-on' 'SET GRPVAL' requests to C and D, informing each that its value has been changed and providing the new value. Param C and D will typically adjust their values to preserve the relationship that existed before Param A issued the 'SET VAL' request.

In the second messaging pattern, the request from Param A is a 'JOIN MSTSLV' request targeting Param B. A 'JOIN MSTSLV' request commands its target to *establish* a parameter group with Param C: the details of the parameter are supplied in the request's value field. For Param B to establish a parameter group with Param C, Param B needs to learn what *existing* parameter groups Param C is a member of. If, for example, Param C is in a parameter group with Params D and E, Param B may need[4] to add Param D and E to the parameter group it forms with Param C. All parameters are obliged to maintain their group membership information during this sequence: there are no implications for Param A, but Param B has now added Params C, D and E to its 'slaves' group list; Param C, D and E have now added Param B to their respective 'masters' group lists.

3.6.3 Connection management

AES-64 uses the 'multicore' metaphor to represent the general concept of an audio stream. The multicore metaphor is adopted from conventional audio distribution (Fig. 3.13), and expresses the idea that an audio stream may carry multiple discrete audio signals (or 'channels'). This generalisation means that an AES-64 audio device may support one or more audio transport technologies while the *control* of those transport technologies may be expressed in general terms.

[4] not 'must'; again, some restrictions apply; see [N. Chigwamba, et al 2012, 10].

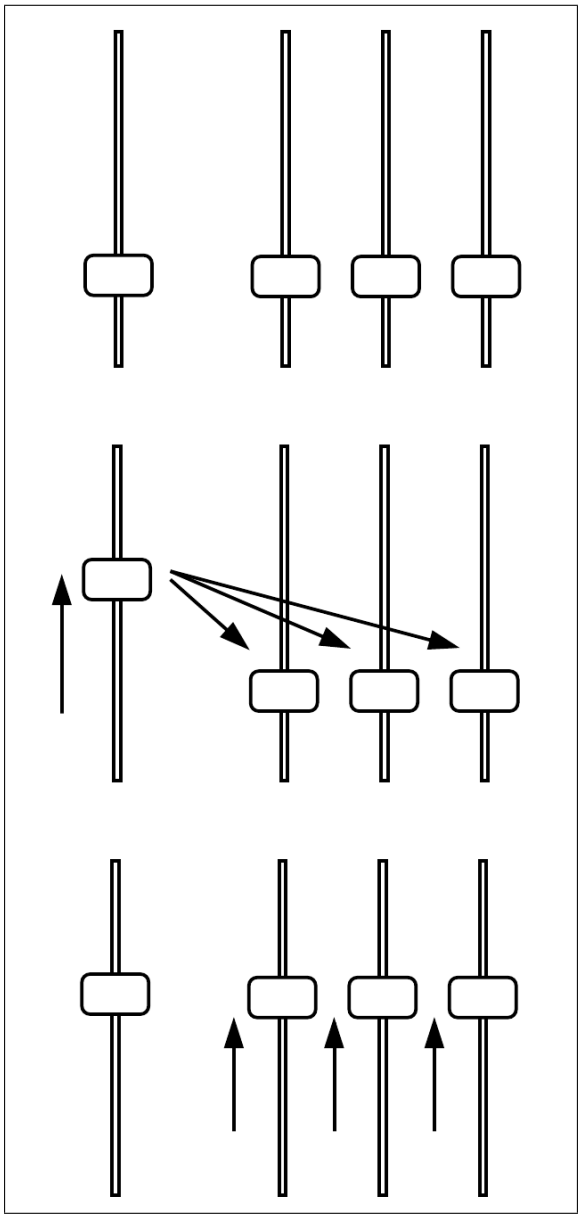


Figure 3.9: Value changes in a parameter group that enforces an 'absolute' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group.

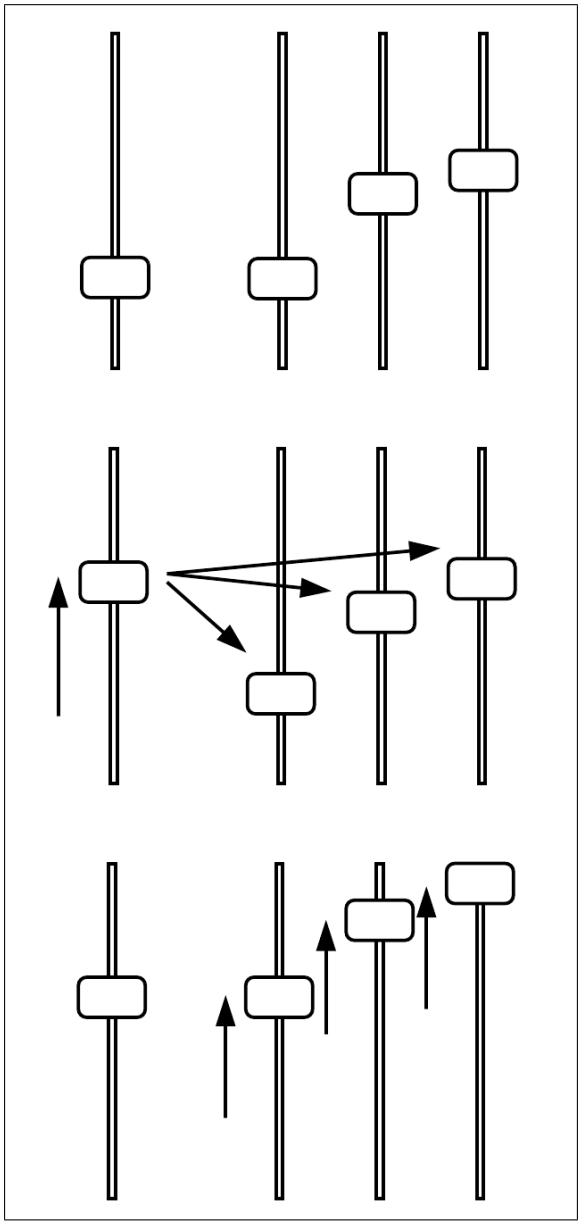


Figure 3.10: Value changes in a parameter group that enforces a 'relative' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group.

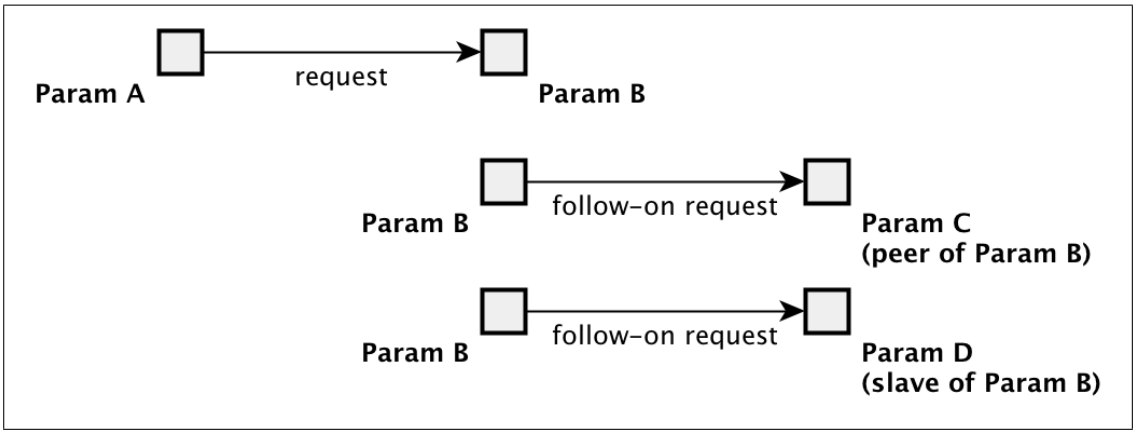


Figure 3.11: An AES64 request that initiates the target parameter, which is a member of a parameter group, to issue group value update requests.

Ethernet AVB connection management

In the case of Ethernet AVB, AES-64 control cannot directly command that a Talker should begin streaming audio to one or more Listeners. It would be unacceptable for AES-64 control to circumvent the quality of service negotiation mechanisms provided by the audio transport system: as shown in Chapter 2, streaming may only take place between two or more stations when it has been guaranteed that a supportive route between the two exists. This is analogous to the operation of the IEEE 1722.1 AVDECC Connection Management Protocol [IEEE P1722.1/D21 2012, 269]; although the pattern of messaging between network controller, Talker and Listener differs, the effect and principles are the same.

Consequently, to initiate an IEEE 1722 streaming connection, AES-64 control must prompt and direct the MSRP components on the Talker and Listener(s) to initiate and negotiate the connection. Foulkes describes this process as follows:

> When the Connection Manager wishes to establish a connection between two XFN[5] devices on an Ethernet AVB network, it performs the following sequence of events:
>
> 1. It issues an XFN *get value* request to the transmitting device to get the value of the stream ID parameter of the multicore that is to transmit the stream.
> 2. It issues an XFN *set value* request to the receiving device to set [the] stream ID parameter of the multicore that is to receive the stream.

[5] At the time Foulkes was writing, AES-64 was officially known as 'XFN'.

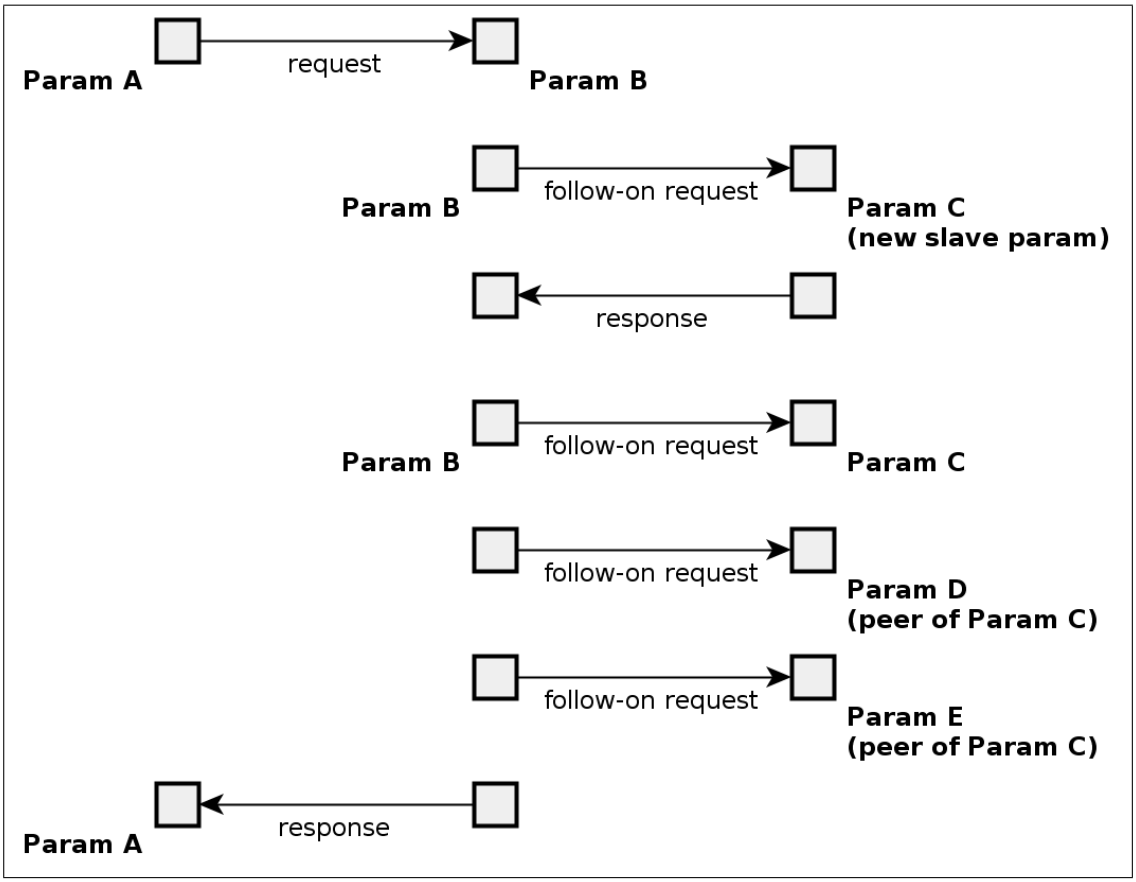


Figure 3.12: An AES64 request that initiates the target parameter to set up a parameter group, which involves the target parameter issuing follow-on requests, receiving and processing responses, and issuing further follow-on requests before finally acknowledging success or failure to the source of the original request. (Details of Param C are made available to Param B in the value field of the initiating request sent by Param A.)

Figure 3.13: A multicore audio distribution cable, which conveys many independent audio signals within a single insulator. AES-64 uses the 'multicore' metaphor to describe the concept of an audio stream, agnostic of the specific audio network in use.

3. It issues an XFN *set value* request to the transmitting device to set the advertise parameter of the multicore that is to transmit the stream.
4. It issues an XFN *set value* request to the receiving device to set the listen parameter of the multicore that is to receive the stream.

[Foulkes 2011, 324]

This messaging sequence may be automated in response to a single user action in operator software; Foulkes describes this automation and the interface of such operator software (2011, 333, 322).

In order for the AES-64 implementation to perform Ethernet AVB stream connection management, the parameters depicted in Fig. 6.7 must be registered. In this example, which depicts AES-64 control over a combined Talker/Listener device, the parameters in the '`INPUT SIGNAL`' section block enact control over the operation of the Listener component, and the parameters in the '`OUTPUT SIGNAL`' section block control the operation of the Talker component.

The control application must register value functions for each of these four parameters, so that:

- Sending a 'get value' request to the '`OUTPUT SIGNAL`' — '`STREAM ID`' parameter will cause its value function to retrieve the current StreamID of the Talker component.

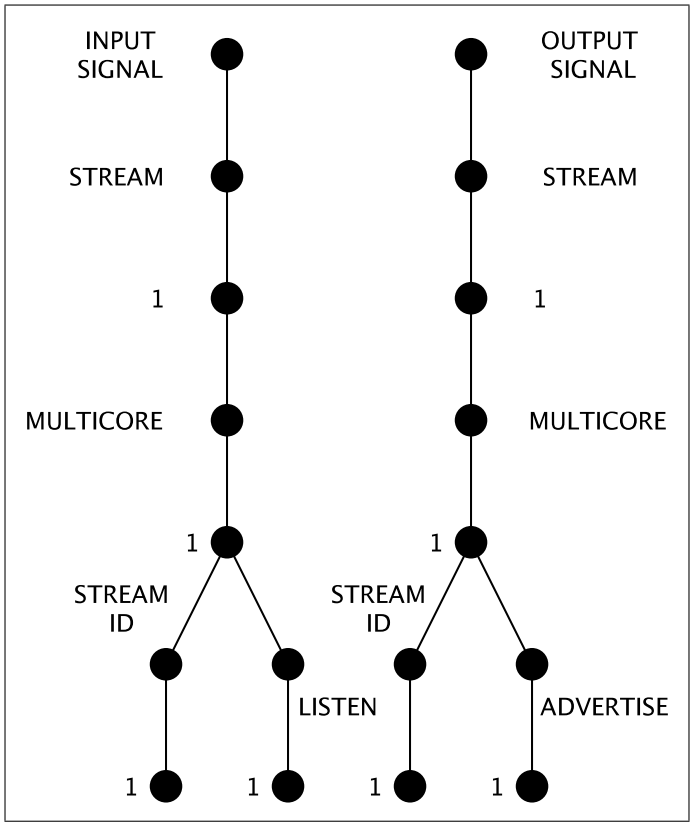


Figure 3.14: AVB stream connection parameters.

- Sending a 'set value' request to the '`OUTPUT SIGNAL`' — '`ADVERTISE`' parameter will cause its value function to enable or disable the Talker component from sending advertise messages to the network[6].
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`STREAM ID`' parameter will cause its value function to control *which* Talker's advertise messages the Listener may express an interest in[7]. In this case, the 'set value' request will supply a StreamID retrieved from a Talker through a previous 'get value' request.
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`LISTEN`' parameter will cause its value function to enable or disable the Listener component from sending 'Listener Ready' messages. The Listener may *only* send a 'Listener Ready' message in response to a Talker advertise if the Talker's stream ID is set on the Listener's '`INPUT SIGNAL`' — '`STREAM ID`' parameter, but it can only do so if the LISTEN parameter is set to 'TRUE'.

3.7 An implementation of AES-64: the UNOS suite

An implementation of the AES-64 standard is provided by Universal Media Access Networks' 'UNOS' suite [Universal Media Access Networks 2012]. This comprises a protocol stack, 'UNOS Core', which has been ported to several architectures, and a PC controller application, 'UNOS Vision' (Fig. 3.15), that may be used to perform connection management, device monitoring and control tasks between audio network devices equipped with the 'UNOS Core' protocol stack or any other implementation of AES-64.

An example network is shown in Fig. 3.16.

3.8 Conclusion

The concept of abstraction is central to the AES-64 standard, which is capable of providing connection management and control services for any IP-capable network audio device. The standard provides a structured yet flexible approach to device representation, a powerful command lexicon and an expressive range of message formats.

[6] Enabling the sending of an advertise message is necessary to set up a stream connection, and this message is periodically re-sent through the lifetime of the stream connection; disabling the Talker from re-sending the message will tear down the stream connection at source.

[7] By sending a 'Listener Ready' message in response to the advertise message.

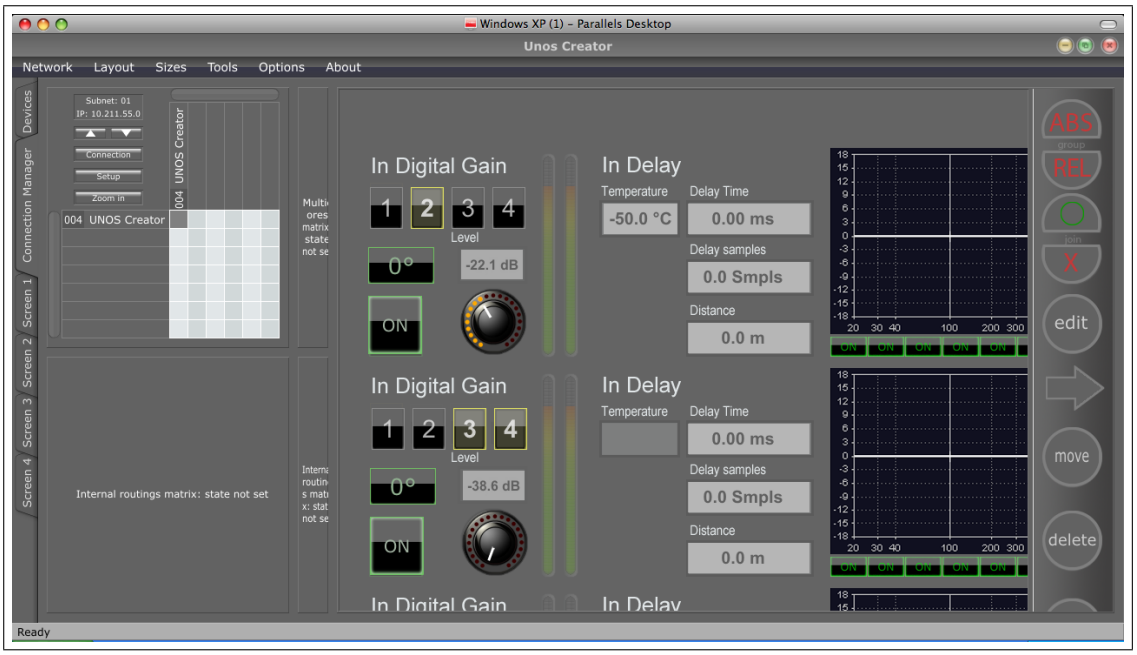


Figure 3.15: The 'UNOS Vision' AES-64 controller application.

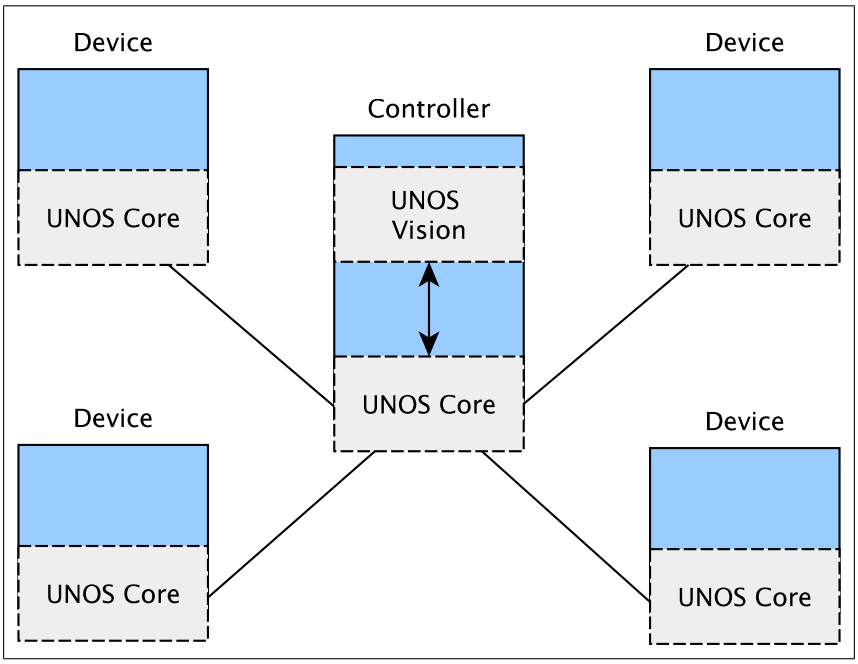


Figure 3.16: An audio network that implements AES-64 control and monitoring. A central controller is running the 'UNOS Vision' application and 'UNOS Core' protocol stack, while four devices run the 'UNOS Core' protocol stack.

AES-64 has been shown to provide a capable encapsulation of Ethernet AVB connection management. It presents a compelling case for research and development of control capabilities over other aspects of Ethernet AVB device configuration and operation.

4 The XMOS XS1 microcontroller architecture

The XMOS XS1 multicore microcontroller provides a powerful platform for research into Ethernet AVB systems. Section 2.4 introduced the XMOS Ethernet AVB 'reference design', which formed the practical basis of the software development in this research project.

The XS1 architecture provides an event-driven model for process execution and hardware support for concurrency, enabling the majority of an Ethernet AVB device (with the exception of an audio codec) to be implemented in software. 'Logical cores' support concurrent execution of independent software processes, with interprocess communication taking place over channels. Comprehensive hardware input-output provides a very direct approach to interfacing the microcontroller with external hardware (e.g., a third-party audio codec).

The architecture provides XC, a programming language that strongly resembles ANSI C with the addition of Occam-like keywords and operators to control channel communication, hardware input-output, and other architectural features. In addition to these new features, XC also enforces some restrictions that do not apply to the C language. XC can also be used to implement communication interfaces that enable software modules developed in C to execute as components of concurrent applications.

4.1 Introduction

The XMOS XS1 is a 32-bit RISC microcontroller architecture intended to provide a general-purpose multi-core microcontroller for embedded systems. The architecture aims to 'make it practical to use software to perform many functions which would normally be done by hardware' [May 2009a, 1]:

> The programmable processors are *general purpose* in the sense that they can execute languages such as C; they also have direct support for concurrent processing (multi-threading), communication and input-output. [Ibid.]

The XS1 architecture's approach to concurrency is strongly informed [May 2009b] by Communicating Sequential Processes [Hoare 1985] [Roscoe 2005], which in turn directly contributed to the joint development of the Occam language and the Inmos Transputer [Hyde 1995, 3]. The XS1 architecture could be considered as an update and adaptation of the Transputer, providing a reduced instruction set, greater input-output capacity and a more familiar programming language.

XS1-based microcontrollers are available in various configurations. Each microcontroller package provides at least one XS1 processing tile, at least one software-configurable switch, between 4 and 32 'logical cores', and between 64KB and 256KB of RAM. Many package configurations incorporate multiple XS1 processing tiles. Fig. 4.1 shows the configuration of the XMOS XS1-L2 microcontroller. Tiles can communicate data and events over the software-configurable switch. Communication between tiles on different microcontroller packages is also possible, if the board design provides links between the packages.

Each XS1 processing tile provides a scheduler, a number of logical cores and 64KB of RAM. An independent process can be allocated to execute on each logical core. Through its scheduler, the tile processes instructions from each logical core on a round-robin basis, ignoring any core that has not been assigned a process or that is temporarily suspended. Communication between processes is provided by message-passing channels.

Most aspects of the XS1 architecture can be controlled via the XC programming language. The following sections describe some aspects of the XS1 architecture, with examples where relevant, focussing on areas relevant to the research project's software implementation.

4.2 Processes and concurrency

The XMOS XS1 architecture is intended to support deterministic execution of many concurrent software-defined processes. Each process executes on a single logical core, and so the configuration of the XS1 microcontroller (the number of logical cores) determines how many concurrent processes can execute on that microcontroller.

For example: the XMOS/Attero Tech 'low-cost AVB endpoint' [XMOS 2011c] is based around an XS1-L2 microcontroller that provides 16 logical cores. The XMOS Ethernet AVB reference design builds a 'demo' application for this platform that executes 15 concurrent processes (Fig. 2.28).

The `par` statement defines a set of tasks that are to be executed by multiple logical cores in fork-join parallelism. The tasks may be trivial (an assignment operation) or

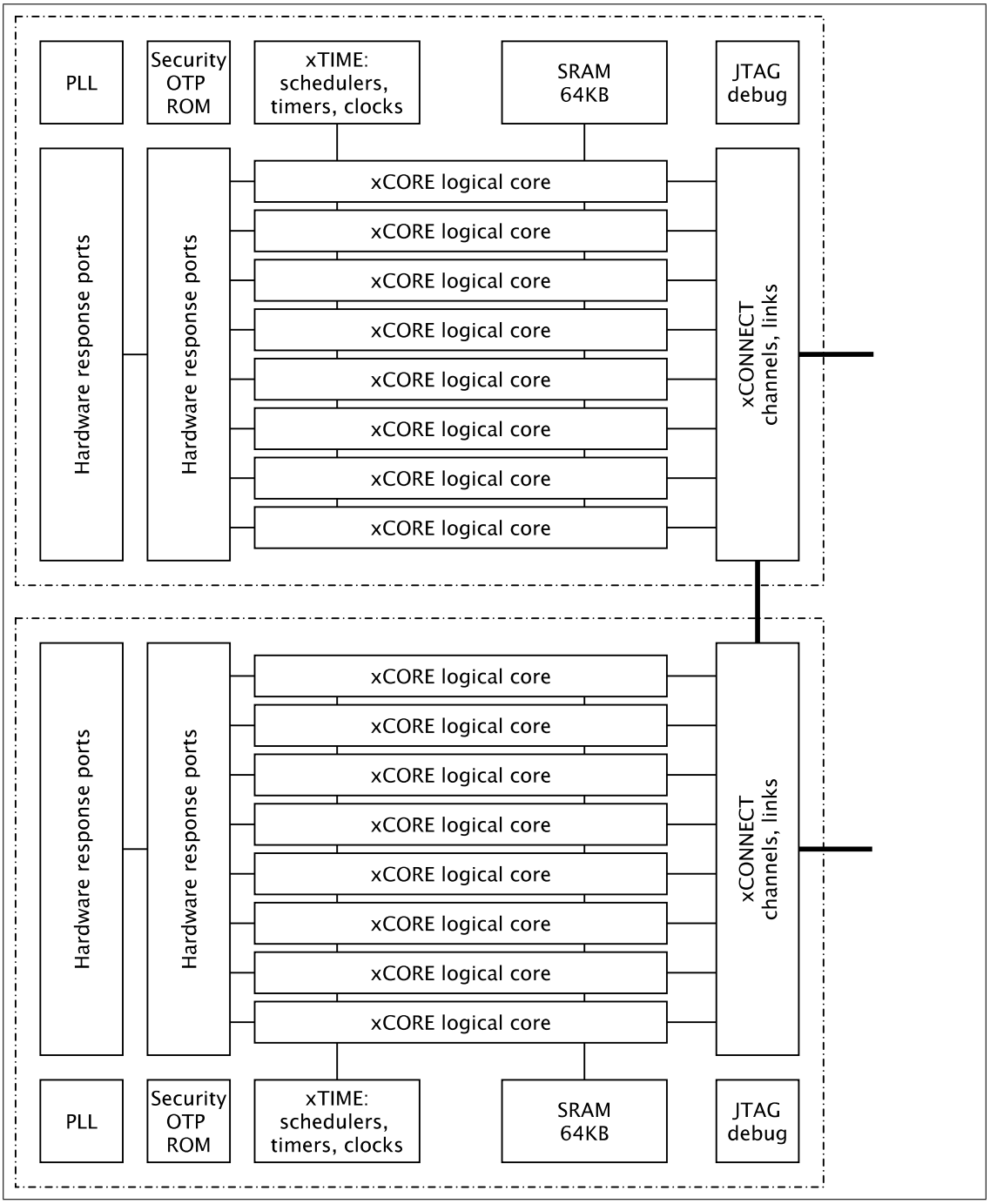


Figure 4.1: Schematic of the XS1-L2 microcontroller, showing two XS1 processing tiles and their software-configurable switches within a single package. Derived from [XMOS 2013].

elaborate (a function call). A task may also be defined as a block of serial commands.

A XS1 processing tile supports a maximum of eight concurrent tasks, and a `par` statement that declares more than eight tasks is invalid unless:

1. it is executing on a processing package with more than one XS1 tile, and
2. the `on ... tile[`n`]:` statement has been used within the `par` statement,

thereby spanning the tasks across multiple XS1 tiles.

The `on` statement allocates a task to execute on an identified processing tile (Listing 4.1). Further examples of use, in the context of a large `par` statement, are shown in the `main()` function of the Ethernet AVB demo application (Listing 2.1 and 2.2).

Listing 4.1: Task allocation

```
    // IP stack
2   on stdcore[1]: uip_server(rx_link[1], tx_link[2], xtcp, 1,
      null, connect_status);
```

The XC `stdcore` keyword shown in these examples does not denote a 'logical core': it denotes a processing tile[1]. For example, the XS1-L2 provides 16 logical cores distributed across 2 processing tiles, as illustrated in Fig. 2.28.

4.3 Events and event-driven execution

The XS1 architecture is 'event-driven' in that software processes do not use conventional interrupts to respond to external conditions. The XS1 architecture replaces interrupts with the concept of 'an event', which can be a condition on hardware input-output pins, a tick from any of the processing tile's configurable clocks and timers, or a message from another process arriving over a channel. Processes can implement periodic 'housekeeping' functions by configuring a timer.

The conventional model of execution for a process is that it declares a set of events that it is interested in, and then suspends. When an interesting event occurs, the process will execute its corresponding handler for that event and return to a suspended state. If an event happens while the process is handling an earlier event, the new event will

[1]This is the consequence of a recent change in the XMOS nomenclature, where 'logical cores' were originally called 'hardware threads' and 'XS1 tiles' were originally called 'XCores': the current version of XC has replaced the `stdcore` keyword with the `tile` keyword, but the version of the Ethernet AVB reference design used as the basis of this research project predates these changes.

be safely preserved[2] until the process returns to a suspended state, at which point the preserved event will be handled immediately.

4.3.1 The select statement

A process may declare the events it is interested in through the XC `select` statement, as shown in Listing 4.2, from the XMOS Ethernet AVB reference design's implementation of an IEEE 802.1AS server. In this example, the `select` statement registers an interest in three different kinds of events:

1. IEEE 802.1AS frames received over a channel from the Ethernet stack process by `ptp_recv_and_process_packet()`;
2. requests received over an array of channels from client processes that need IEEE 802.1AS information, as intercepted by `ptp_process_client_request()`;
3. a periodic timer configured to generate an event every 0.01 milliseconds (the value of `PTP_PERIODIC_TIME`).

Listing 4.2: Event registration using the select statement

```
select
{
case ptp_recv_and_process_packet(c_rx, c_tx):
      break;
case (int i=0;i<num_clients;i++) ptp_process_client_request(client[i]):
      break;
case ptp_timer when timerafter(ptp_timeout) :> void:
      ptp_periodic(c_tx, ptp_timeout);
      ptp_timeout += PTP_PERIODIC_TIME;
      break;
}
```

4.3.2 Parameterised select functions

Listing 4.2 also demonstrates two advanced techniques provided by XC. The case defined by a call to `ptp_recv_and_process_packet()` (line 3) is an example of a *parameterised select function* [XMOS 2011a, 25], which can be used to encapsulate processing for a particular type of event (especially if the processing is complex).

[2] In other words, any input-output operations on the event's source caused by the currently-executing event handler will not interfere with the pending event.

4.3.3 Replicated select functions

The second case (line 5) is also a parameterised select function — `ptp_process_client_request()` — but in this instance it is being used with a *replicator* [XMOS 2011a, 27] to iterate fairly over each IEEE 802.1AS client within the application.

Replicated select functions provide an elegant way to monitor a set of equivalent resources (whether they are chanends, hardware ports, etc.) for like events of the same significance. No IEEE 802.1AS client is 'more important' than any other - but this construct allows the select statement to monitor any non-zero number of clients.

4.3.4 Events and determinism

Uninterruptible event handlers and a scheduler that guarantees each logical core a minimum quantity of processing cycles makes it possible to reason about the worst-case execution times (WCET) of each thread as part of a larger concurrent system. For example, if a logical core is tasked with implementing an interface to external hardware that has hard realtime timing constraints, static analysis of the program can demonstrate both:

1. whether the WCET of the implementation will meet the external hardware's timing constraints;

2. whether the microcontroller provides sufficient processing cycles to the thread, when *also* running the larger concurrent system, for the interface implementation to execute within those timing constraints;

The first analysis establishes whether the interface as defined by software is capable of executing (in isolation) within the external hardware's time constraints; the second analysis establishes whether the interface is *still* capable of executing within the external hardware's time constraints when it is one of several concurrent threads running on the microcontroller:

> The threads[3] in an XCore are intended to be used to perform several simultaneous real-time tasks such as input-output operations, so it is important that the performance of an individual thread can be guaranteed. [May 2009a, 9]

To support these assessments, the XMOS development environment incorporates a comprehensive timing analysis toolkit.

[3]The quoted document predates the current XMOS terminology; for 'thread' read 'logical core', for 'XCore' read 'XS1 tile'.

4.4 Channels

A process may communicate with another process by passing messages over a *channel*. A channel is a synchronous, point-to-point, lossless communications link between two concurrent processes, each of which must have allocated a *channel end* as the terminating point of the link.

The XS1 microcontroller implements a software-controlled interconnect between its logical cores. A channel reserves a temporary route between two processes over the interconnect. Typically[4], the reservation is released when the communication has completed. Channels can provide communications between logical cores on the same processing tile, on different processing tiles, and on different XS1-based microprocessors. Channels support bidirectional communication (i.e., a process can send *and* receive data over a single channel) and implement limited buffering.

4.4.1 Channel input-output in XC

Channels are represented in the XC language by the *chanend* type ('CHANnel END'). XC provides two basic operators for input-output over resources (including hardware ports), and these can be used with chanends:

1. `chanend <: variable` outputs the value of a variable via a channel end;
2. `chanend :> variable` inputs data from a channel end into a variable.

Listing 4.3 shows an example of channel usage taken from the research project's implementation of an AES-64 protocol stack, where a 'message processing' task communicates with a 'device representation' task that maintains a set of AES-64 device nodes (Fig. 6.1). In this case, the message processing task needs to know if the target device node specified in an AES-64 request message exists on the local device. To do so, it executes the `a64_dnm_api_validate_dnode_id` function, which queries the device representation task to find if it has a device node with the ID specified by the `dnode_id` variable. Fig. 4.2 shows a diagram of the communications involved in this process.

In Listing 4.3, lines 4 and 5 perform output to the `c_dnm`[5] chanend: line 4 calls a function that sends a 'validate device node ID' command, and line 5 outputs the ID of the device node. The function will then block on input from the chanend in line 6

[4] A channel can also be *permanently* reserved to support streaming communications between processes.
[5] The 'dnm' abbreviation derives from a phase in the design process where the device representation task was unofficially named the 'device node manager'.

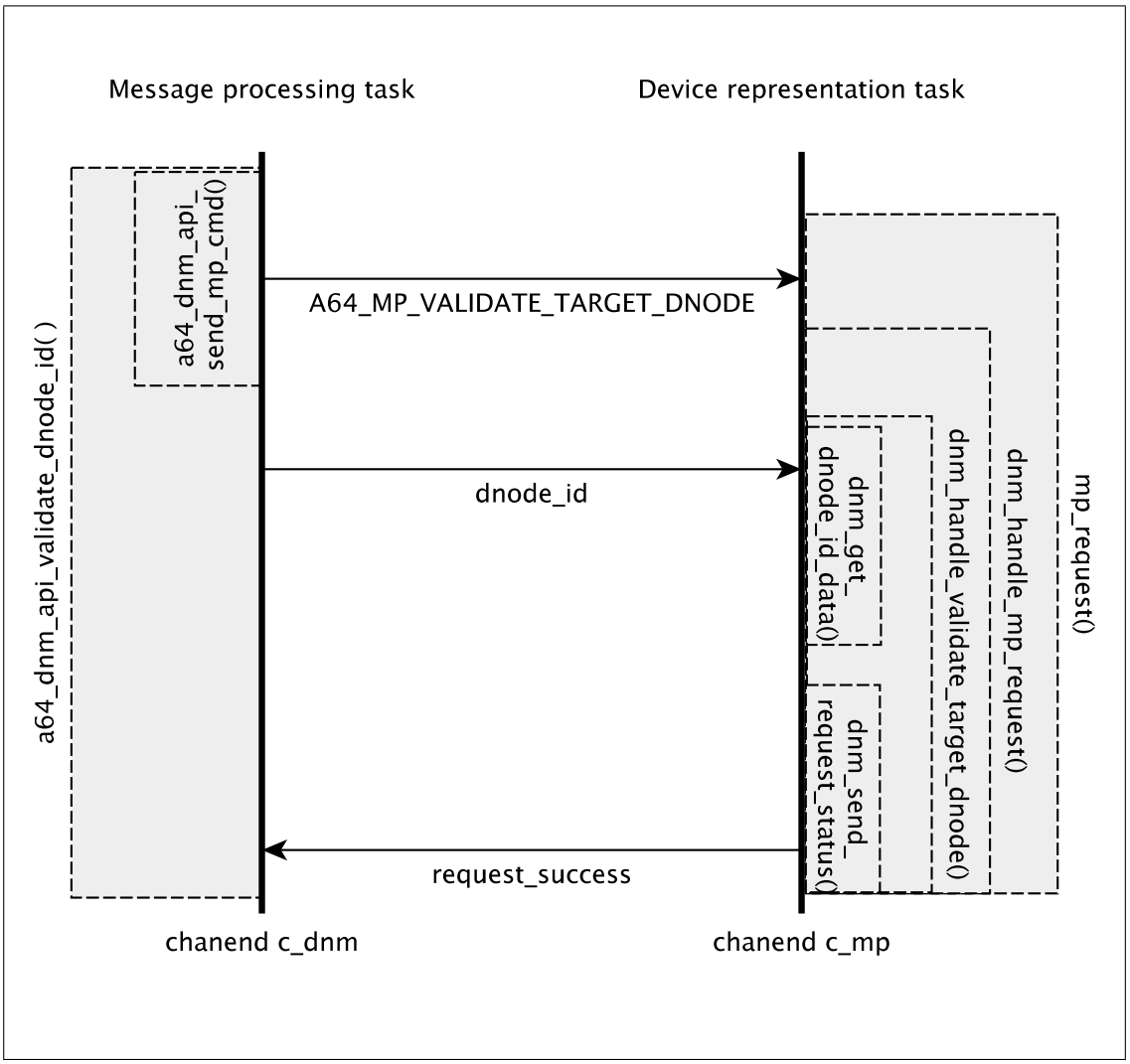


Figure 4.2: Channel communications between the message processing task and the device representation task to verify the existence of a device node.

Listing 4.3: Example of XC channel communications to perform an AES-64 query and receive the result

```
UInt8 a64_dnm_api_validate_dnode_id(chanend c_dnm, UInt32 dnode_id)
{
        UInt8 request_success = A64_DNM_REQUEST_NOK;
        a64_dnm_api_send_mp_cmd(c_dnm, A64_MP_VALIDATE_TARGET_DNODE);
        c_dnm <: dnode_id;
        c_dnm :> request_success;
        return request_success;
}
```

(`c_dnm :> request_success`) until the device representation task returns the result of the query: `A64_DNM_REQUEST_OK` if the specified ID is valid, `A64_DNM_REQUEST_NOK` if it is invalid.

Every channel connects precisely two tasks. For every output operation one task makes to a channel, the other task must make a corresponding input. The device representation task's corresponding communication functions — one to input the device node ID, one to *output* the validation status — are shown in Listing 4.4. (Processing and validation of the device node ID is not shown.)

Listing 4.4: Example of XC channel communications to receive an AES-64 query

```
void dnm_get_dnode_id_data(UInt32 &dnode_id, chanend c_mp)
{
        c_mp :> dnode_id;
}

void dnm_send_request_status(UInt8 status, chanend c_mp)
{
        c_mp <: status;
}
```

Channel inputs and outputs are type-sensitive, so it would be invalid for `dnm_get_dnode_id_data()` to input the 32-bit device node ID into a `UInt8` variable, just as it would be invalid for `a64_dnm_api_validate_dnode_id` to input `request_success` into a `UInt32` variable. The XS1 instruction set traps invalid channel input / output operations at runtime.

As seen in the PTP server's select statement, arrays of channels may be used to allow one task to communicate equally with multiple tasks that are functionally equivalent (e.g., multiple instances of the same process).

4.4.2 Channel transactions

Conventionally, each process involved in communication over a channel must synchronise, handshaking before and after each input-output to ensure the channel buffers are free (if the buffers are not free, the processor traps). However, for extended communications this behaviour adds a considerable overhead and so it can be temporarily suspended through the use of an XC *transaction*.

When processes communicate using transactions, one process must enclose its input-output operations in a *master* block and the other in a *slave* block. Once both processes have entered their respective blocks, they handshake once and then commence transmission. If either process becomes temporarily incapable of performing its input or output (either because the channel buffer is empty and there is nothing for the process to input, or because the channel buffer is full and there is nowhere for the process to output to), it will block until it becomes capable again. Once all of the data has been communicated, the processes will handshake again.

This provides a more efficient way to communicate large quantities of data over channels. In Listing 4.5, the two functions use an XC transaction to communicate a 32-bit device node ID and a 104-bit AES-64 'full address block'.

The `a64_dnm_api_resolve_fab()` function executes on the message processing task. It initiates a channel transaction and then sends a device node ID and a full address block for validation, before closing the transaction. Eventually, the device representation task should respond with information about whether the ID and full address block validated, and if so, which parameters were located. The `a64_dnm_api_resolve_fab()` function will wait on input from the channel to retrieve the validation status and, if validation was successful, details of the parameters that have been selected. The function will return these values to its caller.

The `dnm_get_fab_data()` function executes on the device representation task. It accepts the device node ID and full address block sent by `a64_dnm_api_resolve_fab()`, and returns the received values to its caller, a function that processes and validates the data before eventually sending a response.

Listing 4.5: Use of an XC transaction to pass an AES-64 full address block over a channel

```
UInt8 a64_dnm_api_resolve_fab(chanend c_dnm, UInt32 dnode_id,
  UInt8 sb, UInt8 st, UInt32 sn, UInt8 pb, UInt32 pbi, UInt32 pt,
  UInt32 pi, UInt32 &size_param_select_bitmap,
  UInt32 param_select_bitmap[], UInt32 &located_param_count)
{
        UInt8 request_success = A64_DNM_REQUEST_NOK;
        a64_dnm_api_send_mp_cmd(c_dnm, A64_MP_RESOLVE_FAB);
        master {
                c_dnm <: dnode_id;
                c_dnm <: sb;
                c_dnm <: st;
                c_dnm <: sn;
                c_dnm <: pb;
                c_dnm <: pbi;
                c_dnm <: pt;
                c_dnm <: pi;
        }
        c_dnm :> request_success;
        if (request_success) {
                slave {
                        c_dnm :> located_param_count;
                        c_dnm :> size_param_select_bitmap;
                        for (int i = 0; i < size_param_select_bitmap; i
                            ++)
                                c_dnm :> param_select_bitmap[i];
                }
        }
        return request_success;
}

void dnm_get_fab_data(UInt32 &dnode_id, UInt8 &sb, UInt8 &st, UInt32 &sn,
  UInt8 &pb, UInt32 &pbi, UInt32 &pt, UInt32 &pi, chanend c_mp)
{
        slave {
                c_mp :> dnode_id;
                c_mp :> sb;
                c_mp :> st;
                c_mp :> sn;
                c_mp :> pb;
                c_mp :> pbi;
                c_mp :> pt;
                c_mp :> pi;
        }
}
```

4.5 XC and C

Many features of XC, including the `select` statement, use of channels, and input-output operators previously discussed in this chapter, are reminiscent of the Occam language developed for use with the Inmos Transputer [Pountain and May 1988]. However, in most other respects XC strongly resembles C. This section summarises the most relevant distinctions between the two languages. [XMOS 2011a] provides clear and complete indications of differences between the two languages.

4.5.1 Reinterpretation

Reinterpretation causes a variable to be treated as a variable of a different type without converting the original variable. In Listing 4.6, the `transmitMsg` function transmits the character array `msg` as an array of 32-bit integers, making more efficient use of the width of the chanend data buffers. [XMOS 2011a, 14].

Listing 4.6: Reinterpretation

```
1 void transmitMsg(char msg[], int nwords) {
2   for (int i = 0; i < nwords; i++)
3     transmitInt( (msg, int[])[i] );
4 }
```

On line 3, the array of char `msg` is reinterpreted as an array of type `int` (as specified by the `int[]` expression). The `i`th element of the array of int is then transmitted by `transmitInt()`.

4.5.2 Pointers and passing by reference

XC does not support pointers. As a result, XC also implements a different approach to passing function arguments by reference (Listing 4.7, [XMOS 2011a, 12]), compared to the same code in C (Listing 4.8).

This can be particularly important when calling a C function from an XC function, or vice versa (see the next section for more on this topic).

In XC as in C, array arguments are implicitly passed by reference.

Calling a function with two references to the same object (e.g., `swap(a, a);`) is invalid.

Function prototypes may specify one or more pass by reference arguments as *nullable*, meaning that a `null` reference can be supplied for that argument by a caller. XC provides an `isnull` operator to test such arguments.

Listing 4.7: Passing arguments by reference in XC

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Listing 4.8: Passing arguments by reference in C

```
void swap(int *x, int *y) {     /* see K&R, p80 */
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```

4.6 Integration of C modules in XC programs

The XS1 architecture supports the execution of software modules developed in C, and these modules may be integrated as components of a larger multicore program. Software modules written in C are not subject to the same restrictions that XC modules are: for example, C modules may use pointers freely.

Functions implemented in C may be called from functions implemented in XC (and vice versa). There are minor compatibility issues to navigate, such as the pointer issue identified in the previous section. C function prototypes declare a pass by reference parameter as `type *name`, while XC declares a pass by reference parameter as `type &name`. XMOS provide a `REFERENCE_PARAM(type, name)` macro expansion that conditionally substitutes the appropriate form depending on the language of the implementation file that has included the prototype.

An example of this integration can be taken from the research project's implementation of an AES-64 stack, once again looking at the 'device representation' task (Fig. 6.1). Subsection 4.4.1 of this chapter showed that channel communications for a 'validate

device node' query are implemented by the 'message processing' task and the 'device representation' task as XC functions. This section will show how the XC functions belonging to the 'device representation' task integrate with a C software module.

The 'device representation' task itself is an XC function. On startup (not shown), it is configured to store a device representation consisting of device nodes, parameters and level hierarchies. Once the configuration is loaded, it registers its interest, through a `select` statement (Listing 4.9), in handling command or query events from 'message processing' tasks. These events will be handled by the `mp_request` function (Listing 4.10).

Listing 4.9: The device representation thread waits for events

```
while (1) {
        enum A64MPCommand mp_cmd = A64_MP_INVALID_COMMAND;
        select
        {
        case (int mp_uid = 0; mp_uid < num_mp_threads; mp_uid++)
                mp_request(c_msg_proc[mp_uid], token, mp_cmd);
        }
}
```

The `mp_request()` XC function is another example of a replicated parameterised select function (Subsection 4.3.3), and iterates (l. `5`) over the `c_msg_proc` array of channels connecting the 'device representation' task to the 'message processing' tasks[6], of which there is at least one. The purpose of the `mp_request()` function is to validate that a command sent by a message processing thread[7] falls within a valid range (line 11). It then attempts to locate a handler for the command by calling the C function `dnm_handle_mp_request()`.

[6] As described in Chapter 7, the research project AES-64 implementation supports multiple 'message processing' processes.

[7] Lines 3 — 10 constitute the 'other side' of the conversation initiated by the `a64_dnm_api_send_mp_cmd()` call on line 4 of Listing 4.3.

Listing 4.10: The device representation task receives a request from the message processing task and validates the request's command code

```
select mp_request(chanend c_mp, unsigned char token, enum A64MPCommand
    mp_cmd)
{
case inct_byref(c_mp, token):   /* inputs a command token from a channel
    */
        if (token == A64_CMD_TOKEN) {
            /* handshake to ensure channel is empty before data
                transmission */
              outct(c_mp, XS1_CT_END);
              outct(c_mp, XS1_CT_END);
              mp_cmd = inuint(c_mp); /* inputs the command code, e.g.,
                  A64_MP_VALIDATE_TARGET_DNODE */
              /* handshake to ensure channel is empty after data
                  transmission */
              chkct(c_mp, XS1_CT_END);
              outct(c_mp, XS1_CT_END);
        }
        /* validate command code falls within acceptable range */
        if ((mp_cmd <= A64_MP_READ_DNODE_PARAM_COUNT) &&
                        (mp_cmd >= A64_MP_RESOLVE_FAB))
              dnm_handle_mp_request(c_mp, mp_cmd);
        else
              printstrln("invalid␣MP␣command␣sent␣to␣DNM");
        break;  /* select case break */
}
```

The `dnm_handle_mp_request()` function identifies the value of the command code sent by the message processing task (Listing 4.11, l. 5) and calls the corresponding handler.

Listing 4.11: The device representation thread locates the correct command handler

```
void dnm_handle_mp_request(chanend c_mp, enum A64MPCommand cmd)
{
        switch(cmd) {
        /* handlers for various commands */
        case A64_MP_VALIDATE_TARGET_DNODE: /* 'validate target device
            node' */
                dnm_handle_validate_target_dnode(c_mp);
                break;
        /* handlers for various other commands */
        default:
                break;
        }
}
```

The command handler for validating a device node ID is shown in Listing 4.12. This function implements a response to the query first seen in Listing 4.3, and on line 5 it calls the `dnm_get_dnode_id_data()` function first seen in Listing 4.4.

Listing 4.12: The command handler for validating a device node ID

```
void dnm_handle_validate_target_dnode(chanend c_mp)
{
        struct DNode *target_dnode = NULL;
        UInt32 dnode_id = 0, retval = 0;
        dnm_get_dnode_id_data(&dnode_id, c_mp);   /* call to XC function,
                                passing dnode_id by reference */
    /* call locator function using value retrieved from channel */
        retval = locate_this_dnode(dnode_id, &target_dnode);
        /* send status to MP thread */
        if (retval == -1) { /* dnode not located */
                dnm_send_request_status(A64_DNM_REQUEST_NOK, c_mp);
                return;
        }
        dnm_send_request_status(A64_DNM_REQUEST_OK, c_mp);
}
```

The prototype (Listing 4.13) for the `dnm_get_dnode_id_data()` function illustrates use of the `REFERENCE_PARAM()` macro. This prototype needs to be included from both C and XC implementation files, but valid syntax for an XC pass-by-reference argument is invalid syntax for a C pass-by-reference argument (and vice versa).

Finally, the `dnm_send_request_status()` function performs channel communications and is implemented as an XC function. It was shown in Listing 4.4.

Listing 4.13: The function prototype for dnm_get_dnode_id_data

```
void dnm_get_dnode_id_data(
            REFERENCE_PARAM(UInt32, dnode_id),
            chanend c_mp);
```

4.7 Conclusion

The XMOS XS1 architecture provides a clear and flexible approach to the construction of concurrent software applications that implement real-time software systems. The architecture supports the implementation of processes that can handle several simultaneous real-time tasks in a way that allows performance of a given software process to be guaranteed against external timing or throughput requirements.

The XC programming language provides powerful constructs for the control of XS1 architectural features. Support for message-passing between concurrent processes is efficient and flexible. These techniques have been used extensively in the implementation of interfaces between the processes that implement the AES-64 protocol stack (Chapters 5 and 6).

These characteristics are central to the evaluation of the XMOS XS1 architecture as a platform for development of audio control protocols and are discussed in qualitative terms in Chapter 7.

5 Design

5.1 Introduction

Chapter 2 introduced audio networking applications and technology, and discussed the XMOS Ethernet AVB reference design as an implementation of an audio networking device. Chapter 3 described the Audio Engineering Society's AES-64 standard for control and monitoring of audio networking systems. Chapter 4 discussed the XMOS XS1 platform and illustrated some basic techniques for constructing concurrent applications in the XC programming language.

This chapter describes the process of designing an implementation of the AES-64 standard for the XMOS XS1 platform, with the goal of integrating AES-64 control into the XMOS Ethernet AVB reference design. This chapter covers:

1. requirements capture and analysis,
2. the design methods used to develop a specification (or 'essential model') of the AES-64 protocol stack, and
3. the transfer of the essential model to an implementation model.

The primary objective of this research, as described in Section 1.2, was to investigate the capabilities and limitations of the XMOS XS1 architecture by producting an implementation of the AES-64 standard. The first part of this chapter, Section 5.2, covers the requirements analysis and construction of a requirements specification (Appendix A) for the AES-64 standard.

The second part of this chapter, from Section 5.3 onwards, discusses how the primary research objective guided the selection of a design methodology, and then describes the application of the chosen methodology to produce a structured design (Appendix B) for the AES-64 implementation.

5.2 Requirements capture and analysis

The Audio Engineering Society standard for AES-64 'specifies an approach to the control and monitoring of professional audio devices' Audio Engineering Society [2012]. It outlines a messaging protocol, a model for representation of audio devices of arbitrary complexity, and a set of high-level control mechanisms.

In places, the published standard provides insufficient information to guide an informed design process. For example, the standard states that

> the nature of [an AES-64] command [is] specified in the Command Executive and Command Qualifier fields [of an AES-64 message]'[Audio Engineering Society 2012, 15].

In other words, the nature of an AES-64 request is jointly specified by the value of the Command Executive and the Command Qualifier fields. However, while the AES-64 specification defines the valid identifiers for the Command Executive field [Audio Engineering Society 2012, 20] and Command Qualifier field [Audio Engineering Society 2012, 20] of an AES-64 command, it does not define:

- what *combinations* of Command Executive and Command Qualifier constitute valid requests
  (e.g., 'SET VAL' is a valid request, but 'JOIN VAL' or 'SET MSTSLV' are not.)
- the meaning of each valid request
- details of any data associated with each valid request
- details of the response(s) that a recipient of the valid request is expected to make

In other cases, information was provided by the published standard, but not in explicit terms: for example, the messaging pattern 'variations' described in Chapter 3 (Fig. A.4, A.5, and A.6) are not explicitly discussed by the standard, but implied in its descriptions of the wildcard mechanism [Audio Engineering Society 2012, 16] and the grouping mechanism [Audio Engineering Society 2012, 31-35].

Several techniques were used to perform a complete requirements capture and analysis process for the AES-64 stack implementation. These included:

1. forensic capture of communications between existing implementations of AES-64;
2. code review of an application that uses an existing implementation of AES-64;

3. close reading of the AES-64 standard.

Each of these techniques is discussed in the following section.

5.2.1 Forensic capture of existing AES-64 implementations

Foulkes [2011] describes the implementation of AES-64 connection management between two Ethernet AVB systems:

> One system acts as an Ethernet AVB audio endpoint device and another system acts as an audio gateway between IEEE 1394 and Ethernet AVB networks. These systems, along with existing IEEE 1394 audio devices, were used to demonstrate the ability to transfer audio data between the networking technologies. Each of the devices is remotely controllable via a network neutral command and control protocol, XFN[1].

Foulkes' implementations of Ethernet AVB audio devices and a connection manager application were available for investigation, and enabled forensic capture of network discovery and device enumeration procedures, among others. An example of forensic capture will be provided in the context of the 'GET CLA' command.

Forensic capture of the 'GET CLA' command

[Audio Engineering Society 2012, 24] indicates that an AES-64 controller may discover devices available on the network by sending a broadcast 'GET VAL' request (as described in Subsection 3.6.1); however, it does *not* describe the process by which the controller then enumerates each device's level hierarchy (that is, the structural organisation of a device's control and monitoring parameters).

Forensic capture of Foulkes' application and devices revealed that this device enumeration can be performed through a series of 'GET CLA' requests targeted to each device.

The 'CLA' Command Qualifier is described in the AES-64 standard as:

> Refers to child level aliases, which provide meaningful names to the various levels in the parameter hierarchy. Associated with every level in the AES64 level hierarchy is an alias string. These aliases are set when the levels are created at initialization time. The CLA qualifier shall be used with a GET command executive to get all of the child level aliases for a specified parent level. [Audio Engineering Society 2012, 21]

[1] At the time of publication, the AES-64 standard had the working name 'XFN'.

This description omits several significant pieces of information, specifically that:

1. the 'GET CLA' request specifies the 'parent' level by supplying a 'full address block';
2. the 'GET CLA' response provides the number of 'child' levels, and for each child level it provides:
   a) a 'full address block' locating the child level;
   b) a 'level alias'.

Consequently, the 'GET CLA' request may be used to discover the *number* of child levels for a given level and their locations, as well as their aliases. The following example will show the use of a 'GET CLA' request to enumerate part of the level hierarchy on the 'device configuration node' (Fig. 3.3).

Listing 5.1 shows forensic capture of an AES-64 'GET CLA' request. The two-octet block `00 2a` is formed of the Command Executive (`00`, 'GET') and Qualifier (`2a`, 'CLA') fields. The 13-octet block `f1 d6 00 00 01 d6 00 00 01 ee ee ee ee` shows the request's full address block. Forensic capture reveals that this full address block uses identifiers set to `0xEE`, `0xEEEE`, or `0xEEEEEE` values within the full address block to indicate the 'parent level'[2].

Listing 5.1: A Wireshark capture of an example 'GET CLA' request

```
                                        a9 fe e0 6a 00 00                ...j..
00 00 a9 fe 4e a5 00 00  00 00 00 00 00 01 00 00    ....N...  ........
00 00 00 02 00 2a f1 d6  00 00 01 d6 00 00 01 ee    .....*..  ........
ee ee ee 03 00 00 12 34                             .......4
```

The full address block supplied in the 'GET CLA' request is:

1. 'Section Block' (`0xf1, DEVICE INFO CONFIG`)
2. 'Section Type' (`0xd6, DEVICE`)
3. 'Section Number' (`0x000001`)
4. 'Parameter Block' (`0xd6, CONFIG`)
5. 'Parameter Block Index' (`0x000001`)

[2] As described in Chapter 3, AES-64 level identifiers are either 1, 2, or 3 octets in size. Conventional requests use `0xFF`, `0xFFFF`, or `0xFFFFFF` values to indicate a wildcarded level.

6. 'Parameter Type' (`0xEEEE`)

7. 'Parameter Index' (`0xEEEE`)

The device will parse this full address block to determine that the parent level is the Parameter Block Index '0x000001' level. The device's response to this request is shown in Listing 5.2.

Listing 5.2: A Wireshark capture of the response to the 'GET CLA' request shown in Listing 5.1

```
                                        a9 fe 4e a5 00 00                ..N...
00 00 a9 fe e0 6a 00 00  00 00 ff ff ff ff 00 04   .....j.. ........
00 00 00 02 03 00 00 00  02 00 00 00 18 f1 d6 00   ........ ........
00 01 d6 00 00 01 0f 07  ee ee 64 65 76 69 63 65   ........ ..device
20 6e 61 6d 65 00 00 00  18 f1 d6 00 00 01 d6 00    name... ........
00 01 b0 08 ee ee 64 65  76 69 63 65 20 74 79 70   ......de vice typ
65                                                 e
```

The value field of the 'GET CLA' response shows:

1. `0x00000002`: the number of child levels belonging to the Parameter Block Index '0x000001' level;

2. `0x00000018`: the length in octets of the first entry (the combined length of the first entry's full address block and level alias);

3. `0xf1 d6 000001 d6 000001 0f07 eeee`: the first entry's full address block, specifying its (hitherto unknown) Parameter Type identifier as `0x0f07` ('Device Name');

4. `device name`: the first entry's level alias;

5. `0x00000018`: the length in octets of the second entry;

6. `0xf1 d6 000001 d6 000001 b008 eeee`: the second entry's full address block, specifying its (hitherto unknown) Parameter Type identifier as `0xb008` ('Device Type');

7. `device type`: the second entry's level alias.

This may be compared to the level hierarchy diagram shown in Fig. 3.3.

5.2.2 Review of existing AES-64 applications

Foulkes' implementation of an Ethernet AVB system [2011] was available for investigation. The documentation of Foulkes' Ethernet AVB system clarified the role of 'value functions'

(Subsection 3.4.1) in linking AES-64 parameters and device features[3].

Value functions

[Foulkes 2011, 317] describes the implementation of 'value functions' for AES-64 parameters that represent Ethernet AVB Talker and Listener stream connection management:

> Performing a *set value* request on an advertise parameter with a value of *true* causes the device's `AVBDevice_registerAVTPStream` function to be called for the addressed AVB output stream. This causes the particular stream to be advertised to the attached AVB network.
>
> Performing a *set value* request on an advertise parameter with a value of *false* causes the device's `AVBDevice_deregisterAVTPStream` function to be called for the addressed AVB output stream. This causes the particular stream's advertisement to be withdrawn from the AVB network.
>
> Performing a *set value* request on a listen parameter with a value of *true* causes the device's `AVBDevice_registerAVTPStreamAttach` function to be called for the addressed AVB input stream. This causes a listener ready attribute to be declared for the stream (as identified by the value of the input's stream ID parameter).
>
> Performing a *set value* request on the listen parameter with a value of *false* causes the device's `AVBDevice_deregisterAVTPStreamAttach` function to be called for the addressed AVB input stream. This causes the particular stream's listener attribute to be withdrawn.

This clarifies the interaction between AES-64 parameters and the device functions they represent. All parameters must implement a *get* value function which retrieves the current status of the device function; a control parameter must additionally implement a *set* value function, where a monitoring parameter does not need to.

Value field composition

As described in the first part of this section, the AES-64 standard does not specify the valid requests that may be formed from its specification of Command Executive and

[3] While discussed in a draft informative document [Audio Engineering Society 2011, 7-10], this is not discussed in the published AES-64 standard.

Command Qualifier identifiers; neither does it specify the data *associated* with any such request or response.

In an AES-64 request or response message, this data is held in the *value* field. In some cases, the *value format* field specifies the composition of the data in the value field: for example, the response to a 'GET VAL' request on a 'stream ID' parameter would have a value format specifying a 64-bit integer and a value field containing a stream ID (e.g., `00-22-97-A3-C0-3F-00-00`). In other cases the value field will hold structured data, including lists; this is particularly true of requests and responses related to parameter grouping.

In the absence of specifications, forensic capture of AES-64 operations was sufficient to specify the data associated with AES-64 requests and responses. These specifications are provided in Section A.8.

5.2.3 Close reading of the AES-64 standard

Some areas of AES-64 functionality are not clearly defined by the standard: for example, the messaging patterns that an AES-64 protocol stack is expected to implement. The standard's description of AES-64 messaging is based on the following statements:

> Certain AES64 messages will be requests, and others will be responses. [Audio Engineering Society 2012, 10]

> Every AES64 message will either be a request sent from one device to another, or will be a response from a device after a request has been made. Some message requests require a response, while others do not.
> [Audio Engineering Society 2012, 17]

While both of these statements are accurate, they lend the impression that an AES-64 protocol stack can service any and all situations by parsing requests and sending responses when required to. As discussed in Chapter 3, an AES-64 protocol stack must implement additional messaging patterns, including 'receive one request, issue many responses', 'receive one request, issue further requests', and the still more advanced messaging sequences initiated by commands such as 'JOIN MSTSLV'. These requirements were uncovered through close reading of the AES-64 specification.

While these patterns are made up of requests and responses as described in the quotation above, the patterns are significant because they define what is required of a protocol stack in order for it to process *any single request*.

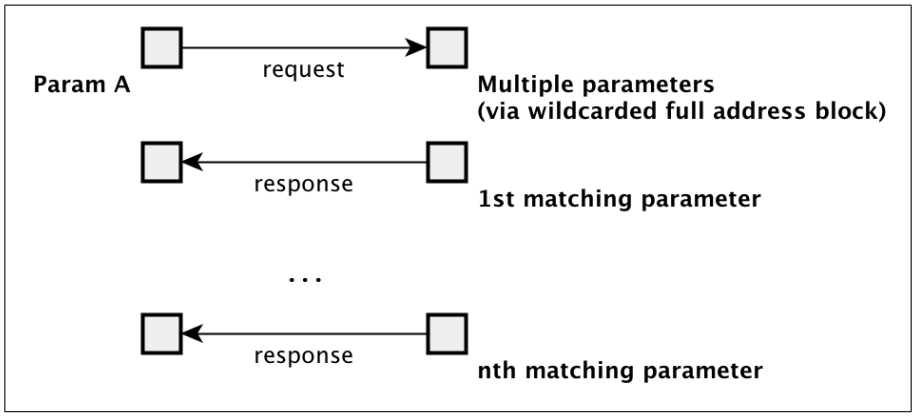


Figure 5.1: The 'receive one request, issue many responses' messaging pattern.

For example, the 'receive one request, issue many responses' pattern (Fig. 5.1) was uncovered by close reading of the standard's discussion of the 'wildcard' mechanism [Audio Engineering Society 2012, 16]. A response message can only be sent from a single parameter [2012, 18], and so a 'GET VAL' request that addresses multiple device parameters will necessarily prompt multiple responses. All of these responses must be dispatched before the protocol stack can complete the initiating request and process another, so that the actions required of the protocol stack are:

1. Receive and parse 'GET VAL' request with a wildcarded full address block
2. For each parameter addressed by the wildcarded full address block, send a response message

Close reading of the standard's account of grouping mechanisms [Audio Engineering Society 2012, 32] indicates two further messaging patterns.

The 'receive one request, issue further requests' messaging pattern (shown in Fig. 5.2) performs value updates between grouped parameters, and is described by the standard as follows:

> When the value of a parameter is initially changed, the change is typically performed via a 'SET VAL' command. Following this change, the parameter's peer-to-peer and slave group lists shall be scanned, and a 'SET GRPVAL' command sent to each member of the group [Ibid.].

In Fig. 5.2, the 'SET VAL' message is the first request sent from Param A to Param B; the following two requests from Param B to Param C and Param D are 'SET GRPVAL' requests, advising each (as peers of Param B) that Param B's value has been changed.

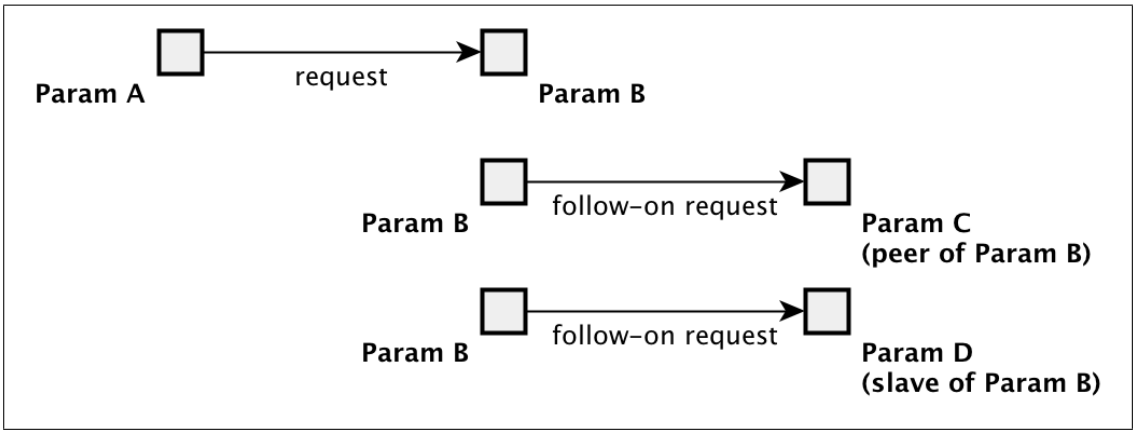


Figure 5.2: The 'receive one request, issue further requests' messaging pattern used to perform group value updates.

The more complex pattern (Fig. 5.3) is outlined by the AES-64 standard's account of parameter group setup [Audio Engineering Society 2012, 32-35], where the AES-64 protocol stack must receive and parse an initiating request (e.g., 'JOIN MSTSLV'), issue a follow-on request ('GET PTPGRP') that requires a response, and receive and parse the response before issuing further follow-on requests (e.g., 'SET MASTER'), before finally issuing a response to the initiating request.
In Fig. 5.3, the first request from Param A to Param B is the 'JOIN MSTSLV' request. This request supplies Param B with the details of a nominated slave parameter (Param C). Before Param B can add Param C as a slave, it must first find out whether Param C has any existing peer relationships: it sends the first follow-on request ('GET PTPGRP') to Param C, which will then return a response to Param B.

Once Param B has received the response from Param C, which in this example supplies the details of Params D & E, it will add Params C, D and E to its list of slaves. Accordingly, it will also inform Params C, D and E that it, Param B, is now their master.

Further discussion of these messaging patterns in the context of the AES-64 protocol stack is provided in the AES-64 requirements specification, Subsection A.6.

5.2.4 The AES-64 requirements document

As indicated in Section 5.2, no comprehensive requirements document for an AES-64 protocol stack was available at the outset of the design process, so it was necessary to develop one. Information for this document was gathered through the procedures outlined above.

The AES-64 protocol stack requirements document is included as Appendix A, and

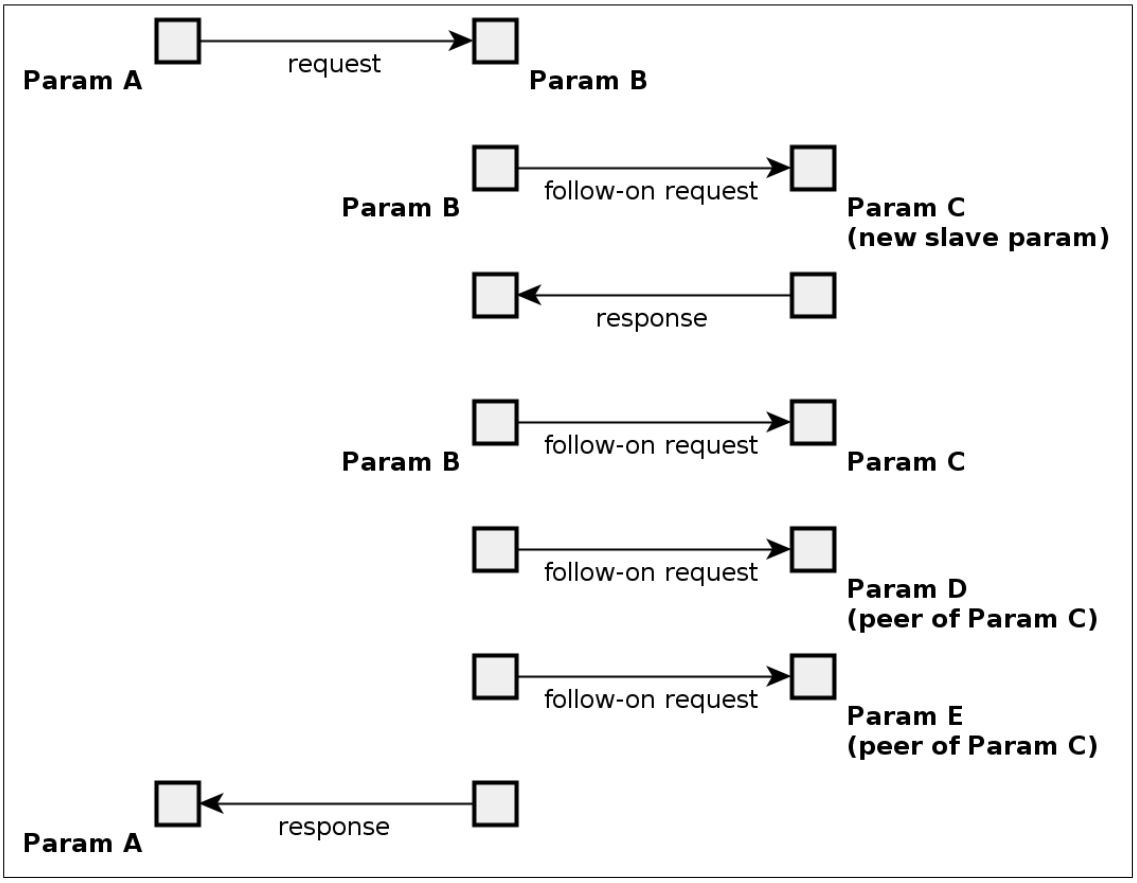


Figure 5.3: An example of the most complex AES-64 messaging pattern, outlining setup of a master-slave parameter group.

provides a detailed account of the AES-64 protocol. Section A.5 specifies the data structures required to implement the AES-64 protocol stack. Section A.6 specifies the messaging patterns required to implement AES-64 messaging in line with the published standard. Section A.7 defines the AES-64 message formats, including the message structures for every command message defined by the standard and their associated responses (Section A.8). Finally, Section A.9 describes the extended messaging interactions defined by the standard.

5.3 Design

The primary research objective of developing an AES-64 protocol stack was to investigate the capabilities and limitations of the XMOS XS1 architecture as a platform for research and development of audio control standards, using the AES-64 audio control standard as a basis for the investigation.

Chapter 4 provided a conceptual overview of the XMOS XS1 architecture, focusing on the XC language's features for concurrency and communications and the 'event-driven' approach, where XC software tasks may be defined in terms of 'events' they may react to, and described how this provides a basis for implementing concurrent software systems as networks of communicating processes.

It followed that an ideal design methodology for the AES-64 stack implementation would be one that was highly congruent to XC language concepts, providing a direct correspondence between *design language* and *implementation language*. This was identified as the primary criterion for the selection of a design method.

The next two subsections describe the selection of a design methodology and introduce the selected method. The remainder of this chapter provides an account of the design process itself.

5.3.1 Selection of a design method

In this case, the execution platform for the AES-64 implementation had been determined before the design process began. The XMOS XS1 architecture had enabled an open-source Ethernet AVB implementation. The availability of this implementation, and the opportunity to integrate a high-level control system *with* it, provided the core motivation for using this platform. Consequently, congruence to the XS1 architecture, specifically the concurrency and communications concepts implemented in the XC language, formed a primary criterion for the selection of a design method. The more direct the correspondence between the design language and implementation language,

- the fewer implementation details intrude into the design process;
- the clearer the design documents can express application requirements;
- the more straightforward it is to verify and debug an implementation against the design documents.

The XMOS XS1 architecture has also influenced the selection of a design method in that the XC language does not provide explicit support for the object-oriented paradigm.

A secondary criterion is provided by the fact that the function of the AES-64 implementation is to provide connection management and control for an audio distribution system. Subsection 2.1.3 described the Audio Engineering Society's definition of quality of service for audio distribution, which includes maximum acceptable latency values for various audio distribution applications. It follows that an effective control system for an audio distribution application must not exceed the application's own allowable latency. Consequently, the AES-64 implementation can be characterised as a 'soft real-time' application.

Thirdly, the requirements analysis process had demonstrated that an AES-64 protocol stack would have to fulfil three central requirements: a flexible framework for device representation, a comprehensive message parser, and the implementation of many command handlers and value functions[4]. Additionally, requirements analysis indicated that some command handlers (e.g., parameter grouping set up and teardown) would need to participate in extended inter-device communications.

This preliminary analysis of an AES-64 protocol stack strongly suggested that the ideal implementation would be composed of multiple concurrent tasks. The ideal design method would support the specification of a system in these terms.

[Williams 2005, 3] observes,

> Perhaps surprisingly, suitable [design methods] for real-time systems design are not very numerous ...

before listing the following:

> Structured Analysis/Structured Design (SA/SD), Concurrent Design Approach for Real-time Systems (CODARTS), Finite State Methods (FSM), and Object-Oriented Design (OOD)

[4] A 'command handler' implements the processing required for a single type of AES-64 command: e.g., 'JOIN MSTSLV'. A 'value function', first introduced in Subsection 3.4.1, implements a link between a single type of AES-64 parameter and the corresponding feature of the device's software.

Of these, SA/SD is a data-driven design methodology that has been extended (as 'Real-Time Structured Analysis and Design, or RTSAD) to provide support for modelling real-time systems [Ward and Mellor 1985]; CODARTS is one of several design approaches that aim to '[extend] the SA/SD approach by providing an approach for structuring the system into tasks as well as a mechanism for defining the interfaces between tasks' [Gomaa 1984, 940]; FSM are a set of design tools that can be used within 'both structured analysis/design and object-oriented design/programming methodologies' [Williams 2005, 97]; and object-oriented design produces a succession of abstract models that reason about the representation of system data in preference to the functional decomposition of a system's intended behaviour.

There is a body of literature that describes object-oriented design approaches for real-time systems: [Selic et al. 1994] provides a book-length tutorial with many examples, and [Williams 2005] provides a general overview of the method. However, an object-oriented approach did not seem to be a good fit for this project: the XC language does not implement OO concepts, and consequently the correspondence between the design language and the implementation language would be quite indirect. One of the object-oriented approach's key advantages is the definition and production of reusable software components: in the context of an AES-64 protocol stack, however, reusability was not a priority. Likewise, an object-oriented approach appeared to be of limited application to the design of a message parser or command handler, or to the integration of the protocol stack overall within the broader Ethernet AVB application.

A large body of literature describes and discusses the SA/SD and RTSAD approaches in the context of real-time systems, including [Gomaa 1984], [Ward and Mellor 1985], [Yourdon 1989], and [Williams 2005][5]. The SA/SD approach performs a top-down decomposition centred on the transformation of data. The primary innovation of the RTSAD approach described in [Ward and Mellor 1985] is the replacement of 'data-flow diagrams' with 'transformation schema', where the latter include 'control transformations' and 'events' to express real-time control and scheduling.

One commonly-cited disadvantage to the RTSAD method is that 'requirements and design changes tend to have a more severe effect on projects using this method' [Kelly 1987, 249]; Gomaa [1989, 22] attributes this to '[use of] this method [is] liable to arrive at a design that is mainly functional'. In the case of the AES-64 protocol stack, the requirements analysis process had already provided a detailed requirements document and it was felt that this was unlikely to change.

[5] More recently, [Fencott 1995] has described the integration of RTSAD and Milner's 'Calculus of Communicating Systems' to build formal specifications of real-time concurrent systems.

Another widely-reported shortcoming in the SA/SD method is the issue of how to partition a design into a set of concurrent tasks. In the case of the AES-64 protocol stack, the requirements analysis process had indicated a preliminary partitioning of the system into three types of software task:

Device representation. Maintaining and providing access to the device's AES-64 representation;

Message parsing and command handling. Processing individual AES-64 requests addressed to the device;

Stack control. Miscellaneous coordinating functions, including control over message input-output as necessary to implement the various AES-64 messaging behaviours.

As a result, this perceived shortcoming was not felt to be a major hazard.

Consequently, the real-time structured analysis and design (RTSAD) methodology was selected as the basis for the design of the AES-64 protocol stack. The perceived advantages of the RTSAD method are presented in the following subsection.

5.3.2 Real-Time Structured Analysis and Design (RTSAD)

RTSAD has a number of advantages as a design method for real-time concurrent systems:

Implementation-independent design. RTSAD separates the logical design of a system from the details of its implementation; the *essential model* specifies the behaviour of an ideal system, excluding any consideration of the technology that implements it. Once completed, the essential model can be mapped onto the resources of a target system to develop an *implementation model* to guide the actual implementation.

Modelling of dynamic data flows and control events. RTSAD enables a dynamic viewpoint of a system in terms of a network of functions that communicate data flows and control events. This is a close fit to the event-based execution model promoted by the XC language. XC's capabilities for the communication of data flows and control events have been demonstrated in Section 4.6.

Common programming paradigm. Both RTSAD and XC support the structured programming paradigm.

RTSAD, as advanced by Ward and Mellor [1985], has also been used to explicitly model the implementation of a system as a set of concurrent tasks, distributed between various

hosts or processors, in a way that has clear relevance to the XMOS/XC model of communicating sequential processes. According to Williams,

> Structured Analysis and Design (SA/SD), although increasingly abandoned in favour of Object-oriented Design (OOD) methods, retains popularity in the field of real-time systems to gain improved performance. [Williams 2005, 241]

Additionally, the use of RTSAD allows the project to investigate the application of this established design method to projects based on the XMOS XS1 architecture: since the XS1 architecture may be used 'to make it practical to use software to perform many functions which would normally be done by hardware' [May 2009a, 1], there is some interest in confirming whether a leading design method for such systems may be applied to the XS1 architecture.

Outline of the RTSAD design process

The first phase of the RTSAD process develops an *essential model*, which forms a structured specification of the intended system behaviour. The essential model is prepared through an incremental decomposition of the system into sets of related functions. In traditional SA/SD this decomposition is wholly data-led, but in RTSAD Ward and Mellor's extensions for real-time systems add 'control processes' which schedule data transformations.

The second phase of the RTSAD process prepares a model to guide implementation of the design. Here, the essential model is mapped onto a model of the target development platform, allocating data stores and data transformations to actual software/hardware resources. Finally, the structure and interfaces of data transformations are specified (for example, using structure charts or structured language specifications).

The remainder of this chapter describes the application of this methodology to the design of the AES-64 protocol stack.

5.3.3 The essential model

The 'essential model' describes the essential (or 'implementation-independent') behaviour of the intended system. Ward and Mellor describe this model as consisting of two parts: an *environmental model*, 'which focuses on describing what the system must interact with' [Ward and Mellor 1985a, 36], and a *behavioral model*, 'which describes the required behaviour of the system' [Ibid.]. Each part consists of a number of *artifacts*.

The environmental model consists of two artifacts:

The context diagram. The context diagram provides graphical notation of the boundary that separates the system from its environment, where discrete agents within its environment are depicted as terminators. The context diagram depicts events that cross this boundary, whether environmental inputs to the system or system outputs to the environment.

The event list. The event list enumerates 'the events that occur in the environment to which the system must respond' [Ibid.]. Ward & Mellor define an 'event' (in the context of the environmental model) as having three characteristics:

> it occurs in the system's environment; it elicits a preplanned response from the system; it occurs at a specific point in time. [Ward and Mellor 1985b, 30]

In other words, an event may be a time-discrete data flow (e.g., a UDP message), a time-continuous data flow (e.g., readings from a temperature sensor), or a control message.

The behavioral model describes the required behaviour of the internal system; it consists of three artifacts:

Data specification. The data specification describes the type, meaning and/or composition of stored data and data flows, including whatever data is delivered by the items on the event list. The data specification can simplify the representation of complex composite data types. Data may be specified using any conventional notation. Ward and Mellor propose DeMarco's notation [Ward and Mellor 1985a, 126].

Transformation schemas. A set of transformation schemas provide graphical notation of a system's behaviour, where the system is depicted as a 'network of activities that accept and produce data and control messages' [Ward and Mellor 1985a, 41]. Fig. 5.4 shows an example transformation schema for an activity that records measurements from a temperature sensor.

Each activity should have an associated 'process specification' that describes the transformation(s) it performs on its input data and control messages. Listing 5.3 shows the specification for the example shown in Fig. 5.4.

The complete set of transformation schemas may be re-organised or 'levelled', where related transformations, events and data flows are grouped to reduce the complexity of the system description [Ward and Mellor 1985a, 134-143].

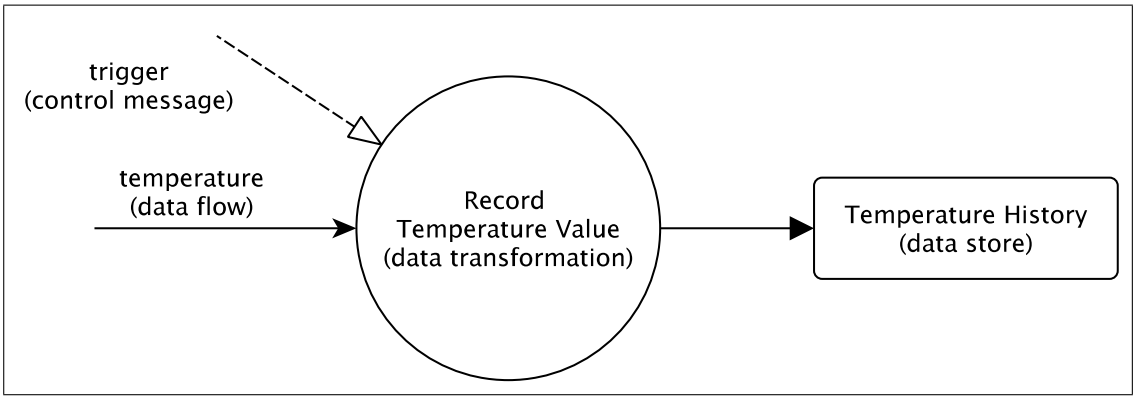


Figure 5.4: An example transformation schema, showing a data transformation and its two inputs, a control message and a data flow, and its output to a data store. Each type of input-output is denoted by a specific arrowhead.

Listing 5.3: Example process specification for Fig. 5.4

```
CASE: trigger
  Read TEMPERATURE VALUE
  Write TEMPERATURE VALUE and TIME to TEMPERATURE HISTORY
```

Process specifications. Each process specification provides a structured language description of an activity (i.e., an activity as depicted on a transformation schema) [Ward and Mellor 1985a, 81]. Listing 5.3 is a very simple example.

The full documentation of the essential model is enclosed as Appendix B. The following subsections describe each component of the model and its overall development.

5.3.4 Building the context diagram and event list

An effective essential model demands a 'precise and formal definition of the system's boundaries' [Ward and Mellor 1985b, 14].

The AES-64 specification defines some of these boundaries: for example, the data flows between the AES-64 protocol stack and the IP stack will be UDP messages. Interactions between the AES-64 protocol stack and the host device are not explicitly discussed in the AES-64 specification, but can be established from considering the context diagram and applying the 'active modelling approach': 'in other words ... we visualise the system in action' [Ward and Mellor 1985b, 36].

The AES-64 protocol stack maintains a logical representation of the device; it follows that this representation is *device-specific*, and therefore must be configured from outside

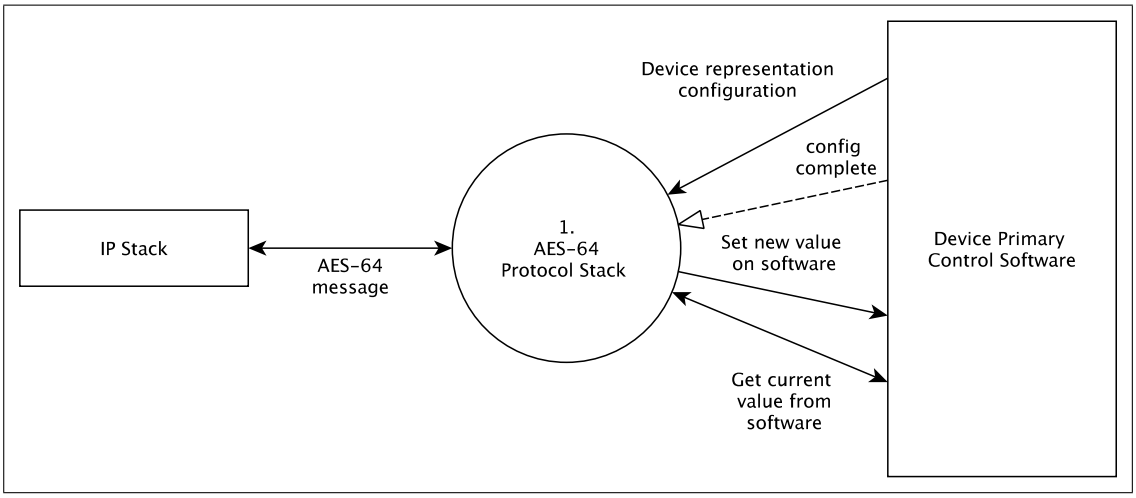


Figure 5.5: The context diagram for the AES-64 protocol stack.

the protocol stack. Furthermore, external AES-64 requests act on the representation of the device, so until this representation is in place the AES-64 protocol stack *cannot process* external requests.

Finally, parameter value changes submitted by AES-64 messages must have some means of influencing the device's operations; conversely, parameter values maintained by the stack must be updated when the corresponding properties of the device are independently altered.

As described in Appendix A Section A.2 – A.4, an AES-64 protocol stack integrates into an audio network device alongside the device's primary control software, which must initialise the device representation loaded on the protocol stack. The device's primary control software also actuates the device in response to AES-64 parameter value changes.

From this, a context diagram (Fig. 5.5) and an event list for the AES-64 protocol stack can be constructed. The AES-64 protocol stack receives one external event from the IP stack:

AES-64 message. This event 'packages' [6] all forms of AES-64 request and response message received by the device. Data transformations internal to the AES-64 protocol stack may easily recognise each form of AES-64 message and act on them accordingly.

The external events received from the device's primary control software (DPCS) are:

[6] 'A single upper-level discrete data flow, continuous data flow or event flow can summarise the details of a lower-level set of flows of the same type.' [Ward and Mellor 1985a, 140]

Device representation configuration. This event 'packages' the registration of successive device nodes, device parameters and level hierarchy entries on the AES-64 protocol stack.

Configuration complete. This event is a control message that the AES-64 protocol stack's device representation has been fully configured, meaning that the stack may now service AES-64 messages from the network. Until this event is received, the protocol stack may not service messages from the IP stack.

In addition to these external events, the context diagram also depicts the following system responses or outputs:

AES-64 message. This 'packages' the transmission of AES-64 response or request messages in the course of processing a previously-received AES-64 request.

Get current value from software. The AES-64 protocol stack retrieves a current value or status from the DPCS in the course of processing a previously-received AES-64 'GET VAL' request.

Set new value on software. The AES-64 protocol stack updates a value and/or control feature on the DPCS, in the course of processing a previously-received AES-64 'SET VAL' or 'SET GRPVAL' request.

In a practical implementation of the software, this set of interactions may be larger and more detailed: but for the purposes of the essential model, 'packaged' interactions such as 'AES-64 message' and 'Device representation configuration' are effective representations. While this context diagram does not adequately describe the requirements of an AES-64 *controller*, it is satisfactory for an AES-64 device.

5.3.5 Building the data specifications

There are three principal sets of data structures that must be specified to enable an implementation of the AES-64 protocol stack:

- the device representation data structures;
- the value field for each AES-64 command and its corresponding response(s);
- auxiliary data structures
  (e.g., headers, lists and entries used in message value fields)

The notation used for specifying these structures follows Ward and Mellor's modifications to DeMarco [Ward and Mellor 1985a, 126]: 'all data compositions can be defined from ... composition, selection and iteration'. Table 5.1 gives a key for the composition notation. Type and meaning specifications are delimited by asterisks.

Table 5.1: Data specification notation

symbol	meaning
=	'is composed of'
+	'together with'
[...]	'select one of'
{ ... }	'repetition of'

Listing 5.4 shows the definition of an AES-64 Device Node and some of its component data structures. Composite data types (e.g., the 'Level Items Table') are defined in terms of their components and their meaning. Integral data types (e.g., the 'Device Node ID') are defined in terms of their type and meaning. The primary source for these definitions is the published AES-64 standard: for instance, the types of each level identifier (Section Block, Parameter Block Index, etc) are shown in [Audio Engineering Society 2012, 50-74].

Listing 5.4: Example of data specification for an AES-64 Device Node

```
Device node =
    Device node ID + Parameter count + {Level hierarchy entry} +
        {Parameter}
    *Container for the protocol stack representation of a functional
        device. The protocol stack may maintain more than one of these.*

Device node ID = *range: unsigned int, 32-bit*
    *Node ID 0 is reserved for device discovery parameters*

Level hierarchy entry =
    Section block + Section block alias + Section type + Section type
        alias + Section number + Parameter block + Parameter block alias
         + Parameter block index + Parameter type + Parameter type alias
         + Parameter index
    *represents the position in the Level Hierarchy of exactly one
        parameter*

Section block = *range: unsigned int, 8-bit*
    *Specification-defined enumerated type.*

Section block alias = *string*
    *Level alias corresponding to a Section Block identifier, e.g.,
        INPUT SIGNAL*
```

The complete data specifications prepared as part of the essential model are included in Appendix B.6.

5.3.6 Building the transformation schemas

The production of transformation schemas is based on decomposition of the data and control flows depicted on the context diagram. The starting point for this process is the event list, and accordingly the complexity of the event list directs the complexity of the transformation schema. The RTSAD modelling process is characterised as 'iterative' [Gomaa 1989] and preliminary attempts may produce very disorganised representations of system behaviour, or reveal crucial oversights in the preparation of the context diagram and event list.

The eventual product, however, is a 'levelled' (i.e., top-down hierarchically organised) set of graphical representations where the activity of the system is graphed as a set of activities that interact by communicating data and events. These activities may be scheduled, if required, by supervisor 'control processes'. For example, compare the context diagram (Fig. 5.5) with the top-level transformation schema (Fig. 5.6). Each interaction depicted on the context diagram between the system and external agents is represented on the top-level transformation schema as a input or output data flow.

Each and every data store ('Device Node', 'Pending Request') and data flow ('Device representation configuration') on the transformation schema is specified in the essential model's data specification.

The activities depicted on the transformation schemas may be specified either by structured language process specifications or by further transformation schema. The only criterion is clarity of representation. As such, the conditional logic of **1.5 Receive AES-64 Message**[7] is represented by a further transformation schema (Fig. 5.7).

Scheduling of transformation schema activities

Traditional SA/SD transformation schemas specify data processing carried out by the system but do not depict scheduling. In RTSAD, activity scheduling *is* depicted through 'control transforms'. Fig. 5.6 and Fig. 5.7 each have a control transform (**1.1 Control AES-64 Protocol Stack** and **1.5.1 Control Receive Message** respectively).

The operation of **1.1 Control AES-64 Protocol Stack** is straightforward: until it receives the 'configuration complete' event from **1.2 Configure Device Node**, it does

[7] 1.5 must only submit an inbound AES-64 request for processing if **1.4 Handle Request** is not already processing an earlier message. On the other hand, if 1.5 receives an inbound AES-64 response, it must pass it immediately to **1.6 Handle Response**.

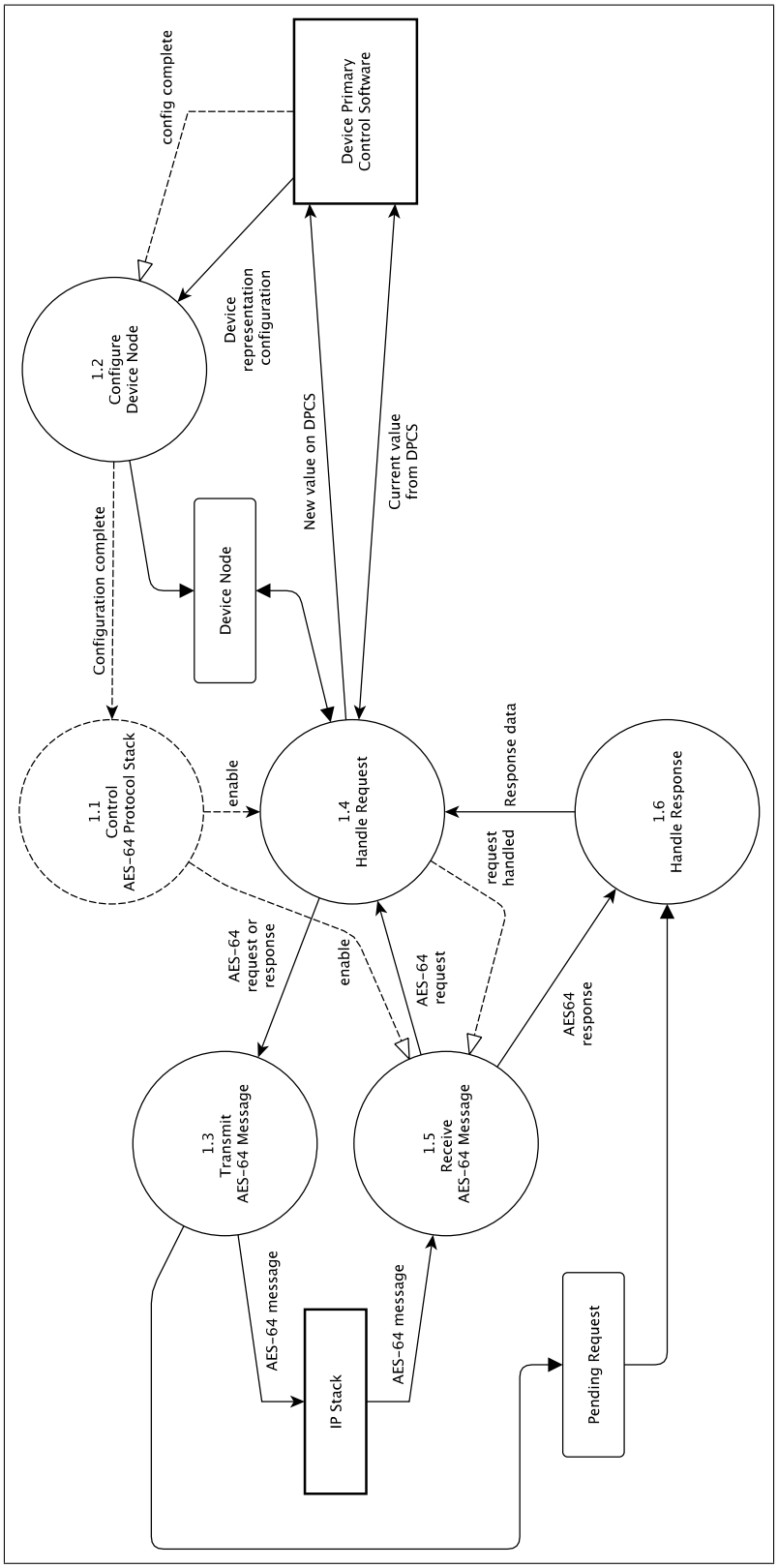


Figure 5.6: The top-level transformation schema for the AES-64 protocol stack.

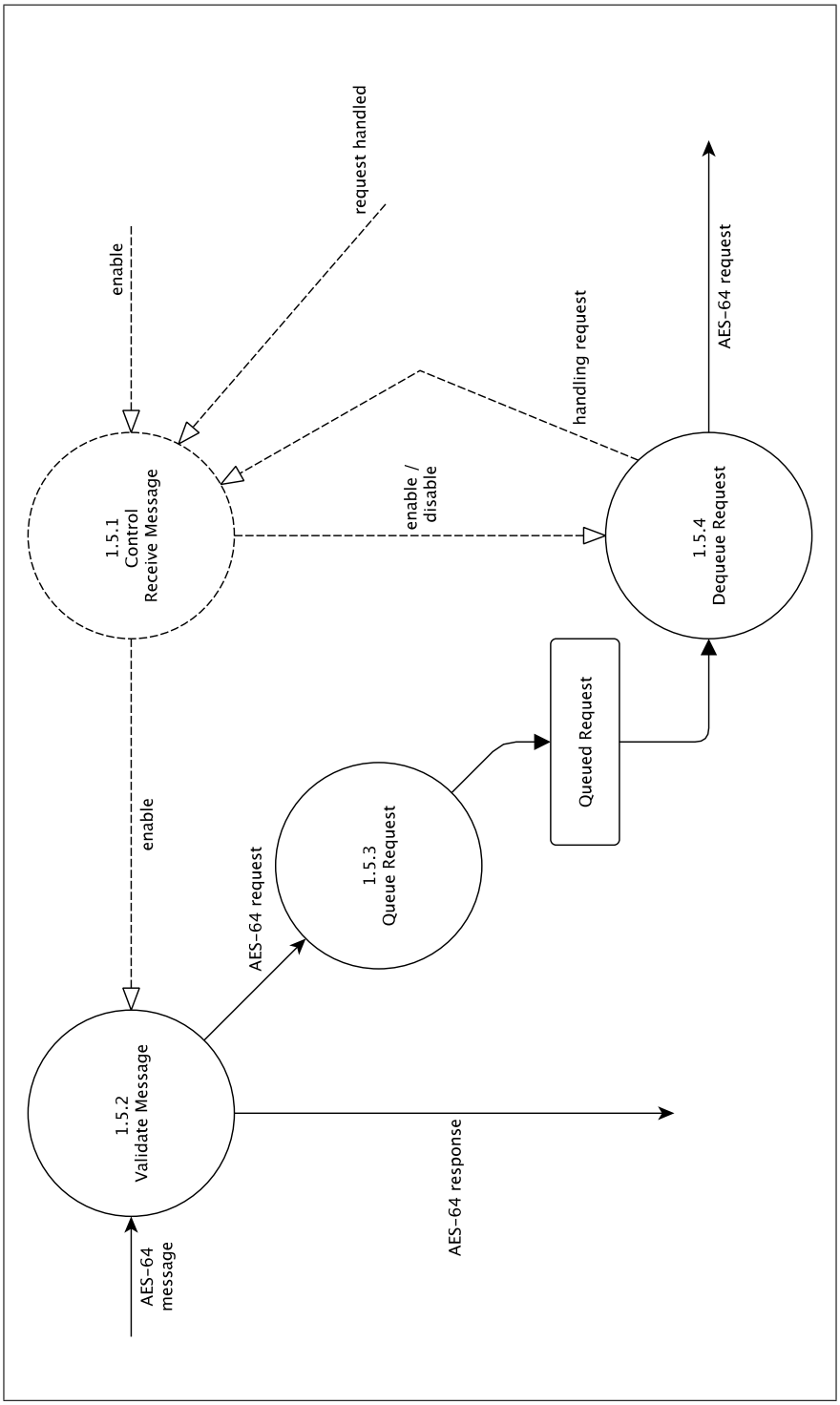


Figure 5.7: The 'Receive Message' transformation schema.

not enable **1.5 Receive AES-64 Message**, thereby preventing the AES-64 protocol stack from receiving or processing external command messages until the full device representation has been configured. This is expressed within the essential model as a structured language process specification (Listing 5.5).

Listing 5.5: Specification of '1.1 Control AES-64 Protocol Stack'

```
CASE configuration complete
        send enable -> 1.5
```

The specification of **1.5.1 Control Receive Message** is shown in Listing 5.6. This specification implements a simple form of flow control over the queuing, dequeuing and handling of inbound AES-64 requests:

- no inbound requests may be handled until the *enable* event from **1.1 Control AES-64 Protocol Stack** has been received, at which time **1.5.2 Validate Message** and **1.5.4 Dequeue Request** are both enabled;
- when **1.5.4 Dequeue Request** submits a message to **1.4 Handle Request**, it additionally sends a *handling request* event to the control activity, which temporarily suspends **1.5.4 Dequeue Request**;
- when **1.4 Handle Request** completes processing of an inbound AES-64 request, it sends the *request handled* event to the control activity, which re-enables **1.5.4 Dequeue Request**.

Consequently, the scheduling of the **1.5 Receive AES-64 Message** activity means that it is not possible for a new inbound AES-64 request to disrupt the processing of an earlier one, while inbound AES-64 responses are exempt from flow control as it is assumed (and later verified, by **1.6 Handle Response**), that they were requested by a current request-processing job.

Listing 5.6: Specification of '1.5.1 Control Handle Receive Message'

```
CASE enable:
        send enable -> 1.5.2
        send enable -> 1.5.3
CASE handling request:
        send disable -> 1.5.4
CASE request handled:
        send enable -> 1.5.4
```

Transformation schema summary

The full set of transformation schemas are presented in Appendix B.4, alongside the rest of the final essential model.

5.3.7 Building the process specifications

Some process specifications have already been shown in earlier sections (Listing 5.5, 5.6). These are constructed from structured language as brief, precise descriptions of individual activities, and for some types of activity are more effective than transformation schema[8].

For example, **1.3 Transmit AES-64 Message** is superficially a straightforward activity: it transmits messages as required by **1.4 Handle Request**. However, if **1.4 Handle Request** needs to transmit a request that expects a response (as it does in the case of processing a 'JOIN' command, for example), then **1.3 Transmit AES-64 Message** must additionally store the sequence ID of the outbound request in order for the protocol stack to reconcile the inbound response message expected at some point in the future. This is specified as shown in Listing 5.7.

Listing 5.7: Specification of '1.3 Transmit AES-64 Message'

```
CASE AES-64 response:
        send AES-64 message
CASE AES-64 request:
        IF request requires a response:
                request sequence ID -> Pending Request store
        send AES-64 message
```

In these process specifications, a unindented `CASE <event>:` statement indicates exactly one of the activity's input events or data flows. The indented block following the `CASE` statement defines the activity's reaction.

The full set of process specifications are presented in Appendix B.5. These include the AES-64 command-specific process specifications described in the following subsection.

5.3.8 Process specifications for handling AES-64 commands

Fig. 5.8 shows the levelled transformation schema for **1.4 Handle Request**. Here, **1.4.2 Validate Request** divides inbound AES-64 requests by their target (Listing 5.8).
The transformation schema for 1.4.3 Handle Level Request is shown in Fig. 5.9.

[8] This is particularly true of activities with lots of conditional outcomes, where on a transformation schema each activity would need to be represented as an individual activity 'bubble'.

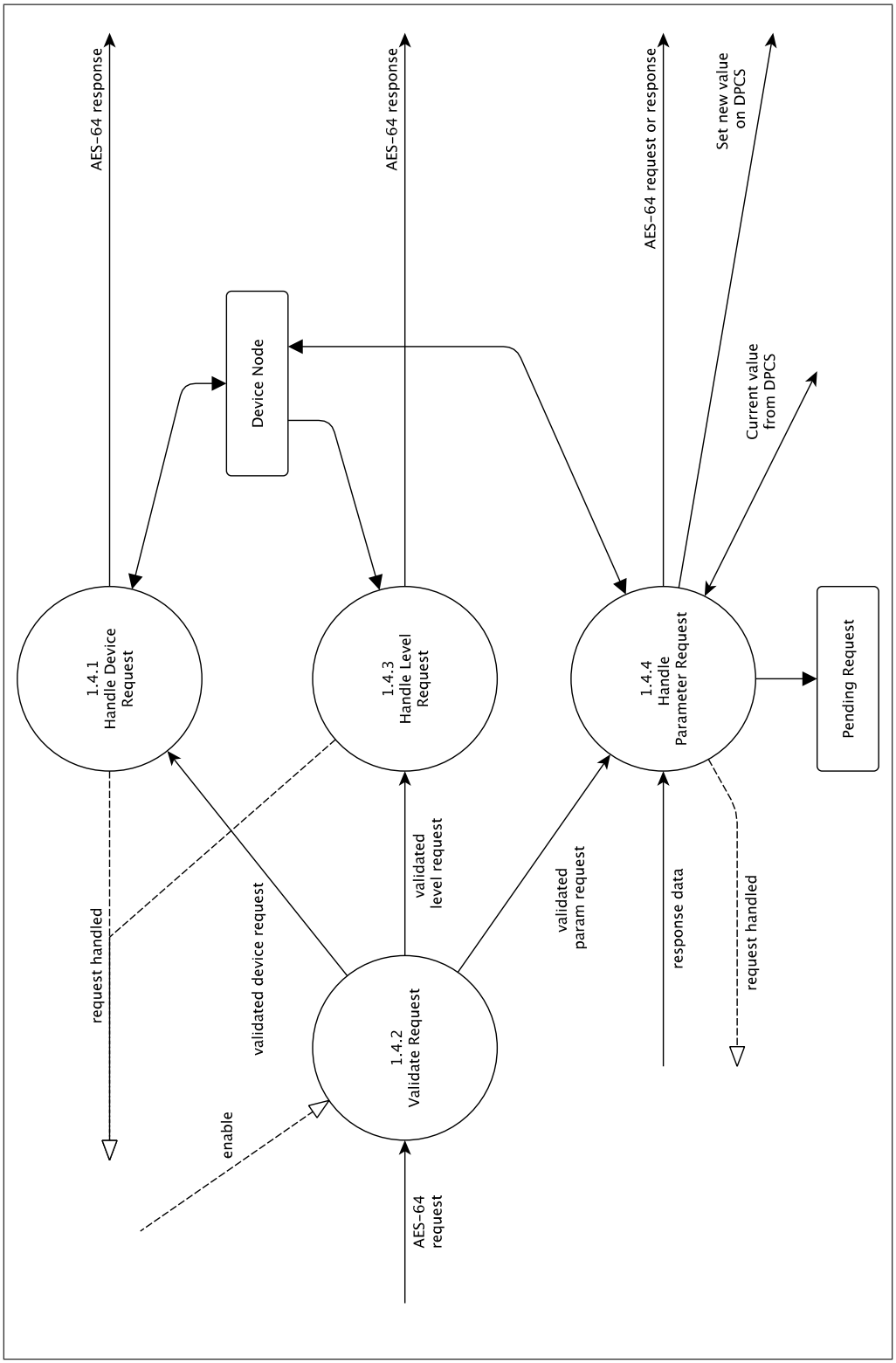


Figure 5.8: The transformation schema for '1.4 Handle Request'.

Listing 5.8: Specification of '1.4.2 Validate Request'

```
CASE AES-64 request:
        SWITCH on AES-64 request Message type:
        CASE device request:
                send AES-64 request -> 1.4.1
        CASE parameter request:
                if AES-64 request Command qualifier == CLA or NAME:
                        send AES-64 request -> 1.4.3
                else
                        send AES-64 request -> 1.4.4
```

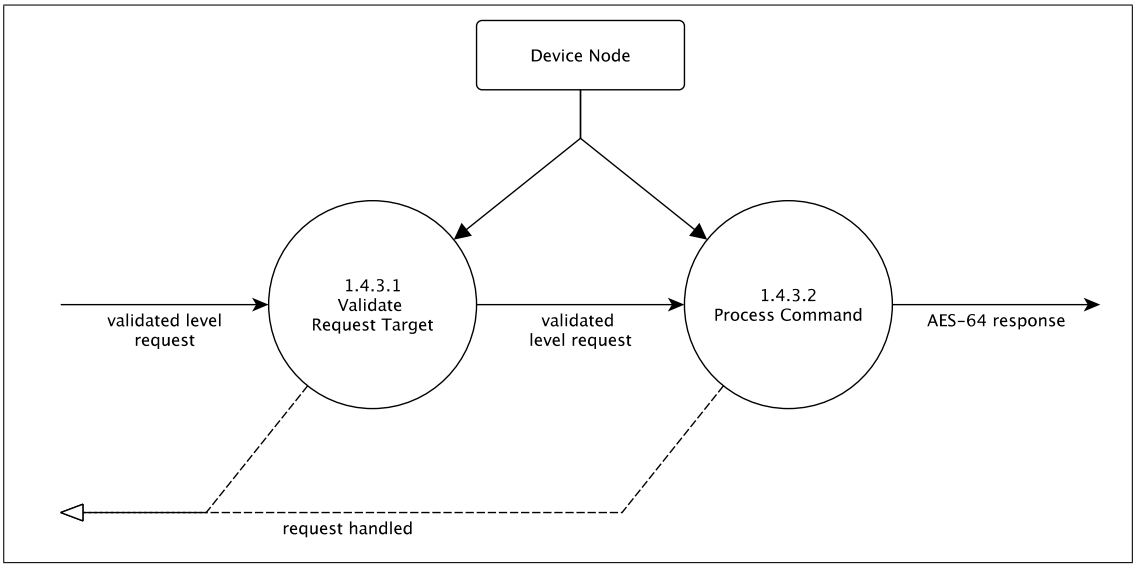


Figure 5.9: The transformation schema for '1.4.3 Handle Level Request'.

Here, **1.4.3.2 Process Command** provides a general-purpose specification of how an AES-64 request is handled (Listing 5.9).

What actually happens during processing of an AES-64 request's command ('Process request command' in Listing 5.9) is specific to the command borne by the request, and so a separate set of command specifications are provided. Listing 5.10 shows an example for the 'GET CLA' level request, which specifies the searching of a device node's level hierarchy to produce a list of child level aliases and their locations.

Listing 5.9: Specification of '1.4.3.2 Process Command'

```
CASE validated level request:
        Process request command
        IF Parameter request with full address block target requiring a
            response:
                send AES-64 response -> 1.3
        send request handled -> 1.5.1
```

Listing 5.10: Specification of the AES-64 'GET CLA' command handler

```
Create response header
Create level alias list
Response value field as per A.8.3
cla resolved = FALSE
search range start = 0
search range end = level items table entry count
FOR EACH level identifier IN request full address block:
        IF level identifier == 0xEE || 0xEEEE || 0xEEEEEE:
                FOR EACH entry in the level items table within search
                    range
                that differs from the previous entry:
                        Create level alias entry
                                (length field, FAB, level alias string)
                        Add entry to level alias list
                cla resolved = TRUE
                EXIT for loop
        ELSE:
                FOR EACH entry in the level items table:
                        IF entry identifier == request level identifier:
                                IF search range start == -1:
                                        search range start = i
                                search range end = i
        IF search range end = -1:
                EXIT for loop
IF cla resolved:
        PACK level alias list into response value field
ELSE:
        DESTROY level alias list
        PACK '0' Entry count into response value field
```

5.4 The implementation model

RTSAD defines the *implementation model* as a mapping or restructuring of an essential model onto the available implementation resources of a system. [Ward and Mellor 1985] discuss this process, particularly with regard to structuring an essential model onto real-time systems with multiple processors and hardware interfaces. A variety of approaches to this phase of design have been proposed, including the Design Approach for Real-Time Systems [Gomaa 1984] and the Modular Approach to System Construction Operation and Test (MASCOT) [Simpson and Jackson 1979].

The starting point of this design work is the finalised essential model. As might be expected, refining the essential model (and its transformation schemas) can significantly simplify the process of developing an implementation model. As seen earlier in this chapter, the levelled transformation schemas (Fig. 5.6 shows the highest level) provide a clear representation of high-level data processing in the AES-64 protocol stack, and the associated process specifications provide a structured description of low-level data processing.

The next step in developing the implementation model is to 'partition' and structure the essential model into a set of concurrent tasks (Fig. 5.10). The top-level transformation schema for the AES-64 protocol stack is straightforward and has good coherence, so the structuring process was not difficult. Once a satisfactory structuring has been achieved, the diagram can be redrawn (Fig. 5.11) showing the concurrent tasks more clearly:

1. the 'stack control task', which configures the device representation, interfaces with the IP stack to receive and transmit AES-64 messages, and integrates with the device primary control software to actuate parameter value functions;

2. the 'message processing task', which parses AES-64 requests and processes the wide variety of standard commands;

3. the 'device processing task', which maintains and provides the message processing task with access to the AES-64 protocol stack's representation of the audio device.

This approach enables the AES-64 protocol stack to execute multiple message processing tasks acting on a common device representation. The following section discusses the structuring of these tasks, their interfaces and their integration into the XMOS Ethernet AVB reference design.

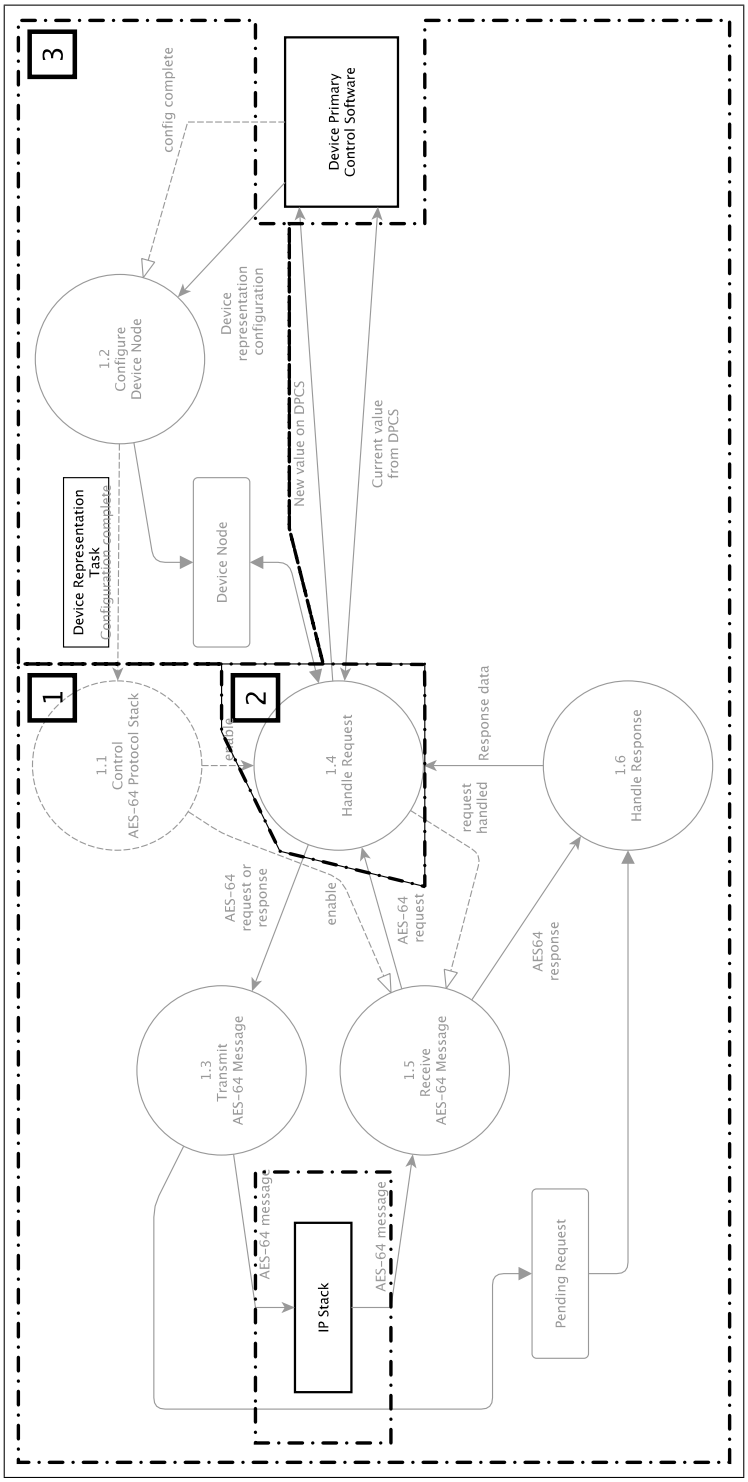


Figure 5.10: Task structuring of the essential model's top-level transformation schema (excluding the 'IP Stack' and 'Device Primary Control Software' terminators) resulting in the specification of three communicating tasks: 1, the 'Stack control task'; 2, the 'Message processing task'; 3, the 'Device representation task'.

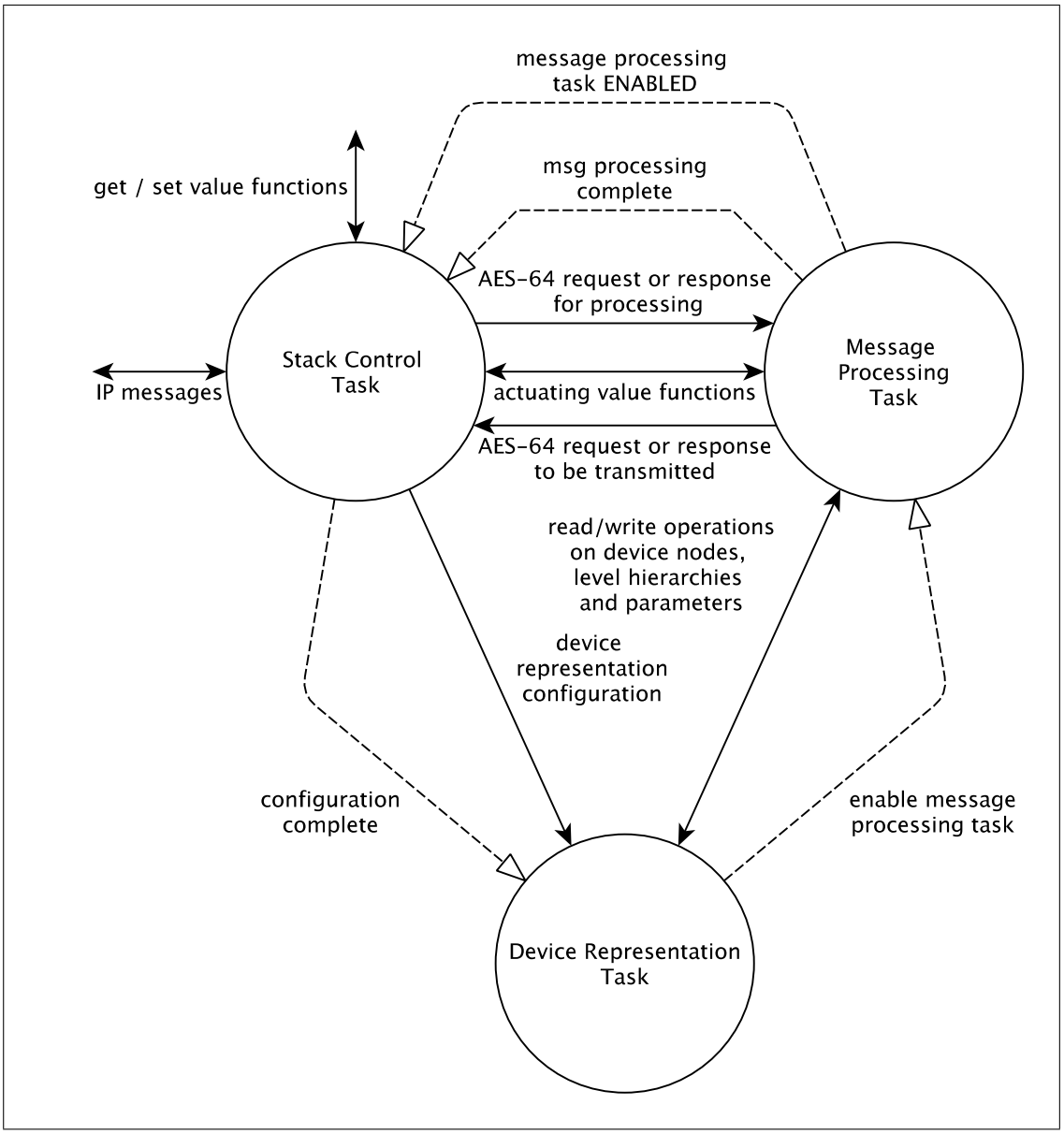


Figure 5.11: The concurrent tasks of the AES-64 protocol stack implementation model.

5.4.1 Task interfaces

As first described in Section 4.6, interfaces between tasks in the AES-64 protocol stack have been designed as lightweight messaging protocols that communicate over XC channels. Each discrete operation (e.g., 'Create device node', 'Validate device node ID') is defined as a protocol command with associated data, where the command, its associated data, and any applicable response are communicated over an XC channel between the task that sends the command and the task that handles it.

Listing 5.11 shows an example implementation of this approach, where the `a64_dnm_api_validate_dnode_id` function executes within a 'message processing' task and communicates with a device representation task as follows:

1. sends a *validate device node ID* command over the channel;
2. sends the associated data (e.g., a device node ID) over the channel;
3. receives an indication of the query status (i.e., whether a device node with the specified ID exists or not).

A detailed illustration of this is provided in Section 4.6, and the implementation of these interfaces is discussed in greater detail in Chapter 6.

Listing 5.11: A message processing function employs an XC channel-based interface to query the existence of an identified device node on the device representation task.

```
UInt8 a64_dnm_api_validate_dnode_id(chanend c_dnm, UInt32 dnode_id)
{
        UInt8 request_success = A64_DNM_REQUEST_NOK;
        a64_dnm_api_send_mp_cmd(c_dnm, A64_MP_VALIDATE_TARGET_DNODE);
        c_dnm <: dnode_id;
        c_dnm :> request_success;
        return request_success;
}
```

5.4.2 The device representation task

In Fig. 5.11, the 'Device Node' data store and its associated functionality has been partitioned as a self-contained 'device representation task'. Fig. 5.12 depicts the device representation task and its inter-task communications in isolation.

The 'Device Node' data store is likely to contain more than one device node, each of which will contain its own level hierarchy and parameter array. These device nodes are configured by the following command messages from the stack control task:

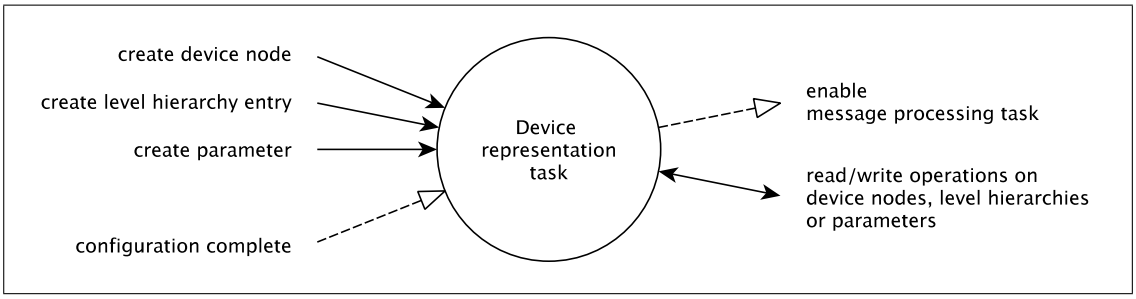


Figure 5.12: The device representation task and its communications with [L] the stack control task and [R] the message processing task.

Create device node. Creates a device node.

Create level hierarchy entry. Associated data includes seven numeric identifiers ('Section Block' – 'Parameter Index') ranging between 8 and 24 bits, and four 'level alias' strings.

Create parameter. Associated data includes value format, value, name and value function information.

Configuration complete. Control message, indicating that the device node(s) is now fully configured and ready to service messaging requests.

In Fig. 5.11 these command messages were packaged as 'Device representation configuration'. Once the device representation task receives the 'Configuration complete' control message, it will transmit the 'enable message processing' control message to the message processing task.

Once it is configured and has enabled the message processing task, the device representation task then receives commands from it:

Read/write operations on device node, level hierarchy, and parameters. This packages all of the read/write operations on the attributes of a device node, its level hierarchy or its parameters, including:

1. validation of device node and parameter IDs;
2. validation of full address blocks (in several distinct contexts);
3. read/write/create/destroy parameter attributes, including values, value function identifiers and group lists (see Table 5.2).

In addition to maintaining the protocol stack's device representation, the task is also responsible for providing the 'message processing thread' with access to the representation in order to enable processing of AES-64 requests. Table 5.2 shows the operations necessary to do so:

Table 5.2: Required operations to enable processing of AES-64 requests

Parameter operations	Device node operations
Read parameter value format	Read device node count
Read parameter value function	Read parameter count on device node
Read parameter name	Validate device node ID
Read parameter flags	Validate parameter ID
Read parameter peer group list count	Resolve target full address block (F.A.B.)
Read parameter peer group list entry	Resolve 'GET CLA' request F.A.B.
Read parameter master group list count	Resolve 'GET NAME' request F.A.B.
Read parameter master group list entry	
Read parameter slave group list count	
Read parameter slave group list entry	
Write parameter flags	
Write parameter peer group list entry	
Write parameter master group list entry	
Write parameter slave group list entry	
Validate parameter peer group list entry	
Validate parameter master group list entry	
Validate parameter slave group list entry	
Remove parameter peer group list entry	
Remove parameter master group list entry	
Remove parameter slave group list entry	

5.4.3 The message processing task

The message parsing and command processing functionality of the AES-64 protocol stack has been partitioned as a self-contained 'message processing task'. The task is shown in the context of the protocol stack in Fig. 5.11, and in isolation in Fig. 5.13.

The message processing task is enabled by the device processing task, as described in Subsection 5.4.2, and in response will signal the *stack control task* that it has been enabled for processing messages; it may also receive the following command from the stack control task:

AES-64 response or request submitted for processing. Submitted by the stack control task.

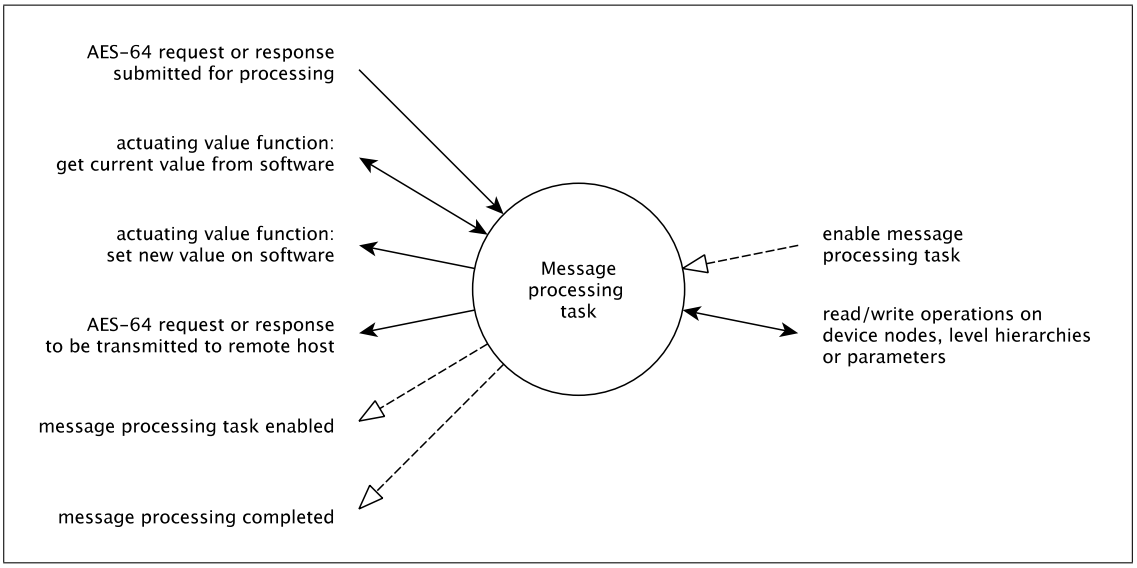


Figure 5.13: The message processing task and its communications with [L] the stack control task and [R] the device representation task.

In the course of processing an AES-64 message, the message processing task may send the following commands to the device representation task:

Read/write operations on device nodes, level hierarchies or parameters. As described in Subsection 5.4.2, this packages the set of commands necessary for the message processing task to process an AES-64 request against the protocol stack's representation of the host device.

In the course of processing an AES-64 message, the message processing task may send the following commands to the stack control task:

AES-64 response or request to be transmitted to remote host. Submits an outbound AES-64 message to the stack control task for transmission via the IP stack.

Actuating value function: Get current value from software. Calls a parameter's registered value function – which executes within the stack control task – in the course of processing a 'GET VAL' (or similar) request.

Actuating value function: Set new value on software. Calls a parameter's registered value function – which executes within the stack control task – in the course of processing a 'SET VAL' (or similar) request.

Message processing completed. The message processing task signals the stack control task that it has completed the current processing job.

5.4.4 The stack control task

The stack control task (Fig. 5.14) controls the AES-64 protocol stack's integration with the XMOS Ethernet AVB reference design, including the IP stack and the Ethernet AVB API. Most notably, this includes the execution of all parameter *value functions* in the course of processing AES-64 'GET VAL' and 'SET VAL' requests.

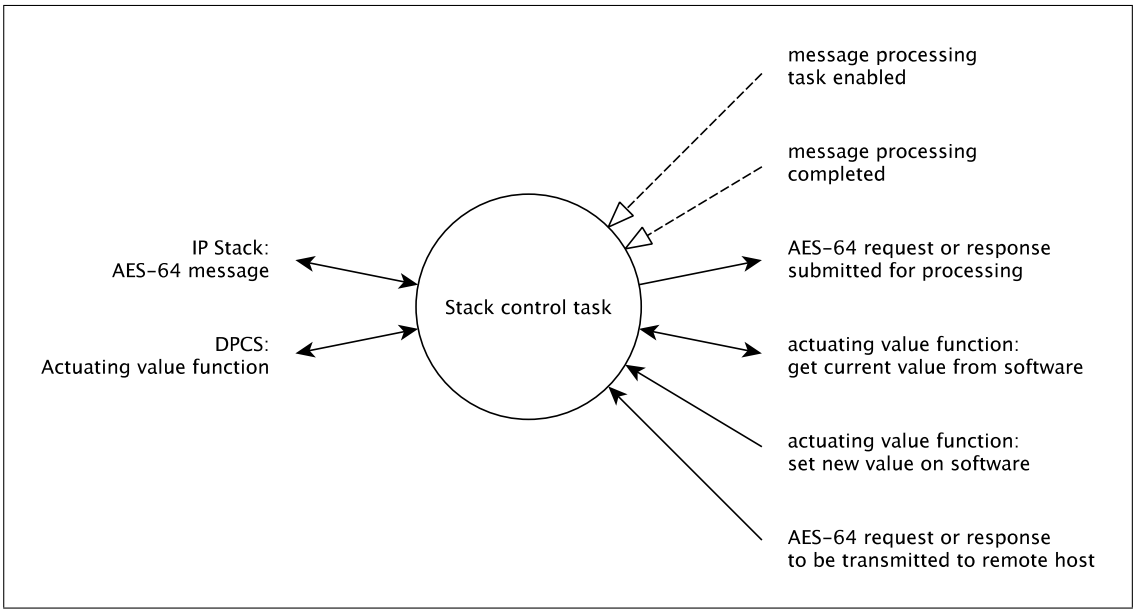


Figure 5.14: The stack control task and its communications with [L] the XMOS IP stack and device primary control software, and [R] the message processing task.

The stack control task's interactions with the XMOS Ethernet AVB reference design are:

XMOS IP Stack. 'Packaged' as per Fig. 5.5:

1. Receives AES-64 request or response from IP stack. This is submitted to the message processing task if the task is available, or queued if it is not.
2. Transmits AES-64 request or response to IP stack, as requested by the message processing task.

Device primary control software. Actuate AES-64 parameter value function.
This happens in response to 'get current value from software' and 'set new value on software' commands received from the message processing task (Subsection 5.4.3).

Each command provides value function information which enables the stack control task to identify and call the corresponding function on the device primary control software. The majority of these functions will be from the XMOS Ethernet AVB reference design's API, but other functions could also be called: e.g., functions that retrieve the device's current IP configuration, or functions that set volume controls on the device's audio hardware.

The stack control task's interactions with the message processing task were described in the preceding subsection, Subsection 5.4.3. The stack control task implements a selective queuing system for inbound AES-64 messages as described in Subsection 5.3.6. The implementation of this queuing system is discussed in greater detail in Section 6.7.

5.5 Conclusion

Designing an implementation of the AES-64 protocol stack involved a thorough requirements analysis process, in part to resolve discontinuities in the published standard. The requirements analysis was completed through close scrutiny of the standard in addition to code review and forensic capture of existing implementations of the standard. The results of the requirements analysis were compiled into a requirements document, included as Appendix A.

The Real-Time Structured Analysis and Design (RTSAD) process produced a system specification or *essential model* providing a structured depiction of system behaviour, data processing and control processes. This model was developed to incorporate implementation details, allowing system behaviour to be structured into a set of communicating tasks. Communication interfaces between these tasks were designed to make use of the message-passing channels provided by the XMOS XS1 architecture and controlled through simple XC constructs, as reviewed in Chapter 4.

Chapter 6 provides a detailed account of the implementation of this design.

6 Implementation

6.1 Introduction

Previous chapters have described the AES-64 standard for control of audio network devices (Chapter 3) and the XMOS Ethernet AVB reference design which implements an Ethernet AVB audio network device (Chapter 2) on the XMOS XS1 platform (Chapter 4). Chapter 5 discussed the design of an AES-64 protocol stack to integrate with the XMOS Ethernet AVB reference design.

This chapter begins with an overview of the implementation process (Section 6.2), followed by a summary of prototyping and testing procedures and the findings these produced (Section 6.3). Sections 6.4–6.9 discuss the structure of the protocol stack (Fig. 6.1) [1] . Fig. 6.2 provides a map for this discussion.

Section 6.4 covers the integration of the AES-64 protocol stack into the XMOS Ethernet AVB reference design. Section 6.5 discusses the implementation of the AES-64 protocol stack in relation to the task model described at the conclusion of Chapter 5, focusing on task communication over XC channels. This is illustrated with a summary description of how task communication operates in the course of processing a simple AES-64 request (Subsection 6.5.3).

Section 6.6 provides an overview of the value functions implemented in order to connect AES-64 parameter values with components of the Ethernet AVB device subsystem(s). This is followed by detailed descriptions of the implementation of the stack control task (Section 6.7), the device representation task (Section 6.8), and the message processing task (Section 6.9).

Section 6.10 describes the implementation of AES-64 control over Talker, Listener, media clock and audio hardware functions of the XMOS Ethernet AVB reference design.

Section 6.11 demonstrates the *configuration* of the AES-64 protocol stack to represent an XMOS Ethernet AVB device.

Finally, Section 6.12 briefly describes the implementation of command-line tools to

[1] Fig. 6.1 represents the software as finally implemented and differs in some details from the model presented at the end of Chapter 5 (Fig. 5.11). For details of these differences, see Section 6.5.

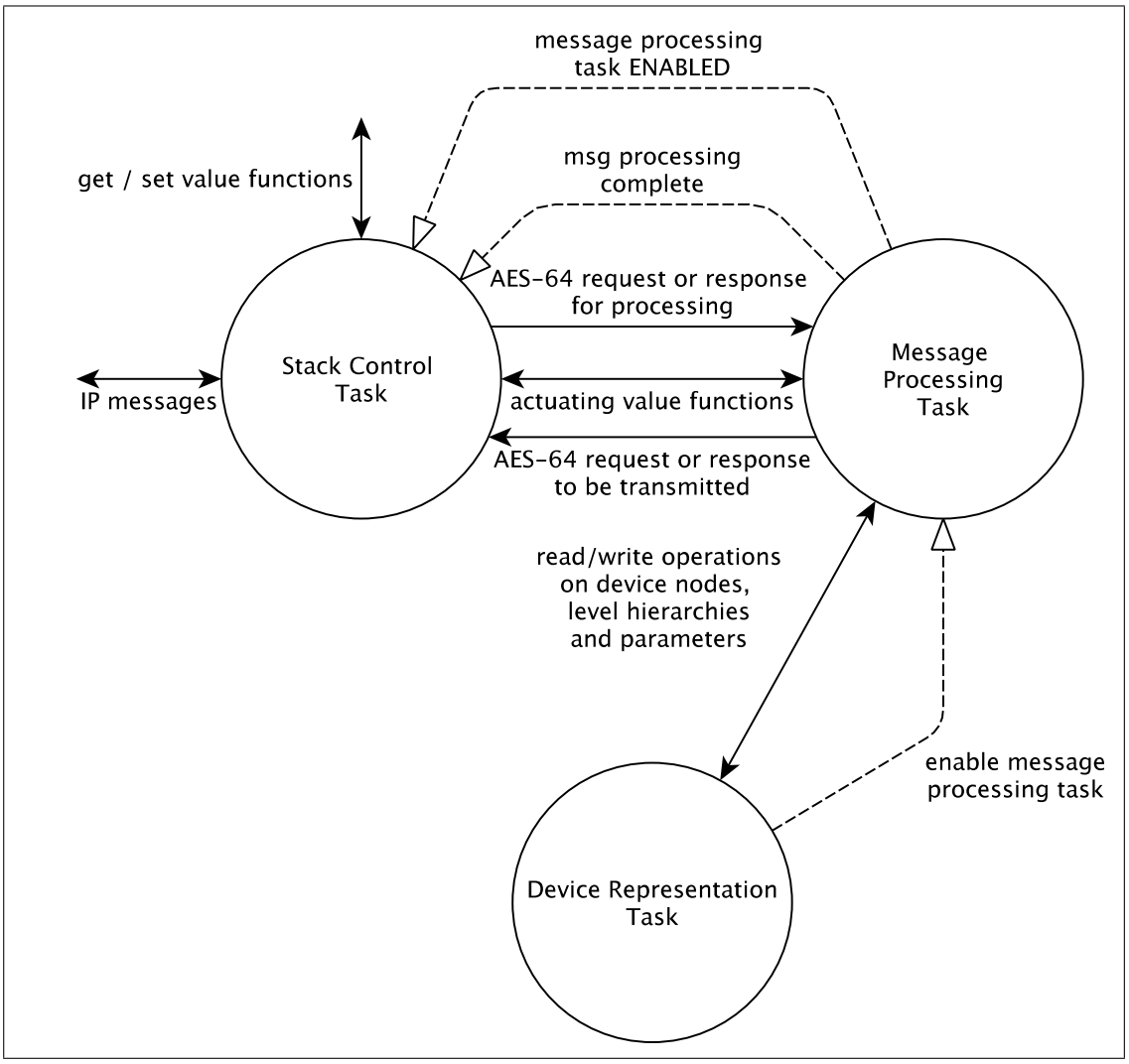


Figure 6.1: The AES-64 implementation tasks and their channels of communication.

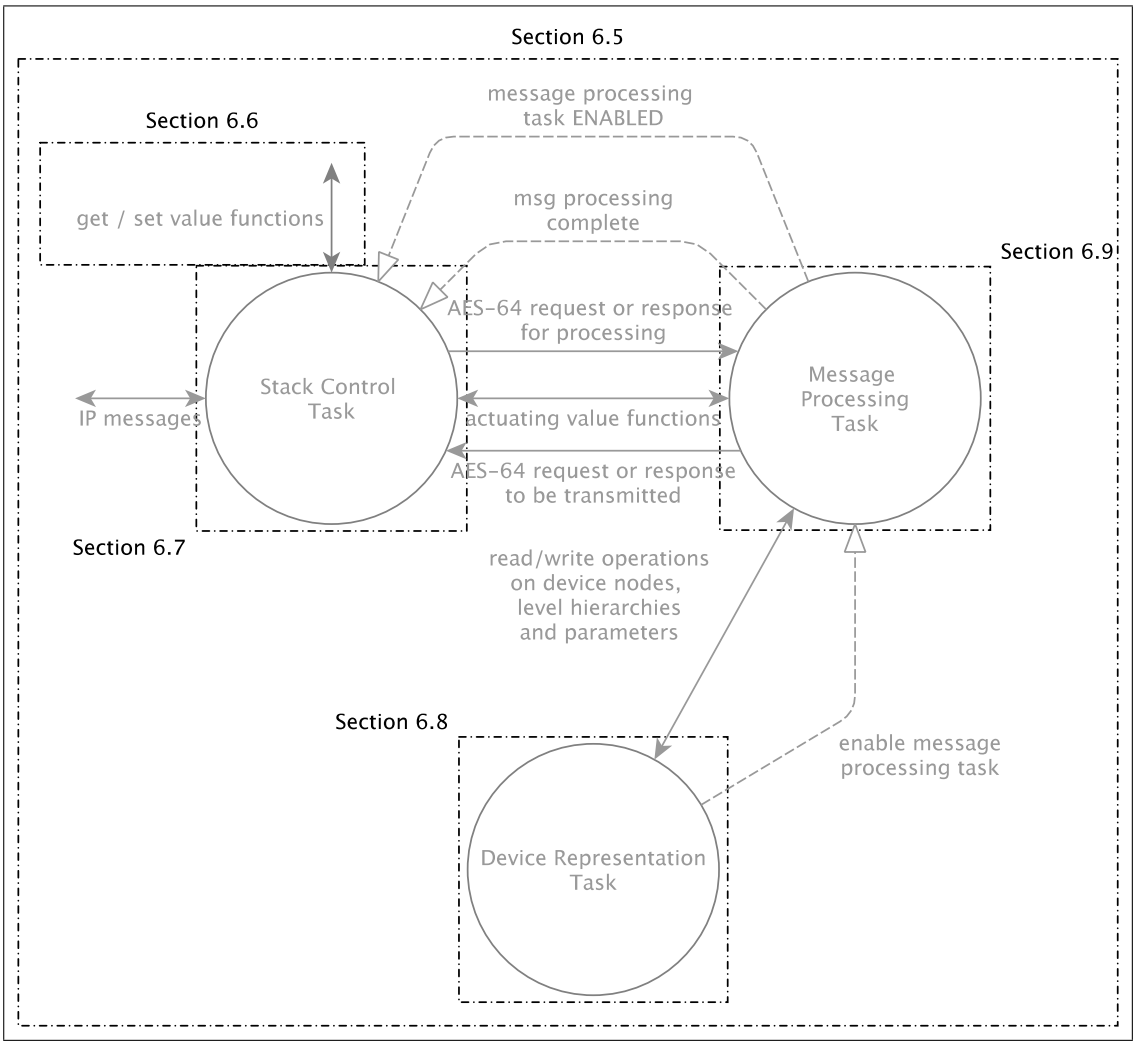


Figure 6.2: Chapter map for discussion of the AES-64 implementation.

perform remote control and monitoring of AES-64-enabled devices.

6.2 Implementation overview

The core functionality of the AES-64 protocol stack – device representation, message processing and command handlers – was implemented in C. The resulting software modules integrate with the XMOS Ethernet AVB reference design (and each other) by task interfaces and communications implemented in XC, as described in Subsection 5.4.1 and (briefly) shown in Section 4.6.

Implementation was an incremental process, beginning with prototyping of core AES-64 functionality on a PC platform, continuing with a port of the prototype to run on the target platform, restructured by the development of the prototype into a set of concurrent tasks as defined by the implementation model, concluding with the integration of the AES-64 implementation with the Ethernet AVB audio device subsystems. Testing ran continuously throughout each phase of implementation. Prototyping away from the target platform also allowed for the use of tools such as Valgrind[2] [Valgrind Developers 2012], and the prototype PC implementation also provided a basis to implement PC control utilities (Section 6.12). This sequence of implementation is shown in Fig. 6.3.

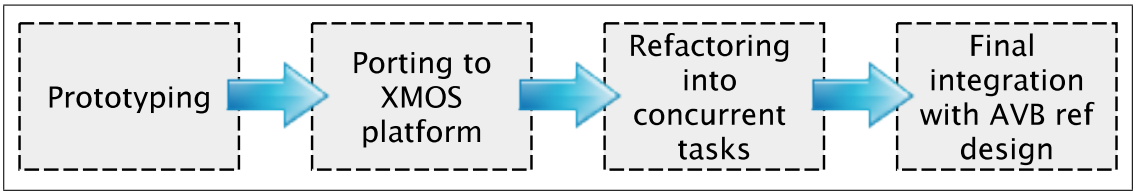


Figure 6.3: Implementation of the AES-64 standard on the XMOS Ethernet AVB reference design.

6.3 Prototyping

The implementation of core AES-64 functionality became possible during the requirements analysis phase of the design. As the formats of message headers and command value fields were revealed, data structures and routines to construct and serialise these structures were implemented. Accurate construction and serialisation of these formats could be tested in isolation, which in turn enabled the development of a preliminary message parser.

[2]Valgrind is a tool for memory debugging, memory leak detection and profiling.

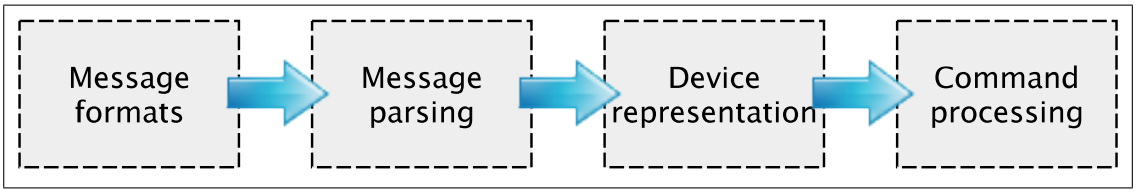


Figure 6.4: Prototyping of core AES-64 functionality.

The development of a parser for AES-64 requests prompted the development of device representation: for instance, to develop and test a parser for 'full address blocks', it was necessary to have in place a device representation (formed of a device node, parameters and a level hierarchy) to evaluate the full address blocks against (Fig. 6.4).

This section provides a summary of the prototyping phase of implementation, illustrating the development of core AES-64 functionality.

6.3.1 Message formats and serialisation

AES-64 message formats and the routines required to construct and serialise them were prototyped and tested using UDP client/server programs communicating over a PC's loopback interface.

As an example, Fig. 6.5 shows the generic AES-64 response message header [Audio Engineering Society 2012, 18]. Listing 6.1 shows the implementation of a data structure for this message header, `MSG_RESPONSE_HDR`, and a serialisation macro, `PACK_BSTR_RESPONSE_HDR()`, which packs the fields of a `MSG_RESPONSE_HDR` struct into a message buffer consistent with Fig. 6.5[3].

[3]The use of a do-while loop to enclose the `PACK_BSTR_RESPONSE_HDR()` macro is recommended by [Torvalds 2001] and [Summit 2005]

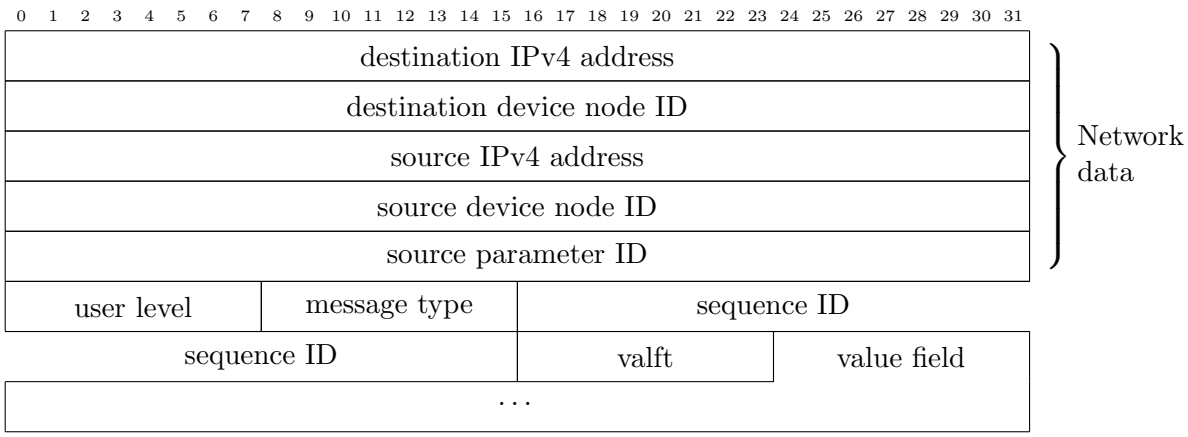


Figure 6.5: The AES-64 response message header.

Listing 6.1: Data structure and serialisation macro for the AES-64 response message header (deserialisation macro not shown)

```
/* For each message-related struct in the AES-64 stack, there is also:
    (a) a LENGTH constant (often used as an offset),
    (b) a PACK macro that serialises the struct contents into a
        bytestream,
    (c) an UNPACK macro that de-serialises a suitable bytestream into a
        struct instance. */
#define MSG_RESPONSE_HDR_LENGTH \
        (A64_NETWORK_DATA_HDR_LENGTH + A64_USER_LEVEL_HDR_LENGTH + \
         A64_MSG_TYPE_HDR_LENGTH + A64_SEQUENCE_ID_HDR_LENGTH + \
         A64_VALFT_HDR_LENGTH)

/* Represents an AES-64 response message. */
struct MSG_RESPONSE_HDR {
    struct A64_NETWORK_DATA_HDR network_data;
    struct A64_USER_LEVEL_HDR user_level;
    struct A64_MSG_TYPE_HDR msg_type;
    struct A64_SEQUENCE_ID_HDR sequence_id;
    struct A64_VALFT_HDR valft;
    struct A64_VALUE_FIELD value;
};

/* Serialises the data from an MSG_RESPONSE_HDR struct into a bytestream
    as defined by AES-64 */
#define PACK_BSTR_RESPONSE_HDR(buf, msg) \
    do { \
        UInt8 *tmp = (UInt8 *) buf; \
        PACK_BSTR_IPv4_ADDR(tmp, msg->network_data.target_ip_addr); \
        tmp += A64_IPv4_ADDR_LENGTH; \
        PACK_BSTR_A64_NODE_ID(tmp, msg->network_data.target_node_id); \
        tmp += A64_NODE_ID_LENGTH; \
        PACK_BSTR_IPv4_ADDR(tmp, msg->network_data.sender_ip_addr); \
        tmp += A64_IPv4_ADDR_LENGTH; \
        PACK_BSTR_A64_NODE_ID(tmp, msg->network_data.sender_node_id); \
        tmp += A64_NODE_ID_LENGTH; \
        PACK_BSTR_PARAM_ID_HDR(tmp, msg->network_data.sender_param_id); \
        tmp += A64_PARAM_ID_HDR_LENGTH; \
        PACK_BSTR_USER_LEVEL_HDR(tmp, msg->user_level); \
        tmp += A64_USER_LEVEL_HDR_LENGTH; \
        PACK_BSTR_MSG_TYPE_HDR(tmp, msg->msg_type); \
        tmp += A64_MSG_TYPE_HDR_LENGTH; \
        PACK_BSTR_SEQ_ID_HDR(tmp, msg->sequence_id); \
        tmp += A64_SEQUENCE_ID_HDR_LENGTH; \
        PACK_BSTR_VALFT(tmp, msg->valft); \
        tmp += A64_VALFT_HDR_LENGTH; \
    } while (0)
```

Since value fields are specific to a given command (or its associated response), additional structures and macros are implemented to construct and serialise these. Listing 6.2 shows an example for the AES-64 'JOIN MSTSLV' command's associated value field. In this listing, the `UNPACK_BSTR_A64_JOIN_MSTSLV_REQ` macro unpacks the bytestream of an 'JOIN MSTSLV' command's associated value field into an `A64_JOIN_MSTSLV_REQ` struct.

Listing 6.2: Data structure and de-serialisation routine for JOIN MSTSLV value field

```
#define A64_JOIN_MSTSLV_REQ_LENGTH      22  /* 1 + 4 + 4 + 13 */
struct A64_JOIN_MSTSLV_REQ {
        UInt8                   join_type;
        struct A64_IPv4_ADDR    slave_ip;       /*  4 bytes */
        struct A64_NODE_ID      slave_dnode_id; /*  4 bytes */
        struct A64_FAB_HDR      slave_fab;      /* 13 bytes */
};

/* This macro unpacks the bytestream of an AES-64 JOIN MSTSLV request's
    value field into an instance of the above struct. */
#define UNPACK_BSTR_A64_JOIN_MSTSLV_REQ(req, buf) \
        do { \
                UInt8 *tmp = (UInt8 *) buf; \
                req.join_type = tmp[0]; \
                tmp += A64_UINT8_LENGTH; \
                UNPACK_BSTR_IPv4_ADDR(req.slave_ip, tmp); \
                tmp += A64_IPv4_ADDR_LENGTH; \
                UNPACK_BSTR_A64_NODE_ID(req.slave_dnode_id, tmp); \
                tmp += A64_NODE_ID_LENGTH; \
                UNPACK_BSTR_A64_FAB_HDR(req.slave_fab, tmp); \
                tmp += A64_FAB_HDR_LENGTH; \
        } while (0)
```

6.3.2 The device node and level hierarchy

The prototype device node provided a basis for the development and testing of an AES-64 message parser, particularly the validation of parameter targets. Its preliminary representation is shown in Listing 6.3 and a schematic of its structure is shown in Fig. 6.6. Multiple device nodes were supported, as indicated by the `next_dnode` pointer. The size of the parameter array (`MAX_PARAMS`) was set in a configuration header file and offset by 1 to account for the fact that AES-64 has a convention of numbering parameters (and level hierarchy indexes like 'Section Number') from '1' rather than '0'.

The prototype implementation of the level hierarchy was based around a linked list of linked lists, after the approach described in Foulkes [Foulkes 2011, 290-293]. Under this approach, a device configuration was built up level by level, as shown in Listing 6.4.

The general practice for implementing a hierarchy as shown in Fig. 6.6 is:

Listing 6.3: Prototype device node

```
struct DNode {
        UInt32 id;
        struct LItem *level_hierarchy;
        struct DNode *next_dnode;
/* AES-64 parameters are indexed from 1, not 0. The +1 in the following
    declarations is there to pad out the [0]th field */
        struct Param *param_array[MAX_PARAMS + 1];
        UInt32 selected_param_bitmap[(MAX_PARAMS / 32) + 1];
        unsigned int selected_param_bitmap_size;
};
```

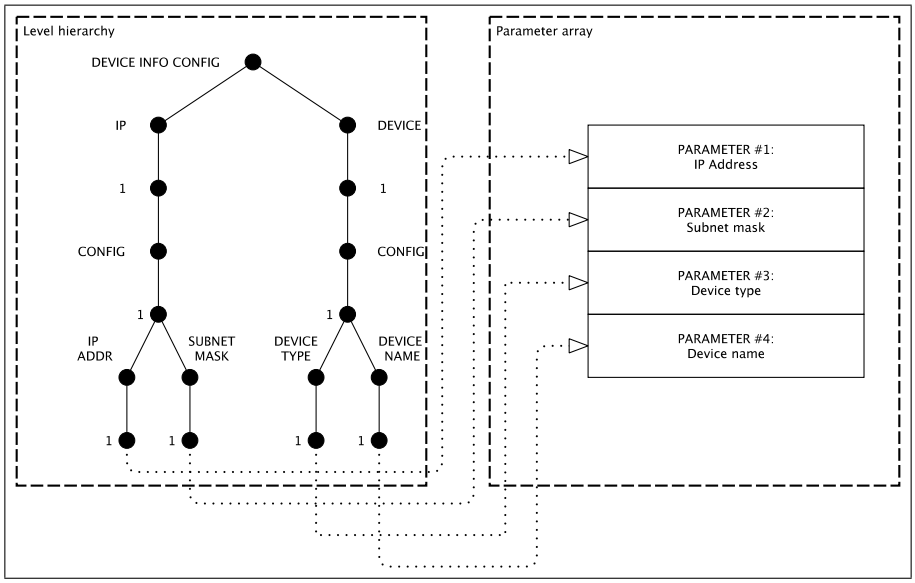


Figure 6.6: Structure of the prototype device node, illustrating the link between the level hierarchy and the parameter array.

1. Build the left-hand branch to completion.
2. Back up to the nearest fork and build the remaining branches from left to right.
3. Repeat until complete.

In Listing 6.4, the 'IP Address' and 'Subnet Mask' parameters shown on the left-hand-side of Fig. 6.6 are created.

This approach to the level hierarchy was later rejected in favour of a table-based implementation that requires marginally more memory but which is substantially more straightforward to process and configure. Subsection 6.3.5 describes and justifies this revision.

Listing 6.4: Creation of two parameters and their level hierarchy entries using the prototype device node

```
/* The general practice for implementing a hierarchy as shown in Fig. 6.6
     is 'build the left-hand branch, then back up to the most recent fork
    and build the right hand branches'. */

/* create Section Block with id 'DEVICE INFO CONFIG' */
        add_section_block_to_last_dnode(get_last_dnode(),
                SB_DEVICE_INFO_CONFIG, "DEVICE␣INFO␣CONFIG");
/* create Section Type level with id 'IP' */
        add_section_type_to_last_section_block(get_last_section_block(),
                ST_INTERFACE_CONFIG, "INTERFACE␣CONFIG");
        add_section_number_to_last_section_type(get_last_section_type(),
                1, "1");
        add_param_block_to_last_section_number(get_last_section_number(),
                PB_IP, "IP");
        add_param_block_idx_to_last_param_block(get_last_param_block(),
                1, "1");
        add_param_type_to_last_param_block_idx(
                get_last_param_block_index(),
                PT_IP_ADDRESS, "IP␣ADDRESS");
        add_param_idx_to_last_param_type(get_last_param_type(),
                1, "1");
/* create and add IP ADDRESS parameter */
        add_parameter_to_last_param_idx(get_last_param_index(),
                "IP", 0, 0xDEADBEEF); /* placeholder value */
/* back up to 'the nearest fork' and build remaining branches from left
    to right */
        add_param_type_to_last_param_block_idx(
                get_last_param_block_index(),
                PT_SUBNET_MASK, "SUBNET␣MASK");
        add_param_idx_to_last_param_type(get_last_param_type(), 1, "1");
/* create and add SUBNET MASK parameter */
        add_parameter_to_last_param_idx(get_last_param_index(),
                "SUBNET", 0, 0xDEADBEEF); /* placeholder value */
```

With this early implementation in place, a basic recursive parser for full address blocks was implemented (Listing 6.5) and tested successfully. The `process_full_address_block()` function parses the full address block supplied with a request, producing a list of matching parameters from the device representation. The `process_fab_request()` function executes the command expressed by an AES-64 request against each of the parameters located by `process_full_address_block()`. An alternate approach is described in Section 6.8.

Listing 6.5: The first basic parser for AES-64 full address block requests

```
/*
process_full_address_block()
This function works recursively through the supplied full address block,
    searching the linked list(s) that comprise the level hierarchy. The
    level hierarchy linking looks like this:

(*) parent
 |
(*)----------(*)----------(*)
child 1       child 2       child 3

If it finds a level matching the current identifier from the full address
     block, it searches the level's children for the next entry in the
    full address block. Under normal circumstances, it should ultimately
    return a parameter.

Wildcards are dealt with by identifying that the current portion of the
    FAB matches a wildcard identifier. If a wildcard is located, the
    function ensures that even *if* a param is located, all of the 'next'
    litems are interrogated too, possibly locating more parameters, before
     the function is allowed to return.
*/
int process_full_address_block(struct LNode *level_hierarchy, UInt8 *
    address_block);

/*
Validate and parse received request message.
Call process_full_address_block() to locate request targets.
Execute command handler for each located request target.
*/
int process_fab_request(UInt8 msg_type, BOOL response_required, struct
    A64RXBuffer *rx_buf, struct A64TXBuffer *tx_buf)
```

6.3.3 The parameter

The prototype parameter implementation (shown in Listing 6.6) provided a basis for message parsing and command handler prototyping. This implementation of an AES-64 parameter was only capable of holding values up to 32 bits in size, did not implement

any other parameter attributes (e.g., group lists, flags), and only provided 'get' and 'set' value command handling. However, this was sufficient to prototype the behaviour of 'GET VAL' and 'SET VAL' messages, and was used to demonstrate very basic control of the XMOS Ethernet AVB subsystem during the porting phase of development.

The final implementation of the AES-64 parameter added several attributes, including a full implementation of parameter grouping. The use of get / set callbacks to implement the parameter *value functions* was found to present difficulties in the multithreaded implementation, so the final AES-64 stack implements an alternative lookup mechanism (described in detail in Section 6.8).

Listing 6.6: The prototype parameter structure

```
/* struct of function pointers to parameter value functions, which
    implement AES-64 'GET' and 'SET' operations on parameter values */
struct Param_callbacks {
        void (*get) (struct Param *, struct A64_NETWORK_DATA_HDR *,
            struct A64_CMD_HDR *, struct A64_SEQUENCE_ID_HDR *,
            struct A64_VALFT_HDR *, struct A64_VALUE_FIELD * );
        void (*set) (struct Param *, struct A64_NETWORK_DATA_HDR *,
            struct A64_CMD_HDR *, BOOL response_required,
            struct A64_SEQUENCE_ID_HDR *, struct A64_VALFT_HDR *,
            struct A64_VALUE_FIELD * );
};

/* Prototype AES-64 parameter struct */
struct Param {
        UInt32 id; /* unique parameter ID */
        char *name; /* human-readable parameter name */
        struct LNode *parent_lnode; /* parent level */
        struct DNode *parent_dnode; /* parent device node */
        UInt8 valft; /* value format */
        UInt32 value; /* prototype value */
        struct Param_callbacks *callbacks;
};
```

6.3.4 Prototyping summary

The prototyping process enabled a simple proof-of-concept implementation of Ethernet AVB subsystem control through AES-64 messaging. This implementation executed as a single thread and provided rudimentary get/set capability over the XMOS Ethernet AVB subsystem: for example, by configuring the parameters shown in Fig. 6.7 with get and set callbacks that call the XMOS Ethernet AVB API functions shown in Listing 6.7, AES-64 requests were able to disable / enable the XMOS Talker and Listener components from

respectively advertising or receiving streams[4].

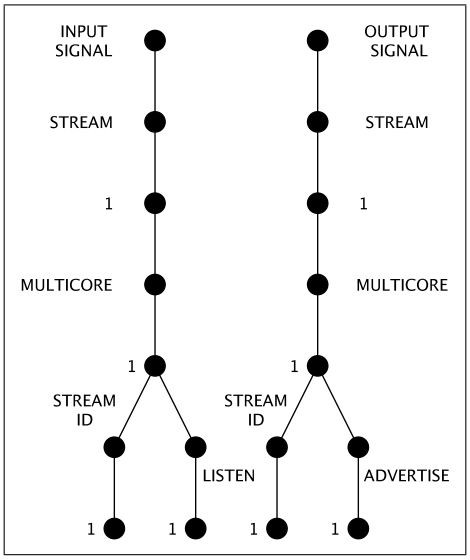


Figure 6.7: AVB stream connection parameters.

Listing 6.7: Some of the XMOS Ethernet AVB API functions for AVB stream connection

```
/*  Talker stream connection management. The Talker can be set to
    broadcast or withdraw an advertisement of its available stream. The
    stream ID is predictably generated and no API call is implemented to
    retrieve it.   */

void set_avb_source_state(unsigned source_num, avb_source_state_t state);

/*  Listener stream connection management. The Listener can be set to
    show interest in an identified Talker's stream and  enabled / disabled
     from expressing interest in _any_ advertised stream. */

void set_avb_sink_id(unsigned sink_num, unsigned stream_id[]);

void set_avb_sink_state(unsigned sink_num, avb_sink_state_t state);
```

While this demonstrated connection management was *possible*, it did not provide a full implementation of Ethernet AVB connection management as it relied heavily on the existing AVB reference design `demo()` function (see Listing 2.3). A more comprehensive implementation of stream and internal connection management is described in Section 6.6.

[4]Since the prototype parameter implementation did not yet support values above 32-bit integers, getting or setting the stream ID was not possible at this stage; however, enabling or disabling the Talker or Listener state was sufficient to interrupt and resume the audio stream connection.

One significant challenge encountered through the prototyping process was that a full-featured AES-64 protocol stack is challenging to implement within a single execution thread. For example, the sequence of interactions involved in establishing a parameter group present difficulties, as (in the case of a master-slave join) the stack must:

1. receive and parse the initial 'JOIN MSTSLV' request;
2. issue a 'GET PTPGRP' query to the slave parameter;
3. service events from the IP stack, queuing any irrelevant messages, until the expected response is received;
4. parse the expected response, return to processing the 'JOIN MSTSLV' request, and issue a series of 'SET MASTER' requests to remote parameters;
5. update the parameter group lists on the parameter targeted by the initial request;
6. finally send an acknowledgment response to the source of the initial request.

This presents two hazards: firstly, the command handler must 'adjourn' after transmitting the 'GET PTPGRP' query to service events from the IP stack until the anticipated response arrives; secondly, any request messages received by the stack between the arrival of the initial request and the dispatch of the acknowledgment response must be *queued* so that they do not disrupt the processing of the 'JOIN MSTSLV' request.

It is possible to evade these and similar hazards, but the resulting logic and code structure is unsatisfactorily complicated. The following sections describe how the task structuring of the protocol stack developed in Chapter 5 produces a more elegant and robust implementation.

6.3.5 Porting the prototype to the XMOS platform

The process of porting the prototype to the XMOS platform presented some issues, which are briefly described here before an overview of the final implementation of AES-64 device representation and message processing.

Interfacing with the XMOS IP stack

The prototype AES-64 stack was implemented as a BSD sockets-based network application. The IP stack provided by XMOS does not implement a sockets interface, but rather a low-level event-based interface [XMOS 2012c, 8-11] implemented over an XC channel. Interactions between the application and the IP stack (e.g., to receive or transmit an

IP message) are not typically atomic operations: for example, transmitting a message through the IP stack involves the following operations:

1. Application calls `xtcp_init_send()` to signal the IP stack that it intends to transmit;
2. The IP stack (eventually) responds with a `XTCP_REQUEST_DATA` event, indicating that the IP stack is ready to transmit;
3. Application catches the `XTCP_REQUEST_DATA` event and calls `xtcp_send()` to send some or all of the message data;
4. The IP stack receives the message data. It transmits the data before sending a `XTCP_SENT_DATA` event to confirm transmission *and* to request more data.
5. Application catches the `XTCP_SENT_DATA` event. If it does have more data to send, it calls `xtcp_send()` to send it. If it does not have more data, it must call `xtcp_complete_send()` to formally complete the transmit operation.

However, these and similar interactions with the XMOS IP stack may be streamlined through the use of a high-level blocking client API [XMOS 2012c, 34-35] that implements socket-style receive and transmit operations. This enabled the sockets-based prototype to be ported without substantial changes to the implementation of the AES-64 message processing and command handlers.

Resources, static analysis and optimisations for binary size

As discussed in Section 4.1, each processing tile within an XS1 microcontroller has 64KB of SRAM. The bootloader and executable binary are stored at one end of the memory, while stack variables arestored at the other end of the memory. Heap allocation takes place in the area between these two blocks.

XMOS programs undergo static analysis at build time to assess that the functions and static data can conceivably fit within the available resources[5]. If the memory usage for the total tasks assigned to any one processing tile is over 64KB, the build will fail.

Particularly in the context of implementing a stack for the AES-64 protocol, which must implement handlers for a large number of commands and which must implement value functions for a large number of parameters, this means that optimising to eliminate redundant code and reduce the size of the resulting binary is a high priority. An example of this is shown in Appendix D, where the functions that configure the AES-64 device

[5]The following two issues, 'use of recursion' and 'use of function pointers', were discovered as a result of this static analysis procedure.

representation do so by iterating over ordered arrays of data (level identifiers, parameter value formats, parameter value function identifiers, etc) rather than by making a long succession of function calls to create individual device parameters.

Use of recursion

The static analysis procedure performed at build time catalogues the memory usage of data structures and functions within the binary. As one might expect, recursion presents difficulties for the analysis tool. If it is possible to confidently estimate the maximum level of recursion, the `#pragma stackfunction` pragma directive can allocate the necessary stack space for that recursion to take place. If it is not possible, an alternative approach to the algorithm should be used.

In the case of the AES-64 prototype, a recursive algorithm was used to match a parameter request's 'full address block' against the device node's level hierarchy, which was implemented as a linked list of linked lists. This was both complex and dependent on recursion, so an alternative method of representing a device node's level hierarchy within a table structure was developed. This is described in full in Section 6.8.

Use of function pointers

The use of function pointers presents similar difficulties for the analysis tool. Here also the `#pragma stackfunction` pragma directive provides a workaround that may be adequate for many purposes. In the case of implementing AES-64 parameter value functions (which vary substantially in length and complexity), it was judged to be inadequate. In response to this, an alternate mechanism of *value function identifiers* was devised for linking AES-64 parameters to value functions. This is described in full in Section 6.6.

6.4 Integration with the Ethernet AVB reference design

Subsection 2.4.2 reviewed the standard XMOS Ethernet AVB reference design application. For the purposes of this project, the reference design application was ported to build and run on the DSP4YOU AVBStreamer hardware target [DSP4YOU 2012]. As indicated in the earlier review, the Ethernet AVB application includes a proprietary remote control service[6]. This provided a starting point for integrating the AES-64 prototype into the Ethernet AVB application, demonstrating how a simple application could be interfaced with the XMOS IP stack and Ethernet AVB subsystem.

[6]This is true of release 5.1.2 of the XMOS Ethernet AVB reference design; subsequent releases have replaced this with an implementation of IEEE 1722.1 (AVDECC).

Listings 2.1 and 2.2 showed the execution threads and communication channels for the standard Ethernet AVB application. Listings 6.8, 6.9 and 6.10 show the execution threads and communication channels for the AVB application integrated with the AES-64 implementation.

Listing 6.8: Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 1)

```
// function prototypes where they differ from the standard XMOS Ethernet
    AVB reference design
void demo(chanend tcp_svr, chanend c_rx, chanend c_tx,
    chanend c_gpio_ctl, chanend c_msg_proc[],
    unsigned int num_msg_proc_threads);

void ptp_gpio_i2c_server(chanend c_rx, chanend c_tx, chanend ptp_link[],
    int num_ptp, enum ptp_server_type server_type, chanend c,
    struct r_i2c &r_i2c);

// other declarations omitted

int main(void)
{
// chanend declarations for inter-thread communication
// ethernet server chanends
  chan tx_link[5];
  chan rx_link[4];
  chan connect_status;

// 802.1AS server chanends
  chan ptp_link[3];

// 1722 thread chanends
  chan listener_ctl[AVB_NUM_LISTENER_UNITS];
  chan buf_ctl[AVB_NUM_LISTENER_UNITS];
  chan talker_ctl[AVB_NUM_TALKER_UNITS];

// media control chanends
  chan media_ctl[AVB_NUM_MEDIA_UNITS];
  chan clk_ctl[AVB_NUM_MEDIA_CLOCKS];
  chan media_clock_ctl;

// streaming channel for audio samples
  streaming chan c_samples_to_codec;

  chan xtcp[1]; // IP stack chanends
  chan c_gpio_ctl; // general-purpose I/O (buttons)

// connects control app and the aes64 msg processing thread(s)
  chan aes64_msg_proc[NUM_MSG_PROCESSING_THREADS];

// connects aes64 msg processing thread(s) and the device node
  chan aes64_dnm_proc[NUM_MSG_PROCESSING_THREADS];
```

Listing 6.9: Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 2)

```
  par
  {
// Ethernet stack
    on stdcore[1]:
    {
      int mac_address[2];
      ethernet_getmac_otp(otp_data, otp_addr, otp_ctrl, (mac_address,
          char[]));
      phy_init(clk_smi, p_mii_resetn, smi[0], mii0);
      ethernet_server(mii0, mac_address, rx_link, 4, tx_link, 5, smi[0],
           connect_status);
    }
// IP stack
    on stdcore[1]: uip_server(rx_link[1], tx_link[2], xtcp, 1,
      null, connect_status);

// 802.1AS server
    on stdcore[1]:
    {
      DSP_RST <: 0;
      audio_clock_CS4272_init(r_i2c, MASTER_TO_WORDCLOCK_RATIO);
      p_codec_gain <: 0x0F;
      audio_codec_CS4272_init(p_codec_ctl, r_i2c, 0);
      ptp_gpio_i2c_server(rx_link[0], tx_link[0], ptp_link, 3,
          PTP_GRANDMASTER_CAPABLE, c_gpio_ctl, r_i2c);
    }

// media clock server
    on stdcore[1]: media_clock_server(media_clock_ctl, ptp_link[1],
        buf_ctl, AVB_NUM_LISTENER_UNITS, clk_ctl, AVB_NUM_MEDIA_CLOCKS);

// Audio codec server
    on stdcore[0]:
    {
      init_media_input_fifos(ififos, ififo_data, AVB_NUM_MEDIA_INPUTS);
      configure_clock_src(b_mclk, p_aud_mclk);
      start_clock(b_mclk);
// Clock signal generation and audio codec run as parallel tasks
      par
      {
        audio_gen_CS4272_clock(p_fs, clk_ctl[0]);
        i2s_master (b_mclk, b_bclk, p_aud_bclk, p_aud_lrclk, p_aud_dout,
            AVB_NUM_MEDIA_OUTPUTS, p_aud_din, AVB_NUM_MEDIA_INPUTS,
            MASTER_TO_WORDCLOCK_RATIO, c_samples_to_codec, ififos,
            media_ctl[0], 0);
      }
    }
```

Listing 6.10: Execution threads and communication channels for the AVB application integrated with the AES-64 implementation (part 3)

```
// 1722 Talker and Listener tasks
    on stdcore[0]: avb_1722_talker(ptp_link[0], tx_link[1], talker_ctl
        [0], AVB_NUM_SOURCES);

    on stdcore[0]: avb_1722_listener(rx_link[3], tx_link[4], buf_ctl[0],
        listener_ctl[0], AVB_NUM_SINKS);

// Media output server
    on stdcore[0]:
    {
      init_media_output_fifos(ofifos, ofifo_data, AVB_NUM_MEDIA_OUTPUTS);
      media_output_fifo_to_xc_channel_split_lr(media_ctl[1],
          c_samples_to_codec, 0, ofifos, AVB_NUM_MEDIA_OUTPUTS);
    }

// AES64 message processing task
    on stdcore[3]: a64_msg_proc(aes64_msg_proc[0], aes64_dnm_proc[0]);
// AES64 device representation task
    on stdcore[1]: a64_dnm(aes64_dnm_proc, NUM_MSG_PROCESSING_THREADS);

// Diagnostic logging server
    on stdcore[0]: xlog_server_uart(p_uart_tx);

// Application tasks
    on stdcore[0]:
    {
// First initialize avb higher level protocols
      avb_init(media_ctl, listener_ctl, talker_ctl, media_clock_ctl,
          rx_link[2], tx_link[3], ptp_link[2]);
// Now run primary event handler application
      demo(xtcp[0], rx_link[2], tx_link[3], c_gpio_ctl, aes64_msg_proc,
          NUM_MSG_PROCESSING_THREADS);
    }
  }
}
```

A summary of the changes made to the base software follows:

1. Two new arrays of channels, `aes64_dnm_proc` and `aes64_msg_proc`, have been allocated:
   a) `aes64_dnm_proc` connects the one and only AES-64 device representation task to the variable number of AES-64 message processing tasks;
   b) `aes64_msg_proc` connects the variable number of AES-64 message processing tasks to the one and only AES-64 stack control task (that is, `demo()`);
2. Two new tasks have been allocated to threads in the `par` block:
   a) `a64_msg_proc()`, an AES-64 message processing task ('MP task'), allocated to a thread on tile 3 (Listing 6.10 l. 14);
   b) `a64_dnm()`, an AES-64 device representation task, allocated to a thread on tile 1 (Listing 6.10 l. 16);
3. The `demo()` task has been extended to accept the new channels for communication with the AES-64 implementation.

Further changes have been made to a task originally named `ptp_server_and_gpio()` (now named `ptp_gpio_i2c_server()`) in order to support AES-64 control of the DSP4YOU hardware's audio codec (via the I^2C interface). This is reviewed in Subsection 6.10.3.

Multiple message processing instances

The AES-64 implementation can be readily configured to execute any viable number of message processing tasks. The configuration file constant `NUM_MSG_PROCESSING_THREADS` must agree with the number of `a64_msg_proc()` tasks declared within the `par` block.

The XC channel construct allows these tasks to be executed on any available processing tile, regardless of which tiles are executing the device representation and control tasks. However, two basic restrictions apply:

1. the number of free logical cores on a given processing tile;
2. the quantity of available memory on a given processing tile.

The memory constraint is more significant, in that the 64KB of RAM on each processing tile stores program code as well as stack variables and heap allocation. Running a second instance of a task on a different processing tile from the first duplicates storage of the task's program code; running a second instance of a task on the same processing tile as the first does not (Fig. 6.8).

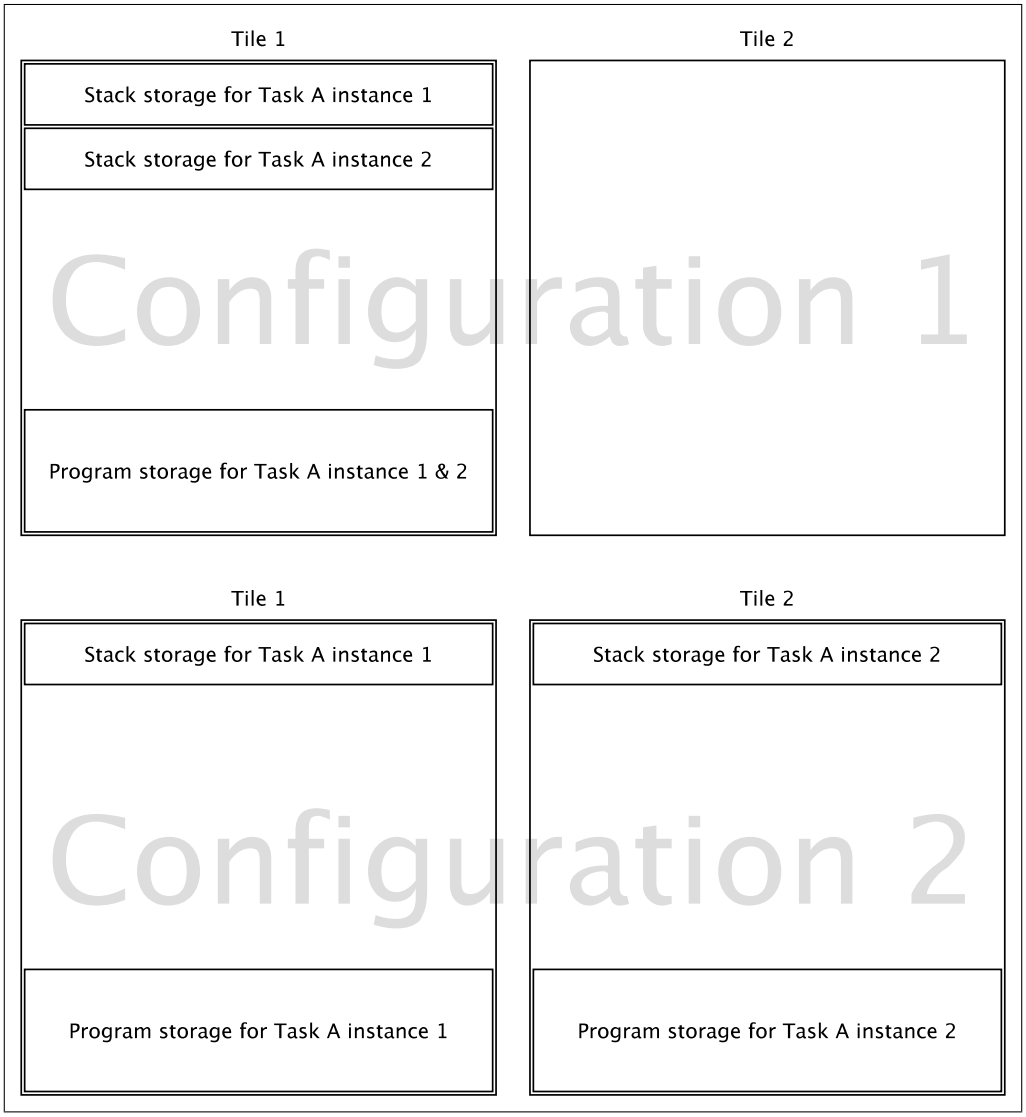


Figure 6.8: Memory overheads associated with multiple instances ('1' and '2') of a single task ('A') in two different configurations.

6.5 Task structuring within the AES-64 implementation

The component tasks of the AES-64 implementation (Fig. 6.1) closely resemble the implementation model described in Chapter 5 (Fig. 5.11), but have diverged in some respects.

The key distinctions are that:

1. the AES-64 device representation is now configured within its own task, rather than receiving its configuration from the stack control task as depicted in the implementation model (Fig. 5.11);

2. the AES-64 implementation supports multiple concurrent instances of the message processing task with minimal alterations to the other components of the stack;

3. the AES-64 'stack control task' is absorbed within the XMOS Ethernet AVB application's `demo` task, which controls all other aspects of the application's behaviour and is therefore the logical place for value functions to execute from.

6.5.1 Event flows within the AES-64 implementation

As described in Subsections 5.4.2 to 5.4.4, each task within the AES-64 implementation responds to at least one source of events:

1. the stack control task (Section 6.7) handles events from:
   a) the XMOS IP stack (inbound AES-64 UDP messages, among other events);
   b) the array of message processing tasks (AES-64 messages to send, notifications of completed or failed processing, etc).

2. the device representation task (Section 6.8) handles events from the array of message processing tasks;

3. each message processing task (Section 6.9) waits on a single configuration event from the device representation task, and then handles events from the stack control task;

All of these events consist of communications over XC 'channels', and each task waits for events to occur in a `select` statement (as described in Chapter 4). Listing 6.11 shows the top-level function for the device representation task in full: line 15 uses the 'replicator' and 'parameterised select' techniques first shown in Section 4.3 to wait fairly on events from any channels connected to message processing tasks. The `mp_request()` function

performs high-level processing of a channel event, identifies the event's specific handler, and calls it. As soon as the event has been handled, the `mp_request()` function returns and the task becomes available to process further events[7].

Listing 6.11: The top-level function for the device representation task

```
void a64_dnm(chanend c_msg_proc[], int num_mp_threads) {
        unsigned char token;
        /* load device configuration as provided by the application */
        a64_device_configuration_init();
        /* enable each of the message processing tasks */
        for (int mp_uid = 0; mp_uid < num_mp_threads; mp_uid++) {
                a64_msg_proc_api_enable_processing(c_msg_proc[mp_uid],
                    mp_uid);
        }
        /* process events from any of the MP tasks */
        while (1) {
                enum A64MPCommand mp_cmd = A64_MP_INVALID_COMMAND;
                select
                {
                case (int mp_uid = 0; mp_uid < num_mp_threads; mp_uid++)
                    mp_request(c_msg_proc[mp_uid], token, mp_cmd);
                }
        }
}
```

The stack control task is implemented by the Ethernet AVB application's `demo()` function, and consequently the task's `select` block is rather more complicated. Listing 6.12 shows an abridged version. Events from the IP stack (line 5) and events from message processing ('MP') tasks (line 15) are handled as individual cases within the select statement, while a timer is used to perform periodic processing, including the processing of any queued AES-64 messages (line 9–13).

As in the device representation task, the message processing tasks are handled by a replicator using a parameterised select function, which (after performing channel communications) calls the `a64_fs_handle_mp_event()` function shown in Listing 6.13.

[7]The processing of events by the device representation task is described from a different perspective in Section 4.6)

Listing 6.12: The select statement for the demo function (including the stack control task)

```
select {
    case avb_get_control_packet(c_rx, buf, nbytes):
        /* handling omitted */
        break;
    case xtcp_event(tcp_svr, conn):
        /* handling of non-AES-64 IP events omitted */
        a64_fs_xtcp_handler(tcp_svr, c_msg_proc, conn);
        break;
    case tmr when timerafter(timeout) :> void:
        timeout += PERIODIC_POLL_TIME;
        /* avb periodic processing omitted */
        a64_fs_periodic(tcp_svr, c_msg_proc, conn);
        break;
    case (int mp_uid = 0; mp_uid < num_msg_proc_threads; mp_uid++)
        a64_fs_mp_handler(c_msg_proc[mp_uid], tcp_svr, token, mp_cmd,
            nbytes, mp_uid, c_gpio_ctl);
}
```

Listing 6.13: Handler for the stack control task's client API

```
void a64_fs_handle_mp_event(chanend c_msg_proc, chanend tcp_svr,
                enum A64MPCommand mp_cmd, UInt32 numbytes, UInt32 mp_uid,
                    chanend c_gpio_ctl) {
        switch(mp_cmd)
        {
/* an MP task wants to transmit a message */
        case A64_MP_TX_REQUEST:
                a64_fs_handle_mp_tx_request(c_msg_proc,
                        tcp_svr, numbytes, mp_uid);
                break;
/* the MP task has been enabled by the device representation task. set MP
     task as 'available' to service any future requests */
        case A64_MP_NOTIFY_PROCESSOR_ENABLED:
                mp_states[mp_uid].state = A64_MSG_PROC_READY;
                break;
/* processing of the MP task's current job has finished (one way or
    another). set MP task as 'available' to service any future requests
    and initialise any stored sequence ID */
        case A64_MP_NOTIFY_PROCESSING_COMPLETED:
        case A64_MP_NOTIFY_PROCESSING_FAILED:
                a64_fs_handle_mp_proc_notify(mp_uid);
                break;
/* an MP task needs a sequence id to use in an outbound GET request */
        case A64_MP_GET_NEW_SEQUENCE_ID:
                a64_fs_handle_mp_get_sequence_id(c_msg_proc);
                break;
/* an MP task needs to actuate a 'get' value function */
        case A64_MP_PROCESS_GET_VALUE_FUNCTION: /*
                a64_fs_handle_mp_process_get_val_vf(c_msg_proc, numbytes,
                    mp_uid, c_gpio_ctl);
                break;
/* an MP task needs to actuate a 'set' value function */
        case A64_MP_PROCESS_SET_VALUE_FUNCTION:
                a64_fs_handle_mp_process_set_val_vf(c_msg_proc, numbytes,
                    mp_uid, c_gpio_ctl);
                break;
        default: break;
        }
}
```

6.5.2 Task communications over XC channels

Communications between the component tasks have been implemented through a lightweight protocol that sends an asynchronous command 'event' and (if it has data to send) blocks for the other end to acknowledge before sending serialised data, as previously depicted in Fig. 4.2.

As an example, Listing 6.14 shows the `a64_dnm_api_resolve_fab()` function, where a message processing task communicates with the device representation task in order to locate the parameters identified a full address block supplied by an AES-64 request. Fig. 6.9 depicts this interaction graphically.

The `a64_dnm_api_resolve_fab()` function first calls the `a64_dnm_api_send_mp_cmd()` function to asynchronously send the `A64_MP_RESOLVE_FAB` command identifier to the device representation task. The function will then block on setting up an XC channel transaction (Subsection 4.4.2), where it takes the master role (l. 5).

The device representation task will process the `A64_MP_RESOLVE_FAB` command identifier and execute the `dnm_handle_resolve_fab()` function (Listing 6.15). The `dnm_handle_resolve_fab()` function calls the `dnm_get_fab_data()` function, which completes the set up of the XC channel transaction, and this enables the AES-64 request's *target device node ID* and *full address block* to be efficiently communicated between the two tasks.

The device representation task processes the information as received and must respond to the message processing task, indicating whether or not it has succeeded in processing the request. Success is defined as locating at least one parameter. If at least one parameter has been located, the `dnm_handle_resolve_fab()` function will follow its success notification with a count of the located parameters, an indication of the size of the selected parameter bitmap, and the bitmap itself[8].

[8]The selected parameter bitmap is a variable-size array of 32-bit integers, where each bit of each integer corresponds to the unique ID of a parameter within the device representation. The array is variable-size because different device nodes within a representation will have different numbers of parameters; for example, while the 'device configuration' device node is unlikely to have more than 32 parameters, it is almost certain than other device nodes will.

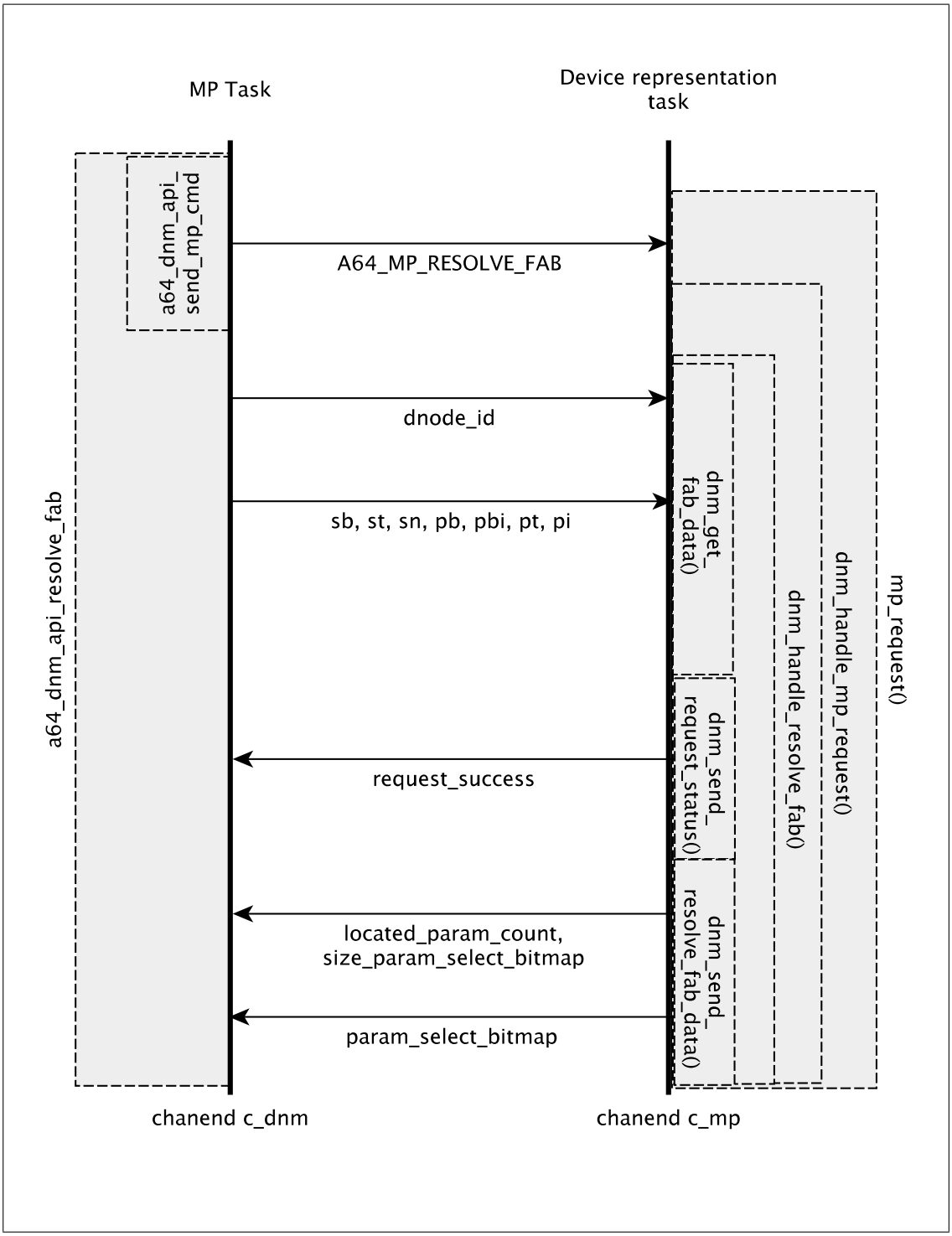


Figure 6.9: Channel communications between the message processing task and the device representation task to process and resolve a full address block supplied by an AES-64 request (see Listing 6.14).

Listing 6.14: Resolution of a full address block performed over an XC channel

```
UInt8 a64_dnm_api_resolve_fab(chanend c_dnm, UInt32 dnode_id, UInt8 sb,
    UInt8 st, UInt32 sn, UInt8 pb, UInt32 pbi, UInt32 pt, UInt32 pi,
    UInt32 &size_param_select_bitmap, UInt32 param_select_bitmap[], UInt32
     &located_param_count)
{
    UInt8 request_success = A64_DNM_REQUEST_NOK;
    a64_dnm_api_send_mp_cmd(c_dnm, A64_MP_RESOLVE_FAB);
    master {
            c_dnm <: dnode_id;
            c_dnm <: sb;
            c_dnm <: st;
            c_dnm <: sn;
            c_dnm <: pb;
            c_dnm <: pbi;
            c_dnm <: pt;
            c_dnm <: pi;
    }
    c_dnm :> request_success;
    if (request_success) {
            slave {
                    c_dnm :> located_param_count;
                    c_dnm :> size_param_select_bitmap;
                    for (int i = 0; i < size_param_select_bitmap; i++)
                            c_dnm :> param_select_bitmap[i];
            }
    }
    return request_success;
}
```

Listing 6.15: The device representation task's handler for resolving full address blocks

```
void dnm_handle_resolve_fab(chanend c_mp)
{
        struct DNode *target_dnode = NULL;
        int retval = 0;
        /* set up variables to pull data into */
        UInt32 dnode_id, located_params_count = 0;
        UInt8 sb = 0, st = 0, pb = 0;
        UInt32 sn = 0, pbi = 0, pt = 0, pi = 0;
        /* pull data in */
        dnm_get_fab_data(&dnode_id, &sb, &st, &sn, &pb, &pbi, &pt, &pi,
            c_mp);
        /* Validate target dnode */
        retval = locate_this_dnode(dnode_id, &target_dnode);
        if (retval == -1) { /* invalid dnode id */
                dnm_send_request_status(A64_DNM_REQUEST_NOK, c_mp);
                return;
        }
        /* call A64 stack function */
        retval = resolve_full_address_block(target_dnode, sb, st, sn, pb,
             pbi, pt, pi);
        if (retval == -1) { /* resolution of FAB has failed */
                dnm_send_request_status(A64_DNM_REQUEST_NOK, c_mp);
                return;
        }
        /* otherwise retval holds the located params count */
        located_params_count = retval;
        /* build response */
        dnm_send_request_status(A64_DNM_REQUEST_OK, c_mp);
        dnm_send_resolve_fab_data(located_params_count, target_dnode->
            selected_param_bitmap_size, target_dnode->
            selected_param_bitmap, c_mp);
        /* and reset selected param bitmap */
        clear_all_selected_param_bitmap(target_dnode);
}
```

As illustrated by this example, each task within the AES-64 protocol stack must implement:

1. a client API of functions that may be called to send channel events to the task;
2. a high-level event handler that intercepts all channel events and directs each type of event to a specific handler;
3. a corresponding handler for each function in the task's client API.

Appendix C provides documentation of the client APIs for each task within the AES-64 protocol stack. As a concise example, Listing 6.16 shows the client API interface for the stack control task, and Listing 6.13 shows the corresponding high-level handler implementation.

The following subsection describes the interactions between each of the tasks in the protocol stack in the course of processing an AES-64 'GET VAL' request.

Listing 6.16: The stack control task's client API

```
void a64_ctrl_api_send_mp_cmd(chanend c_ctrl, enum A64MPCommand cmd,
        UInt32 nbytes);

/* Notify stack control task that this message processing task is now
    available to service requests */
void a64_ctrl_api_notify_mp_enabled(chanend c_ctrl, UInt32 mp_uid);

void a64_ctrl_api_notify_mp_completed(chanend c_ctrl, BOOL success);

/* Retrieve a fresh sequence ID for an outbound request that needs a
    response. */
void a64_ctrl_api_get_new_sequence_id(chanend c_ctrl,
        REFERENCE_PARAM(UInt32, sequence_id));

/* Transmit a request or response message. */
void a64_ctrl_api_submit_tx_request(chanend c_ctrl,
        UInt8 tx_buffer[], UInt32 tx_length);

/* Fire a 'get' value function on the stack control task. Supply the
    parameter's value function identifiers, retrieve the length and data
    returned from the value function. */
void a64_ctrl_api_process_get_value_function(chanend c_ctrl,
        UInt8 vf_class, UInt8 vf_target, UInt8 vf_idx, UInt8 vf_subidx,
        REFERENCE_PARAM(UInt8, value_length), UInt8 value[]);

/* Fire a 'set' value function on the stack control task. Supply the
    parameter's value function identifiers and  the 'set' request's value
    length and data. Retrieve the value format, length and data of any
    response. */
UInt8 a64_ctrl_api_process_set_value_function(chanend c_ctrl,
        UInt8 vf_class, UInt8 vf_target, UInt8 vf_idx, UInt8 vf_subidx,
        UInt8 value_length, UInt8 value_data[],
        REFERENCE_PARAM(UInt8, response_valft),
        REFERENCE_PARAM(UInt8, response_length), UInt8 response_data[]);

/* After sending an outbound request that demands a response (e.g., 'GET
    PTPGRP'), wait on the stack control task to signal that a matching
    response has been received. */
void a64_ctrl_api_wait_for_response(chanend c_ctrl_app, UInt8 rsp_buf[],
        REFERENCE_PARAM(UInt32, rsp_numbytes));
```

6.5.3 Processing example: a 'GET VAL' request

This subsection provides a description[9] of the processing of this request to illustrate how the component tasks of the AES-64 implementation work together.

Fig. 6.10 depicts the communications between tasks in the course of processing a 'GET VAL' request that uses a full address block. Each of the following subsubsections gives a specific task's perspective of the processing of the request, meaning that the description given in 'Processing example: the XMOS IP stack' describes *only* the interactions depicted on the vertical dashed line marked 'XMOS IP Stack'.

All communications between tasks take place over XC channels. For the purposes of this description, these communications (including their associated data) are described as 'events'.

Processing example: the XMOS IP stack

When the 'GET VAL' request arrives as a UDP packet, the XMOS IP stack will notify its clients, which (as shown in Listing 6.12) include the AES-64 stack control task, by sending the 'XTCP RECV DATA' event.

The XMOS IP stack will receive an 'XTCP SEND DATA' event from the AES-64 stack control task when it needs to transmit the 'GET VAL' *response*.

At all other times it may handle other events.

Processing example: the AES-64 stack control task

The AES-64 stack control task handles the 'XTCP RECV DATA' event by checking that the event is relevant (i.e., addressed to the AES-64 UDP port number), and then retrieving the event data from the XMOS IP stack.

After retrieving and validating the event data, the task checks an internal representation of the protocol stack's message processing (MP) tasks to see if any are available to process the message[10].

If no MP task is available, the stack control task queues the packet. If the stack control task locates a message processing task in a 'READY' state *and* the queue is empty, it sets its representation of the task's status to 'PROCESSING' and submits the received data to the task through a 'PROCESS MESSAGE' event. If the queue is not empty, the

[9] A somewhat simplified description; error handling is not described.

[10] Message processing tasks may be marked as 'READY', 'PROCESSING', or 'WAITING' (for an anticipated response from a remote device).

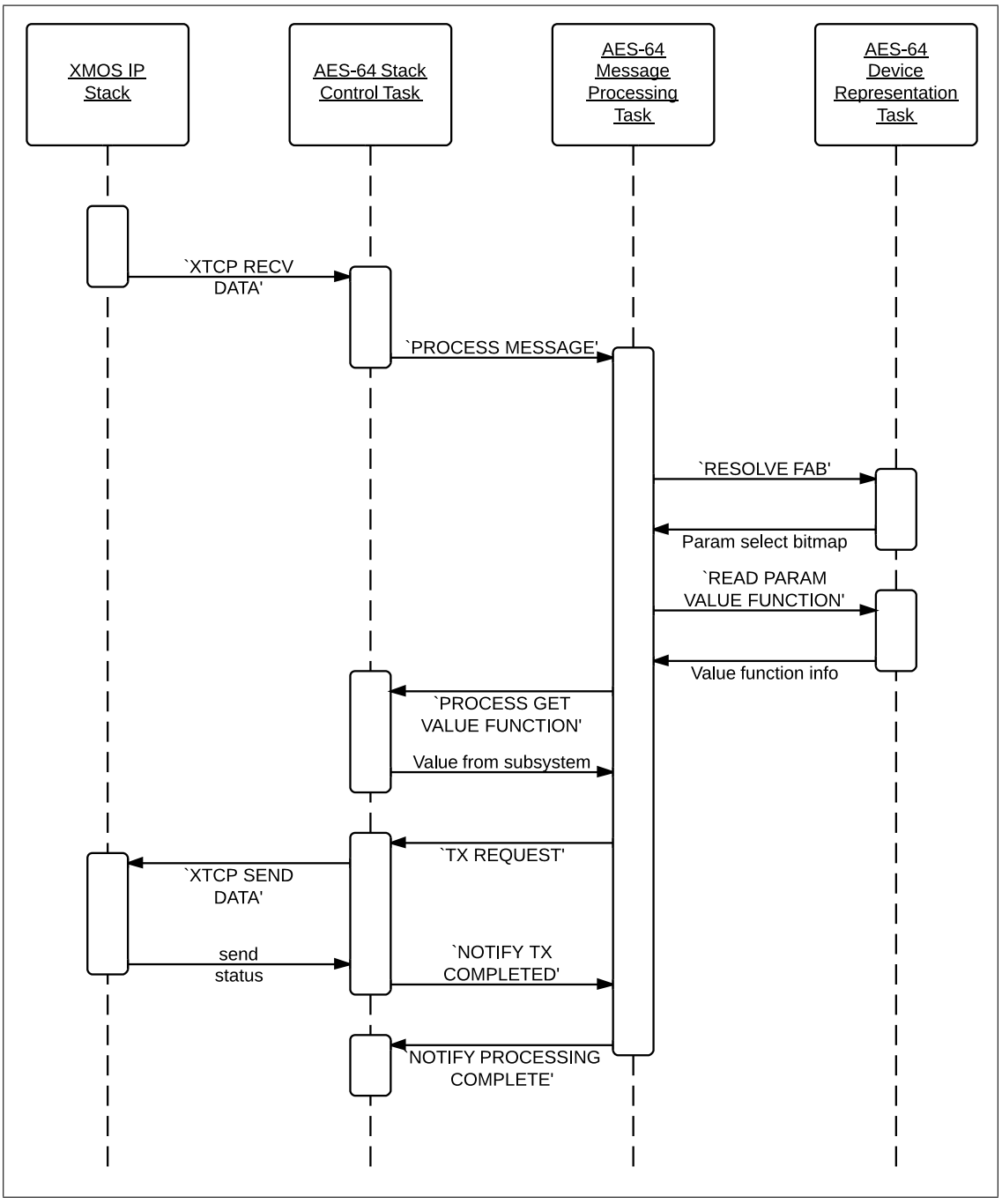


Figure 6.10: Task communications in the AES-64 protocol stack during the successful processing of a 'GET VAL' request.

stack control task pushes the new request onto the tail of the queue and submits the head of the queue for processing instead[11].

The stack control task may then handle any other events.

A short time later, the MP task submits a 'PROCESS GET VALUE FUNCTION' event to retrieve the target parameter's current value (e.g., a stream ID or channel setting) from its value function. The stack control task will locate the correct value function by the identifier information (see Section 6.6) provided with the 'PROCESS GET VALUE FUNCTION' event, execute the value function to retrieve the current value, and return the current value to the MP task.

The stack control task may then handle any other events.

A short time later, the MP task submits a 'TX REQUEST' event, signalling that it has a response message to transmit to the origin of the 'GET VAL' request. The stack control task will process this event by submitting the response message for transmission to the XMOS IP stack. Once the IP stack returns the status of the transmission, the stack control task will relay this to the MP task.

The stack control task may then handle any other events.

The MP task eventually comes to the end of the 'GET VAL' command handler. If the original request only targeted one parameter[12], the MP task will send the 'NOTIFY PROCESSING COMPLETE' event to the stack control task, which updates the task's status as 'READY'.

The stack control task may then handle any other events.

Processing example: the AES-64 message processing task

The AES-64 message processing task will handle the 'PROCESS MESSAGE' event by retrieving and validating the AES-64 request from the stack control task.

Since the request will validate as a parameter request using a full address block, the MP task will deserialise the full address block and send a 'RESOLVE FAB' ('Full Address Block') event to the device representation task. This is processed as per the example provided in Subsection 6.5.2. The device representation's task response to the 'RESOLVE FAB' event provides a *parameter select bitmap*. Each set bit identifies the index of a parameter within the targeted device node's parameter array, meaning that any further

[11] There is also a periodic task (Listing 6.12, line 18) that checks the state of the MP tasks and submits queued entries.

[12] If the original request targeted more than one parameter, the MP task will select the *next* targeted parameter and run the command handler again, repeating until there are no more parameters left to process.

queries to the device representation task can be made in terms of a 32-bit parameter ID rather than a 104-bit full address block.

Since the request validates as a 'GET VAL' request, the MP task must request the *value function* information (see Section 6.6) from the targeted parameter by submitting the 'READ PARAM VALUE FUNCTION' event to the device representation task. Once returned, the value function information can be used to submit a 'PROCESS GET VALUE FUNCTION' event to the stack control task, which should retrieve the current, live value of whatever component of the device system is represented by the targeted parameter.

Once the live value is returned, the MP task will pack this into an AES-64 response message and submit a 'TX REQUEST' event to the stack control thread for the response to be sent out to the remote host.

At this point the MP task will reach the end of the 'GET VAL' command handler. If there are no set bits remaining in the parameter select bitmap, it has processed the command on all of the targets requested by the remote host and must submit a 'NOTIFY PROCESSING COMPLETE' event to the stack control task. It may then handle any other events.

Processing example: the AES-64 device representation thread

The AES-64 device representation thread will handle the 'RESOLVE FAB' query from the MP task by validating first the target device node ID and then the full address block. It will return a parameter select bitmap as long as at least one parameter is successfully identified by the request's full address block.

It may then handle any other events.

Some time later, it will handle other queries or commands from the MP task. In the case of the 'GET VAL' command handler, it will handle a 'READ PARAM VALUE FUNCTION' event, which supplies the 32-bit unique ID of the targeted parameter, and return a unique set of four value function identifiers (described in Section 6.6, below) belonging to the targeted parameter.

It may then handle any other events.

6.6 Value functions

This implementation's introduction of *value function identifiers* addresses two problems:

1. Use of function pointers is complicated by the XMOS platform's requirement to

perform analysis of the binary at build time, as discussed in Subsection 6.3.5;

2. Some control functions can only be called from a thread which has declared communication channels to the relevant subsystem. This is the case, for example, with the XMOS Ethernet AVB subsystem API calls shown in Listing 6.7, which need to be called from the thread which initialises the subsystem (i.e., the one which executes the `demo()` function).

Value function identifiers allow an AES-64 parameter to hold four constant values that define exactly what device function the parameter's value represents. This information may be retrieved by a message processing thread, which may then send a 'PROCESS GET VALUE FUNCTION' or 'PROCESS SET VALUE FUNCTION' event to the stack control task to retrieve or set a new value from/on the device's primary control software.

6.6.1 Value functions and parameters

Each and every parameter on a device node has a unique set of four hierarchical value function identifiers, which together identify precisely what control or attribute of the device's primary control software is associated with the parameter:

1. `vf_class` (Listing 6.17) identifies the approximate type of control or attribute that the parameter's value function addresses;

2. `vf_target` identifies the parameter value function's control or attribute within the stated class — for example, valid targets for the *VFN CLASS NETWORK* class include *NETWORK IP ADDRESS* and *NETWORK SUBNET MASK*;

3. `vf_index` identifies the *instance* of a target that a parameter's value function addresses — this enables parameters to represent multiple network interfaces, multiple Talker units, multiple audio channels, etc;

4. `vf_subindex` identifies, for a few cases, an index within an instance of a target of a particular class. This was specifically devised for Ethernet AVB internal connection management, where connections between a Talker/Listener unit and the XMOS device's audio channels are represented by an array [XMOS 2011, 58, 62], and AES-64 parameters are required to represent individual elements (that is, connections) within the array.

Table 6.1 shows a small set of AES-64 Ethernet AVB device parameters and their value function identifiers. The full set of parameters and their functions are discussed in Section 6.10.1.

Listing 6.17: Value function classes

```
enum A64ValueFunctionClass {
2    VFN_CLASS_INVALID, VFN_CLASS_NETWORK, VFN_CLASS_DEVICE,
        VFN_CLASS_ETHERNET_AVB_SOURCE, VFN_CLASS_ETHERNET_AVB_SINK,
        VFN_CLASS_CS4272_AUDIO, VFN_CLASS_EXPERIMENTAL
};
```

Table 6.1: Example parameters and their value functions

Parameter	VF class	VF target	VF idx	VF subidx
Input / Stream / 1 / Multicore / 1 / Stream ID / 1	AVB SINK	SINK STREAM ID	1	1
Input / Stream / 1 / Multicore / 1 / Listen / 1	AVB SINK	SINK LISTEN	1	1
Input / Stream / 1 / Multicore / 1 / Channel Map / 1	AVB SINK	SINK MAP	1	1
Input / Stream / 1 / Multicore / 1 / Channel Map / 8	AVB SINK	SINK MAP	1	8
Output / Stream / 1 / Multicore / 1 / Stream ID / 1	AVB SOURCE	SOURCE STREAM ID	1	1
Output / Stream / 1 / Multicore / 1 / Advertise / 1	AVB SOURCE	SOURCE ADVERTISE	1	1
Output / Stream / 1 / Multicore / 1 / Channel Map / 1	AVB SOURCE	SOURCE MAP	1	1
Output / Stream / 1 / Multicore / 1 / Channel Map / 8	AVB SOURCE	SOURCE MAP	1	8

6.6.2 Value functions and the stack control task

Listing 6.13 shows the high-level event handler for the stack control task's client API, which is responsible for handling all commands that a message processing task is capable of sending to the stack control task. Lines 25 and 28 show the handlers for value function requests. A value function request can perform either a 'get' or a 'set' operation. The value function identifiers provide a straightforward way to identify the value function corresponding to a given AES-64 parameter. Listing 6.18 shows switching on the `vf_class` identifier to pass execution to a specific 'selector' function.

Each 'selector' function (e.g., `a64_network_vf_selector()` (line 17), `a64_device_vf_selector` (line 20)) process the `vf_target` identifier to find a *specific* function within the class, while `vf_idx` and `vf_subidx` identify specific instances (e.g., in the case of a device with multiple Talkers, these values identify *which* Talker the AES-64 parameter represents).

For example, the value function corresponding to a Listener's 'Stream ID' parameter is shown in Listing 6.19.

Subsection 6.10.1 discusses the value functions implemented to provide AES-64 control of the Ethernet AVB Talker and Listener functionality. Subsection 6.10.3 describes the implementation of AES-64 control over the DSP4YOU hardware's CS4272 audio codec, and the value functions that enable this.

Listing 6.18: The stack control task's handler for 'get' value function requests from the message processing task

```
/* Handle a 'get' value function request from a MP task */
void a64_fs_handle_mp_process_get_val_vf(chanend c_mp, int numbytes, int
    mp_uid, chanend c_gpio_ctl)
{
        enum A64ValueOp value_op = VAL_OP_GET_VALUE;
        enum A64ValueFunctionClass vf_class = VFN_CLASS_INVALID;
        UInt8 vf_target = 0, vf_idx = 0, vf_subidx = 0;
/* get value function identifiers from a MP task */
        a64_ctrl_get_get_value_function_data(c_mp, (UInt8 *) &vf_class, &
            vf_target,
                &vf_idx, &vf_subidx);
/* switch on 'class' identifier and call a subsidiary function to select
    on the target identifier */
        switch (vf_class)
        {
        case VFN_CLASS_NETWORK:
                a64_network_vf_selector(c_mp, value_op, vf_target, vf_idx
                    , vf_subidx, NULL, 0);
                break;
        case VFN_CLASS_DEVICE:
                a64_device_vf_selector(c_mp, value_op, vf_target, vf_idx,
                     vf_subidx, NULL, 0);
                break;
        case VFN_CLASS_ETHERNET_AVB_SOURCE:
                a64_ethernet_avb_source_vf_selector(c_mp, value_op,
                    vf_target, vf_idx, vf_subidx, NULL, 0);
                break;
        case VFN_CLASS_ETHERNET_AVB_SINK:
                a64_ethernet_avb_sink_vf_selector(c_mp, value_op,
                    vf_target, vf_idx, vf_subidx, NULL, 0);
                break;
        case VFN_CLASS_ETHERNET_AVB_MEDIA_CLOCK:
               a64_ethernet_avb_media_clock_vf_selector(c_mp, value_op,
                   vf_target, vf_idx, vf_subidx, NULL, 0);
               break;
        case VFN_CLASS_EXPERIMENTAL:
                a64_experimental_vf_selector(c_mp, value_op, vf_target,
                    vf_idx, vf_subidx, NULL, 0);
                break;
        case VFN_CLASS_CS4272_AUDIO:
                a64_cs4272_audio_vf_selector(c_mp, value_op, vf_target,
                    vf_idx, vf_subidx, NULL, 0, c_gpio_ctl);
                break;
        case VFN_CLASS_INVALID:
        default: break;
        }
}
```

Listing 6.19: The value function handler for an Ethernet AVB Listener's 'Stream ID' parameter

```
void a64_ethernet_avb_sink_stream_id_vf(chanend c_mp,
                enum A64ValueOp value_op,
                UInt8 vf_index, UInt8 *value, UInt8 value_len)
{
        UInt8 retval[8] = {0};
/* Stream ID is 64 bits long */
        unsigned stream_id[2] = {0};
        if (VAL_OP_GET_VALUE == value_op) {
/* Call XMOS Ethernet AVB API function */
                get_avb_sink_id(vf_index, stream_id);
                PACK_BSTR_UINT32(retval, stream_id[0]);
                UInt8 *tmp = &(retval[4]);
                PACK_BSTR_UINT32(tmp, stream_id[1]);
/* Send stream ID data back to MP task */
                a64_ctrl_send_value_function_data(c_mp, 8, retval);
        } else if (VAL_OP_SET_VALUE == value_op) {
                UInt8 *tmp2 = value;
                UNPACK_BSTR_UINT32((stream_id[0]), tmp2);
                tmp2 = &(value[4]);
                UNPACK_BSTR_UINT32((stream_id[1]), tmp2);
/* Call XMOS Ethernet AVB API function */
                set_avb_sink_id(vf_index, stream_id);
/* Send 'set' value status back to MP task */
                a64_ctrl_send_set_value_function_data(c_mp,
                        A64_SET_VAL_STATUS_NO_ERROR, 0);
        } else {
/* This shouldn't ever happen */
                a64_ctrl_send_value_function_no_data(c_mp);
        }
}
```

6.7 The stack control task

This subsection discusses the AES-64 protocol stack control task. Fully integrated into the Ethernet AVB `demo()` control task, this task has two major responsibilities:

1. coordinating input-output of AES-64 messages into/from the AES-64 stack;
2. firing 'value functions' to exert AES-64 control over the Ethernet AVB device primary control software.

Subsection 6.6.2 describes the purpose and implementation of AES-64 value functions. The message co-ordination responsibilities of the stack control task are described below.

6.7.1 Coordinating input-output of AES-64 messages

As indicated by Fig. 6.1, all AES-64 messages enter or leave the protocol stack via the stack control task. This was briefly described in Subsection 6.5.3, which demonstrated the processing of a 'GET VAL' request message. The stack control task receives inbound AES-64 messages from the IP stack and selectively queues or submits them to available message processing tasks. The stack control task transmits outbound AES-64 messages onto the network when message processing tasks send the 'TX REQUEST' event.

To support the full messaging requirements of the AES-64 standard, as described in Subsection 5.2.3, the stack control task has to additionally account for:

1. a message processing task sending an outbound request message that anticipates a future response;
2. an inbound message that is a response to an earlier outbound request message.

Outbound request messages that anticipate a future response

When a message processing task sends a 'TX REQUEST' event and a message for transmission, the stack control task must check the outbound message's type. If the outbound message is a request that anticipates a response, the stack control task must register the message processing task as 'WAITING' and log the sequence ID of the outbound message before transmission.

Inbound responses to earlier requests

The stack control task immediately submits an inbound message to a message processing task if one is available. If there are no available message processing tasks, the message is

queued. The order of arrival is preserved, so that if a new message arrives when a message processing task is available and there is an earlier message waiting in the queue, the new message is queued and the earlier message is dequeued and submitted for processing.

The stack control task must make an exception for inbound *response* messages, such as the response anticipated by the 'GET PTPGRP' request made during setup/teardown of parameter groups. A message processing task is halted waiting for this response, so the response must be *immediately* delivered, bypassing the inbound queue, to the task that has registered a 'WAITING' state and the matching sequence ID.

6.8 The device representation task

This subsection discusses the core implementation of device representation in the multi-threaded AES-64 protocol stack. The final implementation of device representation in the AES-64 protocol stack is based around device nodes (Listing 6.20 and 6.21), parameters (Listing 6.22), and a table-based representation of the level hierarchy: the *level items table* (Listing 6.23).

6.8.1 The device node

The AES-64 device node is shown in Listings 6.20 and 6.21. Both the level hierarchy and the parameter array are now stored in table structures. The device node also has a *selected parameter bitmap* which is used to identify the parameters targeted by any request that addresses the device node. The implementation and use of the selected parameter bitmap is described in detail in Section 6.9; it enables the processing of requests that target multiple parameters. The device node provides functions to mark, clear, search and reset the selected parameter bitmap.

Listing 6.20: The AES-64 device node (part 1)

```
struct DNode {
        UInt32 id;
        struct LevelItemsTable *level_items_table; /* level hierarchy */
        struct DNode *next_dnode;
        struct ParamTable *param_table; /* parameter storage */
        UInt32 *selected_param_bitmap;
        unsigned int selected_param_bitmap_size;
};

struct DNode *dnode_create(UInt8 max_params);
void dnode_destroy(struct DNode *destroy_this);

/* The device representation task maintains the device representation as
    a linked list of device node structs */
int append_to_global_dnodes(struct DNode **head_ref, struct DNode **
    append_this);
UInt8 length_global_dnodes();
struct DNode **get_global_dnodes();
void debug_global_dnodes(struct DNode *list);
void destroy_global_dnodes(struct DNode *list);
struct DNode *get_last_dnode();

/* add a parameter to device node storage */
void add_param_to_dnode_param_array(struct DNode *dnode, struct Param *
    add_this);

/* locate a supplied device node id in the set of all device nodes */
int locate_this_dnode(UInt32 id_to_find, struct DNode **located_dnode);

/* locate a parameter on a specific device node by its index */
int locate_param_on_this_dnode(struct DNode *dnode, UInt32
    parameter_index, struct Param **located_param);

/* resolve a full address block on a specific device node */
int resolve_full_address_block(struct DNode *dnode, UInt8 sb, UInt8 st,
    UInt32 sn, UInt8 pb, UInt32 pbi, UInt32 pt, UInt32 pi);

/* resolve a level identifier from a full address block against an entry
    in the device node's level hierarchy */
int evaluate_supplied_id(UInt32 id, enum LevelPosition this_level, struct
     LevelItemsTableEntry *entry);
```

Listing 6.21: The AES-64 device node (part 2)

```
/*
 * Parameter location and selection via the device node
     param_select_bitmap
 */

/* Mark a parameter as 'located' on the device node's selection bitmap */
void set_on_selected_param_bitmap(struct DNode *this_dnode, UInt32
    param_id);

/* Clear a previously-marked parameter from the selection bitmap */
void clear_on_selected_param_bitmap(struct DNode *this_dnode, UInt32
    param_id);

/* Search for set bits in the bitmap, clear the first set bit found and
    return its position in the bitmap */
void find_param_index_and_clear_from_bitmap(UInt32 *bitmap, UInt32 *
    param_id);

/* Get 'selected parameters count' from the bitmap */
int count_set_bits_in_selected_param_bitmap(struct DNode *this_dnode);

int is_selected_param_bitmap_clear(struct DNode *this_dnode);

void clear_all_selected_param_bitmap(struct DNode *this_dnode);
```

6.8.2 The parameter and parameter table

The AES-64 parameter implementation is shown in Listing 6.22. This implements a large number of the attributes described in the published AES-64 standard, including a fully-functional implementation of parameter grouping and the *value function identifiers* described in Section 6.6. There is also a *parameter table* structure, which encapsulates the parameter array required by AES-64 device nodes.

The value function identifiers are new to this implementation of AES-64 and are described in detail in Section 6.6. The parameter group lists are implemented as described in [Audio Engineering Society 2012, 31-35] and Appendix A.

6.8.3 The level items table

The level items table is a representation of the AES-64 level hierarchy that stores a unique entry for each parameter on a device node. It was developed as a replacement for the prototype level hierarchy implementation in response to the problem described in Subsection 6.3.5. In light of this problem, it was important that the level hierarchy could be structured in a way that would enable full address blocks to be processed without a recursive parser.

The level items table is a sorted array of entries that each individually provide the exact location, including level aliases, of the device node's parameters (Listing 6.23). Each parameter within a device node is indexed at the same position in both the parameter table and level item table, meaning that on a given device node, level item table entry #1 corresponds to parameter table entry #1.

The level items table is 'sorted' in that similar parameters must be grouped as adjacent entries: e.g., parameters belonging to the 'INPUT SIGNAL' section block must be stored in the level items table as one unbroken block (Appendix D). This enforced ordering provides an intuitive organisation of the device node's parameters and enables an efficient matching algorithm for resolving full address blocks.

Listing 6.22: The AES-64 parameter table and parameter implementation

```
/*
 * Parameter storage on device nodes
 */
struct ParamTable {
        UInt8 max_entries;
        UInt8 count;
        struct Param **params;
};

struct ParamTable *init_param_table(UInt8 max_entries);

int add_to_param_table(struct ParamTable *table,
        struct Param **new_param);

/*
 * Parameter implementation
 */
struct Param {
        UInt32 id;
        char *name;
        struct DNode *parent_dnode;
        UInt8 valft;
        UInt32 value;
        /* parameter group lists */
        struct GroupList *peers;
        struct GroupList *masters;
        struct GroupList *slaves;
        /* value function identifiers */
        UInt8 vfn_class;
        UInt8 vfn_target;
        UInt8 vfn_target_idx;
        UInt8 vfn_target_subidx;
};

/* Creates parameter instance and adds it to device node's parameter
    table */
int add_parameter_to_dnode(struct DNode *parent_dnode, char *name, UInt8
    valft, UInt32 value, UInt8 value_function_class, UInt8
    value_function_target, UInt8 value_function_target_index, UInt8
    value_function_target_subindex);

void free_parameter(struct Param *free_this);
```

Listing 6.23: The AES-64 level items table

```
/* Represents the level hierarchy entries for a single unique AES-64
    parameter */
struct LevelItemsTableEntry {
        enum SectionBlock sb;
        char *sb_alias;
        enum SectionType st;
        char *st_alias;
        UInt32 sn;
        enum ParamBlock pb;
        char *pb_alias;
        UInt32 pbi;
        enum ParamType pt;
        char *pt_alias;
        UInt32 pi;
};

/* creates an entry */
struct LevelItemsTableEntry *create_level_items_table_entry(enum
    SectionBlock sb, char *sb_alias, enum SectionType st, char *st_alias,
    UInt32 sn, enum ParamBlock pb, char *pb_alias, UInt32 pbi, enum
    ParamType pt, char *pt_alias, UInt32 pi);

/*
 * Level hierarchy storage on device node
 */
struct LevelItemsTable {
        UInt8 count;
        UInt8 max_entries;
        struct LevelItemsTableEntry **level_items;
};

struct LevelItemsTable *init_level_items_table(UInt8 max_entries);

void destroy_level_items_table(struct LevelItemsTable *level_items);

int add_to_level_items_table(struct LevelItemsTable *level_items, struct
    LevelItemsTableEntry **li_entry);
```

6.9 The message processing task

This subsection discusses the core implementation of the AES-64 message processing (MP) task, focussing on its implementation of message parsing, command handlers and the messaging patterns discussed in Subsection 5.2.3.

The MP task implements a structure (Listing 6.24) to hold important information about a processing job. One instance of this structure is maintained by each MP task, and the instance is reset whenever the MP task sends a 'PROCESSING FAILED' or 'PROCESSING COMPLETED' event to the stack control task.

When the stack control task submits an AES-64 message to the MP task, it is written to the MP task structure's *input buffer*. When the MP task needs to send one or more AES-64 messages via the stack control task, the message is written to the MP task structure's *output buffer*. The *count* and *location* of parameters captured by any request are saved in the MP task structure's remaining attributes.

6.9.1 Parsing of AES-64 request messages

Once the message processing task has received a request message from the stack control task, it must first determine:

1. what type of request it is;
2. what type of target it provides.

The implementation of this is shown in Listing 6.25.

The message processing task recognises six forms of AES-64 request (Table 6.2). Processing of five of these forms is based on the 'Message Type' field. In the case of a parameter request that uses a full address block *and* expects a response, we also have to account for the 'GET CLA' and 'GET NAME' requests[13], which actually target one or more *level items*. Parameter requests that use a full address block and expect a response are therefore additionally checked for their 'Command Qualifier' field (Listing 6.25, l. 8–18).

As previously discussed, the target of an AES-64 request may be a device node, a single parameter on a device node targeted by its unique ID, or one or more parameters on a device node targeted by a full address block. If the target cannot be validated against the AES-64 stack's device representation, the message processing task will abort processing the request. If the target validates successfully, the message processing task

[13]The 'CLA' and 'NAME' qualifiers are only ever supplied with a 'GET' command executive (see Subsection A.8.3 and A.8.7).

will proceed to processing the request's command against the request's target(s). This is achieved, as with validation, by communicating with the AES-64 device representation task.

Table 6.2: Six forms of AES-64 request

Message Type field value	Message
0x0	Parameter request providing a full address block target and expecting a response
0x1	Parameter request providing a full address block target, not expecting a response
0x2	Parameter request providing a parameter index target and expecting a response
0x3	Parameter request providing a parameter index target, not expecting a response
0x9	Device request expecting a response
0xA	Device request, not expecting a response

Listing 6.24: The message processing structure and functions

```
/* MAXBUFLEN is a configuration file constant based on an estimate of the
     largest AES-64 message seen during forensic capture and requirements
    analysis. MAX_PARAMS is a configuration file constant based on the
    number of parameters held by the largest device node configured by the
     application. */
struct A64MessageProcessor {
        unsigned char mp_input_buffer[MAXBUFLEN + 1];
        UInt32 mp_input_length;
        unsigned char mp_output_buffer[MAXBUFLEN + 1];
        UInt32 mp_output_length;
        UInt32 mp_located_param_count;
        UInt32 mp_selected_param_bitmap[(MAX_PARAMS / 32) + 1];
};

struct A64MessageProcessor *initialise_a64_mp();

/* performed whenever the message processing task completes processing a
    request */
void reset_a64_mp();

/* extracts a request's full address block to a set of individual level
    identifiers */
int mp_extract_full_address_block(struct A64_FAB_HDR *fab, UInt8 *sb,
    UInt8 *st, UInt32 *sn, UInt8 *pb, UInt32 *pbi, UInt32 *pt, UInt32 *pi)
    ;

/* processes a full address block parameter request */
int mp_param_request_fab(chanend c_ctrl_app, chanend c_dnm, struct
    A64MessageProcessor *mp, enum A64MsgType msg_type);

/* processes an indexed parameter request */
int mp_param_request_idx(chanend c_ctrl_app, chanend c_dnm, struct
    A64MessageProcessor *mp, enum A64MsgType msg_type);

/* processes a device request */
int mp_device_request(chanend c_ctrl_app, chanend c_dnm, struct
    A64MessageProcessor *mp, enum A64MsgType msg_type);

/* processes a level request (e.g., 'GET CLA', 'GET NAME') */
int mp_level_request(chanend c_ctrl_app, chanend c_dnm, struct
    A64MessageProcessor *mp);
```

Listing 6.25: Basic message parsing in the AES-64 stack

```
1       enum A64MsgType msg_type =  a64_msg_proc_mp->mp_input_buffer[
            A64_MSG_TYPE_HDR_OFFSET];
        switch (msg_type)
3       {
        case A64_PARAM_REQ_FAB_WITH_RESPONSE:
5               {
                enum A64CmdQual cq =
7                       a64_msg_proc_mp->mp_input_buffer[(
                            A64_SEQ_ID_HDR_OFFSET +
                            A64_SEQUENCE_ID_HDR_LENGTH + 1)];
                switch (cq)
9               {
                case CQ_NAME:
11              case CQ_CLA:
                        status = mp_level_request(c_ctrl_app, c_dnm,
                            a64_msg_proc_mp);
13                      break;
                default:
15                      /* process a param request */
                        status = mp_param_request_fab(c_ctrl_app, c_dnm,
                            a64_msg_proc_mp, msg_type);
17                      break;
                }
19              }
                break;
21      case A64_PARAM_REQ_FAB_NO_RESPONSE:
                status = mp_param_request_fab(c_ctrl_app, c_dnm,
                    a64_msg_proc_mp, msg_type);
23              break;
        case A64_PARAM_REQ_IDX_WITH_RESPONSE:
25      case A64_PARAM_REQ_IDX_NO_RESPONSE:
                status = mp_param_request_idx(c_ctrl_app, c_dnm,
                    a64_msg_proc_mp, msg_type);
27              break;
        case A64_DEV_REQ_WITH_RESPONSE:
29      case A64_DEV_REQ_NO_RESPONSE:
                status = mp_device_request(c_ctrl_app, c_dnm,
31                      a64_msg_proc_mp, msg_type);
                break;
33      default:
                break;
35      }
```

6.9.2 Accessing the device representation

The message processing task performs read/write operations on device nodes and parameters by communicating with the device representation task via an API, as shown in Section C.4. In this context, the message processing task specifies the targets of these operations in terms of their unique 32-bit identifiers (Listing 6.26), as many AES-64 requests do (Table 6.2).

Listing 6.26: Excerpt of device representation task API showing operation targets

```
// excerpt from a64_dnm_client_api.h

/*
 * Read attributes from parameter target. Returns success of request and
     places attribute in reference parameters.
 */
UInt8 a64_dnm_api_read_param_valft(chanend c_dnm, UInt32 dnode_id, UInt32
     param_id, REFERENCE_PARAM(UInt8, valft));

UInt8 a64_dnm_api_read_param_gu_value(chanend c_dnm, UInt32 dnode_id,
    UInt32 param_id, REFERENCE_PARAM(UInt32, gu_value));

UInt8 a64_dnm_api_read_param_flags(chanend c_dnm, UInt32 dnode_id, UInt32
     param_id, UInt32 flags[]);

UInt8 a64_dnm_api_read_param_peer_count(chanend c_dnm, UInt32 dnode_id,
    UInt32 param_id, REFERENCE_PARAM(UInt8, peer_count));

UInt8 a64_dnm_api_read_param_peer_entry(chanend c_dnm, UInt32 dnode_id,
    UInt32 param_id, UInt8 entry_idx, UInt32 peer_entry[]);
```

For requests that specify one or more parameter targets with a full address block, the message processing task must first resolve the full address block to one or more unique 32-bit identifiers, determining:

- how many parameters are targeted, and
- which parameters are targeted.

This process was introduced in Subsection 6.5.2 as an illustration of inter-task communications. It is now described in detail.

6.9.3 Resolution of full address blocks

The actual parsing of a request full address block is performed by the device representation task, but the process is initiated by a message processing task, which sends a copy of the request's target device node and target full address block to be validated (Fig. 4.2).

The device representation task parses the full address block against its level items table one identifier at a time, from the most general identifier ('Section Block') to the most specific ('Parameter Index') (Fig 6.11). Entries on the level items table that match the current identifier from the request full address block are then evaluated against the *next* identifier in the full address block; entries that don't match the current identifier are not evaluated for any remaining identifiers. Each entry that matches the current identifier is marked on the device node's parameter select bitmap[14]; entries that do not match are cleared from the parameter select bitmap.

If the device representation task parses the full address block and selects at least one parameter, it sends the count of selected parameters and the parameter select bitmap to the message processing task. If the device representation task parses the full address block and fails to select any parameters, it sends an error code to the message processing task, which must then abort processing the request.

Once the message processing task has received the count of selected parameters and the parameter select bitmap, it will process the parameter select bitmap, recovering each selected parameter's identifier by locating and clearing set bits in the bitmap until none remain.

Alternate forms of processing the level hierarchy (as required by 'GET CLA' and 'GET NAME' requests) are also supported by this implementation.

6.9.4 Support for messaging patterns

The message processing task can send AES-64 requests or responses to the stack control task at any point in the processing of a inbound message. It must simply write the outbound request or response to its *output buffer* and send the stack control task a 'TX REQUEST' event, followed by the buffered message data (Listing 6.27). Implementing the AES-64 messaging patterns is therefore straightforward: a command handler can simply iterate over the MP task's output buffer and call `a64_ctrl_api_submit_tx_request()` until the required multiple responses or follow-on requests have been sent.

Requests that require responses are transmitted in the same way, with the distinction that the stack control task will record that a response is anticipated in the future, the specific MP task that is waiting for it, and the unique sequence ID that should be attached to the response. Section 6.7 describes how this is achieved.

Once a request that requires a response has been transmitted, the message processing

[14] Each bit within the parameter select bitmap corresponds to a parameter's unique 32-bit identifier; the bitmap dynamically expands if any device node within the AES-64 protocol stack holds more than 32 parameters.

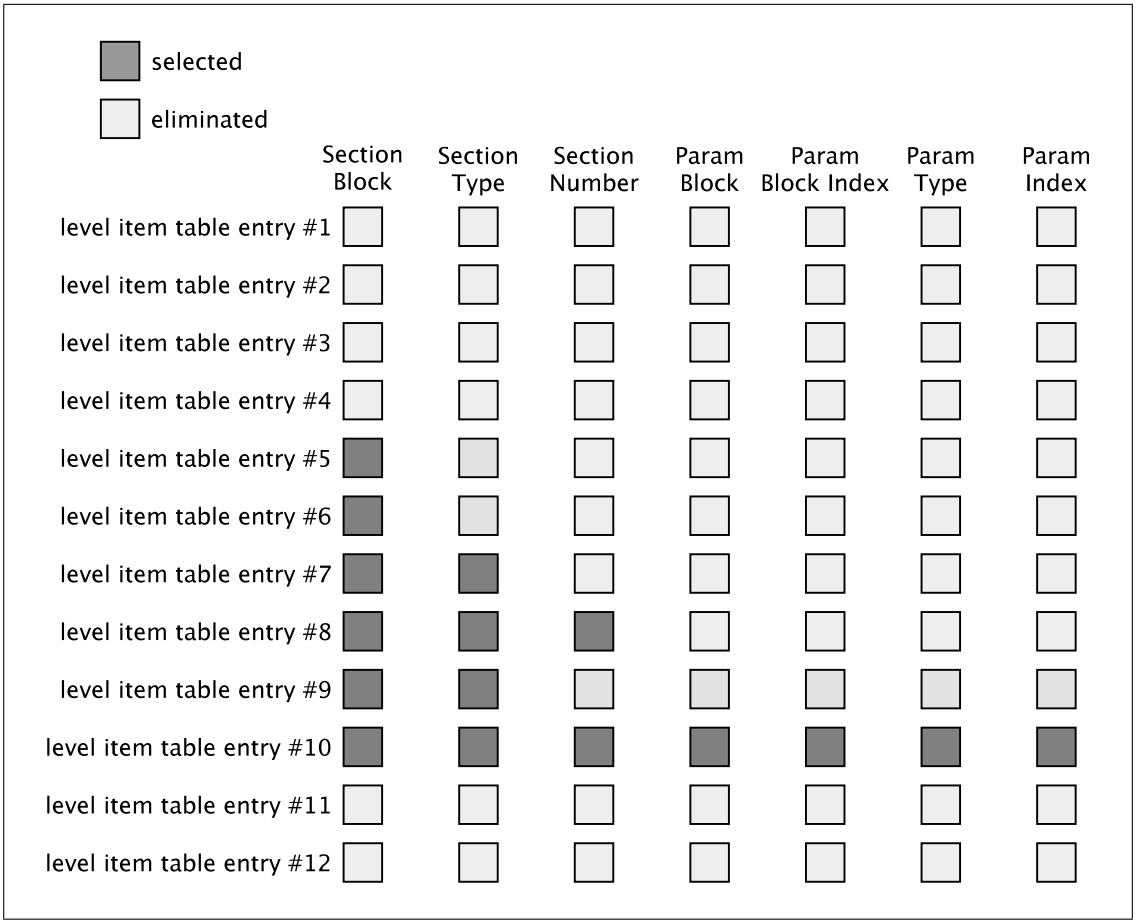


Figure 6.11: Selection and elimination of level item table entries as successive identifiers in a request full address block are parsed, resulting in one level item table entry (and parameter), #10, being identified as the target of the request.

Each identifier is checked against the range of level item table entries that matched the previous identifier (in the case of the Section Block identifier, that matched the device node ID), so that:

Section Block: checked against entry #1 — #12
Section Type: checked against entry #5 — #10
Section Number: checked against entry #7 — #10
Parameter Block: checked against entry #8 — #10
all subsequent identifiers: checked against entry #10

Listing 6.27: Implementation of transmit request via the stack control task

```
void a64_ctrl_api_submit_tx_request(chanend c_ctrl, UInt8 tx_buffer[],
    UInt32 tx_length)
{
        int tx_status = 0;
        a64_ctrl_api_send_mp_cmd(c_ctrl, A64_MP_TX_REQUEST, tx_length);
        master {
                for (int i = 0; i < tx_length; i++)
                        c_ctrl <: tx_buffer[i];
        }
        /* Now wait on the TX completed status from ctrl app */
        c_ctrl :> tx_status;
}
```

task may wait for the stack control task to deliver the expected response by calling the `a64_ctrl_api_wait_for_response()` API function (Section C.2).

6.10 AES-64 device control

Section 6.6 discussed the implementation of value functions as the linking mechanism between AES-64 parameter values and the device functions they represent. This section describes the application of this mechanism in three areas of the Ethernet AVB device: the Ethernet AVB Talker and Listener functionality, the Ethernet AVB media clock server, and the audio codec functionality.

6.10.1 AES-64 control of the Ethernet AVB Talker and Listener

Fig. 6.12 and 6.13 show the AES-64 representations of a single XMOS Talker unit and a single XMOS Listener unit respectively. Since the standard XMOS Ethernet AVB reference design is configured by default to provide one Talker unit and one Listener unit, the AES-64 representation is configured to do the same; however, if the XMOS Ethernet AVB reference design is configured[15] to provide a different number of Talker or Listener units, the AES-64 representation will automatically add or remove blocks of parameters, each block indexed at the 'Parameter Block Index' identifier. The listing in Appendix D shows how the device representation can be configured to achieve this: on lines 153 and 203, the functions creating the 'Listener' and 'Talker' parameter blocks will create as many blocks as defined by the reference design's master configuration file.

[15] By the XMOS Ethernet AVB reference design's `avb_conf.h` file [XMOS 2011, 42-3], which provides the `AVB_NUM_SINKS` and `AVB_NUM_SOURCES` constants.)

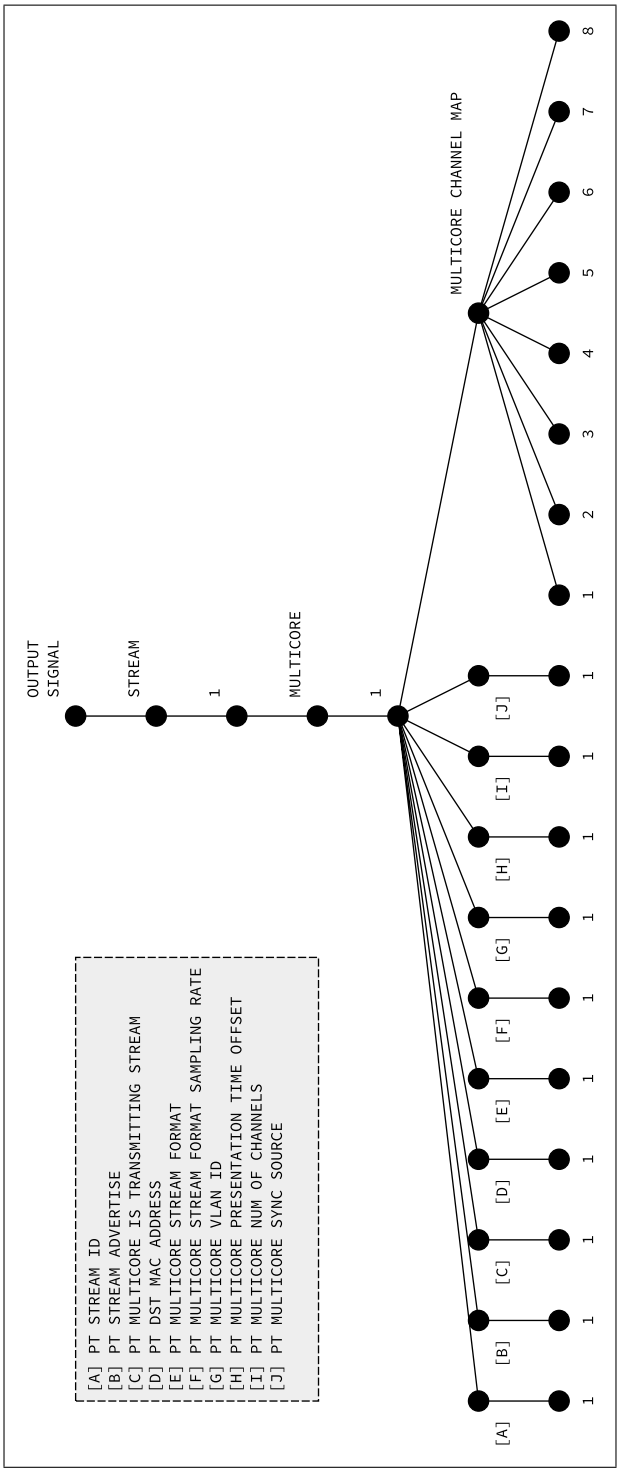


Figure 6.12: The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB Talker unit. Multiple Talker units are represented by multiple sets of these parameters, indexed at the 'Parameter Block Index' level.

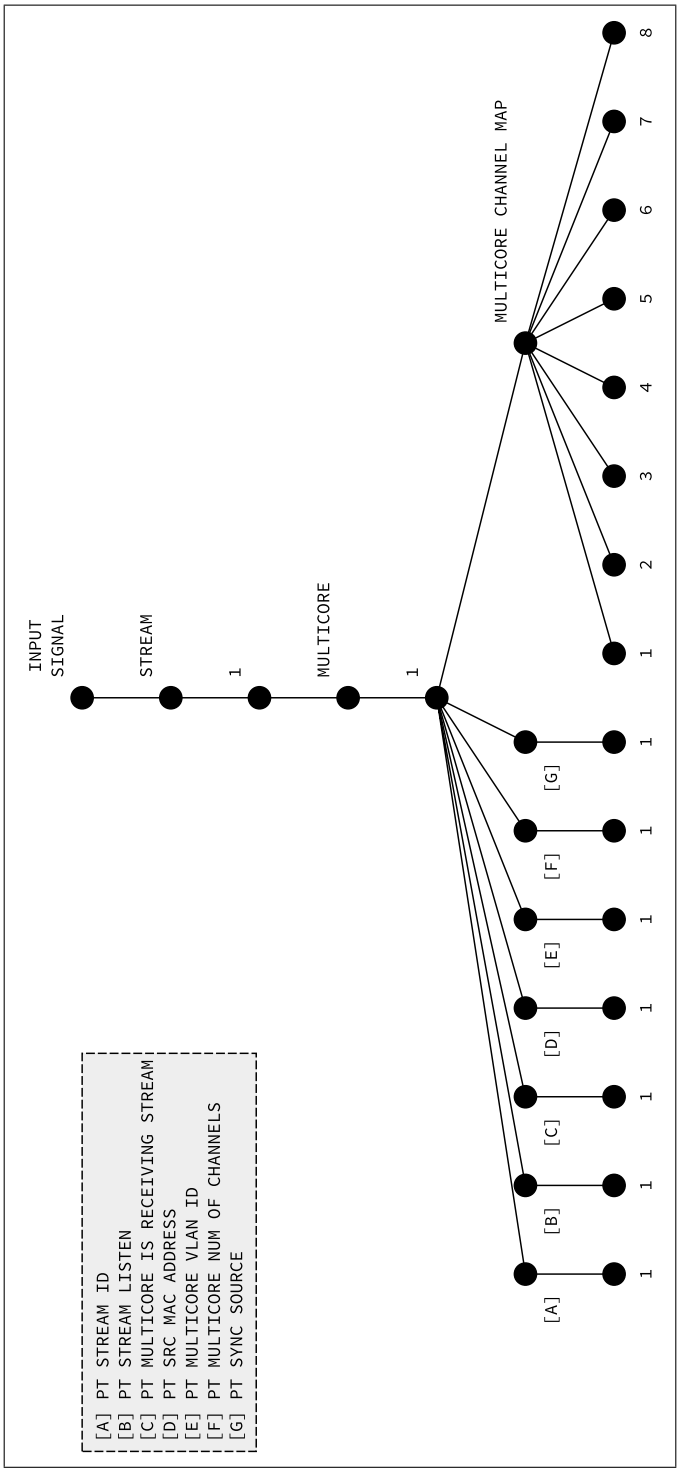


Figure 6.13: The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB Listener unit. Multiple Listener units are represented by multiple sets of these parameters, indexed at the 'Parameter Block Index' level.

The AES-64 parameters provided to represent Talker and Listener units correspond to the AVB Source and Sink control sections of the XMOS Ethernet AVB reference design API [XMOS 2011, 56–65], and their respective value functions provide interfaces between the XMOS Ethernet AVB reference design's API and AES-64 'GET VAL' / 'SET VAL' commands. Listing 6.28 shows the value function implemented for the Talker unit's 'VLAN ID' parameter.

Representation of the Talker unit

The AES-64 representation of the Talker unit (Fig. 6.12) is composed of the following parameter types, which are identified in their relation to the XMOS Ethernet AVB reference design API where applicable:

STREAM ID represents (but does not control) the Stream ID of the Talker unit's one and only AVB stream. The 64-bit Stream ID is conventionally formed of the Talker device's MAC address as the most significant 48 bits, followed by a 16-bit value that identifies the specific Talker *unit* within the device.

STREAM ADVERTISE controls whether the Talker unit is declaring 'Talker Advertise' via MSRP. It corresponds to the `set_avb_source_state()` API function [XMOS 2011, 61].

MULTICORE IS TRANSMITTING STREAM represents (but does not control) whether the Talker unit is currently transmitting a stream. It corresponds to the `get_avb_source_state()` API function [XMOS 2011, 61], but its behaviour is distinct from the previous parameter in that a Talker can be declaring Talker Advertise while *not* transmitting a stream.

DST MAC ADDRESS represents and controls the multicast MAC address that the Talker unit's stream is being (or would be) transmitted to. It corresponds to the `get_avb_source_dest()` and `set_avb_source_dest()` API function [XMOS 2011, 59-60].

MULTICORE STREAM FORMAT represents and controls the digital audio encoding format used by the Talker unit. It corresponds to the `get_avb_source_format()` and `set_avb_source_format()` API functions [XMOS 2011, 57].

MULTICORE STREAM FORMAT SAMPLING RATE represents and controls the sampling rate applied to the digital audio encoding performed by the Talker unit. It corresponds to the `get_avb_source_format()` and `set_avb_source_format()` API

functions [XMOS 2011, 57], which govern the encoder's sampling rate as well as the format applied.

MULTICORE VLAN ID represents and controls the destination VLAN which a Talker unit transmits its stream on. It corresponds to the `get_avb_source_vlan()` and `set_avb_source_vlan()` API functions [XMOS 2011, 60].

PRESENTATION TIME OFFSET represents and controls the 'presentation time' offset embedded into the Talker's audio stream. This parameter corresponds to the `get_avb_source_presentation()` and `set_avb_source_presentation()` API functions [XMOS 2011, 59].

NUM OF CHANNELS represents and controls how many channels of audio are delivered in the Talker's multicore stream. It corresponds to the `get_avb_source_channels()` and `set_avb_source_channels()` API functions [XMOS 2011, 57].

SYNC SOURCE represents and controls the media clock server unit used to prepare the stream. It corresponds to the `get_avb_source_sync()` and `set_avb_source_sync()` API functions [XMOS 2011, 58].

MULTICORE CHANNEL MAP represents and controls the mapping of audio hardware inputs to channels in the Talker's multicore stream. The parameter index identifies a unique channel within the multicore stream; the value assigns a numbered audio input to that channel. This parameter type corresponds to the `get_avb_source_map()` and `set_avb_source_map()` API functions [XMOS 2011, 58], although some processing is required to adapt the AES-64 approach to that used in the XMOS reference design (Listing 6.29).

Listing 6.28: An AES-64 value function for the Talker 'VLAN ID' parameter

```
void a64_ethernet_avb_source_vlan_id_vf(chanend c_mp, enum A64ValueOp
    value_op, UInt8 vf_index, UInt8 *value, UInt8 value_len)
{
        UInt8 retval[4];
        unsigned vlan_id = 0;
        if (VAL_OP_GET_VALUE == value_op) {
// processing a 'GET VAL' request
                get_avb_source_vlan(vf_index, &vlan_id);
                PACK_BSTR_UINT32(retval, vlan_id);
                a64_ctrl_send_value_function_data(c_mp, 4, retval);
        } else if (VAL_OP_SET_VALUE == value_op) {
// processing a 'SET VAL' request
                UNPACK_BSTR_UINT32(vlan_id, value);
                set_avb_source_vlan(vf_index, vlan_id);
                a64_ctrl_send_set_value_function_data(c_mp,
                        A64_SET_VAL_STATUS_NO_ERROR, 0);
        } else {
                a64_ctrl_send_value_function_no_data(c_mp);
        }
}
```

Listing 6.29: The AES-64 value function for Ethernet AVB Talker channel map parameters

```
void a64_ethernet_avb_source_map_vf(chanend c_mp, enum A64ValueOp
    value_op, UInt8 vf_index, UInt8 vf_subindex, UInt8 *value, UInt8
    value_len)
{
    UInt8 retval[4];
    unsigned new_map_setting = 0;
    unsigned map[AVB_NUM_MEDIA_INPUTS];
    unsigned num_channels = 0;
    enum avb_source_state_t curr_state;
    get_avb_source_state(vf_index, &curr_state);
/* AES-64 counts from 1, XMOS Ethernet AVB counts from 0 */
    vf_subindex -= 1;
/* We need to 'know' the current map whether we are getting it or setting
     it */
    get_avb_source_map(vf_index, map, &num_channels);

    if (VAL_OP_GET_VALUE == value_op) {
        if (vf_subindex < num_channels) {
            PACK_BSTR_UINT32(retval, map[vf_subindex]);
            a64_ctrl_send_value_function_data(c_mp, 4, retval);
        } else {
            a64_ctrl_send_value_function_no_data(c_mp);
        }
    } else if (VAL_OP_SET_VALUE == value_op) {
        UNPACK_BSTR_UINT32(new_map_setting, value);
/* check subindex and new value are both valid */
        if ((vf_subindex < num_channels) && (new_map_setting <
            num_channels)) {
            map[vf_subindex] = new_map_setting;
/* XMOS docs say that we must toggle the talker state for changing the
    map to take effect */
            set_avb_source_state(vf_index, AVB_SOURCE_STATE_DISABLED);
            set_avb_source_map(vf_index, map, num_channels);
            if ((curr_state == AVB_SOURCE_STATE_POTENTIAL) ||
                    (curr_state == AVB_SOURCE_STATE_ENABLED)) {
                set_avb_source_state(vf_index, AVB_SOURCE_STATE_POTENTIAL
                    );
            }
            a64_ctrl_send_set_value_function_data(c_mp,
                A64_SET_VAL_STATUS_NO_ERROR, 0);
        } else {
            a64_ctrl_send_set_value_function_data(c_mp,
                A64_SET_VAL_STATUS_ERROR, 0);
        }
    } else {
        a64_ctrl_send_value_function_no_data(c_mp);
    }
}
```

Representation of the Listener unit

The AES-64 representation of the Listener unit (Fig. 6.13) is composed of the following parameter types, which are identified in their relation to the XMOS Ethernet AVB reference design API, where applicable:

STREAM ID represents and controls the Stream ID of the one and only stream that the Listener unit is interested in receiving. It corresponds to the `get_avb_sink_id()` and `set_avb_sink_id()` API functions [XMOS 2011, 63].

STREAM LISTEN controls whether the Listener unit is declaring 'Listener Ready' via MSRP. It corresponds to the `set_avb_sink_state()` API function [XMOS 2011, 65].

MULTICORE IS RECEIVING STREAM represents (but does not control) whether the Listener unit is currently receiving a stream. It corresponds to the `get_avb_sink_state()` API function [XMOS 2011, 65], but its behaviour is distinct from the previous parameter in that a Listener can be declaring Listener Ready while *not* receiving a stream.

SRC MAC ADDRESS represents and controls the multicast MAC address that the Listener unit intends to receive a stream from. It corresponds to the `get_avb_sink_addr()` and `set_avb_sink_addr()` API function [XMOS 2011, 63-64].

MULTICORE VLAN ID represents and controls the VLAN that a Listener unit intends to receive a stream from. It corresponds to the `get_avb_sink_vlan()` and `set_avb_source_vlan()` API functions [XMOS 2011, 64].

NUM OF CHANNELS represents and controls how many channels of audio are received in the Listener's incoming multicore stream. It corresponds to the `get_avb_sink_channels()` and `set_avb_sink_channels()` API functions [XMOS 2011, 61].

SYNC SOURCE represents and controls the media clock server unit used to recover a received stream. It corresponds to the `get_avb_sink_sync()` and `set_avb_sink_sync()` API functions [XMOS 2011, 62].

MULTICORE CHANNEL MAP represents and controls the mapping of channels in the Listener's incoming multicore stream to audio hardware outputs. The parameter index identifies a unique channel within the multicore stream; the value

assigns a numbered audio output to that channel. This parameter type corresponds to the `get_avb_sink_map()` and `set_avb_sink_map()` API functions [XMOS 2011, 62].

6.10.2 AES-64 control of the Ethernet AVB media clock server

The default configuration of the XMOS Ethernet AVB reference design provides a single 'media clock server', tasked with embedding clock information in outbound streams and recovering it from inbound streams. XMOS provide a control API for the media clock server component, and these control features have been implemented as AES-64 parameters (Fig. 6.14). These correspond to the media clock server control section of the XMOS Ethernet AVB reference design [XMOS 2011, 55-56], and each parameter's value function interfaces between one or more API calls and AES-64 'GET VAL' and 'SET VAL' commands.

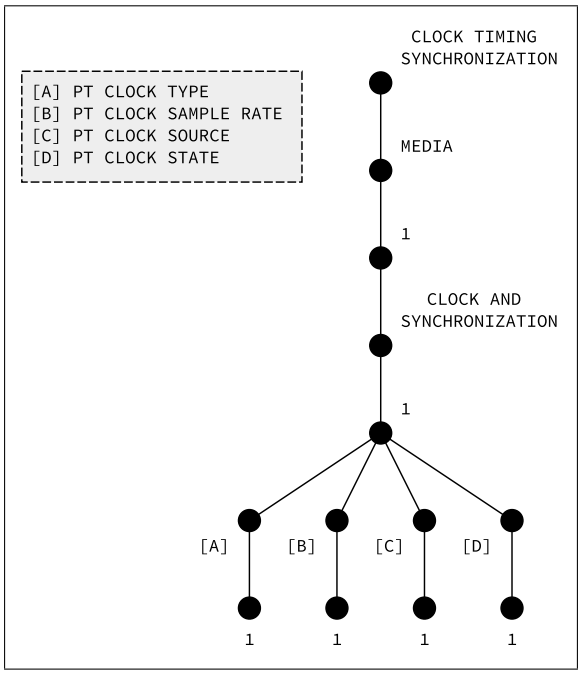


Figure 6.14: The level hierarchy for the AES-64 parameter representation of an XMOS Ethernet AVB media clock server. Multiple media clock server units may be represented by multiple sets of these parameters indexed at the 'Parameter Block Index' level.

CLOCK TYPE represents and controls whether the media clock server generates a clock that is derived from the local IEEE 802.1AS server, or derived from an inbound IEEE 1722 stream. It corresponds to the `get_device_media_clock_type()` and `set_device_media_clock_type()` functions [XMOS 2011, 55].

CLOCK SAMPLE RATE represents and controls the sampling rate (in Hz) generated by the media clock server. It corresponds to the `get_device_media_clock_rate()` and `set_device_media_clock_rate()` functions [XMOS 2011, 55].

CLOCK SOURCE represents and controls, in the case where **CLOCK TYPE** is set to generate a clock derived from an inbound IEEE 1722 stream, which channel of the stream is used to derive the clock. It corresponds to the `get_device_media_clock_source()` and `set_device_media_clock_source()` functions [XMOS 2011, 56].

CLOCK STATE represents and controls the media clock server's generation of a clock. It corresponds to the `get_device_media_clock_state()` and `set_device_media_clock_state()` functions [XMOS 2011, 56].

6.10.3 AES-64 control of the CS4272 audio codec

As described in Subsection 2.4.1, the AES-64-capable Ethernet AVB device was implemented on a DSP4YOU AVBStreamer module interfaced with a DSP4YOU E-mic stereo preamp module (Fig. 2.27). The E-mic module is based around a Cirrus Logic CS4272 dual-channel audio codec, which the XMOS microcontroller may communicate with through an I^2C interface [Cirrus Logic 2005, 35].

The XMOS Ethernet AVB reference design implements a reusable I^2C component, which enables the XS1-G4 processor on the AVBStreamer module to initialise the CS4272 codec and provide it with a clock signal.

The CS4272 has a number of control parameters, including (for each of its two input and output channels, 'A' and 'B'):

1. *mute* controls on the 'A' and 'B' analogue inputs [Cirrus Logic 2005, 43];
2. *mute* controls on the 'A' and 'B' analogue outputs [Cirrus Logic 2005, 42];
3. *gain* (volume) controls on the 'A' and 'B' analogue outputs [Ibid.].

The XMOS reference design does not implement an API or control mechanisms for these functions[16]. The following subsection describes the implementation of a control mechanism to enable AES-64 control of the CS4272. The AES-64 parameters devised to represent these control features are shown in Fig. 6.15.

The AES-64 representation of the audio hardware is composed of the following parameter types:

[16] Understandably, since the target hardware for the XMOS Ethernet AVB reference design is not the DSP4YOU system, but an Attero Tech development board with two simpler audio codecs, the CS5343 and CS4344 [XMOS 2011d, 4].

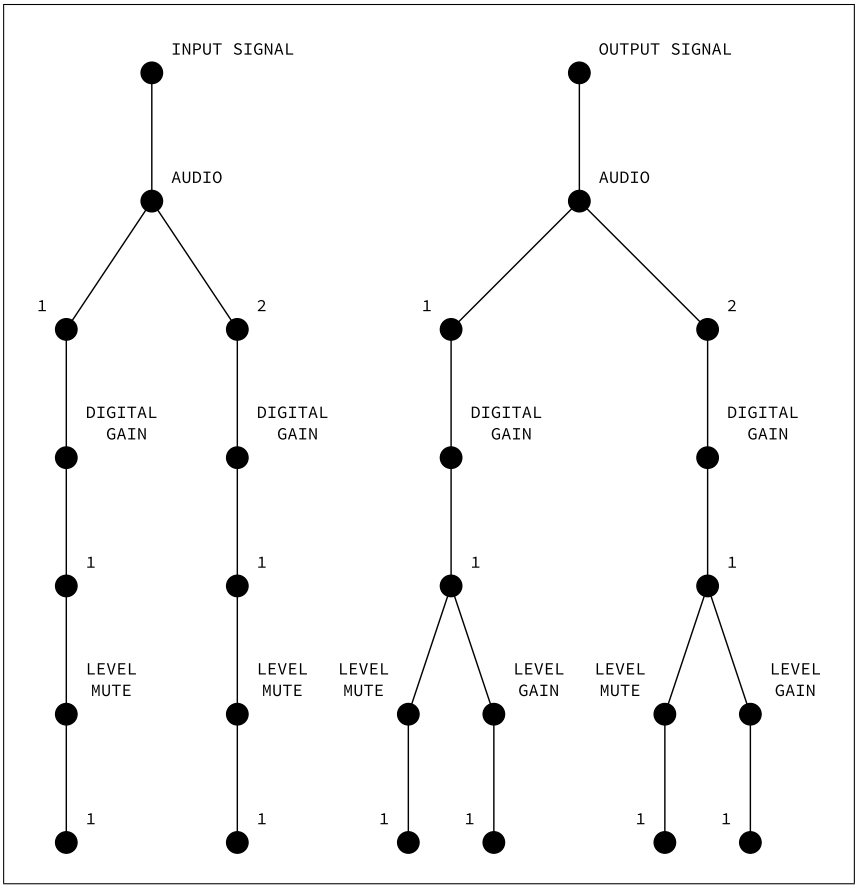


Figure 6.15: The level hierarchy for the AES-64 'audio' params.

LEVEL GAIN corresponds to the register that controls gain on each of the two output channels on the CS4272 [Cirrus Logic 2005, 42]. The value function for this parameter is shown in Listing 6.35.

LEVEL MUTE corresponds (depending on its position in the level hierarchy, as in Fig. 6.15) to either:

- the register that mutes each of the two input channels on the CS4272;
- the register that controls gain on each of the two output channels on the CS4272 (which can, therefore, be used to mute the output signal).

Implementation of audio hardware control

The Ethernet AVB reference design controls the CS4272 audio codec through an I^2C interface which encapsulates a set of hardware input-output ports belonging to a single processing tile within the XS1 package. There is exactly one instance of the I^2C interface (Listing 6.30). This instance is subject to the XC thread disjointness rules [XMOS 2011, 29], with the result that any I^2C read/write operations have to be performed by the same thread responsible for initialising the codec.

Listing 6.30: The I^2C interface instance

```
on stdcore[0]: struct r_i2c r_i2c = { PORT_I2C_SCL, PORT_I2C_SDA };
```

The initialisation of the codec in the Ethernet AVB reference design is shown on line 19 of Listing 6.9. Consequently, the reference design's original `ptp_server_and_gpio()` function was amended and expanded (and renamed `ptp_gpio_i2c_server()`) to enable AES-64 value functions executing on the stack control task to initiate I^2C read/write operations. Figure 6.16 expands the protocol stack task diagram to include these interactions.

Communication between the stack control task and the PTP-GPIO-I^2C server employs the protocol outlined in Subsection 6.5.2. The `ptp_gpio_i2c_server()` task receives commands from the `demo()` task over the `c_gpio_ctl` channel. Listing 6.31 shows the available commands, and Listing 6.32 shows the `ptp_gpio_i2c_server()` task's `select` statement.

The functions shown in Listing 6.32 perform I^2C read/write operations to the CS4272 codec's control registers, in some places performing rudimentary translation between the range of the AES-64 parameter value and the range of the CS4272 register's admissible values. Listing 6.33 shows the `ptp_gpio_i2c_handle_set_output_level()` function, which calls the `audio_codec_CS4272_set_output_level()` function (Listing 6.34).

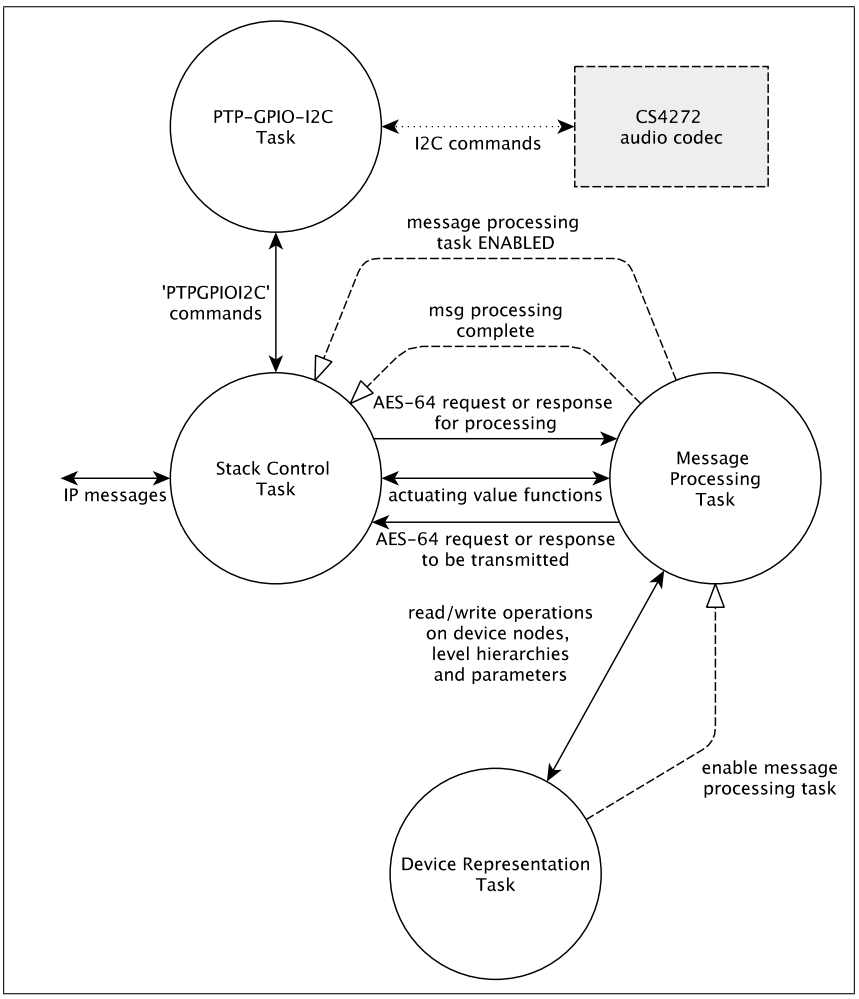


Figure 6.16: An expanded protocol stack task diagram showing interactions between the stack control task and the `ptp_gpio_i2c_server()` task to control the CS4272 audio codec.

Listing 6.31: The audio hardware control commands

```
enum PTPGPIOI2CCommand {
  PTPGPIOI2C_INVALID_COMMAND,
  PTPGPIOI2C_I2C_SET_OUTPUT_LEVEL,
  PTPGPIOI2C_I2C_GET_OUTPUT_LEVEL,
  PTPGPIOI2C_I2C_SET_INPUT_MUTE,
  PTPGPIOI2C_I2C_GET_INPUT_MUTE
};
```

Each I^2C write operation returns an acknowledgment or an error code. These are returned (via the `c_gpio_ctl` channel) to the AES-64 value function on the stack control task. Listing 6.35 shows a value function that communicates with the `ptp_gpio_i2c_handle_set_output_level()` function shown in Listing 6.33 to call the `audio_codec_CS4272_set_output_level()` function shown in Listing 6.34.

Listing 6.32: Event selection in the `ptp_gpio_i2c_server()` task

```
while (1) {
    enum PTPGPIOI2CCommand cmd = PTPGPIOI2C_INVALID_COMMAND;
    select
    {
      /* case 1 - handle PTP events */
      do_ptp_server(c_rx, c_tx, ptp_link, num_ptp);
      /* case 2 - handle commands from the stack control task */
      case inct_byref(c, token):
        if (token == PTP_GPIO_I2C_COMMAND_TOKEN) {
          outct(c, XS1_CT_END);
          outct(c, XS1_CT_END);
          cmd = inuint(c);
          chkct(c, XS1_CT_END);
          outct(c, XS1_CT_END);
        }
        switch(cmd)
        {
        case PTPGPIOI2C_I2C_SET_OUTPUT_LEVEL:
          res = ptp_gpio_i2c_handle_set_output_level(c, r_i2c);
          break; /* switch case break */
        case PTPGPIOI2C_I2C_GET_OUTPUT_LEVEL:
          res = ptp_gpio_i2c_handle_get_output_level(c, r_i2c);
          break; /* switch case break */
        case PTPGPIOI2C_I2C_SET_INPUT_MUTE:
          res = ptp_gpio_i2c_handle_set_input_mute(c, r_i2c);
          break; /* switch case break */
        case PTPGPIOI2C_I2C_GET_INPUT_MUTE:
          res = ptp_gpio_i2c_handle_get_input_mute(c, r_i2c);
          break; /* switch case break */
        default:
          break; /* switch case break */
        }
        /* handle errors, if any */
      break; /* select case break */
    }
  }
}
```

Listing 6.33: The `ptp_gpio_i2c_handle_set_output_level()` function

```
void ptp_gpio_i2c_handle_set_output_level(chanend c, struct r_i2c &r_i2c)
{
  unsigned output_channel;
  unsigned new_level;
  unsigned status;

  c :> output_channel;
  c :> new_level;

  status = audio_codec_CS4272_set_output_level(r_i2c, output_channel,
      new_level);
  c <: status;
}
```

Listing 6.34: Setting output channel level via the I^2C interface

```
/* write value to I2C register */
unsigned REGWR(unsigned reg, unsigned val, struct r_i2c &r_i2c)
{
  struct i2c_data_info data;
  data.master_num = 0;
  data.data_len = 1;
  data.clock_mul = 5;
  data.data[0] = val;
  return i2c_master_tx(DEVICE_ADRS, reg, data, r_i2c);
}

int audio_codec_CS4272_set_output_level(struct r_i2c &r_i2c, unsigned
    channel, unsigned level)
{
  /* Each channel has its own register: {0 == Channel A (register 4h), 1
      == Channel B (register 5h)} */
  unsigned res = 1;
  if (channel == 0) {
    res = REGWR(0x04, volume_levels[level], r_i2c);
    if (res == 0) {
      /* handle error */
    }
  } else if (channel == 1) {
    res = REGWR(0x05, volume_levels[level], r_i2c);
    if (res == 0) {
      /* handle error */
    }
  }
  if (res != 0)
    return 1;
  else
    return 0;
}
```

Listing 6.35: The AES-64 value function for CS4272 output channel level

```
/* Prototypes for PTP GPIO I2C server API functions */
void ptp_gpio_i2c_server_get_output_level(chanend c_server, unsigned
    output_channel, REFERENCE_PARAM(unsigned, current_level));

unsigned ptp_gpio_i2c_server_set_output_level(chanend c_server, unsigned
    output_channel, unsigned level);

/* The value function */
void a64_cs4272_audio_output_level_gain_vf(chanend c_mp, enum A64ValueOp
    value_op, UInt8 vf_index, UInt8 vf_subindex, UInt8 *value, UInt8
    value_len, chanend c_gpio_ctl)
{
  UInt8 retval[4] = {0};
  unsigned new_level;
  unsigned current_level = 0;
  unsigned status = 0;
    if (VAL_OP_GET_VALUE == value_op) {
        ptp_gpio_i2c_server_get_output_level(c_gpio_ctl, vf_index, &
            current_level);
        a64_ctrl_send_value_function_data(c_mp, 4, retval);
    } else if (VAL_OP_SET_VALUE == value_op) {
        UNPACK_BSTR_UINT32(new_level, value);
/* Check supplied level is within valid range */
        if ((new_level <= 11) && (new_level >= 0))
          status = ptp_gpio_i2c_server_set_output_level(c_gpio_ctl,
              vf_index, new_level);
          if (status == 1)
            a64_ctrl_send_set_value_function_data(c_mp,
                A64_SET_VAL_STATUS_NO_ERROR, 0);
          else
            a64_ctrl_send_set_value_function_data(c_mp,
                A64_SET_VAL_STATUS_ERROR, 0);
    } else {
    a64_ctrl_send_value_function_no_data(c_mp);
    }
}
```

6.11 Configuration

Configuration of the AES-64 protocol stack and its representation of the host device are described in this section.

6.11.1 Protocol stack configuration

The AES-64 protocol stack can be configured to run with multiple message processing tasks. The `a64_conf.h` header defines the `NUM_MSG_PROCESSING_THREADS` constant[17]. The value given here must be matched by an equal number of calls to `a64_msg_proc()` within the `main()` function's `par` block. Each call to `a64_msg_proc()` must be provided with an entry in each of the two channel arrays, `aes64_msg_proc` and `aes64_dnm_proc`. Listing 6.36 shows an example of the AES-64 protocol stack configured to run with two message processing tasks.

The `a64_conf.h` header also defines constant values to set the maximum size of a level item alias, the buffer size allocated to inbound or outbound AES-64 messages, the size of the parameter flags register, and the greatest number of parameters held by any one device node within the AES-64 representation of the host device. The configuration of this representation is discussed in the following subsection.

[17] This constant is used, for example, to declare an array of channels between the device representation task and the set of message processing tasks, and to inform the device representation task that it must select communication events on any of those channels (see Listing 6.10 lines 15–16, and Listing 6.11, line 14).

Listing 6.36: Configuration of multiple message processing tasks

```
/* a64_conf.h */
#define NUM_MSG_PROCESSING_THREADS (2)

/* app_demo.xc excerpt */
#include "a64_conf.h"

/* ... */

int main(void) {

  /* ... */
  // channel declarations:

  // array of channels that
  // connect control app and the aes-64 msg processing thread(s)
  chan aes64_msg_proc[NUM_MSG_PROCESSING_THREADS];

  // array of channels that
  // connects aes-64 msg processing thread(s) and the device node
  chan aes64_dnm_proc[NUM_MSG_PROCESSING_THREADS];

  /* ... */

  par {

    /* ... */

    on stdcore[3]: a64_msg_proc(aes64_msg_proc[0], aes64_dnm_proc[0]);

    on stdcore[3]: a64_msg_proc(aes64_msg_proc[1], aes64_dnm_proc[1]);

    on stdcore[1]: a64_dnm(aes64_dnm_proc, NUM_MSG_PROCESSING_THREADS);

    /* ... */

  }
}
```

6.11.2 Configuration of the device representation task

An application making use of the AES-64 implementation must define a function, `a64_device_configuration_init`, that configures the device representation task. A valid configuration consists of at least one device node, a level hierarchy, and a set of parameters.

This subsection presents excerpts from the configuration of device representation for the XMOS Ethernet AVB reference design. The full configuration file is included (and briefly discussed) as Appendix D.

Creating a device node

The creation of a device node is shown in Listing 6.37. The caller must indicate the maximum number of parameters that the device node will hold.

Once the device node has been created, it must be added to the device representation task's list of device nodes. Once the device node has been created and added, its parameters must be configured in full before any further nodes are created or configured.

Listing 6.37: Configuration of the device representation task

```
int a64_device_configuration_init()
{
        /* General Usage:
         *
         * 1. First create at least one device node.
         *
         * 2. Then create (a) a level block, (b) a parameter,
         *      as many times as necessary.
         *    Declare level blocks and parameters in matched
         *      pairs.
         */
        int retval = 0;
        /* One device node for the device config information */
        struct DNode *new_node = dnode_create(NUM_CONFIG_PARAMS);
        if (new_node == NULL) return -1;
        append_to_global_dnodes(get_global_dnodes(), &new_node);
        /* ... */
```

Creating a device parameter

Creating an AES-64 parameter involves the following steps:

1. create a level items table entry;
2. add the level items table entry to the device node;

3. create a parameter;

4. add the parameter structure to the device node.

An example of this is shown in Listing 6.38, where the creation of the 'IP address' parameter and its addition to the device node are both handled by the `add_parameter_to_dnode` call.

Listing 6.38: Creating a level items table entry and a device parameter

```
int retval = 0;
enum A64ValueFunctionClass vf_class = VFN_CLASS_NETWORK;

/* acquire pointer to current device node */
struct DNode *last_dnode = get_last_dnode();

/* create level hierarchy entry */
struct LevelItemsTableEntry *new_lite01 = create_level_items_table_entry(
    SB_DEVICE_INFO_CONFIG, "device␣config", ST_IP, "ip", 0x1, PB_CONFIG, "
    config", 0x1, PT_IP_ADDRESS, "ip␣address", 0x1);
if (new_lite01 == NULL) return -1;

/* add to device node's level hierarchy */
add_to_level_items_table(last_dnode->level_items_table, &new_lite01);

/* create parameter and add to device node's parameter table */
retval = add_parameter_to_dnode(last_dnode, "ip␣address", VALFT_INT_32,
    0, VFN_CLASS_NETWORK, NETWORK_IP_ADDRESS, 1, 0);
if (retval == -1) return -2;
```

Large numbers of related parameters can be created efficiently by iterating over arrays of AES-64 identifiers, value formats, value function identifiers and level aliases. This technique is particularly useful as it limits the memory footprint of the code required to set up an AES-64 representation, and is used to configure the AES-64 representation of the Ethernet AVB reference design and the CS4272 audio codec, as discussed in Subsection 6.10.

6.11.3 Configuration file for the AES-64-enabled Ethernet AVB audio device

The full configuration file for the device representation task is included as Appendix D.

6.12 The AES-64 connection management and control utilities

A set of simple connection management and control utilities were developed to test and demonstrate the capabilities of the AES-64-enabled Ethernet AVB device. The AES-64 prototype described earlier in this chapter provided a basis for the implementation of these utilities.

The utilities are simple — far simpler than the UNOS suite described in Section 3.7 — in that they do not maintain a local AES-64 stack or device representation of their own: they simply send AES-64 compliant messages to effect changes, retrieve device status and (in some cases) display or capture any device responses.

Chapter 7 discusses the use of these tools to control the AES-64 implementation, and Appendix E provides a description and usage instructions for each of these tools, including diagrams of the AES-64 messages sent and received .

a64_discover provides device discovery and rudimentary device enumeration;

a64_connect performs stream connection between a nominated Talker and a nominated Listener;

a64_disconnect instructs a Talker to withdraw its Talker Advertise or a Listener to withdraw its Listener Ready, in each case ending a stream connection;

a64_query queries the status of the Talker and Listener components on a device;

a64_channel sets the mapping between a stream multicore's channels and a Talker's audio inputs or a Listener's audio outputs;

a64_join_setup_1 sets up a proof-of-concept parameter group between two parameters on one device, and **a64_join_teardown_1** tears down the parameter group;

a64_join_setup_2 sets up a proof-of-concept parameter group between two parameters on two different devices, and **a64_join_teardown_2** tears down the parameter group;

a64_get_mstgrp queries the 'master' group list of a parameter on the device to verify the operation of the join/unjoin implementation;

a64_get_slvgrp queries the 'slave' group list of a parameter on the device to verify the operation of the join/unjoin implementation;

a64_mute mutes an audio input or output channel on a device;

a64_volume sets the level of an audio output channel on a device.

a64_level_check gets the level of the audio output channels on the device.

6.13 Conclusion

Implementing the AES-64 standard for the XMOS XS1 platform followed a reasonably direct path from the models and specifications developed by the design process. It was possible to prototype basic functionality while the implementation design was being finalised. The XS1 platform provides the opportunity to adapt and integrate existing C code within larger, concurrent XC applications. This meant that once the design work *was* finalised, substantial parts of the prototype could be ported to the XS1 platform for use in the protocol stack. This was particularly convenient as it enabled the use of conventional verification tools like Valgrind.

The final implementation of the AES-64 standard implements a significant part of the functionality defined by the published standard, and conforms closely to the implementation design described in Chapter 5. It has been shown to integrate with the XMOS Ethernet AVB reference design. Chapter 7 discusses how this implementation enables control and monitoring over the functionality of the device. The AES-64 implementation provides a detailed representation of the XMOS Ethernet AVB reference design's parameters through a clear interface. Appendix D shows the code that constructs this representation.

A set of flexible command-line utilities were also developed to demonstrate and test AES-64 control of the XMOS Ethernet AVB end stations. The use of these is described in Chapter 7, and additionally Appendix E provides reference description and usage information for each utility.

7 Evaluation

7.1 Introduction

The primary goals of the development process presented in Chapters 5 – 6 were:

- to produce an AES-64-capable Ethernet AVB endpoint, and
- to investigate the capabilities and limitations of the XMOS XS1 architecture for development of audio control protocols.

This chapter presents an evaluation of the AES-64 implementation and the findings of the investigation.

Section 7.2 evaluates the XS1-based implementation of the AES-64 standard in qualitative terms. This evaluation considers the following points:

- the performance of the AES-64 implementation for device control;
- the clarity, consistency and maintainability of the code that comprises the AES-64 implementation.

Section 7.3 presents quantitative evaluation of the performance of the AES-64 implementation, focusing on the latency (or reaction time) of the AES-64 implementation to various external commands.

Section 7.4 presents an evaluation of the XMOS XS1 architecture as a platform for designing and implementing the AES-64 audio control standard.

7.2 Qualitative evaluation: the AES-64 implementation for Ethernet AVB device control

As originally conceived, the AES-64 implementation was intended to enable connection management and control of Ethernet AVB streams, as well as network discovery and device enumeration.

As implemented, the AES-64 implementation provides connection management of Ethernet AVB multicore streams and their channels, as well as control of the device's audio hardware. It provides a basic implementation of network discovery, as well as proof-of-concept implementations of device enumeration and parameter grouping.

As such, the AES-64 implementation supports the following connection management and control operations:

- discovery of AES-64 devices on the network
- enumeration of a device's basic capabilities
- connection of an Ethernet AVB stream between a Talker and a Listener
- connection of further Listeners to a Talker's existing stream
- disconnection of single Listeners from receiving a Talker's stream
- disconnection of a Talker from transmitting a stream to a set of Listeners
- assignment of audio inputs to channels of a stream transmitted by a Talker
- assignment of channels of a stream received by a Listener to its audio outputs
- querying of a device for the status of its Talker and Listener units
- volume control of the audio inputs of a device
- volume control of the audio outputs of a device
- setup and teardown of an AES-64 absolute master-slave parameter group, where the slave is a local parameter to the master
- setup and teardown of an AES-64 absolute master-slave parameter group, where the slave is a parameter on a remote device to the master
- automatic relaying of value updates from a master parameter to its slave(s) for the duration of a master-slave parameter group
- querying of parameter group list membership (i.e., for verification of group setup and teardown)

7.2.1 Network discovery

Network discovery is implemented as outlined in Subsection 3.6.1. The discovery is performed by the `a64_discover` utility (Subsection E.2.1), which broadcasts a AES-64 'GET VAL' request targeting the 'IP Address' parameter on the 'configuration device node' (Fig. 7.1). This parameter's *value function* retrieves the device's current IP address from the stack control task. This value is updated whenever the IP stack reports an `XTCP_IFUP` event (i.e., whenever the device is physically attached to a network).

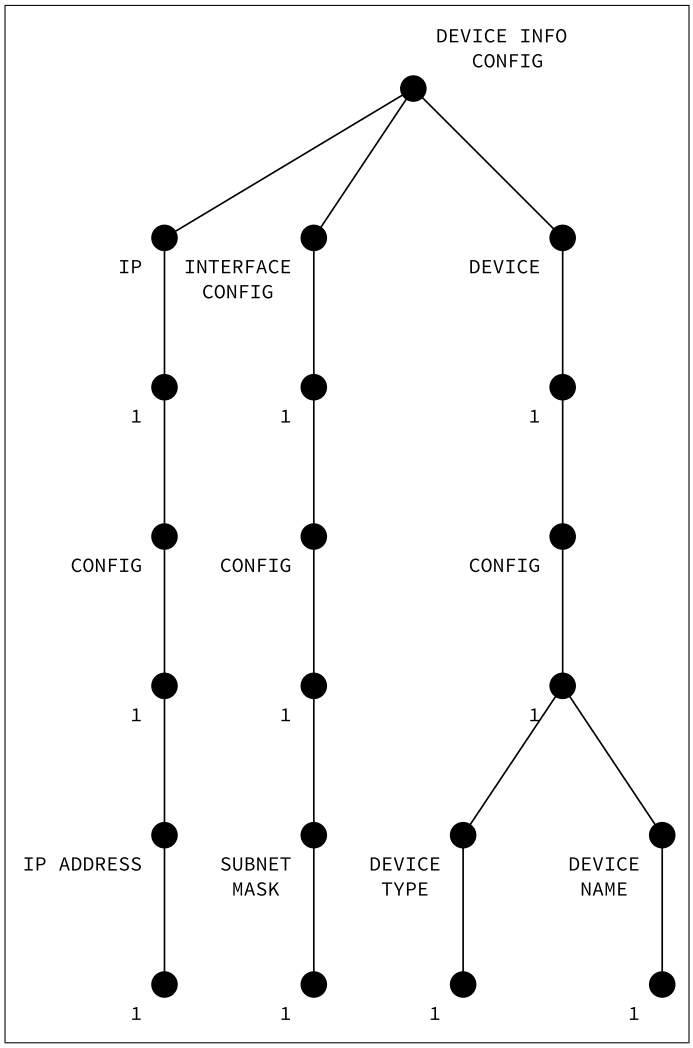


Figure 7.1: The level hierarchy for the AES-64 'configuration device node'.

The `a64_discover` utility performs rudimentary device enumeration, as well as retrieving the 'Device Name' parameter from the 'configuration device node'. In addition to printing

its findings onscreen (Fig. 7.2), the `a64_discover` utility dumps a CSV file that could be loaded and processed programmatically by a dedicated device enumeration utility.

```
aes64_tools$ ./a64_discover
Address from discovery: 169.254.92.141
Address from discovery: 169.254.179.48

Discovery period ended.
Discovery table has 2 entries
Index    Stream IFs      Input MCs       Output MCs      Audio Ins       Audio Outs              IP Address      Device Name
0               1               1               1               2               2          169.254.92.141      DIBLEY BOARD 2
1               1               1               1               2               2          169.254.179.48      DIBLEY BOARD 1
aes64_tools$
```

Figure 7.2: The `a64_discover` utility displays discovered devices and rudimentary enumeration, including the value of the 'Device Name' parameter.

7.2.2 Proof of concept: device enumeration

Enumeration of AES-64-capable devices is based around the level hierarchy. With the exception of mandatory parameters and locations (e.g., the 'IP Address' parameter that enables device discovery), an AES-64 controller does not have preordained knowledge of the structure of a device it discovers on the network.

The 'GET CLA' command, as described in Subsubsection 5.2.1, may be used to build up a local representation of each level hierarchy on each device node of a remote device. The `a64_discover` utility uses a sequence of 'GET CLA' requests to build up an approximate idea of a discovered device's capabilities by enumeration. As shown in Fig. 7.2, it enumerates:

1. the number of stream-capable network interfaces;
2. the number of Listener units ('Input MultiCores');
3. the number of Talker units ('Output MultiCores');
4. the number of audio inputs on the device;
5. the number of audio outputs on the device.

The more literal usage of 'GET CLA' – to recover a list of child nodes, their full address block locations and their level aliases – is also supported by the implementation (Fig. 7.3).

The enumeration process described above could be extended to discover the number of child nodes, and their identifiers, for every node in the level hierarchy.

```
aes64_tools$ ./a64_get_cla 169.254.155.173
Querying the child levels at SB INPUT SIGNAL - ST STREAM - 1 - PB MULTICORE - 1 - 0xEEEE - 0xEEEE...
Number of GET CLA entries returned: 7
Entry #1 has full address block [1 - aa - 001 - d1 - 001 - d50 - eeee]
        and level alias: stream id
Entry #2 has full address block [1 - aa - 001 - d1 - 001 - d52 - eeee]
        and level alias: listen
Entry #3 has full address block [1 - aa - 001 - d1 - 001 - d57 - eeee]
        and level alias: listener state
Entry #4 has full address block [1 - aa - 001 - d1 - 001 - d58 - eeee]
        and level alias: src maap addr
Entry #5 has full address block [1 - aa - 001 - d1 - 001 - d60 - eeee]
        and level alias: vlan id
Entry #6 has full address block [1 - aa - 001 - d1 - 001 - d62 - eeee]
        and level alias: num channels
Entry #7 has full address block [1 - aa - 001 - d1 - 001 - d63 - eeee]
        and level alias: channel map
aes64_tools$
```

Figure 7.3: 'GET CLA' data returned from the AES-64 capable Ethernet AVB device in response to the insert 'GET CLA' request.

7.2.3 Stream connection management

Stream connections are established through the `a64_connect` utility (Subsection E.2.2). For the XMOS Ethernet AVB reference design, the pertinent parameters are, for a Talker:

```
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   STREAM ID         1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   DST MAC ADDRESS   1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   MULTICORE VLAN ID 1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   ADVERTISE         1
```

and, for a Listener:

```
INPUT SIGNAL    STREAM  1   MULTICORE  1   STREAM ID         1
INPUT SIGNAL    STREAM  1   MULTICORE  1   SRC MAC ADDRESS   1
INPUT SIGNAL    STREAM  1   MULTICORE  1   MULTICORE VLAN ID 1
INPUT SIGNAL    STREAM  1   MULTICORE  1   LISTEN            1
```

The stack control task implements *value functions* for each of these parameters, such that:

- Sending a 'get value' request to the '`OUTPUT SIGNAL`' — '`STREAM ID`' parameter will retrieve the current StreamID of the Talker component.
- Sending a 'get value' request to the '`OUTPUT SIGNAL`' — '`DST MAC ADDRESS`' parameter will retrieve the current MAAP address reservation allocated to the Talker component.

- Sending a 'get value' request to the '`OUTPUT SIGNAL`' — '`MULTICORE VLAN ID`' parameter will retrieve the current VLAN ID allocated to the Talker component.
- Sending a 'set value' request to the '`OUTPUT SIGNAL`' — '`ADVERTISE`' parameter (setting a 'true' or 'false' value) will enable or disable the Talker component from sending advertise messages to the network[1].
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`STREAM ID`' parameter (setting a StreamID retrieved from a Talker) will control *which* Talker's advertise messages the Listener may respond to.
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`SRC MAC ADDRESS`' parameter (setting the destination MAAP address retrieved from a Talker) will prompt the Listener to join the Talker's MAAP group.
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`MULTICORE VLAN ID`' parameter (setting the VLAN ID retrieved from a Talker) will prompt the Listener to register as a member of that VLAN.
- Sending a 'set value' request to the '`INPUT SIGNAL`' — '`LISTEN`' parameter (with a 'true' or 'false' value) will enable or disable the Listener component from sending a 'Listener Ready' response to a Talker Advertise message that matches the StreamID held by the '`INPUT SIGNAL`' — '`STREAM ID`' parameter.

The `a64_connect` utility performs stream connection management between one Talker and one Listener. The use of this utility and its subsequent output is shown in Fig. 7.4, and the sequence of messages implemented by this utility is shown in Fig. 7.5.

(This does represent an 'unsophisticated' approach to connection management: with further modifications to the XMOS Ethernet AVB reference design, it would be unnecessary to set the VLAN ID and multicast MAC address parameters over AES-64. The AES-64 StreamID parameter could control which Talker Advertise SRP messages a Listener is allowed to respond to; consequently, once a Stream ID value has been specified on the Listener, the Listener may recover both the VLAN ID and multicast MAC address from any Talker Advertise SRP message bearing the correct StreamID.)

Additional Listeners can be connected to an existing stream (resources permitting) by invoking the `a64_connect` utility with the Talker's IP address and an IP address of one of the prospective Listeners.

[1] Enabling the sending of an advertise message is necessary to initiate a stream connection, and through the lifetime of the stream connection the Talker periodically re-sends this message; disabling the Talker from re-sending the message will tear down the stream connection at source.

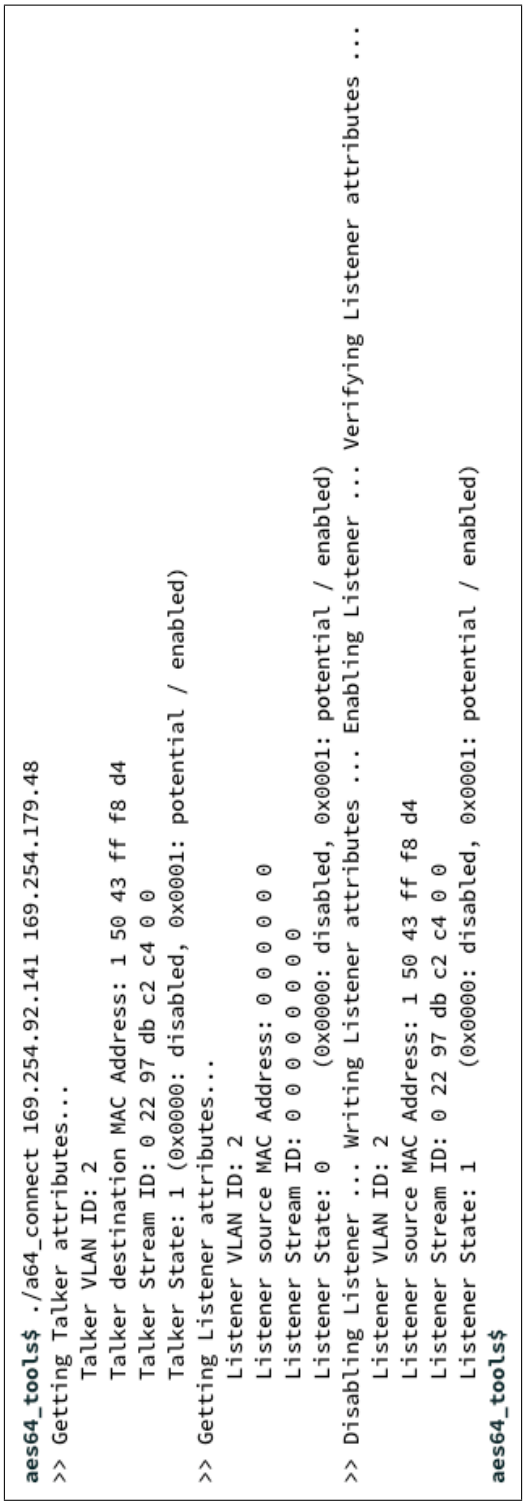


Figure 7.4: Connection management between AES-64-capable Ethernet AVB devices, via the `a64_connect` utility.

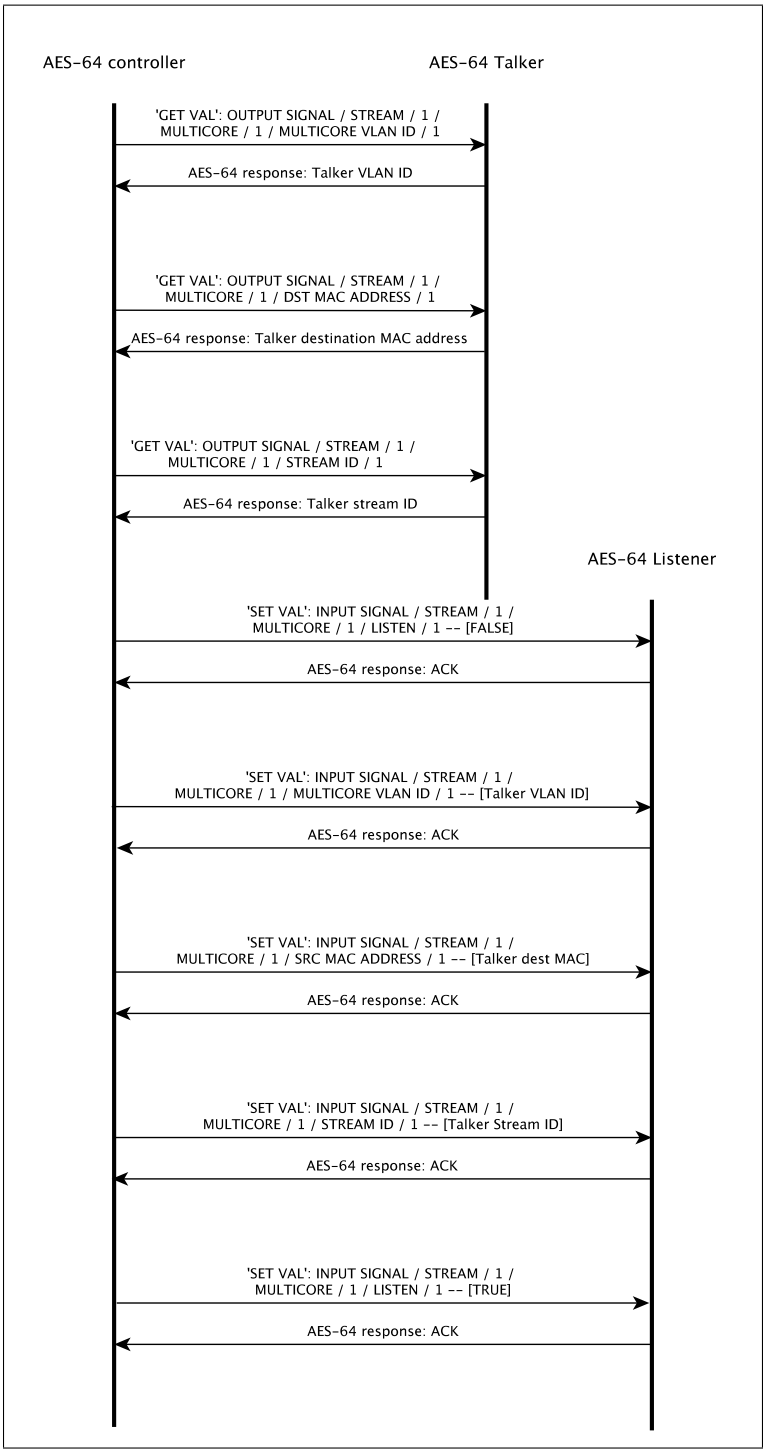


Figure 7.5: AES-64 messaging for the a64_connect utility.

A stream can be torn down either at source (the Talker), by setting the Talker unit's 'Advertise' parameter to 'false', or at one of its destinations (a Listener), by setting the Listener unit's 'Listen' parameter to 'false'. The `a64_disconnect` utility can request either of these actions, and the messaging for each case is shown in Figs. 7.6 and 7.7.

```
aes64_tools$ ./a64_disconnect 169.254.92.141 t
Disabled Talker on device 169.254.92.141
```

Figure 7.6: Teardown of a stream at the Talker, via the `a64_disconnect` utility.

```
aes64_tools$ ./a64_disconnect 169.254.179.48 l
Disabled Listener on device 169.254.179.48
```

Figure 7.7: Teardown of a stream at the Listener, via the `a64_disconnect` utility.

7.2.4 Internal connection management

Internal connection management involves assignment of the channels in a multicore stream to or from the media buffers implemented by the XMOS audio subsystem. There are eight input media buffers and eight output media buffers [XMOS 2011, 11-12]. The input media buffers can be dynamically mapped to the channels of the Talker's multicore stream (Fig. 7.8). The output media buffers can be dynamically mapped from the channels of the Listener's multicore stream (Fig. 7.9).

In the standard XMOS Ethernet AVB reference design, the first two input media buffers are fed by the two input channels of a Cirrus Logic CS4272 audio codec [Cirrus Logic 2005], while the first two output media buffers feed into the two output channels of the CS4272.

Internal connection management at either a Listener sink or a Talker source is represented by the XMOS Ethernet AVB reference design as an array where each element of the array assigns a specific media buffer to a channel of the multicore stream.

The AES-64 representation diverges from this to present each 'point of assignment' as an individual parameter, resulting in the 'fan' of parameters with a `MULTICORE CHANNEL MAP` 'Parameter Type' identifier visible in Fig. 6.12 and 6.13.

Each `MULTICORE CHANNEL MAP` parameter corresponds to a channel of a multicore stream. In the representation of a Talker, the *value* held by each `MULTICORE CHANNEL MAP` parameter controls which media input buffer that channel sources its audio from; in the representation

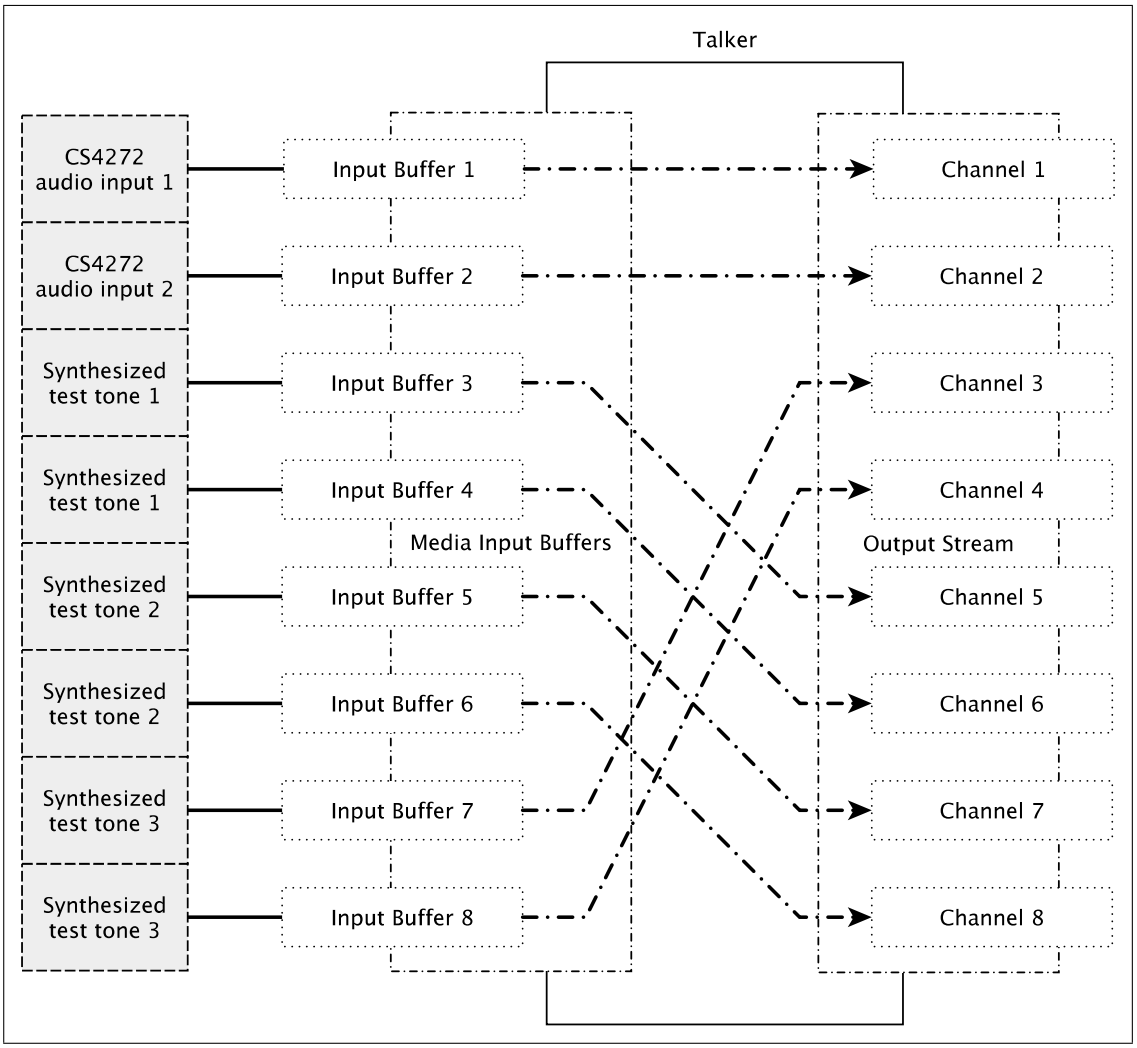


Figure 7.8: Mapping between the media input buffers and the output stream channels on a Talker. The dot-dashed lines indicate dynamic mappings from a media input buffer to a channel in the Talker's multicore stream. In this diagram, the default mappings have been changed: by default, input buffer 1 is mapped to output channel 1, input buffer 3 is mapped to output channel 3, and so on.

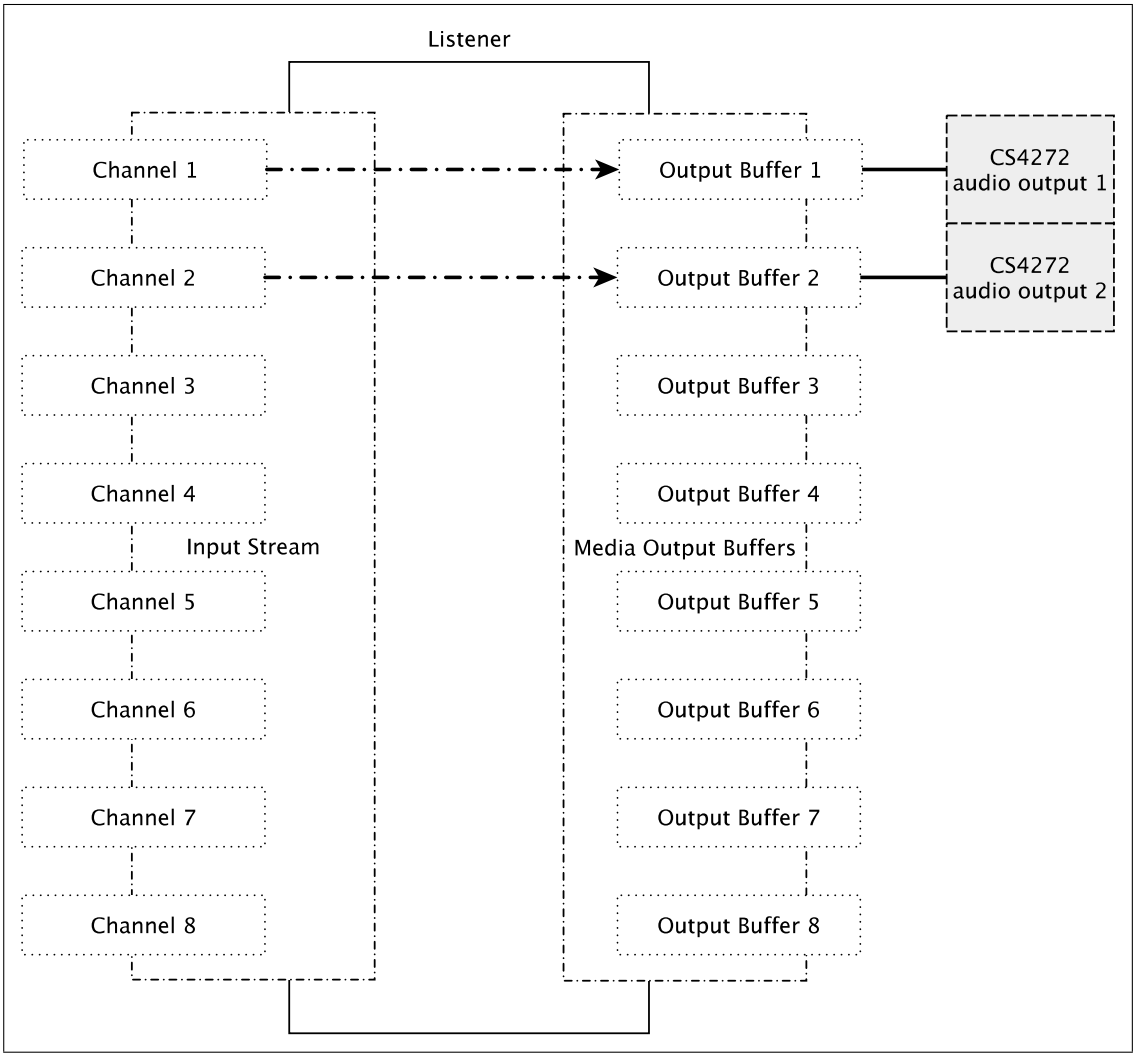


Figure 7.9: Mapping between the input stream channels and media output buffers of a Listener. The dot-dashed lines indicate dynamic mappings from the multicore stream received by the Listener to one of its media output buffers. On the DSP4YOU AVBStreamer board, only the first two media output buffers are connected to actual audio outputs - in this case, the output channels of the CS4272 audio codec.

of a Listener, the value held by each `MULTICORE CHANNEL MAP` parameter controls which media output buffer the corresponding channel sinks its audio into.

The `a64_channel` utility is capable of setting these channel assignment parameters on a Talker or Listener component (Subsection E.2.5). Fig. 7.10 shows a series of invocations of the `a64_channel` utility to control the channel assignment on a Listener:

```
aes64_tools$ ./a64_channel 169.254.92.141 l 1 3
Set stream channel 1 to audio output 3 on device 169.254.92.141
aes64_tools$ ./a64_channel 169.254.92.141 l 2 4
Set stream channel 2 to audio output 4 on device 169.254.92.141
aes64_tools$ ./a64_channel 169.254.92.141 l 4 1
Set stream channel 4 to audio output 1 on device 169.254.92.141
aes64_tools$ ./a64_channel 169.254.92.141 l 4 4
Set stream channel 4 to audio output 4 on device 169.254.92.141
aes64_tools$ ./a64_channel 169.254.92.141 l 1 1
Set stream channel 1 to audio output 1 on device 169.254.92.141
aes64_tools$ ./a64_channel 169.254.92.141 l 2 2
Set stream channel 2 to audio output 2 on device 169.254.92.141
```

Figure 7.10: Multicore channel assignment at an Ethernet AVB device's Listener component using the `a64_channel` utility. The '`l`' argument indicates that the channel assignment is to be performed on a Listener component; '`t`' would specify a Talker component.

1. The first two invocations assign the stream multicore channels 1 and 2 to the Listener component's media output buffer 3 and 4. Since output buffers 3 and 4 are not connected to the audio codec, this effectively silences audio playback.

2. The next invocation assigns the stream multicore channel 4 to media output buffer 1. Media output buffer 1 is connected to one of the CS4272 audio codec's output channels, resulting in playback of the synthesised sine wave generated by media input buffer 4 on the Talker component of the device producing the stream.

3. The next invocation assigns the stream multicore channel 4 back to media output buffer 4, resulting in silence.

4. The following two invocations assign the stream multicore channels 1 and 2 to media output buffer 1 and 2, restoring playback of audio from the Talker's media input buffer 1 and 2 (that is, from the Talker's CS4272 *input* channels).

Changing channel assignment on a Talker is very similar, and is briefly described in Subsection E.2.5.

7.2.5 Media clock server control

Control of the media clock server as provided by the XMOS Ethernet AVB API [XMOS 2011, 55-56] is represented by the AES-64 parameters described in Subsection 6.10.2. Simple get / set utilities have been implemented to control these parameters, described in Subsection E.2.13—E.2.16.

7.2.6 Audio hardware control

The standard XMOS Ethernet AVB reference design has been altered (as described in Subsection 6.10.3) to enable the representation and control of audio input and output channel controls on the CS4272 audio codec.

The `a64_mute` utility is capable of muting the input or output channels of the CS4272 (Fig. 7.11). The CS4272 also provides 'volume' registers for each of its two output channels, and the `a64_volume` utility can set the output level on either output channel (Fig. 7.12).

```
aes64_tools$ ./a64_mute 169.254.92.141 o 1 e
Muted audio output 1 on device at 169.254.92.141
aes64_tools$ ./a64_mute 169.254.92.141 o 2 e
Muted audio output 2 on device at 169.254.92.141
aes64_tools$ ./a64_mute 169.254.92.141 o 2 d
Un-muted audio output 2 on device at 169.254.92.141
aes64_tools$ ./a64_mute 169.254.92.141 o 1 d
Un-muted audio output 1 on device at 169.254.92.141
```

Figure 7.11: The `a64_mute` utility mutes and un-mutes output channels on the CS4272 audio codec.

```
aes64_tools$ ./a64_volume 169.254.92.141 2 0
Set volume level 0 on audioport 2 on device at 169.254.92.141
aes64_tools$ ./a64_volume 169.254.92.141 1 0
Set volume level 0 on audioport 1 on device at 169.254.92.141
aes64_tools$ ./a64_volume 169.254.92.141 1 11
Set volume level 11 on audioport 1 on device at 169.254.92.141
aes64_tools$ ./a64_volume 169.254.92.141 2 11
Set volume level 11 on audioport 2 on device at 169.254.92.141
```

Figure 7.12: The `a64_volume` utility sets volume level on output channels of the CS4272 audio codec.

The opportunity to use an existing I^2C implementation to enable expanded software control of the CS4272 audio codec, and the elegance of linking that expanded software

control to the AES-64 representation, both indicate that the XMOS XS1 architecture provide a highly compelling platform for development of audio control protocols: especially since audio control protocols may easily be controlling functions other than or in addition to Ethernet AVB devices.

7.2.7 Proof of concept: parameter grouping

Parameter grouping is the most complex mechanism presently defined by the AES-64 standard. Subsection 3.6.2 provides a conceptual overview, and Subsection A.9.3 – A.9.7 provides a detailed account of the setup, lifetime and teardown of parameter groups.

The AES-64 implementation provides a proof-of-concept implementation of parameter grouping, providing:

1. set-up of master-slave groups;
2. teardown of master-slave groups;
3. group value updates from masters to slaves;
4. an absolute value relationship between the parameters;
5. query and write capability on parameter group lists (i.e., slaves list, masters list, peers list).

Master-slave groups may be formed between parameters on the same device, parameters on different devices, or both. Once formed, value changes on the master parameter will be relayed to the nominated slaves.

The set-up and teardown of parameter groups involves the most complex messaging required of an AES-64 device. Two utilities demonstrate the set-up of a master-slave parameter group. `a64_join_setup_1` sets up a master-slave parameter group between two parameters on the same device (Fig. 7.13); `a64_join_setup_2` sets up a master-slave parameter group between two parameters on two different devices (Fig. 7.14).

In the first case, the master-slave group is setup internally within the single device involved[2]. In the second case, the device holding the master parameter conducts an exchange of messages with the device holding the slave parameter: this exchange is defined in Appendix A Subsection A.9.4.

[2]When the master and slave parameters are on the same device, processing of setup, teardown and value updates must take place without sending IP messages; the slave and master list entries are therefore stored with an (obviously illegal) IP address value of '255.255.255.255'.

```
aes64_tools$ ./a64_join_setup_1 169.254.218.31
Sending JOIN MSTSLV request. Master param is Audio Output 1 Level Mute
Slave param is Audio Output 2 Level Mute
aes64_tools$ ./a64_get_slvgrp 169.254.218.31
1 slave(s) on target parameter on device 169.254.218.31
Slave #0:
        has device address 255.255.255.255
        and dnode ID 1 and param ID 19
aes64_tools$ ./a64_get_mstgrp 169.254.218.31
1 master(s) on target parameter on device 169.254.218.31
Master #0:
        has device address 255.255.255.255
        and dnode ID 1 and param ID 17
aes64_tools$ ./a64_join_teardown_1 169.254.218.31
Tearing down the master-slave join...   ...done.
aes64_tools$ ./a64_get_mstgrp 169.254.218.31
No masters on target parameter on device 169.254.218.31
aes64_tools$ ./a64_get_slvgrp 169.254.218.31
No slaves on target parameter on device 169.254.218.31
aes64_tools$
```

Figure 7.13: Set-up, verification and teardown of a master-slave group between two parameters on the same device (see Footnote 2 for clarification of '255.255.255.255' address).

```
aes64_tools$ ./a64_join_setup_2 169.254.75.100 169.254.217.103
Sending JOIN MSTSLV request.
Master param is Audio Output Level 1 Mute on device 169.254.75.100
Slave param is Audio Output Level 1 Mute on device 169.254.217.103
aes64_tools$ ./a64_get_slvgrp 169.254.75.100
1 slave(s) on target parameter on device 169.254.75.100
Slave #0:
        has device address 169.254.217.103
        and dnode ID 1 and param ID 17
aes64_tools$ ./a64_get_mstgrp 169.254.217.103
1 master(s) on target parameter on device 169.254.217.103
Master #0:
        has device address 169.254.75.100
        and dnode ID 1 and param ID 17
aes64_tools$ ./a64_join_teardown_2 169.254.75.100 169.254.217.103
Un-joining master-slave join......done.
aes64_tools$ ./a64_get_slvgrp 169.254.75.100
No slaves on target parameter on device 169.254.75.100
aes64_tools$ ./a64_get_slvgrp 169.254.75.100
No slaves on target parameter on device 169.254.75.100
aes64_tools$
```

Figure 7.14: Set-up, verification and teardown of a master-slave group between two parameters on different devices.

Once the parameter group has been set up, it can be verified by querying the master parameter to return a list of its slaves ('GET SLVGRP'), or querying the slave parameter to return a list of its masters ('GET MSTGRP'). Examples of verification for the first type of join (master and local slave parameter) are shown in Fig. 7.13. Examples of verification for the second type of join (master and remote slave parameter) are shown in Fig. 7.14.

For the lifetime of the join, a value update to the 'master' will relay a value update to the 'slave' parameter. Once the join is torn down, no further value updates will be relayed. This behaviour is shown for the first type of join in Fig. 7.15 and for the second type of join in Fig. 7.16.

```
aes64_tools$ ./a64_level_check 169.254.217.103
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_mute 169.254.217.103 o 1 e
Muted audio output 1 on device at 169.254.217.103
aes64_tools$ ./a64_level_check 169.254.217.103
Querying level of audio output 1...     Audio output 1 level is: 0.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_mute 169.254.217.103 o 1 d
Un-muted audio output 1 on device at 169.254.217.103
aes64_tools$ ./a64_join_setup_1 169.254.217.103
Sending JOIN MSTSLV request. Master param is Audio Output 1 Level Mute
Slave param is Audio Output 2 Level Mute
aes64_tools$ ./a64_mute 169.254.217.103 o 1 e
Muted audio output 1 on device at 169.254.217.103
aes64_tools$ ./a64_level_check 169.254.217.103
Querying level of audio output 1...     Audio output 1 level is: 0.
Querying level of audio output 2...     Audio output 2 level is: 0.
aes64_tools$ ./a64_join_teardown_1 169.254.217.103
Tearing down the master-slave join...   ...done.
aes64_tools$ ./a64_mute 169.254.217.103 o 1 d
Un-muted audio output 1 on device at 169.254.217.103
aes64_tools$ ./a64_level_check 169.254.217.103
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 0.
aes64_tools$
```

Figure 7.15: Group value updates in effect between a master and slave parameter on the same device.

'Peer-to-peer' parameter groups and 'relative' value relationships have not been im-

```
aes64_tools$ ./a64_level_check 169.254.75.100
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_level_check 169.254.224.106
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_join_setup_2 169.254.75.100 169.254.224.106
Sending JOIN MSTSLV request.
Master param is Audio Output Level 1 Mute on device 169.254.75.100
Slave param is Audio Output Level 1 Mute on device 169.254.224.106
aes64_tools$ ./a64_mute 169.254.75.100 o 1 e
Muted audio output 1 on device at 169.254.75.100
aes64_tools$ ./a64_level_check 169.254.75.100
Querying level of audio output 1...     Audio output 1 level is: 0.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_level_check 169.254.224.106
Querying level of audio output 1...     Audio output 1 level is: 0.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_mute 169.254.75.100 o 1 d
Un-muted audio output 1 on device at 169.254.75.100
aes64_tools$ ./a64_level_check 169.254.75.100
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_level_check 169.254.224.106
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_mute 169.254.75.100 o 1 e
Muted audio output 1 on device at 169.254.75.100
aes64_tools$ ./a64_join_teardown_2 169.254.75.100 169.254.224.106
Un-joining master-slave join......done.
aes64_tools$ ./a64_mute 169.254.75.100 o 1 d
Un-muted audio output 1 on device at 169.254.75.100
aes64_tools$ ./a64_level_check 169.254.75.100
Querying level of audio output 1...     Audio output 1 level is: 10.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$ ./a64_level_check 169.254.224.106
Querying level of audio output 1...     Audio output 1 level is: 0.
Querying level of audio output 2...     Audio output 2 level is: 10.
aes64_tools$
```

Figure 7.16: Group value updates in effect between a master and slave parameter on separate devices.

plemented, but the implementation of 'master-slave' parameter groups fulfils the major prerequisites for a future implementation of these features.

7.2.8 Limitations of the AES-64 implementation

The limitations of the AES-64 implementation are presented in two contexts:

1. as a means of control for audio network devices based on the XMOS Ethernet AVB reference design;

2. as an implementation of the AES-64 standard.

Limitations with regard to the control of Ethernet AVB devices

As indicated in the introduction to this chapter, the scope of the AES-64 implementation in this context was to provide Ethernet AVB connection management and volume control capabilities. The AES-64 implementation provides an implementation of control over the published XMOS Ethernet AVB 'Source' (Talker) and 'Sink' (Listener) API, as well as the Ethernet AVB media clock server. However, some limitations apply.

As discussed in Subsection 7.2.3, establishing a stream connection by individually getting and setting the VLAN ID, multicast MAC address and StreamID is not the most efficient approach: correct values for the first two parameters could be obtained by parsing an SRP Talker Advertise message whose StreamID matches the value of the third parameter. The AES-64 'StreamID' parameter could control which Talker Advertise SRP messages a Listener is allowed to respond to. Once a Stream ID value had been specified on the Listener, the Listener could retrieve the VLAN ID and multicast MAC address from any Talker Advertise SRP message with a matching StreamID.

The motivations for *not* using this approach were:

- Getting and setting individual values for stream connection management focuses quantitative measurements on the latency *added* by processing AES-64 'get / set' messages. Statistically, more opportunities to measure processing latency are preferable to fewer opportunities. The 'more efficient' approach would be superior for an application but inferior for the purposes of evaluating performance.

- Presenting quantitative measurements in terms of a 'worst case' establishes a baseline for the maximum expected latency contributed by AES-64 processing. For example, another optimisation that could improve performance would be to send connection management messages with indexed parameter targets, making it

unnecessary to parse an AES-64 full address block to perform each 'get' and 'set' request.

Lastly, the AES-64 implementation is presently limited to running on the XMOS XS1-G4 class of microcontrollers. This is a resource issue. Fig. 7.17 shows the distribution of software tasks for an Ethernet AVB device incorporating the AES-64 implementation across the logical cores of an XS1-G4 microcontroller. Fig. 7.18 shows the distribution of software tasks for an Ethernet AVB device *without* the AES-64 implementation across the logical cores of an XS1-L2 microcontroller. There are insufficient unused logical cores to integrate the AES-64 implementation in its current state, except perhaps at the expense of removing either the Talker or the Listener task from the Ethernet AVB reference design.

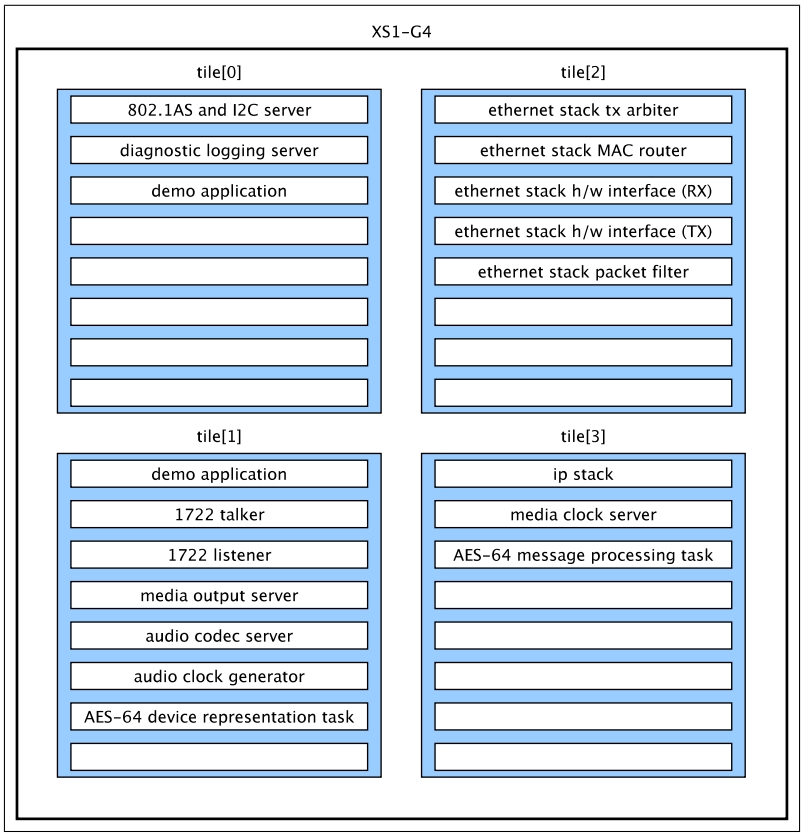


Figure 7.17: Task mapping of the AES-64-capable Ethernet AVB device on an XS1-G4 microcontroller. The 'demo application' task incorporates the AES-64 stack control task.

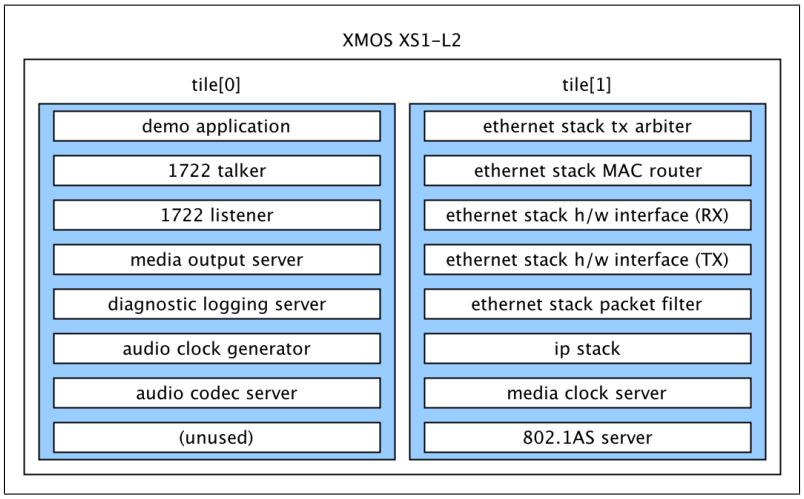


Figure 7.18: Mapping of AVB end station software tasks on the XMOS 'low-cost AVB endpoint'.

Limitations with regards to implementation of the AES-64 standard

The major AES-64 features that are *not* supported by the implementation are:

parameter flags — each AES-64 parameter is intended to hold a *flags register* [Audio Engineering Society 2012, 25-29] which external messages can retrieve, set or clear elements of. These flags are primarily intended to implement basic state control over parameter values (e.g., a number of flags are dedicated to implementing security measures), and the remainder are only of relevance to the following excluded features;

device-to-controller 'push' mechanism — an 'set-and-forget' update mechanism [Audio Engineering Society 2012, 29-30] where a device can be configured to regularly send status updates to a controller, supplying the values of a set of interesting parameters;

'modifiers' — specialised parameters [Audio Engineering Society 2012, 36-46] that can be used, in combination with parameter groups, to process and alter parameter values, address blocks, or message timing.

'desk items' — each 'desk item' is an embedded graphical representation [Audio Engineering Society 2012, 47-49] of a parameter type that may be retrieved and rendered by a desktop controller, and then linked via a parameter group, to provide standardised representation and control of parameters of that type.

The AES-64 implementation supports 18 of the 25 commands identified in the AES-64 requirements specification (Appendix A, Table A.8 and A.9). The unsupported commands are:

ACT SNP and SAVE SNP — these commands save or recall 'snapshots' of a set of parameters within the device node. This is discussed briefly in the published specification [Audio Engineering Society 2012, 22-23] but is not considered to be a core feature.

CLEAR FLAG, GET FLAG and SET FLAG — these commands perform erase, read and write operations on the parameter flags register described above.

JOIN PTP and UNJOIN PTP — this more sophisticated 'peer-to-peer' form of parameter grouping is not supported by the current implementation, but the proof-of-concept implementation of 'master-slave' parameter grouping fulfils the major prerequisites.

Lastly, the AES-64 implementation supports all of the messaging patterns derived in Subsection 5.2.3.

7.3 Quantitative evaluation: the AES-64 implementation

This section presents quantitative evaluation of the AES-64 implementation, focusing on the latency (or reaction time) of the AES-64 implementation to various external commands. Given the division of command processing into three cooperating tasks and the real-time requirement of the audio application area, this issue was of key interest. The timing measurements used to investigate this issue are presented, alongside minor quantitative metrics.

7.3.1 Timing measurements methodology

The timing measurements presented in this section were produced through a fixed test sequence involving a PC running the AES-64 control utilities and two DSP4YOU AVBStreamer devices running the AES-64 implementation, as follows:

1. Power on both Ethernet AVB devices;
2. Using the `a64_discover` utility, discover and enumerate both devices;
3. Using the `a64_connect` utility, set up an Ethernet AVB stream between the two DSP4YOU devices;
4. Using the `a64_channel` utility, set a multicore-channel-to-media-buffer mapping on the Listener;
5. Using the `a64_disconnect` utility, tear down the Ethernet AVB stream by detaching the Listener;
6. Power off both Ethernet AVB devices.

This test sequence was repeated fifty (50) times in ten blocks of five cycles, while AES-64 traffic was captured using the WireShark network analyser. Each AES-64 command operation was sent as a 'full address block request, requiring response', meaning that the device(s) sent a response on completion of every command. The time deltas between commands and the resulting response were catalogued and compiled as follows:

1. The latency between the network discovery 'GET VAL' request and the earliest response from either of the two devices;
2. The latency between the stream connection 'GET VAL' request that retrieves the Talker's VLAN ID parameter value, and the device's response containing the VLAN ID;

3. The latency between the stream connection 'GET VAL' request that retrieves the Talker's multicast MAC address parameter value, and the device's response containing the multicast MAC address;
4. The latency between the stream connection 'GET VAL' request that retrieves the Talker's StreamID parameter value, and the device's response containing the StreamID;
5. The latency between the stream connection 'SET VAL' request that sets the Listener's VLAN ID parameter value, and the Listener's acknowledgment response;
6. The latency between the stream connection 'SET VAL' request that sets the Listener's multicast MAC address parameter value, and the Listener's acknowledgment response;
7. The latency between the stream connection 'SET VAL' request that sets the Listener's StreamID parameter value, and the Listener's acknowledgment response;
8. The latency between the stream connection 'SET VAL' request that sets the Listener's 'Listen' parameter value [to ENABLED], and the Listener's acknowledgment response;
9. The latency between the channel mapping 'SET VAL' request that sets the Listener's 'Channel Map' parameter, and the Listener's acknowledgment response;
10. The latency between the stream disconnection 'SET VAL' request that sets the Listener's 'Listen' parameter value [to DISABLED], and the Listener's acknowledgment response.

The timings were subsequently extracted from the WireShark captures, compiled, and are presented in Appendix F. A summary of the findings follows.

7.3.2 Findings

Table 7.1 presents the mean and median latency for each of the tested operations in microseconds. The mean and median values for these operations fall within the commonly-cited latency requirement of 5 milliseconds (or 5000 microseconds) [Gross 2006, 65], including the compound operation of establishing a stream connection which takes an average of 2.175 milliseconds[3].

[3] As discussed earlier, this approach to establishing stream connections may be made more efficient by modifying the XMOS application to automatically recover the VLAN ID and multicast MAC address settings from Talker Advertise frames matching the specified StreamID.

This is a satisfying result, particularly given that all of these operations are performed using 'full address block' requests. In a real-world scenario, an AES-64 controller would reasonably be expected to enumerate all device(s) and issue indexed requests, obviating the processing overhead involved with parsing a 'full address block' against the target device node's level hierarchy. As such, these measurements can be interpreted as 'worst case' latency timings.

Table 7.1: AES-64 'full address block' request processing times in microseconds

Request	Mean (μs)	Median (μs)
Network discovery:		
Get 'IP Address' param value	423	415
Stream connection setup:		
Get Talker 'VLAN ID' param value	323	323
Get Talker 'multicast MAC address' param value	258	258
Get Talker 'Stream ID' param value	361	382
Set Listener 'VLAN ID' param value	263	258
Set Listener 'multicast MAC address' param value	364	383
Set Listener 'Stream ID' param value	292	297
Enable Listener 'LISTEN' param	316	325
Total	2175	
Stream channel assignment:		
Set Listener 'CHANNEL MAP' value	1092	430
Stream connection teardown:		
Disable Listener 'LISTEN' param	355	351

For most of the operations the mean and median timing measurements track very closely, demonstrating very consistent performance; this can be verified by referring to Appendix F.

The exception to this is the 'stream channel assignment' operation, which displays wide variance: a maximum reaction time of 5.4 milliseconds, or 5412 microseconds, was recorded during the test sequence. The value function invoked in this operation (Listing 6.29) is substantially more complex than the others, not least since the state of the Listener unit has to be toggled from its current state to 'disabled' and back to 'potential' in order for the new value to take effect [XMOS 2011, 62].

Additionally, the 'SET VAL' request is addressing one element of the array used to configure the entire Listener unit. The XMOS Ethernet AVB API only allows the entire array to be overwritten. Consequently, before the new value can be set on a single element, the current values of the array need to be retrieved.

In total, the AES-64 value function for this parameter performs the following tasks via

the Ethernet AVB API (Listing 6.29):

1. Get the current configuration of the Listener unit's channel map;
2. Get the current state of the Listener unit;
3. Set the state of the Listener unit to 'disabled';
4. Set the addressed element of the retrieved channel map to its new value;
5. Write the updated channel map to the active configuration;
6. Set the state of the Listener unit to its original state.

Despite the observed variance in these tests, the performance of the channel mapping request is still within acceptable standards. However, further investigation and optimisation is clearly required.

7.4 Qualitative assessment: the XMOS XS1 architecture

On the basis of the AES-64 implementation, the XMOS XS1 architecture appears to present a highly capable platform for development of audio networking control standards and protocols.

The XC language provides flexible, coherent and powerful tools for structuring concurrent software, and additionally enables the straightforward integration of software modules written in ANSI C. In terms of XC, the AES-64 implementation has demonstrated:

- a clear and straightforward path from design to implementation as a set of communicating XC tasks;
- the integration of those communicating tasks into an existing XC concurrent application;
- the ease of integrating C software modules into a concurrent embedded application by implementing data and event communications in XC;
- efficient and elegant support for parallel replication of tasks.

The XS1 microcontroller provides highly desirable features for the development of audio control protocols. Event-based processing and scheduling enables worst-case execution time estimation, which in turn enables the implementation of an Ethernet AVB-compatible MAC layer. In other respects, the determinism of the XS1 processor

facilitates direct interfacing to external hardware with critical timing commitments, as is the case with the CS4272 audio codec.

The availability of open-source software components (e.g., the Ethernet AVB reference design, the Ethernet and IP stack reference designs, the I^2C component) greatly simplifies the process of integrating the AES-64 implementation with the hardware and/or software subsystems it represents and controls.

7.4.1 The XC language

The research project has demonstrated that the XC language facilitates a concurrent implementation of the AES-64 stack which enables the logical structure and execution of AES-64 tasks to be clearly expressed.

For example, when the stack control task needs to submit a request for processing, it sends the 'PROCESS REQUEST' command and the message data through a channel to an available message processing task. This interaction is a basic producer-consumer problem. Implementing this basic interaction safely in a shared-memory system would require the use of semaphores and mutexes (or higher-level primitives), as well as an underlying operating system of some kind (Fig. 7.19).

However, an AES-64 implementation requires advanced interactions: for example, while a stack control task and message processing task might communicate inbound requests through one section of shared memory, communicating *inbound responses* to previously-sent requests (as required by parameter group operations) would need to be communicated through another section of shared memory with its own semaphores and mutexes.

As another example, implementing interactions between multiple producers and a single consumer, as in the situation where a single device representation task services requests from multiple message processing tasks, would be substantially more complex in a shared-memory system.

The AES-64 standard defines a range of message processing behaviours, from single request messages that produce multiple response messages (in the case of a wildcarded 'GET VAL' request), to single request messages that produce secondary requests and response processing (in the case of the 'JOIN' and 'UNJOIN' requests). An AES-64 implementation must support all of these message processing behaviours. It must also guard message processing from interruption, e.g., new request messages must be queued for processing if a previous request has not yet been completely processed. Implementing the AES-64 standard as a set of concurrent co-operating tasks is far simpler than implementing it as a single-threaded application (which is technically possible, as the

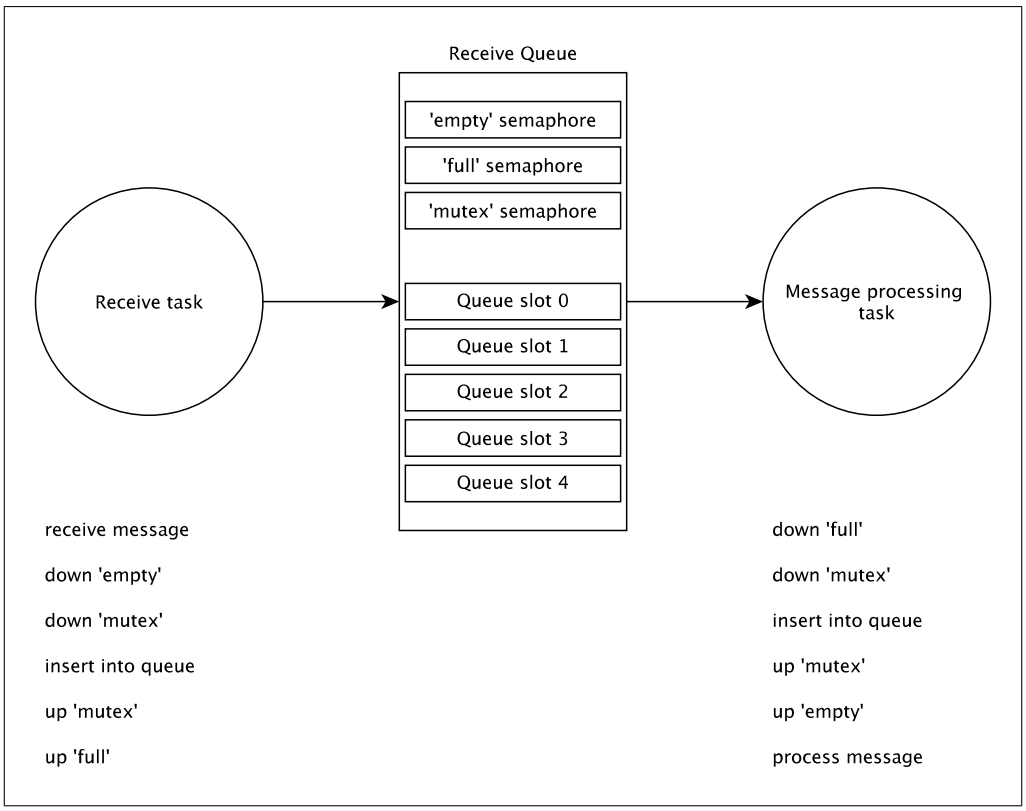


Figure 7.19: AES-64 requests are passed between a Stack Control Task and a Message Processing Task in a shared-memory system, requiring the use of semaphores and mutexes.

prototyping process demonstrated, but far from advisable): however, implementing the AES-64 standard as a set of concurrent tasks that communicate by passing messages over XC channels is significantly simpler than implementing it as a set of concurrent tasks that communicate through shared memory.

The high-level result of using the XC language in the AES-64 implementation is that it has enabled a complex set of requirements to be structured through a well-established design process, and translated efficiently and lucidly in program code. Each task in the AES-64 implementation presents clear interfaces for interaction with other tasks, including external tasks (e.g., the Ethernet AVB and audio control subsystems). The command and communication protocols implemented over XC channels make explicit the data flows between tasks, with the result that subtle conditional processing of inputs or outputs (e.g., the stack control task's processing of outbound requests that expect responses) can be expressed without being obscured by implementation logistics.

Integrating modules written in ANSI C into one or more tasks of an XC application is straightforward and effective: a method for interfacing between C modules and XC channels was described in Chapter 4, and applications of this method are described throughout Chapter 6. A few unusual caveats apply, as presented in Subsection 6.3.5: C code should be mindful of the XMOS XS1 compiler's requirement to perform static analysis on each function within the binary, which inhibits (but does not in fact prohibit) use of recursive algorithms and function pointers.

As described in Subsection 6.3.5, this presented some obstacles for the implementation of the AES-64 level hierarchy (the linked-list-of-linked-lists approach, requiring a recursive parser) and parameter value functions (requiring the use of function pointers). However, for the uses intended within the AES-64 implementation, the most appropriate response was to implement alternative mechanisms (as described in Subsection 6.8.3 and Section 6.6 respectively).

The XC language provides powerful abstractions for constructing adaptable concurrent applications. The use of the parameterised select mechanism with a replicator enables elegant parallel replication of the AES-64 implementation's message processing task: additional instances of this task (within the resources of the XS1 processor) may be added simply by editing a precompiler constant and adding channel and task declarations to the top-level `main` function, as described in Section 6.11.

Within XC itself, some may find the prohibition on use of pointers problematic. However, the pass-by-reference features added to XC are a suitable replacement for pointer manipulation in most contexts; where this is not the case, it is straightforward to integrate C code within an XC task without surrendering the major benefits of XC. This

has been done with each component task of the AES-64 implementation (Section 5.4, Section 6.5).

7.4.2 The XMOS XS1 microcontroller

The XMOS XS1 microcontroller directly enables most of the benefits attributed to the XC language and the XMOS development environment, such as:

- the use of channels for task communication,
- 'fair' task scheduling and the determinacy it enables,
- the ability to build and execute concurrent programs without the overhead and indeterminacy of a real-time operating system,
- the use of `select` by tasks that need to wait fairly on large and diverse sets of events from the concurrent program or external hardware.

The microcontroller's implementation of channel communications is particularly elegant: for the purposes of an XC program, it does not matter whether each end of a given channel are resident on the same processing tile, on different processing tiles within a package, or even resident on different processing packages. This offers the prospect of constructing very large scalable programs with ease. In simpler configurations, the flexibility of the channel system maximises the use of limited resources and means that demanding applications like the Ethernet AVB reference design can still accommodate the integration of something like the AES-64 implementation, which itself has significant task communication requirements.

The microcontroller's support for determinism (or, alternatively, for worst-case execution timing estimates) is critical to its value as a platform for the development of audio control protocols. Without this support for determinism, an implementation of the Ethernet AVB networking standards would be impossible;

Memory is a significant constraint of the XS1 architecture. As described in Sections 4.1, Subsection 6.3.5 and Section 6.4. Since the 64KB available to each XS1 processing tile is used to store executable code in addition to stack and heap memory, the development of the AES-64 implementation involved regular optimisation drills to minimise the memory footprint of the executable's tasks and functions. Here, the development environment provided static analysis of compiled program binaries, allowing the developer to pinpoint functions or data structures that were consuming the greatest amount of any given processing tile's storage. In the context of audio control protocols, implementing substantial

representation schemes, complicated processing behaviour, or very large sets of control actions on this architecture may involve challenges.

Chapters 5 and 6 have shown that the path from the established RTSAD design method to an XC implementation of the AES-64 standard was a linear one. For the greater part, obstacles encountered during the design process were typically caused by oversights made in the requirements analysis process, and were rarely introduced by the XC language or characteristics of the XS1 architecture. Lastly, the XMOS development environment's debugger (in combination with the XTAG2 programming adapter) provided very capable monitoring and debugging of the AES-64 implementation executing on the target hardware.

7.4.3 Reference designs

The XMOS reference designs provide an extensive basis for development of audio network devices and control standards. The XMOS open-source implementation of the Ethernet Audio/Video Bridging standards provides a compelling platform for the development of monitoring or control standards, including the development of high-level PC control software. The availability of the Ethernet AVB implementation in the first place is a substantial motivation to use the XMOS XS1 as a platform for audio control standard development.

The range of open-source components (such as the XMOS IP stack and I^2C component) simplified the extension of AES-64 control to incorporate control of audio hardware. Once the relevant register addresses and values on the CS4272 audio codec were identified from its published datasheet, it was straightforward to implement value functions that read and wrote to those register values via XC channels and the I^2C component. The amendments necessary to integrate hardware control routines, value functions and AES-64 parameters for the CS4272 audio codec within the Ethernet AVB reference design (as described in Subsection 6.10.3) were simple to conceptualise and implement.

7.4.4 Limitations

Memory constraints

The most significant technical limitation of the XMOS XS1 architecture is the memory constraint: 64KB per processing tile, such that the XS1-G4, used to develop the AES-64 implementation, has in total 256KB. An interface to external memory could in theory be implemented, but the available target hardware did not support this. This memory must

accommodate the executable for that tile in addition to the executable's stack and heap requirements.

This presented difficulties for the AES-64 implementation, which have been described in the previous section. Given that a key motivation for using the XMOS XS1 architecture in research and development of audio control standards is the Ethernet AVB reference design, and given that the Ethernet AVB reference design of course has its own independent memory footprint, the AES-64 implementation benefitted from thorough optimisation. For example, one area of the implementation that demanded a great deal of attention was the full address block parser, both for generic AES-64 'full address block' parameter requests and for the specialised processing required of 'GET CLA' and 'GET NAME' commands.

Documentation

This project engaged with a significantly complex concurrent application to develop and integrate a novel implementation of device representation, message parsing and command processing. As might be expected, in places this process was less than straightforward.

Some aspects of working with the current version of the XC language are at first unfamiliar, and at the outset of this research project the scarcity of published documentation on the advanced use of channels, or the integration of C code, was a hindrance. Meanwhile, the XMOS reference designs do provide thorough, if somewhat complex, demonstrations of channel communications and the integration of C code, and the language's implementation of these concepts is consistent and elegant, once the reader is familiar with the basic idioms.

The difficulties encountered were chiefly due to a scarcity of available and accessible documentation of 'best practices' for software development on the XMOS XS1 architecture. For example, the most complete illustration of communicating AES-64-sized messages over channels was provided by the XMOS TCP/IP reference design, which is an advanced and 'difficult' program to 'read'. Further examples include the issues with use of recursive functions and function pointers in C code (as described in Subsection 6.3.5), descriptions of which could not be found in the published literature.

This is not a technical limitation but a social or educational condition. During the later phases of the research project, XMOS began (and continue) to address this situation directly with expanded tutorials.

7.5 Conclusion

The AES-64 implementation provides capable connection management and control of the XMOS Ethernet AVB reference design and additional features (e.g., audio hardware control), while also providing proof-of-concept implementations of significant AES-64 mechanisms.

A set of rigorous timing measurements has established that the AES-64 implementation performs routine application tasks - network discovery, stream and internal connection management - consistently within commonly-cited timing requirements for the application area.

The implementation of the AES-64 standard on the XMOS Ethernet AVB reference design has illustrated the strengths (as well as a few limitations) of the XMOS XS1 architecture as a platform for the research and development of audio control standards. In this analysis, the architecture's most significant strengths are:

- the range and quality of the XMOS open-source reference design components, such as:
  - the XMOS Ethernet AVB reference design, which by itself motivates highly for audio control standard research employing XMOS XS1 systems;
  - the I^2C component, which made implementing audio hardware control efficient and fast.
- the microcontroller's flexible support for concurrency, hardware input-output and task communications, which directly enable the powerful XC language and indirectly enable the implementation and integration of demanding concurrent systems (e.g., the Ethernet AVB reference design and the AES-64 implementation);
- the XC language, which provides clear, powerful tools for implementing and integrating concurrent software, particularly:
  - use of the `select` statement to wait on sets of events from diverse sources;
  - its ability to integrate and interface software modules written in ANSI C;
  - use of parameterized select statements and replicators to fairly wait on events from equivalent but distinct sources;
  - the clarity of channel input-output and transactions to implement complex data exchange between tasks;
  - enabling the use of a well-established design method, offering an unusually direct route from design to implementation.

8 Conclusion

8.1 Introduction

This thesis has investigated whether the XMOS XS1 architecture is a viable platform for the research and development of audio control standards, focusing on the benefits and limitations of the XC language and its approach to concurrency and event-driven programming. This was achieved by researching and developing formal requirements for the AES-64 audio control standard, designing and developing an implementation of the standard for the XMOS XS1 architecture, and integrating the implementation with the XMOS open-source implementation of an Ethernet AVB streaming audio device.

The AES-64 implementation was then evaluated in qualitative and quantitative terms by considering the completeness of the software product as both an implementation of AES-64 and as a control system for the Ethernet AVB streaming audio device, and by compiling timing measurements to assess the latency of command processing by the software product.

The XMOS XS1 architecture was evaluated in terms of the primary research question by considering the qualities and limitations of the AES-64 implementation, the efficiency of the development process, and the contexts and extent to which the features of the XS1 architecture enabled the AES-64 implementation to fulfil its requirements, including real-time performance requirements, while doing so in a clear, expressive and extensible way.

This section presents a review of the research objectives, including a short discussion of the research. This is followed by discussion of the limitations of the research, recommendations for further work, and closing remarks.

8.2 Summary of the previous chapters

The primary objective of this research was to determine whether the XMOS XS1 architecture presents a compelling platform for the research and development of audio control standards, focusing on the architecture's associated 'XC' programming language and

its features for concurrency, event-driven programming and integration with ANSI C software modules.

Chapter 2 and 3 discussed the audio distribution domain, its technical challenges, and argued for the significance of audio control standards in this application area, before introducing the AES-64 audio control standard.

Chapter 4 discussed the XMOS XS1 architecture, its primary features (including the XC language), and presented some fundamental techniques for constructing concurrent software on this architecture.

Chapter 5 presented an account of the requirements analysis and design processes that developed a design for the AES-64 implementation, and established that software projects centered around the XMOS XS1 architecture can make profitable use of well-established design methods.

Chapter 6 examined the implementation of the AES-64 protocol stack in detail, focusing on the structuring of the protocol stack as a set of co-operating tasks and the XC language's capacity to express such structuring. The chapter asserted the efficiency of moving from the RTSAD design to an XC implementation, and of integrating this implementation into an existing software project of some complexity.

Chapter 7 evaluated the software product of the research both as an implementation of the AES-64 audio control standard and as a means of controlling the Ethernet AVB streaming audio device, and presented an evaluation of the XMOS XS1 architecture's viability as a platform for research and development of audio control standards.

8.3 Review of the research question

This research investigated whether the XMOS XS1 architecture and its supporting features could offer a compelling platform for the research and development of audio control standards.

The research approached this question by designing and implementing a protocol stack for the AES-64 audio control standard and integrating it into the XMOS implementation of an Ethernet AVB streaming audio device. The final software product was then evaluated against qualitative criteria and quantitative criteria, considering the software product separately as an implementation of the audio control standard and as a means of controlling the Ethernet AVB streaming audio device.

The research contributes to the body of research on audio control standards by demonstrating that this flexible and powerful architecture is a compelling platform for the development of audio control standards, and by implementing AES-64 connection

management and control over standards-compliant Ethernet AVB streaming audio devices where no such implementation previously existed. The research contributes to the literature concerning the XMOS XS1 architecture by describing a linear design method and various useful implementation techniques.

These findings may be of use to the research of high-level control applications for use with Ethernet AVB or other audio distribution networks. The AES-64 implementation forms an extensible basis for more sophisticated applications: it may be configured as required to provide connection management and control services over different configurations of the Ethernet AVB streaming audio device, and an example of its extension to implement control over the external audio hardware has been demonstrated. Additionally, the requirements documentation developed by the research provides a baseline implementation reference where none previously existed.

8.4 Review of the research objectives

The research objectives stated in Chapter 1 are now re-evaluated by considering the work done:

- Through the process of requirements analysis and the application of an established design method, as described in Chapter 5, a requirements document (Appendix A) and design document (Appendix B) were prepared for an implementation of the AES-64 standard.

- The AES-64 protocol stack was implemented to integrate with the XMOS implementation of an Ethernet AVB streaming audio device. The XMOS implementation was extended to provide control of the target hardware's external audio codec. The AES-64 protocol stack provided proof-of-concept implementations of advanced AES-64 control mechanisms. These implementations thoroughly investigated the capabilities and limitations of the XMOS XS1 architecture to support the development of audio control standards, focusing on the XC language's mechanisms for concurrency, event-driven programming and integration of ANSI C software modules. This work is described in Chapter 6.

- The assessment of the AES-64 protocol stack, both as an implementation of the standard and as a means of controlling Ethernet AVB streaming audio devices, was carried out through qualitative and quantitative evaluation of the stack's performance under sustained testing, as described in Chapter 7. Qualitative evaluation (Section 7.2) found that the AES-64 protocol stack provides a comprehensive control

system for the Ethernet AVB streaming audio device; additionally, considered as an implementation of the AES-64 standard, the stack implements most of the major AES-64 control features and all of the messaging patterns. Quantitative evaluation (Section 7.3) found that the AES-64 protocol stack was capable of executing a variety of Ethernet AVB control tasks consistently within the commonly-cited latency requirement of 5 milliseconds: for example, the multiple operations involved in stream connection management took an average total of 2.175 milliseconds.

- The qualitative evaluation of the XMOS XS1 architecture, presented in Chapter 7, is based on the evaluation of the AES-64 protocol stack (also presented in Chapter 7) and the account of design and development presented in Chapters 5 and 6. The capabilities and limitations of the architecture to support the development of the AES-64 protocol stack, and by extension other audio control standards, were discussed in Section 7.4. The evaluation found that:
  - the XC language's features for interprocess communication and 'fair' waiting on a diverse range of events;
  - the ease of adapting legacy C code to execute within XC tasks that form a larger concurrent program;
  - compatibility with an existing design methodology (Real-Time Structured Analysis and Design);
  - open-source reference design components, including the Ethernet AVB reference design and the I^2C component

  all provide a compelling basis for research and development of audio control standards. The existence of the open-source Ethernet AVB reference design was a key motivation for using the XMOS XS1 architecture in this project; however, the project has demonstrated significant additional benefits, with few limitations. Of these, the most important limitation is the microcontroller's memory constraint: throughout the software development, regular optimisation drills managed this limitation effectively.

8.5 Achievements and limitations

8.5.1 The AES-64 implementation

Control features for Ethernet AVB devices. The AES-64 protocol stack and control application provides a comprehensive implementation of the XMOS Ethernet AVB

reference design's programming API, including control over Talker, Listener and media clock server functionality. It adds a feature by providing additional control over the DSP4YOU hardware's audio codec, providing input and output mutes and output volume control.

AES-64 control over XMOS Ethernet AVB streaming audio devices is available through the set of command-line controller applications (Appendix E) developed to enable evaluation of the protocol stack. At present this is the only means of AES-64 control over the XMOS devices; UNOS Creator, for example, cannot discover, enumerate or perform connection management. This was not a research objective, because existing AES-64 controllers such as UNOS Creator employ forms of device enumeration that have not yet been published in the official standard.

The AES-64 protocol stack can presently be deployed on XMOS Ethernet AVB devices based around the XS1-G4 microcontroller. As shown in Fig. 7.17 and Fig. 7.18, the AES-64 protocol stack cannot currently be deployed on the XMOS 'low-cost AVB endpoint', which is based around an XS1-L2 microcontroller.

Research and implementation of the AES-64 protocol stack. The AES-64 protocol stack implements the majority of features and requirements identified in the course of the research (Appendix A). It implements 18 of the 25 forms of AES-64 command, all of the AES-64 messaging patterns, and provides proof-of-concept implementations of advanced mechanisms including master-slave parameter grouping and device enumeration.

The protocol stack does not implement the AES-64 parameter flags register, the device-to-controller 'push' mechanism, modifiers, or desk items. It therefore does not implement the seven forms of AES-64 command associated with these features. Additionally, it does not implement the 'peer-to-peer' form of parameter grouping, but the implementation of the 'master-slave' form provides adequate proof of concept, demonstrating that the AES-64 protocol stack is also capable of supporting the 'peer-to-peer' form.

Evaluation of the XMOS XS1 architecture. This research has shown that the XMOS XS1 architecture forms a compelling platform for the research and development of a modern audio control standard. It has demonstrated a linear route from requirements capture to implementation on target hardware, showing that the XC language enables clear expression of characteristically complex behaviours, such as the messaging involved in the setup and teardown of a parameter group. The

research has evaluated the utility of the open-source Ethernet AVB reference design and demonstrated how it may be extended and integrated with an implementation of an IP-based control application.

8.6 Possibilities for further research

The work presented in this thesis establishes the XMOS XS1 architecture as a powerful and compelling platform for the development of audio control standards. The AES-64 protocol stack may be used as an extensible basis for research into standards-based audio distribution applications, and has been designed and implemented with this in mind.

The optimisation of the AES-64 protocol stack to enable its deployment on the XS1-L microcontroller series is a desirable possibility for further research. The XMOS development tools for the XS1-L series provide advanced instrumentation of executing code and this may be desirable for more detailed investigations of Ethernet AVB streaming audio devices. There are also a greater variety of development boards based around the XS1-L series than the XS1-G series.

This investigation has not examined the architecture and XC language features for interfacing with external hardware, since this falls outside the implementation of an IP-based protocol stack. While these features are unlikely to form an integral part of any audio control standard, it may be useful to consider them in the context of a device that combines an audio control standard implementation with a human-computer interface (e.g., a physical control panel).

A AES-64 requirements specification

A.1 Introduction

This appendix provides a requirements specification for implementing an AES-64 protocol stack on an audio networking device. For a conceptual overview of AES-64, refer to Chapter 3. For an account of the development of this requirements specification, refer to Chapter 5.

This requirements specification describes how an AES-64 protocol stack integrates into an audio network device (Section A.2 – A.4). Section A.5 specifies the data structures required to implement an AES-64 protocol stack. Section A.6 defines the messaging patterns that an AES-64 protocol stack must support. Section A.7 defines the basic AES-64 message formats. Section A.8 describes each of the command requests defined by the AES-64 standard and their associated responses. Section A.9 describes the device-to-device messaging interactions defined by AES-64.

A.2 Prerequisites

The prerequisites for a device to support the AES-64 standard are as follows:

1. The device must have at least one network interface.
2. The device must implement a UDP-capable IP stack.
3. The device must implement control software for the device's primary functions, e.g., audio, network, clock and media transport systems.
4. This control software must provide a programming interface.
5. It must be capable of running an AES-64 protocol stack.
6. The configuration of the device's AES-64 protocol stack must provide the 'configuration' parameters described in Subsection 3.3.1.

A.3 The AES-64 protocol stack

An AES-64 protocol stack enables the representation and remote control/monitoring of a networked multimedia device, communicating with other networked devices by AES-64 messages. The AES-64 protocol stack represents the device's properties through one or more device nodes, level hierarchies and parameters. Other networked devices may identify and manipulate any of these parameters to monitor or control the functions of the device, and the device may update its own parameters to reflect changes in operating state or control changes received through other interfaces. The AES-64 protocol stack also enables a range of high-level control mechanisms as defined by the AES-64 standard.

A.4 The device primary control software

The AES-64 protocol stack interacts with the device's primary control software (DPCS). The DPCS will initialise and configure the AES-64 protocol stack as it does the device's other component subsystems.

The DPCS must register and initialise the protocol stack with a set of AES-64 device nodes, level hierarchies and parameters. Each AES-64 parameter must be registered to a specific device node; within that device node, it must have a unique parameter ID and a unique position within the device node's level hierarchy, as described in Subsection 3.4.2. Each parameter must also be registered with a function or functions within the DPCS that will respond to changes in the parameter value, with the result that read/write operations on parameter values requested by AES-64 messages will make something or other happen on the device itself.

A.5 Fundamental data structures

The following data structures must be implemented within the AES-64 stack to support basic device functionality:

1. The Parameter
2. The Parameter Group List
3. The Level
4. The Device Node

This section documents the purpose, attributes and functions of each data structure.

A.5.1 The Parameter

Most control and monitoring operations defined by the AES-64 standard manipulate attributes of parameters. Parameter attributes can be changed by AES-64 messages received from remote devices and controllers, or by the DPCS.

Parameters can be classified as:

1. Control parameters, which can be written and read from;
2. Monitor parameters, which can only be read from.

Monitor parameters indicate observable states, e.g., metering the amplitude of an audio signal. Control parameters represent controllable states or variables. Monitor parameters implement 'read' value functions; control parameters implement 'read' and 'write' value functions.

The key attribute of a parameter is its value. An associated attribute expresses the *value format*.

To support the full range of AES-64 features, a parameter must also have other attributes. These, in turn, require access functions. These are shown in Table A.1 and are introduced below.

Unique ID. This 32-bit identifier uniquely identifies an AES-64 parameter within a device node. It is assigned when the parameter is attached to its parent device node. This ID *must* be the same value as the parameter's corresponding entry in the device node's level items table.

Name. This text string provides a human-readable string to identify the parameter's function. Typically it is the same as the parent 'Parameter Type' level alias.

Value format. The *value format* attribute is an 8-bit identifier associated with the parameter's *value* attribute. The default value format is the 32-bit Global Units format, as described in [N. Chigwamba, et al 2012, 11].

Value. The AES-64 stack provides storage for a 32-bit Global Units value; if the parameter represents any other kind of value, this must be stored within the device's primary control software, and it can then be read or written to via *value functions*.

Value functions. The parameter must (in some way) identify the primary control functions associated with operations (e.g., get / set) on its value, so that the AES-64 stack can initiate actions on the device primary control software in response.

Flags register. This attribute records the state of the parameter and its AES-64 attributes. It is a 64-bit bitfield indexed into 64 flags. Access functions can specify individual flags within the register by index. Specified in [Audio Engineering Society 2012, 25-29].

Master, slave and peer group lists. A parameter can be grouped with an arbitrary number of 'other' parameters on the same device or other devices. Each parameter-to-parameter grouping is maintained as a single 'relationship'.

There are three types of relationship that a parameter can maintain: with a Peer, where a value change will be relayed in either direction; with a Master, where a value change on the 'other' parameter will be relayed to the first; and with a Slave, where a value change on the first parameter will be relayed to the 'other' parameter. Since each type of relationship is handled differently, a parameter maintains three lists identifying (and classifying) any parameters that it has a relationship with (see Subsection A.5.2).

A.5.2 The Parameter Group List

A key mechanism of the AES-64 protocol is parameter grouping, which enables value changes to be tracked between sets of functional parameters on the same device or on different devices. Subsection 3.6.2 provides a conceptual overview of parameter grouping; see also Subsection A.9.3.

Each parameter maintains three *parameter group lists*:

1. the Masters group list
2. the Peers group list
3. the Slaves group list

A *parameter group list entry* describes a parameter that is grouped with the owner of the group list. The description consists of the following information:

1. The grouped parameter's device address, device node ID and parameter ID.
2. The offset between the value of the grouped parameter and the value of the owner parameter.

Table A.1: Parameter data specification

Parameter
Unique ID
Name
Value format
Value
Flags register
Parent device node
Master group list
Slave group list
Peer group list
Create parameter
Destroy parameter
Get ID
Get / Set value format
Get / Set value
Clear / Set flag
Get flag status
Get / Reset flag register
Get / Set master group list
Get / Set slave group list
Get / Set peer group list

3. The status of the group relationship, which can be either online or offline, to make it possible to enable or disable a group relationship without deleting it.

4. The join type of the group relationship is either *absolute* or *relative* (see Subsection A.9.3).

5. The join flags of the group relationship can indicate that the relationship is *permanent*, *passive*, or *paired* (see Subsection A.9.3).

Table A.2: Parameter group list data specification

Parameter group list
Entry count Type Group entries
get entry count add entry find entry find and remove entry find and replace entry clear all entries

A.5.3 The Level

Each parameter is identified and located within its parent device node by its position in the device node's *level hierarchy*, which is composed of connected *levels*.

The hierarchy is ordered by the generality of level identifiers: for example, a Section Block level (e.g., `INPUT` or `DEVICE INFO CONFIG`) is more general than a Parameter Type level (e.g., `LEVEL MUTE` or `IP ADDRESS`). The full ordering is:

1. Section Block levels

2. Section Type levels

3. Section Number levels

4. Parameter Block levels

5. Parameter Block Index levels

6. Parameter Type levels

7. Parameter Index levels

Of these, the Section Block, Section Type, Parameter Block and Parameter Type level identifiers represent strings, or *level aliases*. The Section Number, Parameter Block Index and Parameter Index levels are numeric identifiers (indexing from 1 upwards).

Every level has exactly one parent level unless it is a Section Block level, in which case it has a parent device node instead. Every level has at least one child level unless it is a Parameter Index level. Parameter Index levels point to the one and only parameter that they locate within the device node (Fig. A.1).

Each level within the hierarchy is uniquely identified by the combination of its parent level and the value of its level identifier. For example, the Parameter Index levels shown in Fig. A.1 all share the '1' identifier, but each has a different parent level.

The attributes of a Level are summarised in Table A.3 and described below.

Identifier. A level's identifier is selected from the tables of identifiers provided in the AES-64 specification [Audio Engineering Society 2012, 50-74]. The identifier uniquely identifies a level among its parent's child levels.

Type. This gives the type of the level, indicating whether it is a Section Block level, a Section Type level, etc.

Alias. This is a text string that provides a human-readable translation of the level's identifier. It may be retrieved through the device enumeration process (e.g., a 'GET CLA' request) or by a 'GET NAME' request.

Parent level reference. A reference to the level's parent level. If the level is a 'Section Block' level, this reference will be null.

Parent device node reference. A reference to the level's parent device node.

Next level reference. This is a reference to the next level that shares the level's parent level. If the parent level has only one child, this reference will be null.

Child level reference. This is a reference to the level's *first* child level. Further child levels can be reached through the first child level's *next level reference*.

Child level count. This is a simple counter of the Level's child levels.

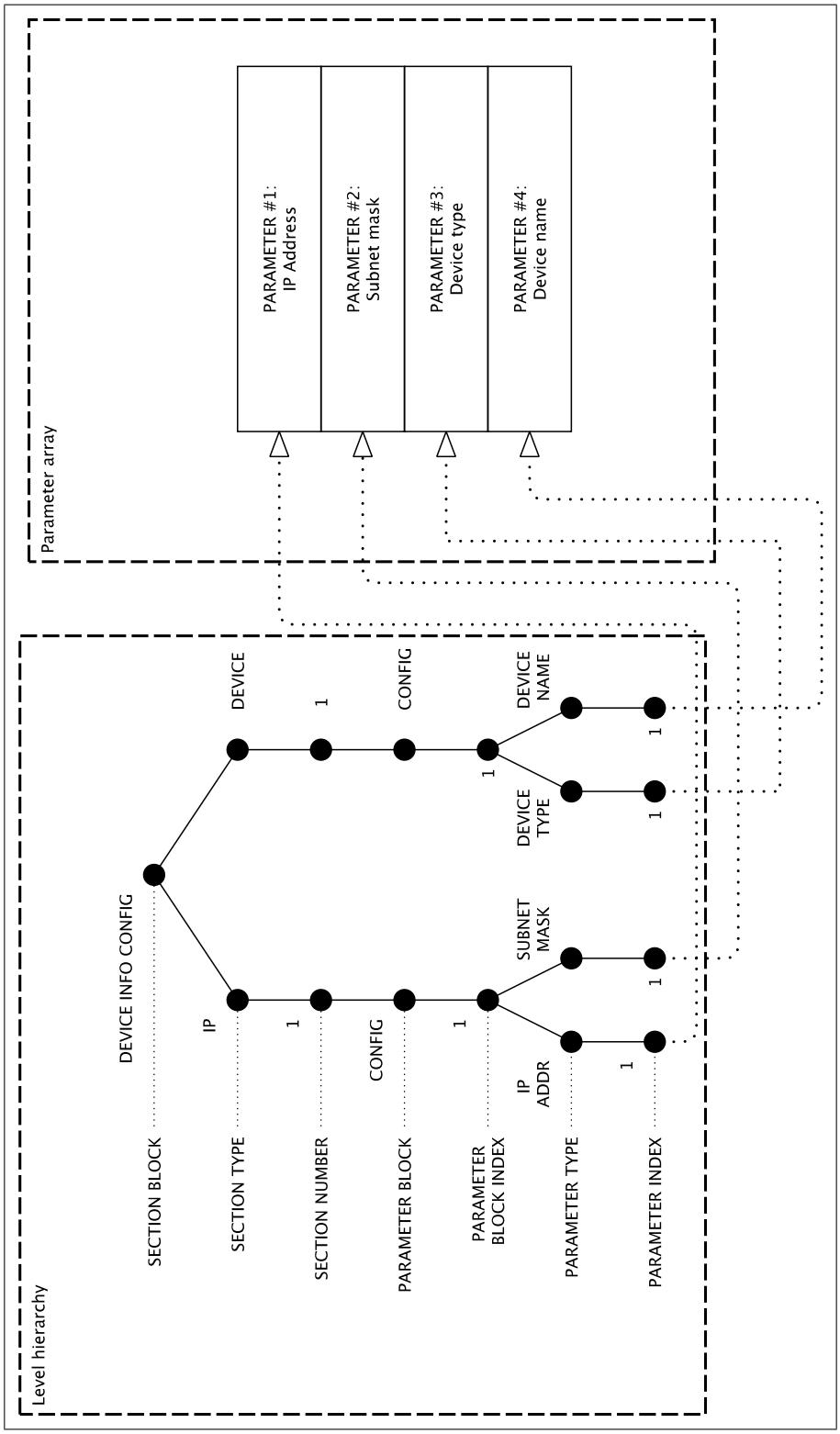


Figure A.1: Location and identification of parameters within a device node through the level hierarchy.

Table A.3: Level data specification

Level
Numeric ID
Position
Alias
Parent level reference
Parent device node reference
Next level reference
Child level reference
Child level count
create level node
find level node
get most recently created level
get ID
get alias
attach to parent level
attach to parent device node
get next level
get child level
get child level count
set child level count

Implementing the level hierarchy

There are a number of approaches to implement the level hierarchy: this section has described one of them. The level hierarchy has to be processed in a variety of ways (to locate one or more parameters from a full address block, to fulfil a GET NAME request, to fulfil a GET CLA request, etc). This approach may not be the optimal approach for a given platform.

However, the central responsibilities of the level hierarchy are:

1. To uniquely identify and locate each parameter within the device node;
2. To enable one or more parameters to be identified based on one or more wildcarded level identifiers in a full address block;
3. To enable one or more levels to be identified based on the special full address block provided with a 'GET CLA' or 'GET NAME' request (see Subsections A.8.3 and A.8.7).
4. To enable level aliases to be retrieved for enumeration purposes.

Point 1 and 2 imply that each level has an identifier, and must be able to reference its child levels and the levels with which it shares a parent. Point 3 implies that a level must be able to reference its parent level. Point 4 implies that each level has an alias string that corresponds to its identifier, and it must be possible to retrieve this alias. Any implementation that fulfils all of these responsibilities and implications is a viable one.

A.5.4 The Device Node

A device node encapsulates the representation of a single functional device. Instances of the level and parameter structures belong to one and only one device node.

The attributes of a device node are summarised in Table A.4, and described below.

Unique ID. This numeric ID uniquely identifies the device node within an AES-64 protocol stack.

Level hierarchy pointer. This pointer is linked to the device node's one and only level hierarchy.

Parameter count. This is a simple counter of the number of parameters configured on the device node. It is incremented when an initialised parameter is joined to the device node.

Parameter array. The parameters configured on the device node are stored in this array. It is important to understand that while the level hierarchy uniquely locates device parameters, they are actually *stored* in one contiguous block (see Fig. A.1). Initialised parameters are appended to this array when they are joined to a device node.

A.6 Messaging patterns

There are two basic types of AES-64 message: requests and responses:

> Every AES-64 message will either be a request sent from one device to another, or will be a response from a device after a request has been made. [Audio Engineering Society 2012, 16]

As shown in Figs. A.2 and A.3, an AES-64 request message may or may not demand a response (depending in part on the command expressed by the request; for example, a 'GET' request will always demand a response).

Table A.4: Device node data specification

Device node
Unique ID Level hierarchy pointer Parameter count Parameter array
create device node append device node to global device nodes find device node destroy device node get ID get level hierarchy pointer add parameter to parameter array

Figure A.2: An AES64 request message which does not demand a response.

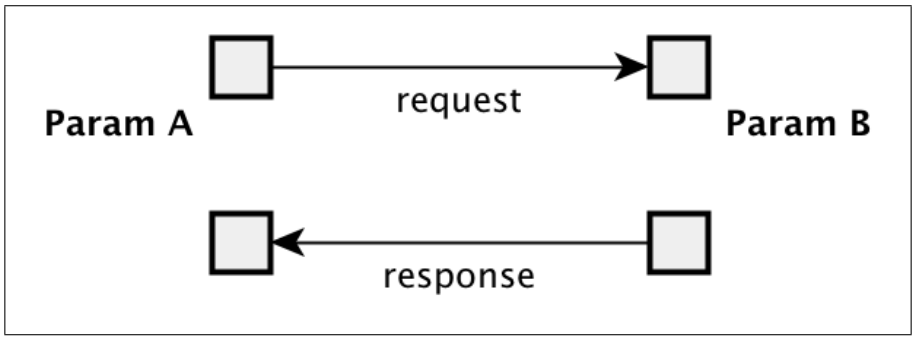


Figure A.3: An AES64 request which demands a response, and the corresponding response.

When a request that demands a response is addressed to multiple parameters, each parameter will issue a response (Fig. A.4).

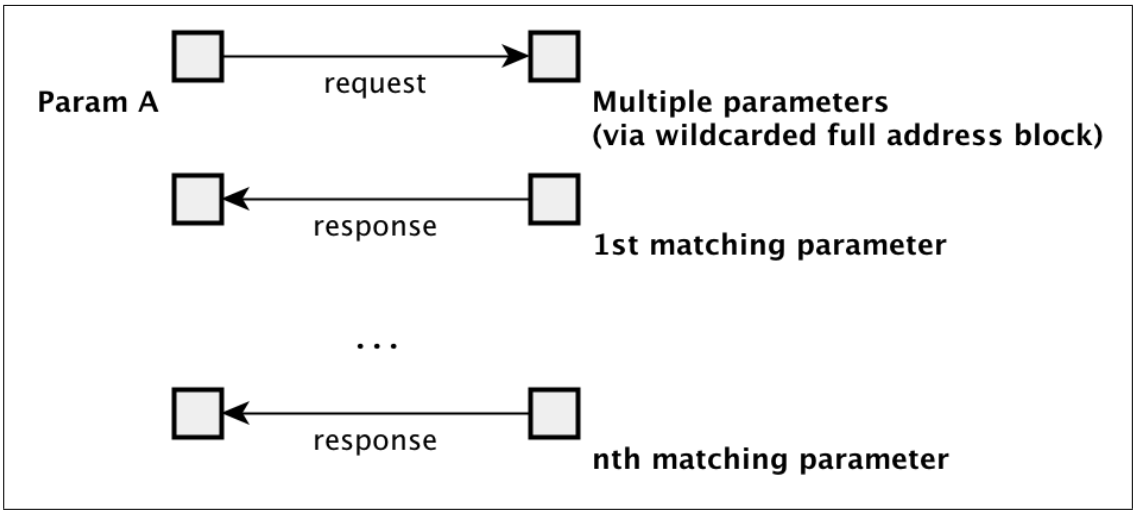


Figure A.4: An AES64 request that expects a response, addressing n parameters through the use of a wildcarded address block. Each parameter returns a response message.

Parameter grouping introduces two final messaging patterns. As described in [Audio Engineering Society 2012, 32], a value change on a parameter that is part of a group may result in that parameter issuing value updates to the other members of the group (Fig. A.5).

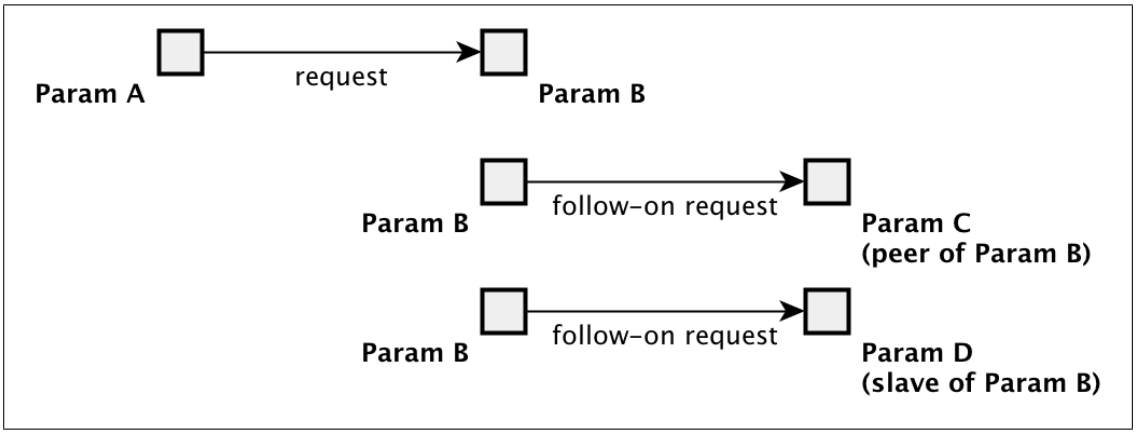


Figure A.5: An AES64 request that initiates the target parameter, which is a member of a parameter group, to issue group value update requests.

The final messaging pattern is implicated by the AES-64 specification's descriptions of parameter group setup. While the setup for a peer-to-peer parameter group [Audio

Engineering Society 2012, 33] involves a longer sequence of messages than the setup for a master-slave parameter group [Audio Engineering Society 2012, 34], they both use the same basic pattern (Fig. A.5).

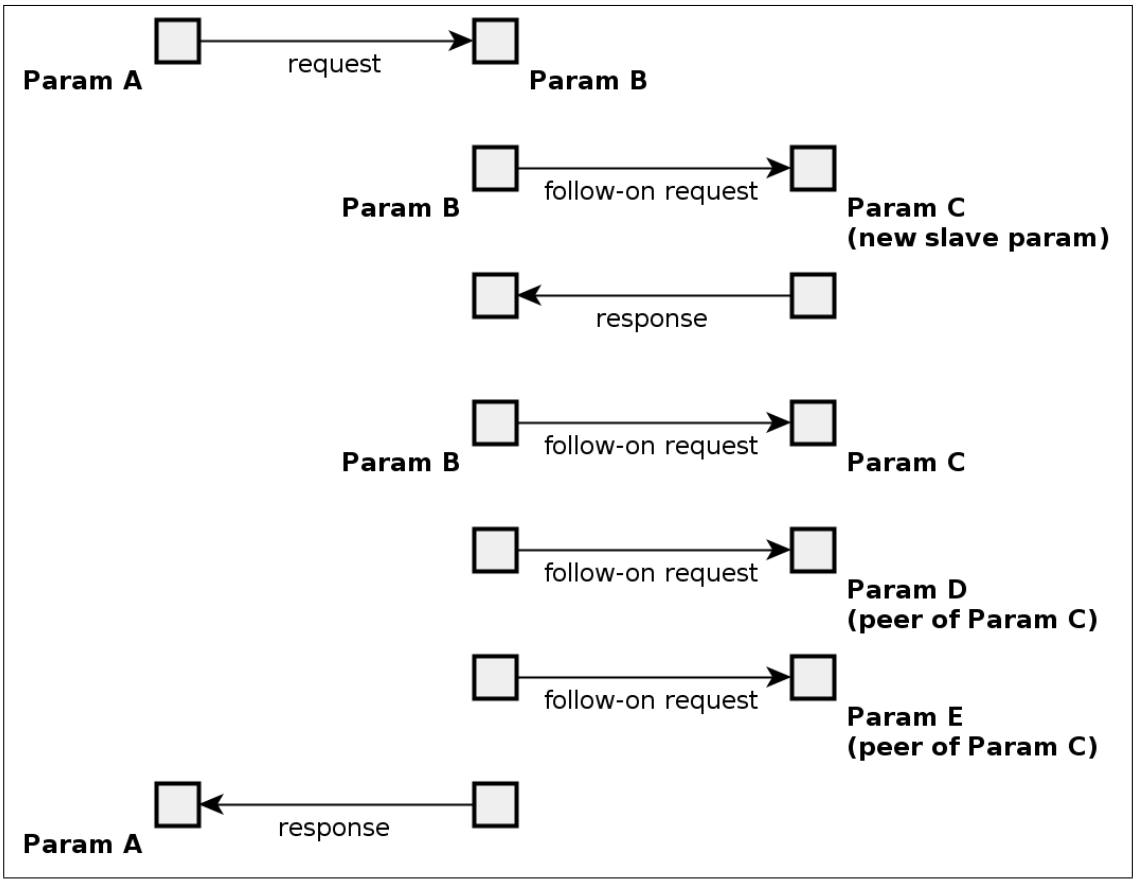


Figure A.6: An AES64 request that initiates the target parameter to set up a parameter group, which involves the target parameter issuing follow-on requests, receiving and processing responses, and issuing further follow-on requests before finally acknowledging success or failure to the source of the original request. (Details of Param C are made available to Param B in the value field of the initiating request sent by Param A.)

A.7 Message formats

Since AES-64 requests may be targeted at parameters or at a device node, there are multiple subtypes of request. All response messages use a common format (Fig. A.12).

The request subtypes are:

1. Parameter request providing a full address block target

2. Parameter request providing a parameter ID target
3. Parameter request with two parameter targets
4. Device node request

A request may optionally expect its recipient(s) to respond. All subtypes of request are 'sent from' a parameter, and the response will be addressed to this parameter. Additionally, a request that expects a response must supply a 32-bit 'Sequence ID' to match responses with requests.

As a result of the additional 'Sequence ID' field, there are actually two message formats for every request subtype. All AES-64 message formats are identified by a universal 'Message Type' field, as shown in Table A.5.

Table A.5: Message Type field values

Msg Type value	Message
0x0	Parameter request providing a full address block target and expecting a response
0x1	Parameter request providing a full address block target, not expecting a response
0x2	Parameter request providing a parameter index target and expecting a response
0x3	Parameter request providing a parameter index target, not expecting a response
0x4	Response message
0x9	Device request expecting a response
0xA	Device request, not expecting a response

Request and response messages are composed of a subtype-specific Message Header and a Value field (Fig. A.7). The size and composition of the Value field is specific to the command that the message either expresses or provides a response to.

A.7.1 Message Header formats

This section of the document defines the Message Headers for AES-64 requests and responses.

Parameter request providing a full address block target and expecting a response. This request expects its recipient(s) to answer with a response message. It targets a parameter or set of parameters by providing a 104-bit 'full address block' that

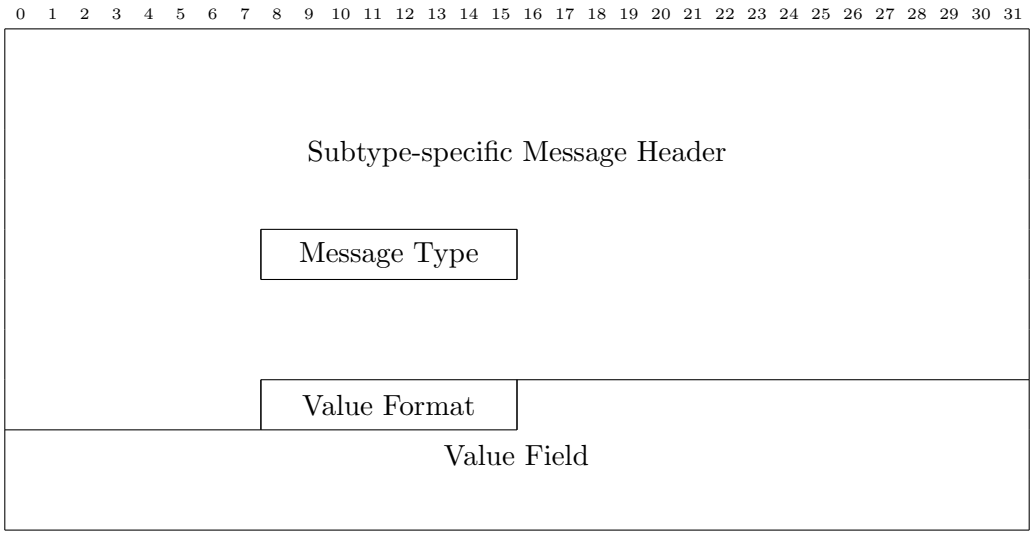


Figure A.7: Generic AES-64 message structure.

specifies an identifier for each position of the level hierarchy (see Subsection A.5.3). The range of usable identifiers are specified in the AES-64 specification [Audio Engineering Society 2012, 50-76] and include 'wildcard' identifiers. Fig. A.8 shows the header structure.

Parameter request providing a full address block and not expecting a response. This request does not expect a response from its recipient(s), and therefore omits the 'Sequence ID' field. It uses the 'full address block' addressing scheme, again allowing the use of 'wildcard' identifiers. Fig. A.9 shows the header structure.

Parameter request providing a parameter index target and expecting a response. This request expects its recipient(s) to answer with a response message. Unlike the 'full address block' address scheme, this request can only address a *single* parameter by its unique ID. The sender will have learned this ID previously through device enumeration or the response to a 'GET ID' request. Fig. A.10 shows the header structure.

Parameter request providing a parameter index target, not expecting a response. This request doesn't expect a response, and therefore omits the 'Sequence ID' field. As before, the sender will have learned this ID previously through device enumeration or the response to a 'GET ID' request. Fig. A.11 shows the header structure.

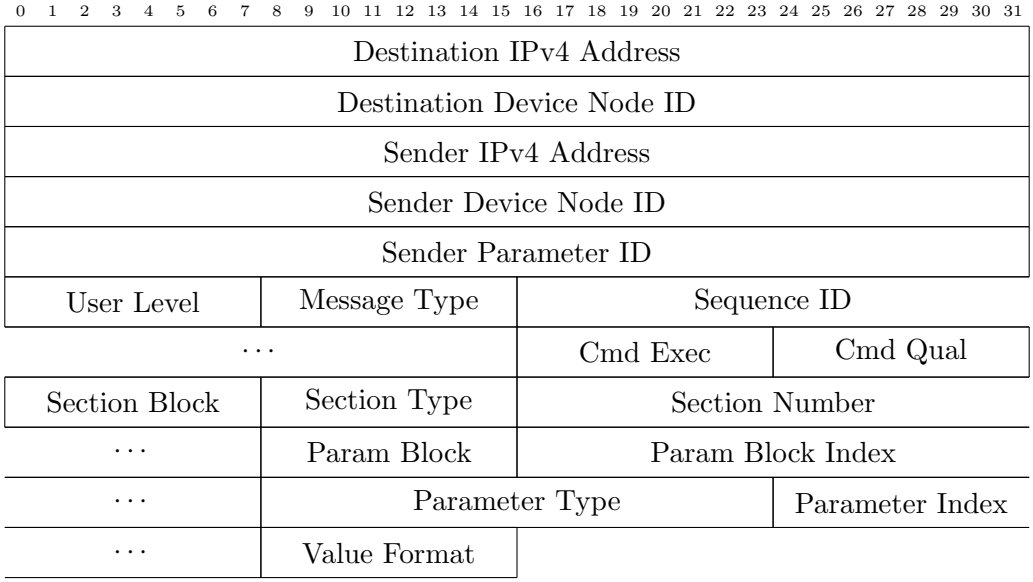


Figure A.8: AES-64 header for a parameter request providing a full address block target and expecting a response.

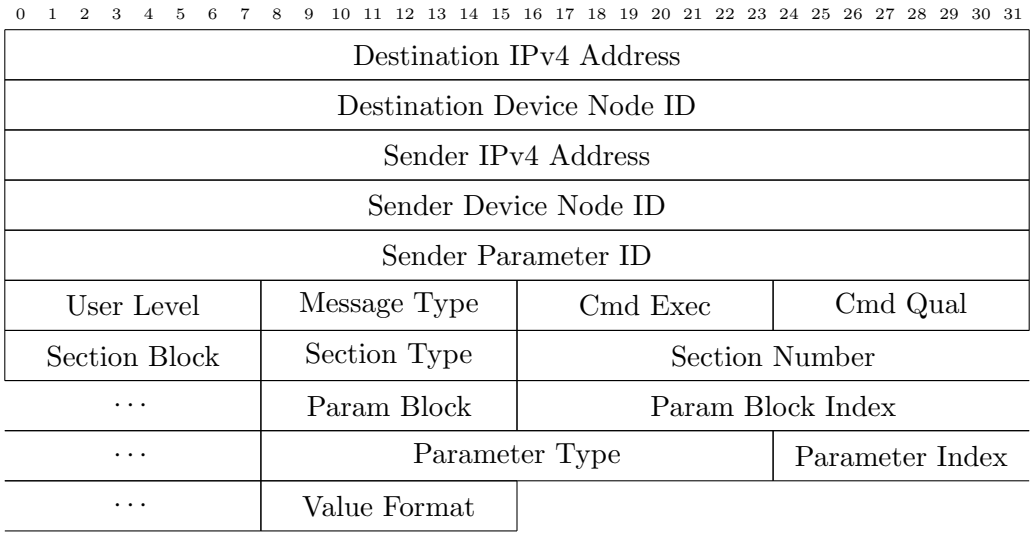


Figure A.9: AES-64 header for a parameter request providing a full address block target and not expecting a response.

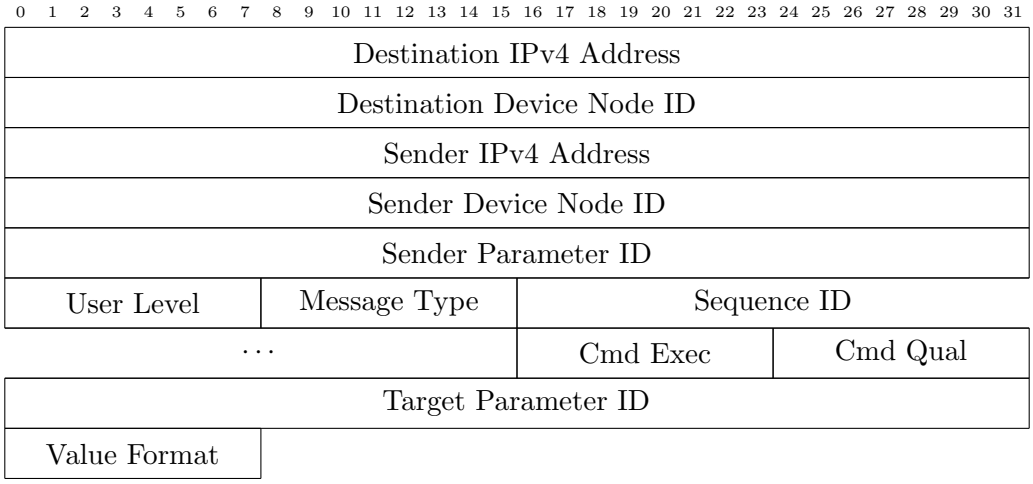


Figure A.10: AES-64 header for a parameter request providing a parameter index target and expecting a response.

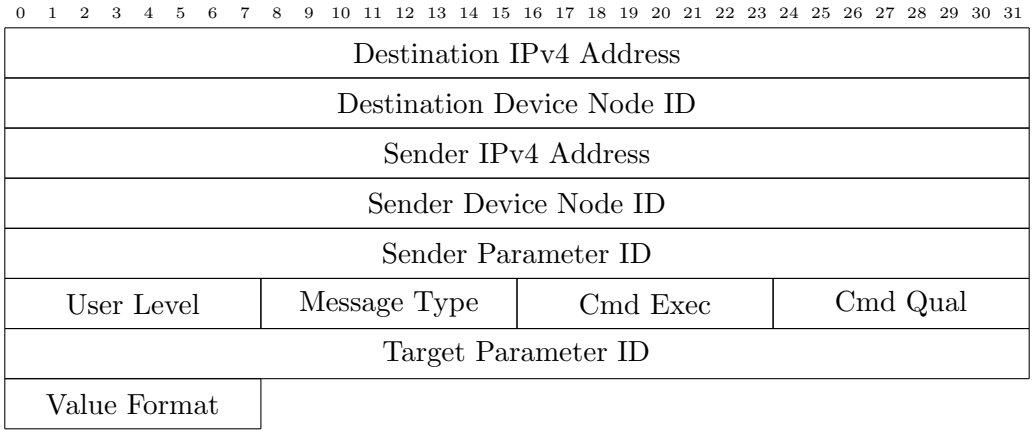


Figure A.11: AES-64 header for a parameter request providing a parameter index target and not expecting a response.

Response message. This is a universal response message. A recipient receiving any type of request expecting a response should produce a message in this form, although the structure of the Value field (not shown) will depend on the command expressed by the original request. The response message copies the Sequence ID provided with the original request. The recipient of the response–which should be the sender of the original request–uses the Sequence ID to return the response data to the AES-64 application. Fig. A.12 shows the header structure.

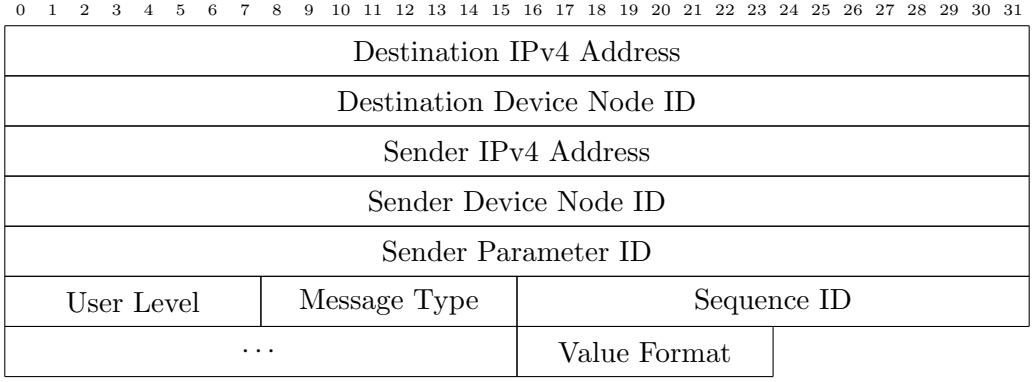


Figure A.12: AES-64 header for a response message (note mandatory 'Sequence ID' field)

Device node request expecting a response. A small number of AES-64 commands address an entire device node. These include the commands used in the device discovery and enumeration process. Fig. A.13 shows the header structure.

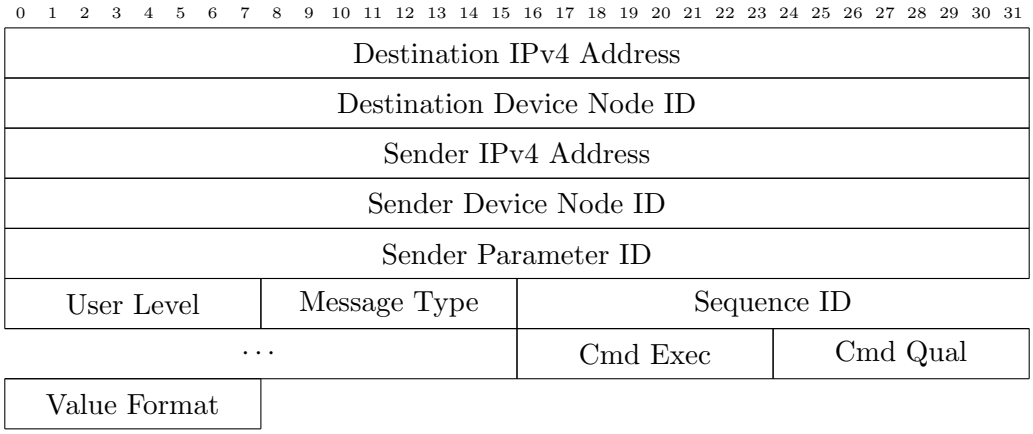


Figure A.13: AES-64 header for a device node request expecting a response.

Device node request not expecting a response. A smaller number of AES-64 commands address an entire device node and *do not* expect a response. Fig. A.14 shows the header structure.

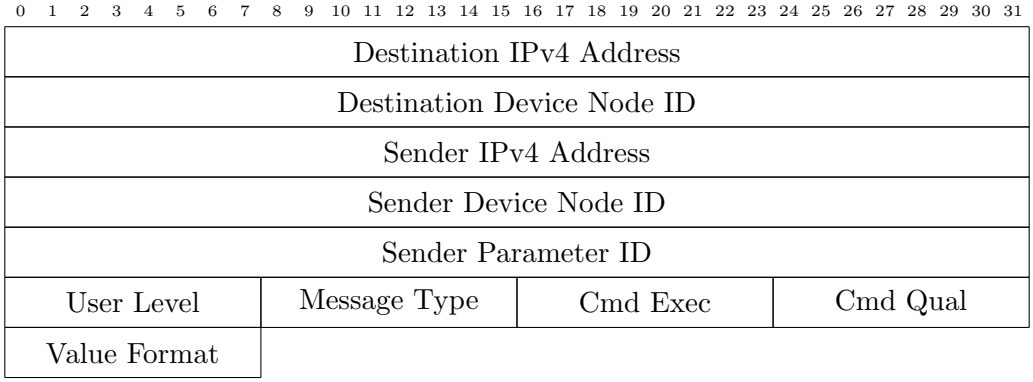


Figure A.14: AES-64 header for a device node request not expecting a response.

A.7.2 AES-64 commands

The command content of an AES-64 request is determined by the Command Executive and Command Qualifier fields, which determine a command in terms of a *verb* (Command Executive, Table A.6) and an *object* (Command Qualifier, Table A.7).

AES-64 requests can be classified by the type of their target. For example, the 'JOIN' and 'UNJOIN' Command Executives will always target a parameter; likewise, the 'VAL', 'FLAGS' or 'VTBL' Command Qualifiers specify attributes on a parameter. This leads to the classifications of 'Parameter Requests' (Table A.8) and 'Device Requests' (Table A.9). Section A.8 defines each of these requests in detail.

Table A.6: Command Executive values

Value	Executive	Meaning
0x00	GET	Retrieve *object*
0x01	SET	Set a value or attribute on *object*
0x02	ACT	Perform an action on *object*
0x05	RESET	Reset allocated storage on *object*
0x06	JOIN	Join target parameter in a grouping relationship of *object* type with (supplied parameter)
0x07	UNJOIN	Remove target param from grouping relationship of *object* type
0x08	CREATE	Create *object*
0x09	SAVE	Save *object*
0x10	IS ALIVE	Establish if destination device is online
0x13	CLEAR	Clear *object* attribute on target

Table A.7: Command Qualifier values

Value	Qualifier	Meaning
0x00	VAL	A *parameter value* attribute.
0x08	FLAG	A *parameter flags register* attribute.
0x0A	NAME	A *parameter name* attribute.
0x0B	VTBL	A *parameter value table* attribute.
0x0C	ID	A *parameter ID* attribute.
0x0D	SNP	A *device node snapshot* as created and stored by the device primary control software.
0x11	PTPGRP	A *parameter peer-to-peer group list* attribute.
0x12	SLVGRP	A *parameter slave group list* attribute.
0x13	MSTGRP	A *parameter master group list* attribute.
0x15	GRPVAL	A *parameter value* attribute, as set via a group update.
0x2A	CLA	The *child level aliases* of a level.
0x2D	MASTER	An entry in a *parameter master group list* attribute.
0x2E	SLAVE	An entry in a *parameter slave group list* attribute.
0x2F	PEER	An entry in a *parameter peer-to-peer group list* attribute.
0x30	MASTER OFF	An entry in a *parameter master group list* attribute.
0x31	SLAVE OFF	An entry in a *parameter slave group list* attribute.
0x32	PEER OFF	An entry in a *parameter peer-to-peer group list* attribute.
0x35	PTP	A *peer-to-peer* group relationship. Used with JOIN and UNJOIN.
0x36	MSTSLV	A *master-slave* group relationship. Used with JOIN and UNJOIN.

Table A.8: Parameter Requests

Command	Subsection
ACT SNP	A.8.1
CLEAR FLAG	A.8.2
GET CLA	A.8.3
GET FLAG	A.8.4
GET ID	A.8.5
GET MSTGRP	A.8.6
GET NAME	A.8.7
GET PTPGRP	A.8.8
GET SLVGRP	A.8.9
GET VAL	A.8.10
GET VTBL	A.8.11
JOIN MSTSLV	A.8.13
JOIN PTP	A.8.14
SAVE SNP	A.8.15
SET FLAG	A.8.16
SET GRPVAL	A.8.17
SET MASTER	A.8.18
SET MASTER OFF	A.8.19
SET MSTGRP	A.8.20
SET PTPGRP	A.8.21
SET SLAVE	A.8.22
SET VAL	A.8.23
UNJOIN MSTSLV	A.8.24
UNJOIN PTP	A.8.25

Table A.9: Device Requests

Command	Subsection
IS ALIVE	A.8.12

A.8 AES-64 commands and responses

Each AES-64 command and its primary response is described in this section. A synopsis of the command or response is followed by a diagram depicting the contents of its associated Value Format and Value fields, as shown in Fig. A.15.

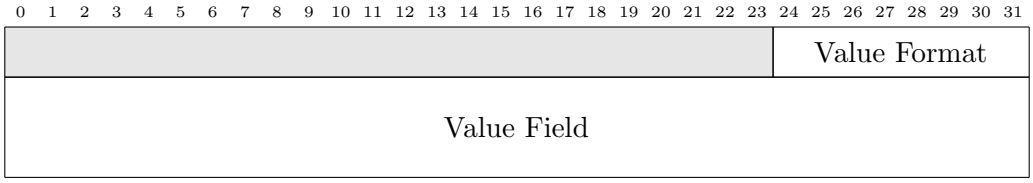
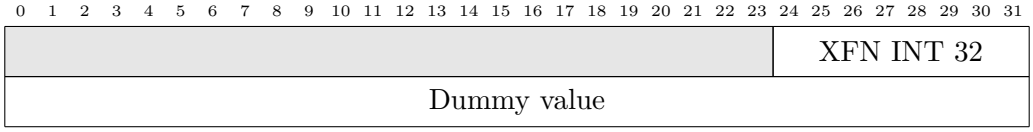


Figure A.15: Example command / response message diagram

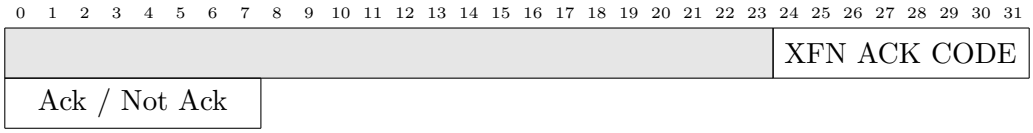
Dummy values

For a number of AES-64 commands, particularly many GET requests, the Value field is strictly unnecessary, and is therefore packed with a 'dummy value':



Acknowledgment response

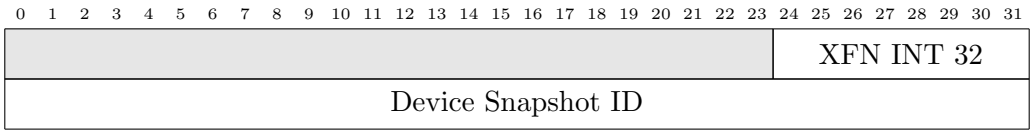
There is a generic response for many AES-64 commands, the 'acknowledgment', which indicates success ('ACK') or failure ('NACK'). The content of the Value field is shown below:



A.8.1 ACT SNP

Request. Instructs the AES-64 protocol stack to load a specified snapshot of parameter values.

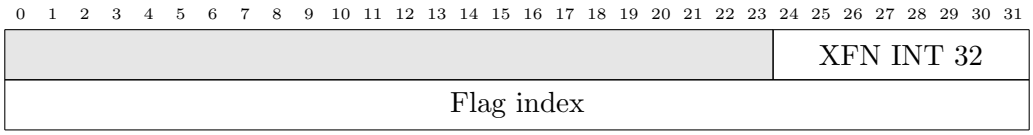
Response. Provides acknowledgment.



A.8.2 CLEAR FLAG

Request. Clears the specified flags on the target parameter's flags register.

This command may have additional side-effects depending on the flag(s) cleared, e.g., resetting the parameters alternate value attribute.



Response. Provides acknowledgment.

A.8.3 GET CLA

Request. This command can only be sent in the 'parameter request providing a full address block target expecting a response' subtype. The command provides a full address block that identifies a level on the target device node, and retrieves a count and a list of pairs of full address blocks and level aliases for every child level belonging to the target level.

The request full address block must include at least one level identifier ('Section Block' — 'Parameter Type') set to the reserved value `0xEE` (Fig. A.16). The preceding level identifiers must identify a unique position on the level hierarchy (i.e., they must not include wildcards). The request then returns a count and list of the child levels at this position in the level hierarchy.

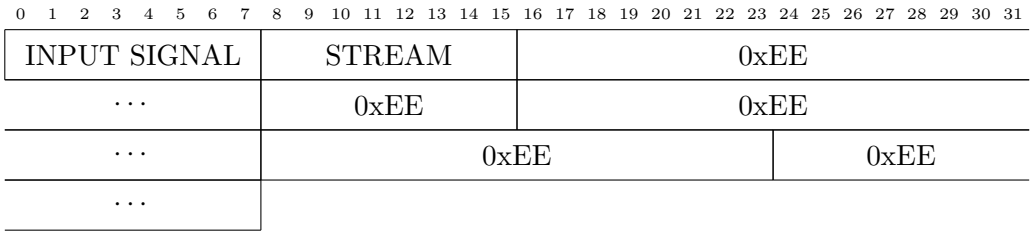
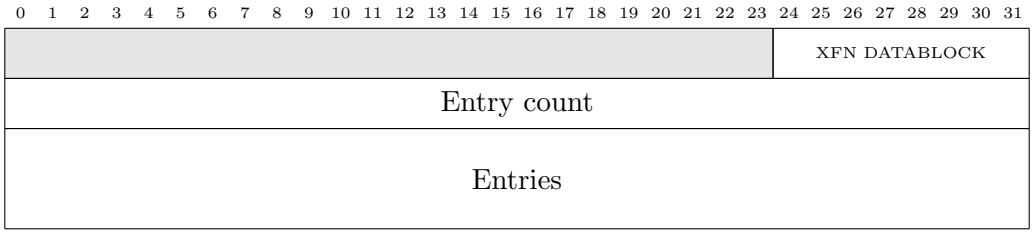


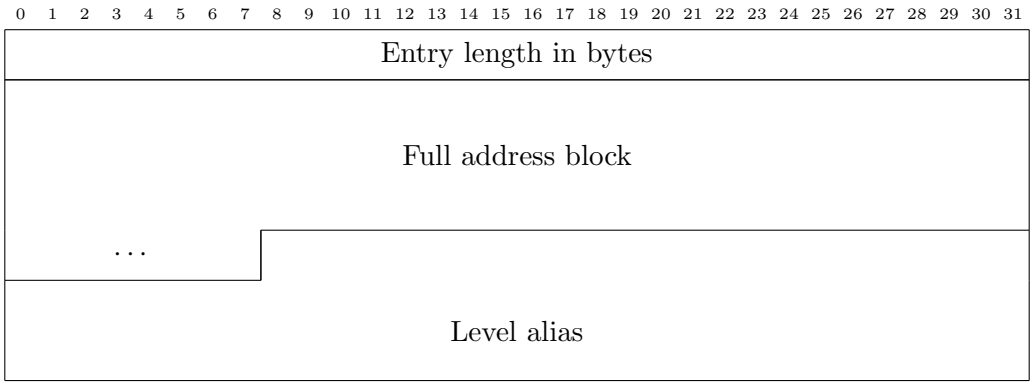
Figure A.16: Example GET CLA full address block

The request value field supplies a dummy value.

Response. An entry count, followed by a number of level entries.

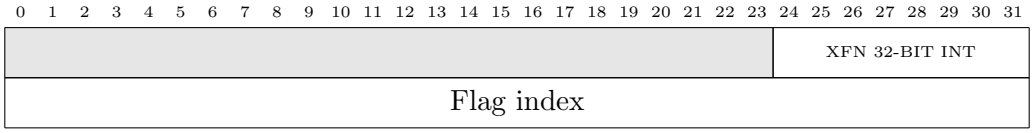


The format of each entry is given below:

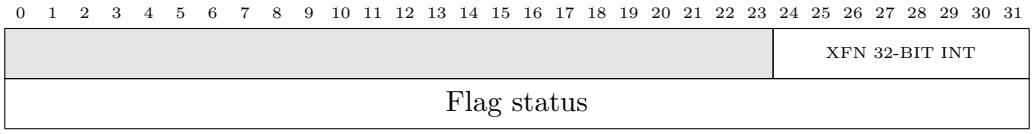


A.8.4 GET FLAG

Request. Gets the status of the specified flag within the target parameter's flag register.



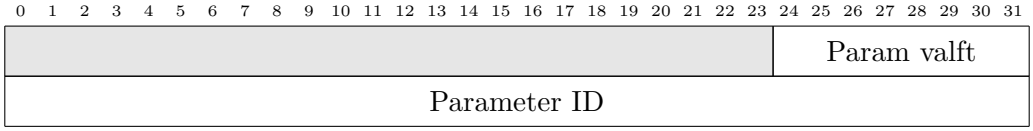
Response. Provides the status of the flag register index.



A.8.5 GET ID

Request. Sent as a parameter request using a full address block target. Retrieves the targeted parameter's unique ID. The value field provides a dummy value.

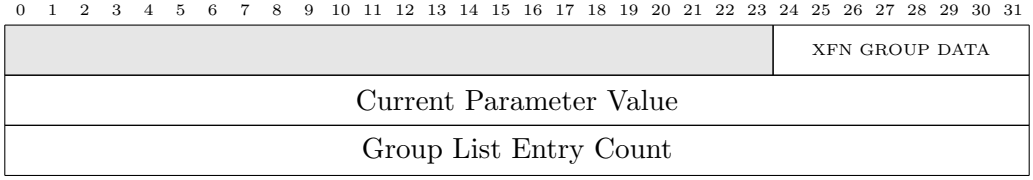
Response. Provides the parameter ID (and the parameter value format).



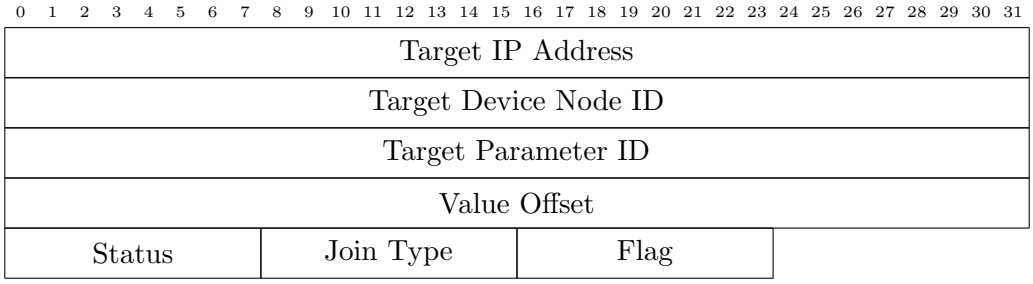
A.8.6 GET MSTGRP

Request. Retrieves the contents of the target parameter's masters group list. The request provides a dummy value field.

Response. A Group Static Header, supplying the current parameter value and an entry count:



Followed by a number of group entries, as counted:



The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this offset will be zero (0).

Status is either 'ONLINE' or 'OFFLINE'.

Join Type is either 'NOT SET' (0), 'ABSOLUTE' (1) or 'RELATIVE' (2).

Flag is either 'NOT SET' (0), 'PERMANENT' (1), 'PASSIVE' (2), or 'PAIR' (3).

A.8.7 GET NAME

Request. This command can only be sent in the 'parameter request providing a full address block target expecting a response' subtype. The command provides a full address block that identifies a level on the target device node, and retrieves the alias of the targeted level. It supplies a dummy value.

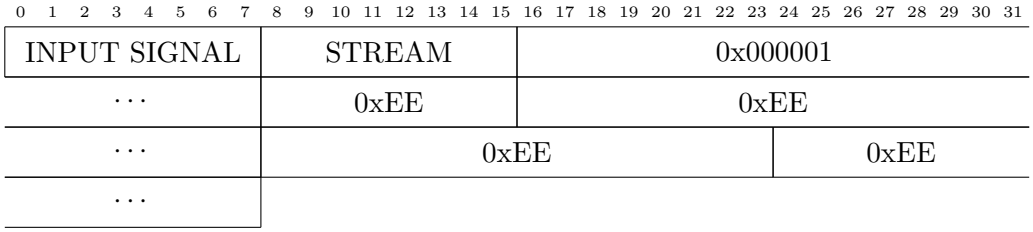
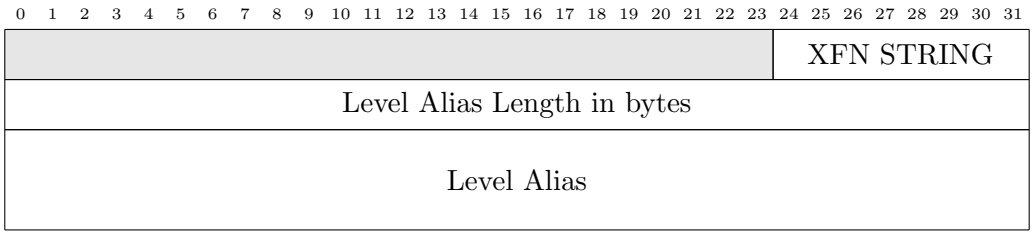


Figure A.17: Example GET NAME full address block

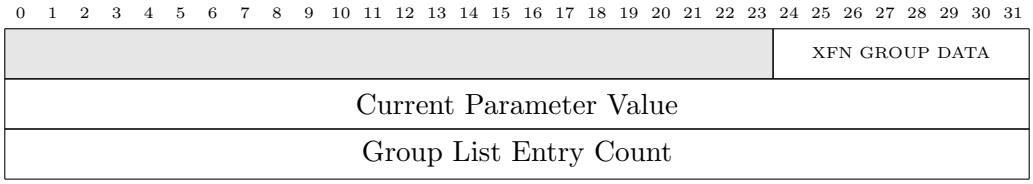
Response. Returns a single Level Alias.



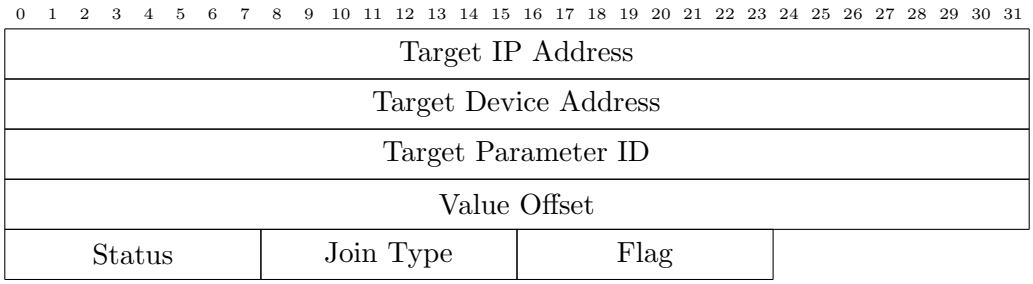
A.8.8 GET PTPGRP

Request. Retrieves the contents of the target parameter's peers group list. The request provides a dummy value field.

Response. A Group Static Header, supplying the current parameter value and an entry count:



Followed by a number of group entries, as counted:



The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this offset will be zero (0).

Status is either 'ONLINE' or 'OFFLINE'.

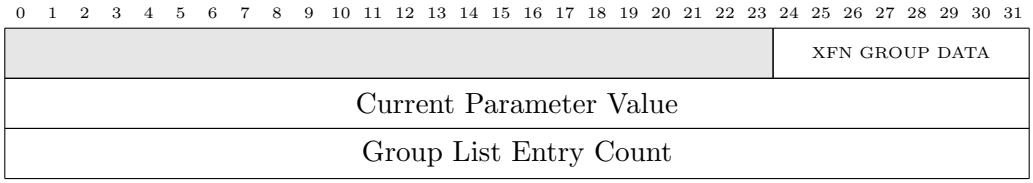
Join Type is either 'NOT SET' (0), 'ABSOLUTE' (1) or 'RELATIVE' (2).

Flag is either 'NOT SET' (0), 'PERMANENT' (1), 'PASSIVE' (2), or 'PAIR' (3).

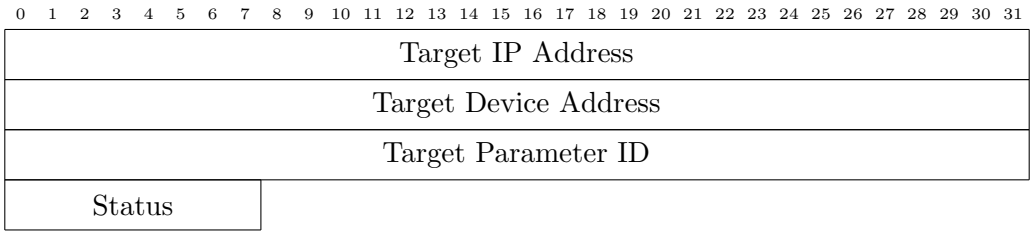
A.8.9 GET SLVGRP

Request. Retrieves the contents of the target parameter's slaves group list. The request provides a dummy value field.

Response. A Group Static Header, supplying the current parameter value and an entry count:



Followed by a number of group entries, as counted:



The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this

offset will be zero (0).

Status is either ‘ONLINE’ or ‘OFFLINE’.

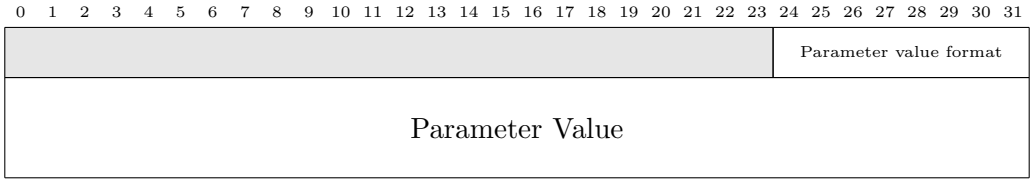
Join Type is either ‘NOT SET’ (0), ‘ABSOLUTE’ (1) or ‘RELATIVE’ (2).

Flag is either ‘NOT SET’ (0), ‘PERMANENT’ (1), ‘PASSIVE’ (2), or ‘PAIR’ (3).

A.8.10 GET VAL

Request. Retrieves the target parameter’s value attribute. The request supplies a dummy value field.

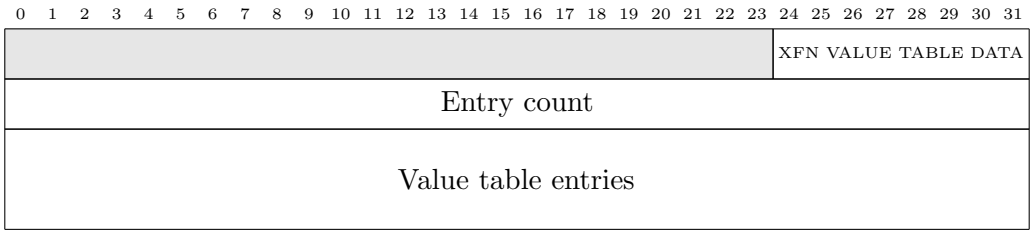
Response. The response provides the parameter value. The value format field holds the parameter’s value format, and dictates the length of the following value (i.e., the size of the parameter value).



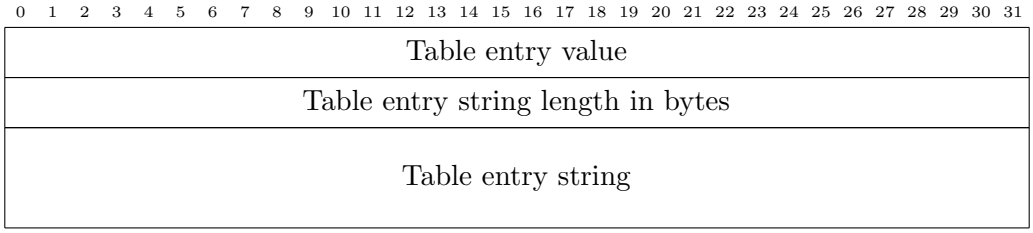
A.8.11 GET VTBL

Request. Retrieves the target parameter’s value table. Provides a dummy value field.

Response. A list of entries representing the range and quantisation of the parameter value. The list header is:



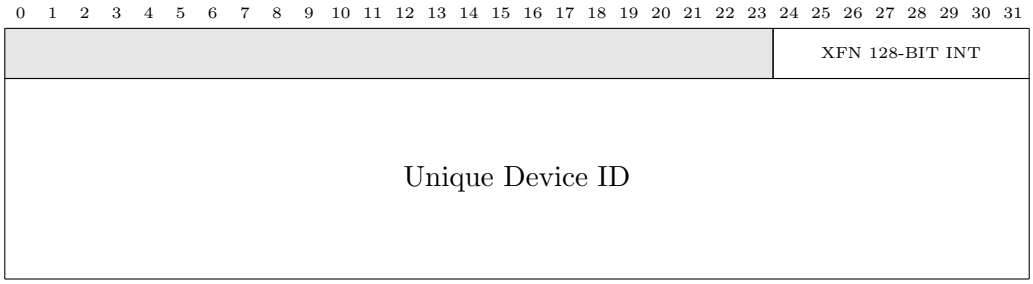
The format of each value table entry is:



A.8.12 IS ALIVE

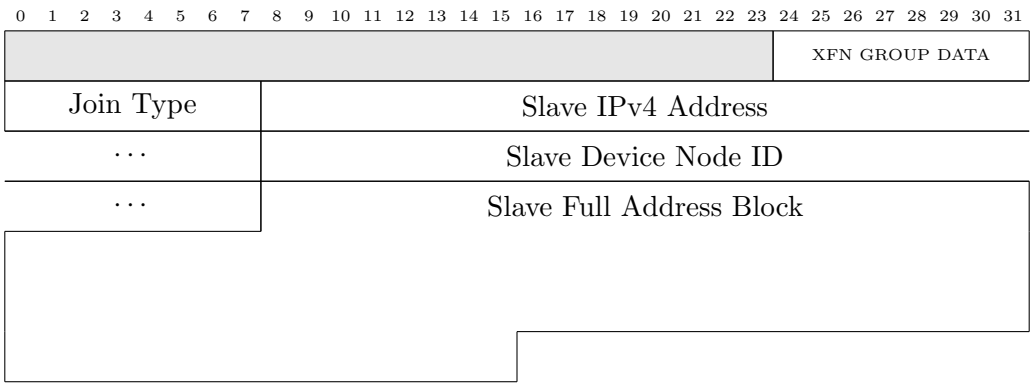
Request. Attempts to discover whether a device is still online, analogous to the 'ping' utility. Supplies a dummy value field.

Response. Supplies a unique device ID.



A.8.13 JOIN MSTSLV

Request. Initiates a master-slave join relationship between the target parameter and a slave parameter, the details of which are provided in the request value field:

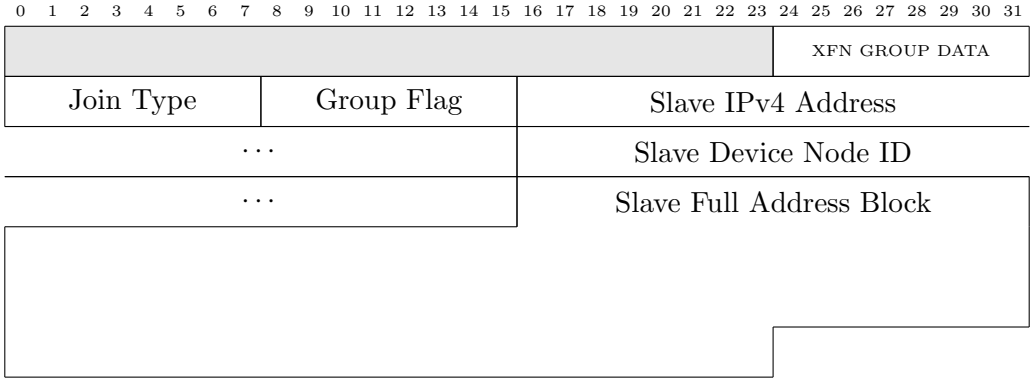


Follow-on requests. The set up process initiated by the 'JOIN MSTSLV' command leads to the *target* parameter issuing 'GET PTPGRP' and 'SET MASTER' commands.

Response. Provides acknowledgment.

A.8.14 JOIN PTP

Request. Initiates a peer-to-peer join relationship between the target parameter and another parameter, the details of which are provided in the request value field:



The Group Flag field indicates whether the intended Join relationship is permanent, passive or paired.

Follow-on requests. The set up process initiated by the 'JOIN MSTSLV' command leads to the *target* parameter issuing 'GET PTPGRP', 'SET PTPGRP', 'GET MSTGRP' and 'SET MASTER' commands.

Response. Provides ACK code.

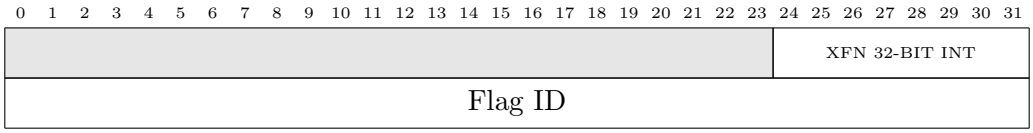
A.8.15 SAVE SNP

Request. Instructs the AES-64 control application to save a snapshot of the device node's parameter values. Provides a dummy value field.

Response. Provides acknowledgment.

A.8.16 SET FLAG

Request. Sets the supplied Flag ID in the target's flag register to 'on'. Flag ID 0 is 'all flags'.

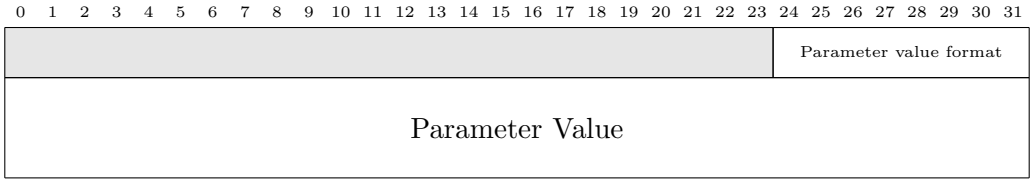


Response. Provides acknowledgment.

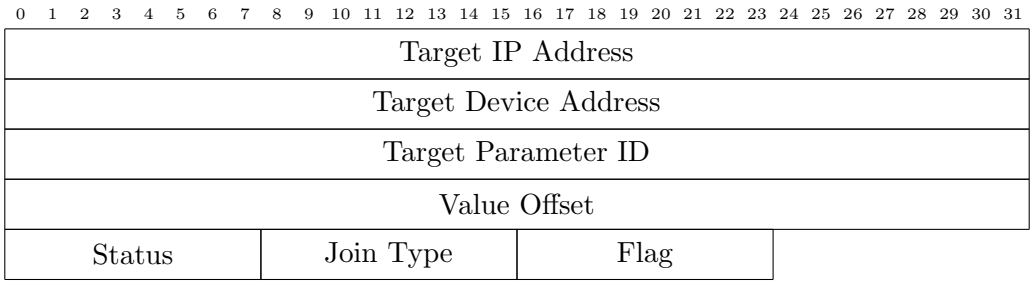
A.8.17 SET GRPVAL

Request. Sets value on the target parameter as a result of the target's group relationship with the source parameter. The sender parameter must be found within the target parameter's Masters or Peers group list.

The value change caused by this command is *not* relayed to any of the target parameter's Peers or Slaves.



Response. None defined.



A.8.18 SET MASTER

Request. The supplied Group List Entry is to be added or merged onto the target's Masters group list.

The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this offset will be zero (0).

Status is either 'ONLINE' or 'OFFLINE'.

Join Type is either 'NOT SET' (0), 'ABSOLUTE' (1) or 'RELATIVE' (2).

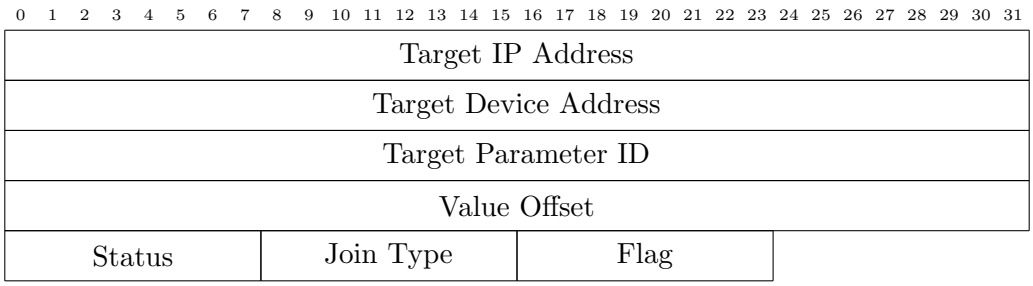
Flag is either 'NOT SET' (0), 'PERMANENT' (1), 'PASSIVE' (2), or 'PAIR' (3).

If the 'Target' attributes of an entry on the target parameter match those of an entry in the supplied list, the entry in the supplied list will overwrite the entry on the target parameter.

Response. Provides acknowledgment.

A.8.19 SET MASTER OFF

Request. The supplied Group List Entry is to be removed from the target's Masters group list.



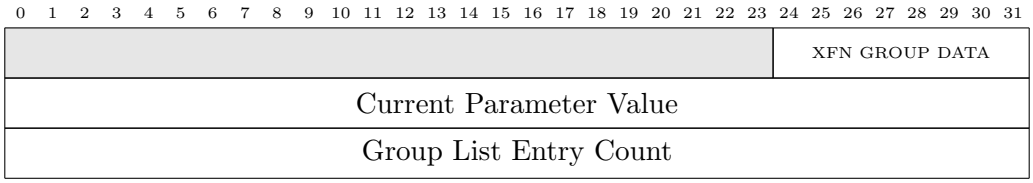
Any entry on the target parameter's Masters list that matches the 'Target IP Address',

'Target Device Address' and 'Target Parameter ID' values will be removed from the list.

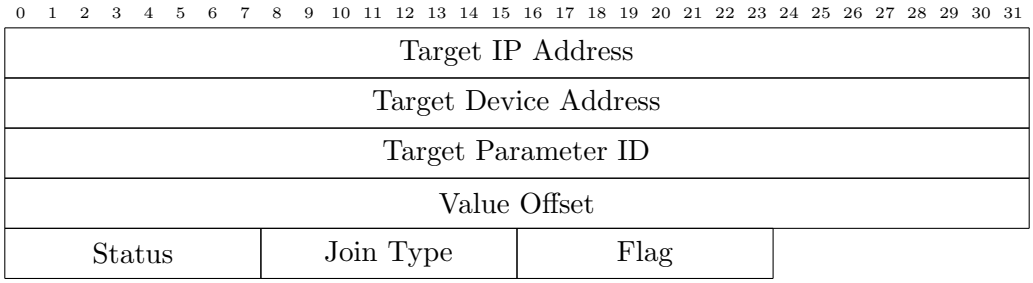
Response. Provides acknowledgment.

A.8.20 SET MSTGRP

Request. The supplied Group List Entries are to replace the target's Masters group list.



Followed by a number of group entries, as counted:



The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this offset will be zero (0).

Status is either 'ONLINE' or 'OFFLINE'.

Join Type is either 'NOT SET' (0), 'ABSOLUTE' (1) or 'RELATIVE' (2).

Flag is either 'NOT SET' (0), 'PERMANENT' (1), 'PASSIVE' (2), or 'PAIR' (3).

Response. Provides acknowledgment.

A.8.21 SET PTPGRP

Request. The supplied Group List Entries are to replace the target's Peers list. On receiving a SET PTPGRP request, the target parameter has to determine the effective join type (ABSOLUTE or RELATIVE) between itself and the sender parameter. This is done by two checks:

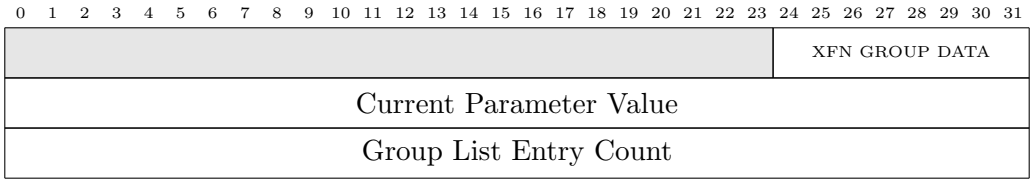
1. The target parameter checks its PTP Group List for the sender parameter. If it finds an existing Group List Entry for the sender parameter, it copies the join type stated in that Entry.

2. The target parameter checks the Group List Entries supplied by the sender parameter *for itself*. If it finds a matching Group List Entry, it copies the join type stated in that entry.

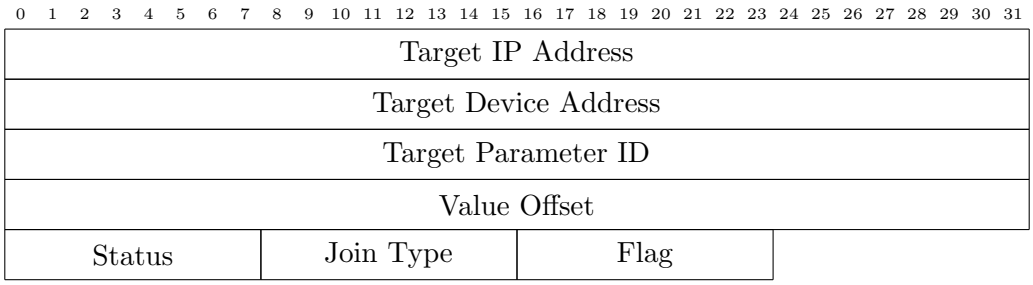
If neither of these checks results in a known join type, the default is relative. Note that this is not expected to happen, as the context for SET PTPGRP is a communication between two parameters in an existing join relationship.

The Value field of the request is formed of a Group Static Header followed by one or more Group List Entries.

A Group Static Header is followed by one or more Group List Entries:



Followed by a number of group entries, as counted:



The Value Offset is a signed value giving the offset between the parameter's current value and the value of the Master parameter. If the Join Type is 'ABSOLUTE' this offset will be zero (0).

Status is either 'ONLINE' or 'OFFLINE'.

Join Type is either 'NOT SET' (0), 'ABSOLUTE' (1) or 'RELATIVE' (2).

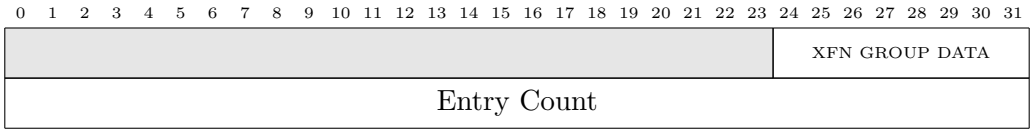
Flag is either 'NOT SET' (0), 'PERMANENT' (1), 'PASSIVE' (2), or 'PAIR' (3).

Response. Provides acknowledgment code.

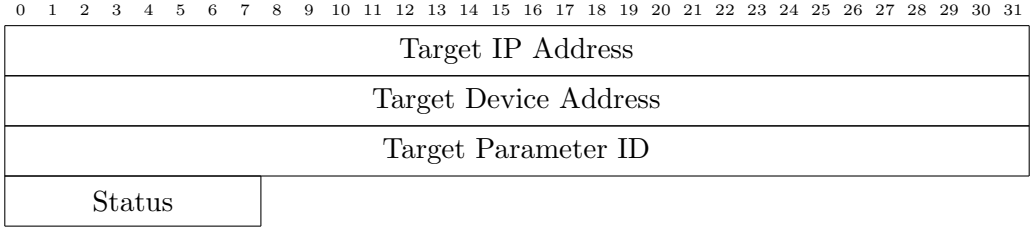
A.8.22 SET SLAVE

Request. The supplied Group List Entries are to be added to the target's Slaves list.

The request is formed of a Slave Group Static Header followed by one or more Slave Group List Entries.



A Slave Group List Entry contains everything we need to know about a single Slave:

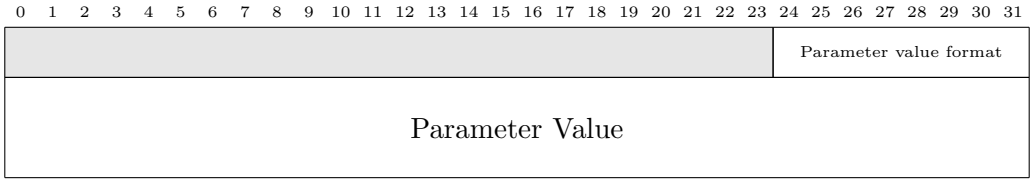


Response. Provides acknowledgment.

A.8.23 SET VAL

Request. Set the supplied value on the target parameter. This can be a Global Units value or a value of some other type.

This value change must be relayed to any members of the target parameter's Peers or Slaves group lists.



Response. The valid responses for this command are:

1. A broadcast, if the value is set through a client application callback that wants to send a broadcast response;

2. No response at all, if the value is set through a client application callback that does not want to send a response;

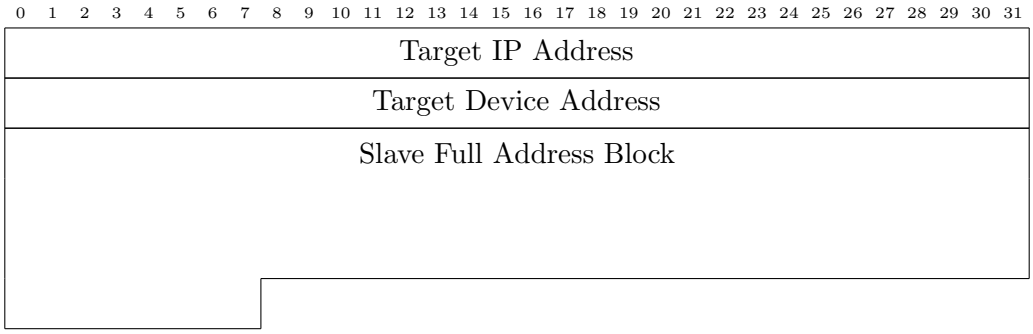
3. A response containing the original request's valft and data, if the client application callback wants to acknowledge by echoing the original request data;

4. An ACK, if there was no error;

5. A NACK, if none of the preceding cases are true.

A.8.24 UNJOIN MSTSLV

Request. Terminates a Join MSTSLV relationship by supplying details of a slave that should be removed from the target master:



Response. Provides acknowledgment.

A.8.25 UNJOIN PTP

Request. Terminates every peer-to-peer join involving the target parameter, except for relationships marked as 'permanent'.

When this command is received, the addressed parameter sorts its current PTP list into two sets:

1. Peers with a 'permanent' relationship

2. Peers without a 'permanent' relationship

Having done so, the addressed parameter updates set 1 and set 2 by sending two SET PEER OFF requests:

1. To the permanent peers, instructed to remove the non permanent peers.

2. To the non-permanent peers, instructed to remove the permanent peers.

Supplies a dummy value field.

Response. Provides an acknowledgment.

A.9 Messaging interactions

As indicated in Section A.7, most interactions within AES-64 are handled within a single request-and-response cycle. For some of AES-64's advanced features this is not the case, and this section describes those interactions in detail.

The features of AES-64 which involve extended interactions correspond roughly to those described in Section 3.6, and these are:

1. Network discovery
2. Device enumeration
3. Push notification
4. Parameter grouping
    a) Setup
    b) Maintenance
    c) Value changes
    d) Teardown

A.9.1 Network discovery

This process allows an AES-64 software controller to poll the network for AES-64-enabled devices to interact with.

The simplest form of network discovery can be performed by sending a broadcast 'GET VAL' request specifying device node ID '0' and the following full address block:

```
DEVICE INFO CONFIG  IP   1   CONFIG  1   IP ADDRESS  1
```

Following this request, the AES-64 controller may wait for a brief interval and catalogue the received responses.

More sophisticated forms of network discovery could be implemented - for example, an AES-64 device's response could usefully specify the size of its device representation (how many device nodes exist, and how many parameters are present on each), or include the contents of the 'Device Name' parameter. This behaviour has not been described in the published AES-64 standard.

A.9.2 Device enumeration

This process allows an AES-64 software controller to perform bulk retrieval of the parameters on a discovered device (i.e., a device that has identified itself to the controller via the network discovery process).

The simplest form of this process can be carried out using the 'GET CLA' request. For example, an AES-64 controller device can discover the following details:

- the number of stream interfaces
- the number of stream multicore inputs
- the number of stream multicore outputs
- the number of audio hardware inputs
- the number of audio hardware outputs

by sending the following full address blocks to device node '1' within 'GET CLA' requests:

```
SB INPUT SIGNAL  ST STREAM  0xEEEEEE    0xEE          0xEEEEEE   0xEEEE   0xEEEE
```

and for each stream interface counted by the response to that request:

```
SB INPUT SIGNAL  ST STREAM [i/f index] PB MULTICORE 0xEEEEEE   0xEEEE   0xEEEE
SB OUTPUT SIGNAL ST STREAM [i/f index] PB MULTICORE 0xEEEEEE   0xEEEE   0xEEEE
```

followed by:

```
SB INPUT SIGNAL  ST AUDIO   0xEEEEEE    0xEE          0xEEEEEE   0xEEEE   0xEEEE
SB OUTPUT SIGNAL ST AUDIO   0xEEEEEE    0xEE          0xEEEEEE   0xEEEE   0xEEEE
```

A.9.3 Parameter grouping - overview

Terminology. A *parameter group* is a set of parameters that have established a group relationship. A *grouped parameter* is a parameter that is part of a parameter group. An *external value change* is a message that changes the value of a grouped parameter from outside the group. A *group value change* is a message that relays an external value change to the other parameters in that group.

Parameter grouping. This mechanism allows the linking of AES-64 parameters within devices, between devices, or between a controller and a device.

The purpose of parameter grouping is to enable simultaneous changes to parameter values. A value update mechanism is provided that can selectively relay changes on parameter values to any or all other parameters in the group. See Subsection A.5.2 for a description of how parameters store grouping data.

AES-64's implementation of this mechanism is complex and subtle. Since an established parameter group does not merely exist in its own right but may also interact with other groups, and since a single parameter can simultaneously be a member of multiple parameter groups, understanding the implementation requires close attention to three interrelated concepts:

1. Value relationships
2. Dominance
3. Permanence

Value relationships. This describes the interaction between the values held by grouped parameters. There are two kinds of value relationship:

1. *absolute value relationship*: all grouped parameters maintain identical values.
2. *relative value relationship*: at group setup, each grouped parameter learns the relative offsets between *its own current value* and the current values of *all other grouped parameters*. For the lifespan of the group relationship, each grouped parameter adjusts its own current value to maintain these original offsets.

Value relationships can be complicated by the interaction of several parameter groups through a common parameter. These advanced issues are discussed by N. Chigwamba, et al [2012].

Dominance. This describes the basis on which parameters may exchange *group value updates*. AES-64 provides two possibilities:

1. *master-slave groups*: one or more grouped parameters are nominated masters, while the remainder are slaves. On receiving an external value change, a master parameter sends a group value update to the slaves. If a slave receives an external value change, it does not send a group value update, but remains silently 'out of step' with its master(s) until the master sends the next group value update.

2. *peer-to-peer groups*: any grouped parameter relays a group value update to the entire parameter group if it receives an external value change.

Permanence. This describes the requirement that some parameter groups need to persist until they are explicitly deactivated. The 'UNJOIN PTP' command summarily removes the targeted parameter from all of its peer-to-peer groups: permanence describes the need for some groups to be exempted from removal. This requirement is recorded in the 'Group Flags' field of a Parameter Group Entry (see Subsection A.5.2).

A.9.4 Parameter grouping - setup

A parameter group is established by one of two parameter requests:

1. 'JOIN MSTSLV', creating a *master-slave* group (see Subsection A.8.13).

2. 'JOIN PTP', creating a *peer-to-peer* group (see Subsection A.8.14).

In each case, it should be noted that the sender of the request is *not necessarily* forming a parameter group with the target parameter. Each JOIN message *supplies* a parameter address in its value field: this might be the sender parameter, but it can also be any other valid parameter.

Setting up a parameter group requires the targeted parameter to issue some queries to discover the context in which it's joining to the supplied parameter. This process is common to setting up master-slave and peer-to-peer parameter groups, but the messaging sequences are different.

Setting up a master-slave parameter group. The 'JOIN MSTSLV' parameter request sets up a master-slave parameter group. The targeted parameter is the master, the supplied parameter is the slave, and the peers (if any) of the supplied parameter also become slaves. The messaging sequence for this command is shown in Figure A.18.

1. The controller sends a 'JOIN MSTSLV' request to the targeted parameter ('Parameter 1' in the figure). The value field of this request supplies the address of a second parameter ('Parameter 2' in the figure).

2. 'Parameter 1' issues a 'GET PTPGRP' request to 'Parameter 2'.

3. 'Parameter 2' responds, supplying its peers list. (The peers of a slave are also slaves.) 'Parameter 1' merges the received peers list into its *slaves* list.

4. 'Parameter 1' issues a 'SET MASTER' request to each of its slaves, including the new entries. 'Parameter 2' and all of its peers register 'Parameter 1' as a master.

5. 'Parameter 1' issues a 'JOIN MSTSLV' response to the controller, if one has been requested.

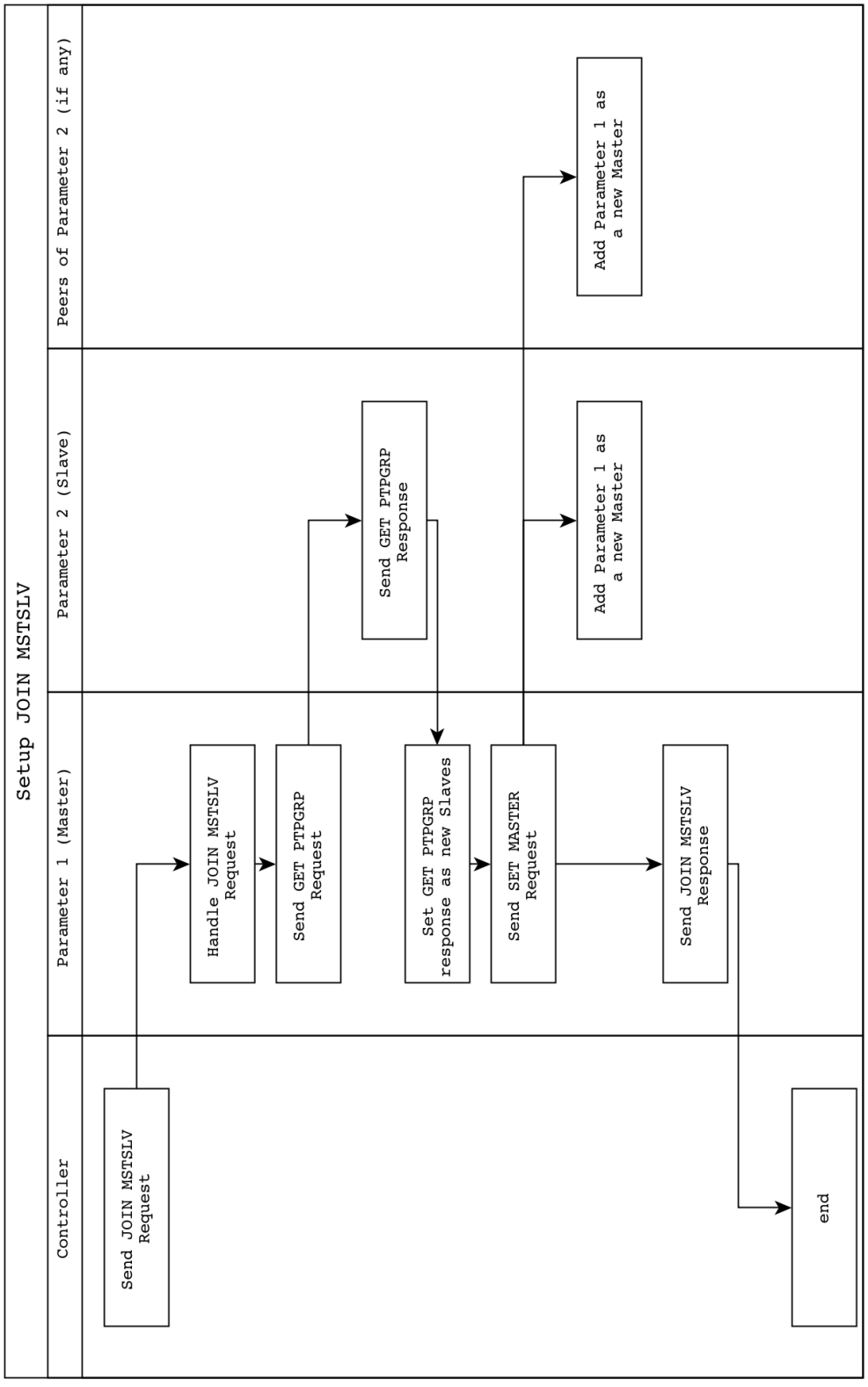


Figure A.18: Setup of a master-slave parameter group by JOIN MSTSLV

Setting up a peer-to-peer parameter group. The 'JOIN PTP' parameter request sets up a peer-to-peer parameter group. The targeted parameter is a peer of the supplied parameter. In a peer-to-peer parameter group, the grouped parameters must synchronise their peer group lists and master group lists; as a result, the targeted parameter has to query its new peer and re-issue a merged peers list and a merged masters list.

The messaging sequence for this command is shown in Figures A.19 and A.20.

1. The controller sends a 'JOIN PTP' request to the targeted parameter ('Parameter 1' in the figure). The value field of this request supplies the address of a second parameter ('Parameter 2').

2. 'Parameter 1' issues a 'GET PTPGRP' request to 'Parameter 2'.

3. 'Parameter 2' responds, supplying its peers list. 'Parameter 1' merges the list received from 'Parameter 2' into its own peers list.

4. 'Parameter 1' issues a 'SET PTPGRP' request to its new peers list, supplying the updated peers list. 'Parameter 2' and its peers overwrite their peers list with the list issued by 'Parameter 1'.

5. 'Parameter 1' issues a 'GET MSTGRP' request to 'Parameter 2'

6. 'Parameter 2' responds, supplying its masters list. 'Parameter 1' merges the list received from 'Parameter 2' into its own masters list.

7. 'Parameter 1' issues a 'SET MASTER' request to its peers list, supplying its merged masters list. 'Parameter 2' and its peers overwrite their masters list with the list issued by 'Parameter 1'.

8. 'Parameter 1' issues a 'SET SLAVE' request to its masters list, supplying its merged peers list. 'Parameter 1"s masters merge the list received from 'Parameter 1' into their slaves list.

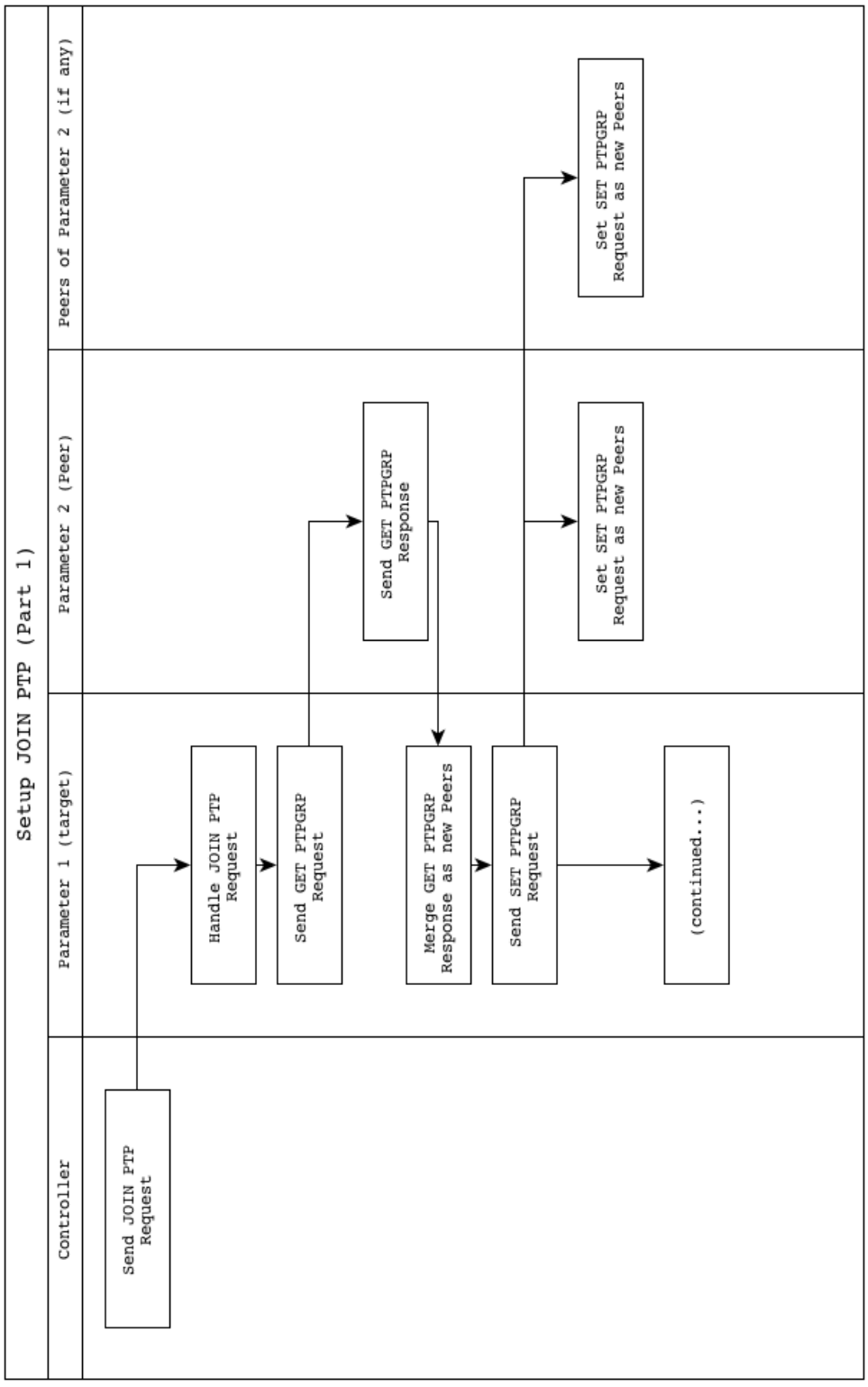


Figure A.19: Setup of a peer-to-peer parameter group by JOIN PTP (1)

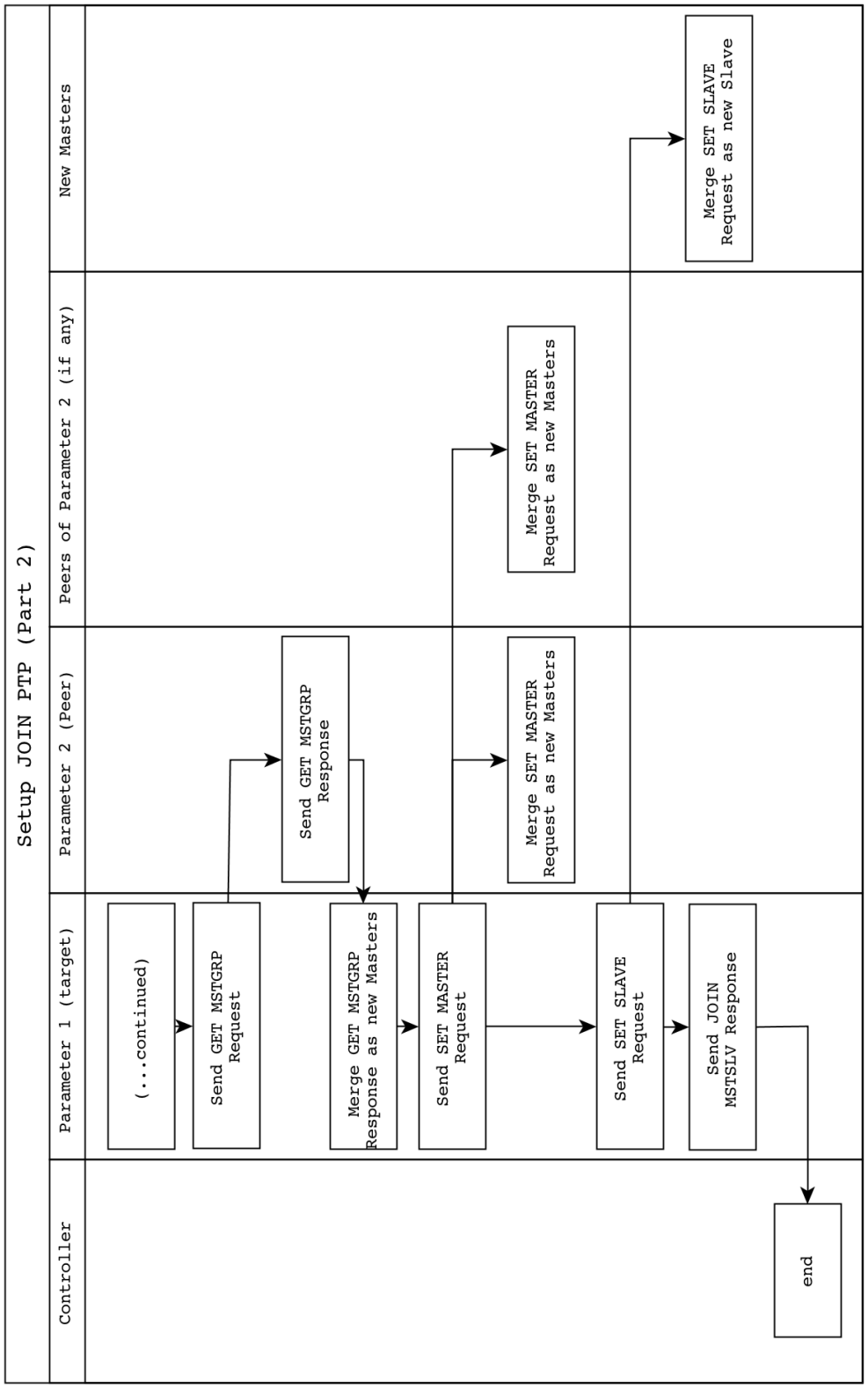


Figure A.20: Setup of a peer-to-peer parameter group by JOIN PTP (2)

A.9.5 Parameter grouping - maintenance

As the previous section suggests, AES-64 defines a number of requests which do maintenance on the parameter group lists.

The group list maintenance commands can be categorised into four types:

1. Queries that retrieve the targeted list ('GET PTPGRP', 'GET MSTGRP').
2. Commands that merge the supplied list of group entries into the targeted group list ('SET MASTER', 'SET PEER', 'SET SLAVE').
3. Commands that remove the supplied list of group entries from the targeted group list ('SET MASTER OFF', 'SET PEER OFF').
4. Commands that *replace* the targeted group list with the supplied list of group entries ('SET MSTGRP', 'SET PTPGRP').

For further details, refer to the command reference (Table A.8).

A.9.6 Parameter grouping - value changes

An external value change to a grouped parameter can come from the device control application or from a 'SET VAL' request. The external value change is then relayed to the grouped parameter's peers and/or slaves using the 'SET GRPVAL' parameter request.

Depending on the *value relationship* of a parameter group, one of two things can happen:

1. in an *absolute* value relationship, the grouped parameter's peer or slave will adjust its value to the grouped parameter's new value (Fig. A.21).
2. in a *relative* value relationship, a grouped parameter's peer or slave will adjust its value to maintain the original offset between its value and the grouped parameter's value (Fig. A.22).

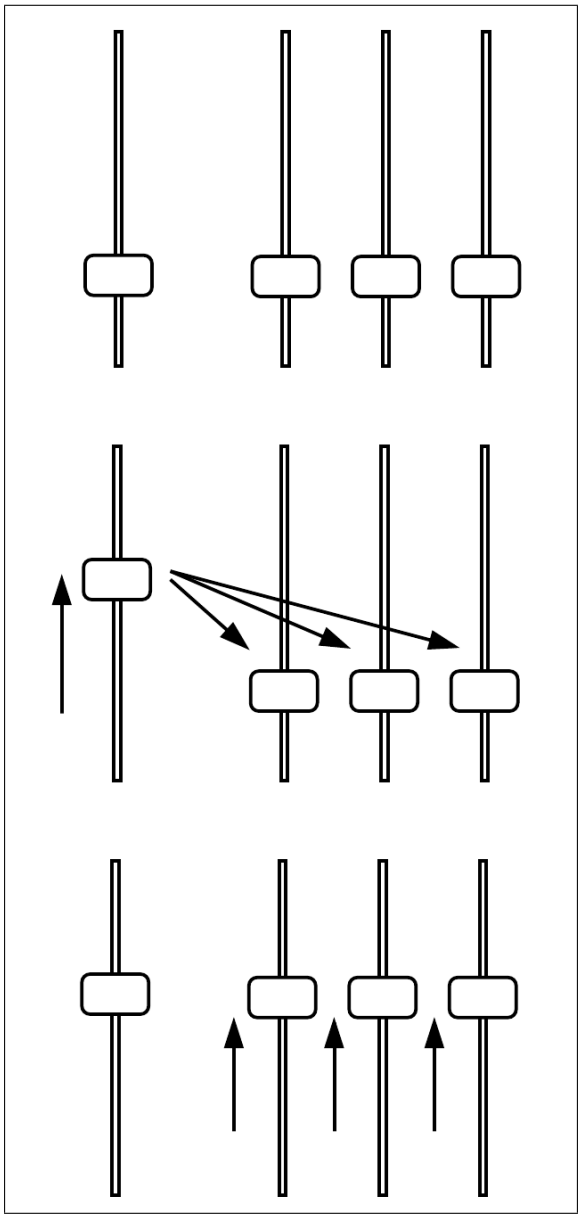


Figure A.21: Value changes in a parameter group that enforces an 'absolute' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group.

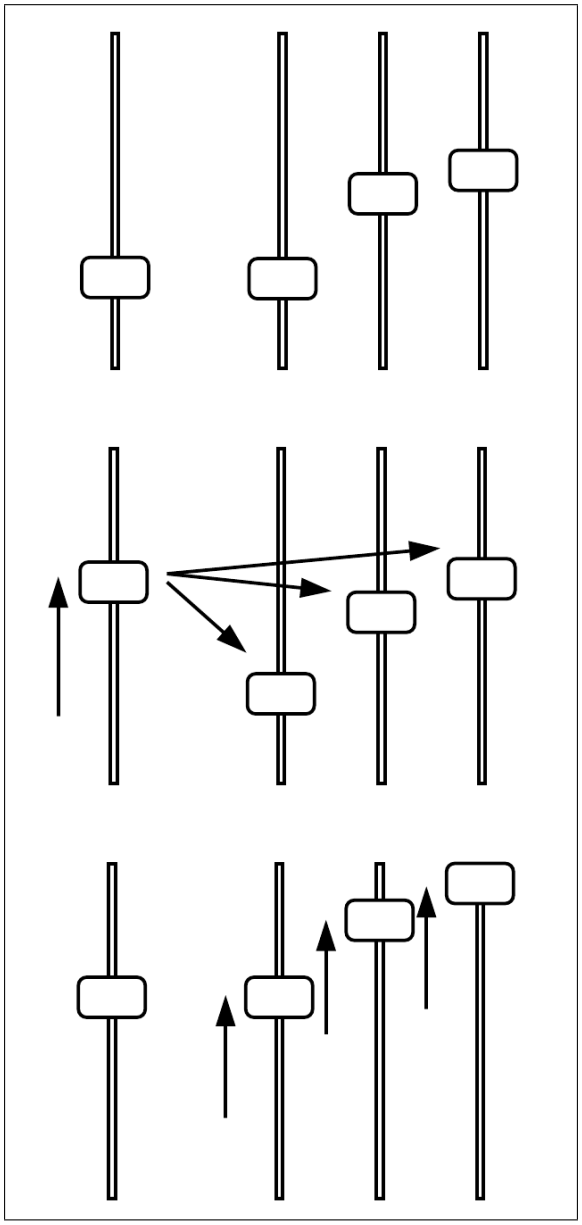


Figure A.22: Value changes in a parameter group that enforces a 'relative' value relationship. Frame 1 shows the resting state of the group; frame 2 shows the first parameter receiving a value change and the effective value changes on the other members of the group; frame 3 shows the new resting state of the group.

A.9.7 Parameter grouping - teardown

Tearing down a master-slave parameter group. The 'UNJOIN MSTSLV' command tears down a master-slave parameter group. The targeted parameter is the master, and the supplied parameter is one of the master's slaves. The targeted parameter removes itself as a master from the supplied parameter and from the supplied parameter's peers.

The messaging sequence for this command is shown in Figure A.23.

1. The controller sends an 'UNJOIN MSTSLV' request to the targeted parameter ('Parameter 1' in the figure). The value field of the request supplies the address of a second parameter assumed to be a slave of the targeted parameter ('Parameter 2' in the figure).
2. 'Parameter 1' issues a 'GET PTPGRP' request to 'Parameter 2'.
3. 'Parameter 2' responds, supplying its peers list.
4. 'Parameter 1' issues a 'SET MASTER OFF' request to 'Parameter 2' and each of its peers. 'Parameter 1' removes 'Parameter 2' and its peers from its slaves list.
5. 'Parameter 1' issues an 'UNJOIN MSTSLV' response to the controller, if one was requested.

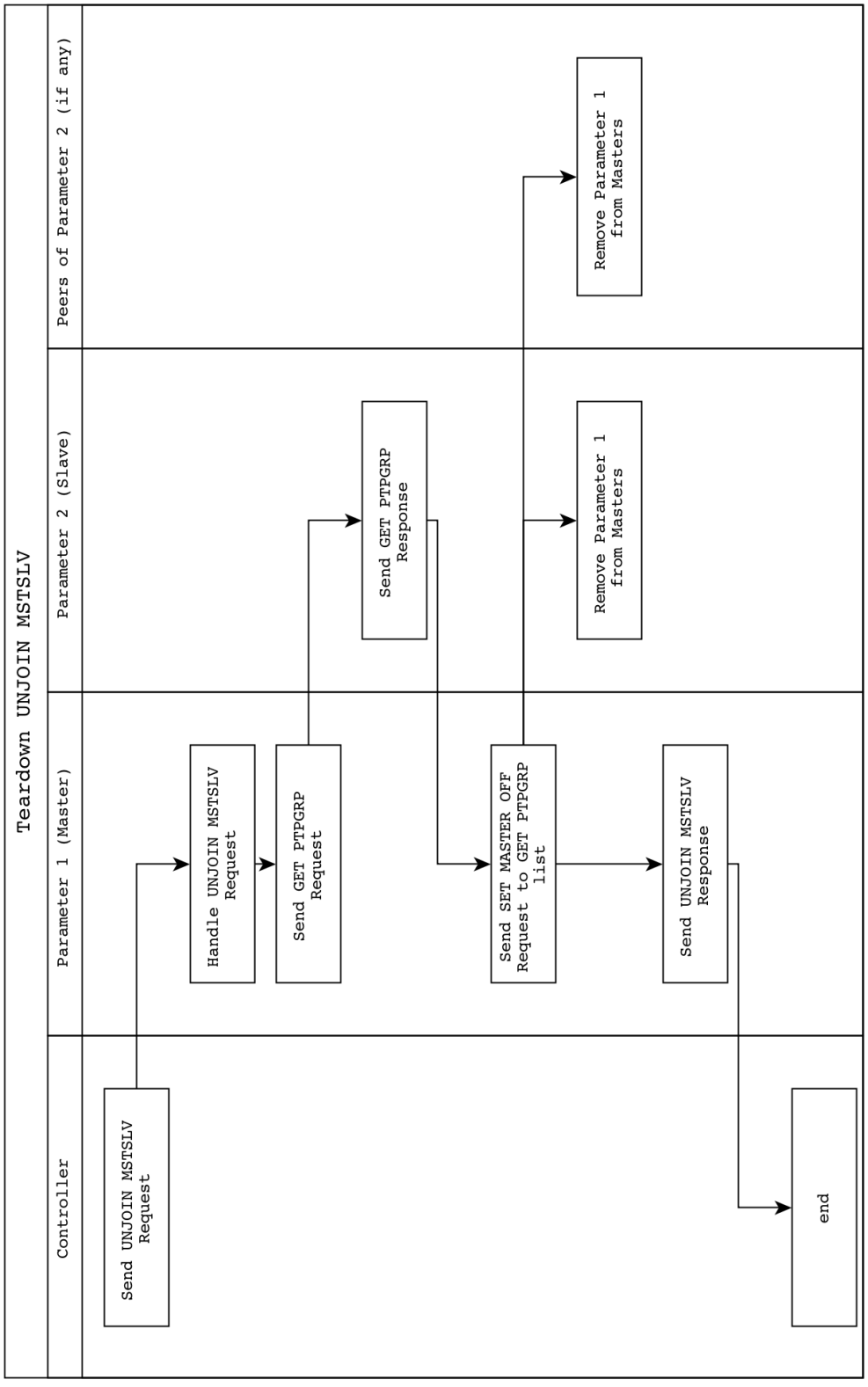


Figure A.23: Teardown of a master-slave parameter group by UNJOIN MSTSLV

Tearing down a peer-to-peer parameter group. The 'UNJOIN PTP' command tears down a peer-to-peer parameter group. The targeted parameter will remove itself from *any and all* non-permanent relationships with its peers.[1] The targeted parameter will preserve any relationships flagged as 'permanent'.

It is important to note that grouping consistency must be maintained after the teardown. Before teardown, peers in a permanent relationship with the targeted parameter know about peers in a non-permanent relationship with the targeted parameter; likewise, peers in a non-permanent relationship with the targeted parameter know about peers in a permanent relationship with the targeted parameter. After teardown is complete, neither group may retain any knowledge of the other group.

The messaging sequence is shown in Figure A.24.

1. The controller sends an 'UNJOIN PTP' request to the targeted parameter ('Parameter 1' in the figure). The value field of the request does not supply a second parameter, as the command terminates *all* non-permanent peer relationships. 'Parameter 1' sorts its peers list into two groups:
   a) Peers flagged as 'permanent'
   b) Peers flagged as 'non-permanent'
2. 'Parameter 1' issues a 'SET PEER OFF' request to all its permanent peers, supplying the list of its non-permanent peers. The permanent peers remove the non-permanent peers from their peer lists.
3. 'Parameter 1' issues a 'SET PEER OFF' request to all its non-permanent peers, supplying the list of its permanent peers. The non-permanent peers remove the permanent peers from their peer lists.
4. 'Parameter 1' issues a 'UNJOIN PTP' response to the controller, if one was requested.

[1] To selectively remove peers from a peer-to-peer parameter group would break the consistency rule of peer-to-peer parameter groups.

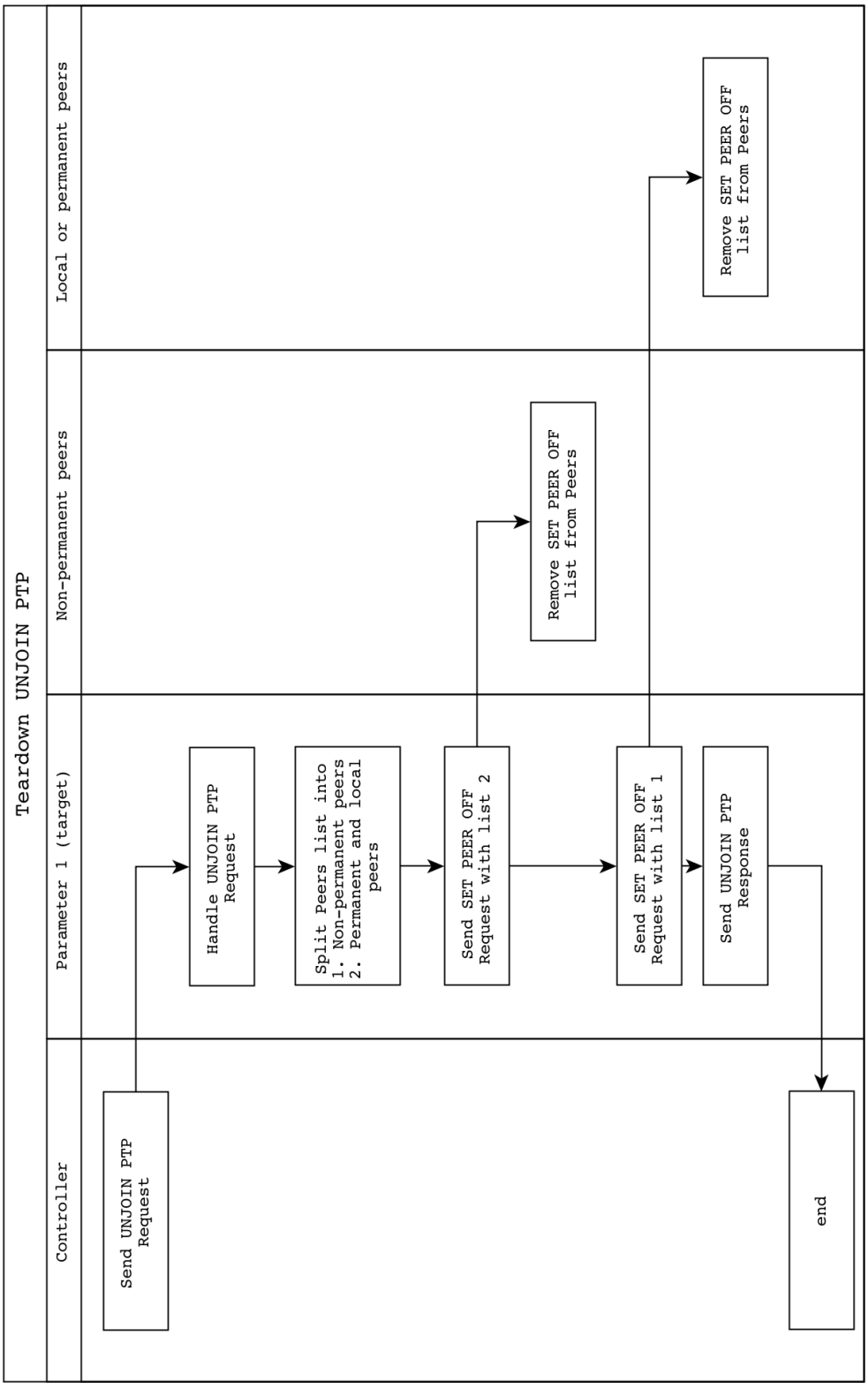


Figure A.24: Teardown of a master-slave parameter group by UNJOIN PTP

B AES-64 protocol stack essential model

B.1 Introduction

This appendix provides documentation of the AES-64 protocol stack's essential model, as developed from the requirements document presented in Appendix A. The production of this model is described in Section 5.3.

The essential model comprises:

- A context diagram, which provides graphical notation of the system's interactions with external agents (Section B.2).
- An event list, which enumerates the stimuli and reactions through which the system interacts with its environment (Section B.3).
- Transformation schemas, which provide graphical notation of the system's activities in terms of data transformation. These are 'levelled' to express the hierarchical organisation of the system, on a top-down basis. For each transformation, there is either a lower-level transformation schema or a written process specification (Section B.4).
- Process specifications, which describe in structured language an activity depicted on a transformation schema (Section B.5).
- A data specification, which describes the type, meaning and/or composition of each data flow or item of stored data within the system, as depicted on transformation schemas and/or described in transformation specifications. Data may be specified using any conventional notation (Section B.6).

B.2 Context diagram

The context diagram is shown in Fig. B.1.

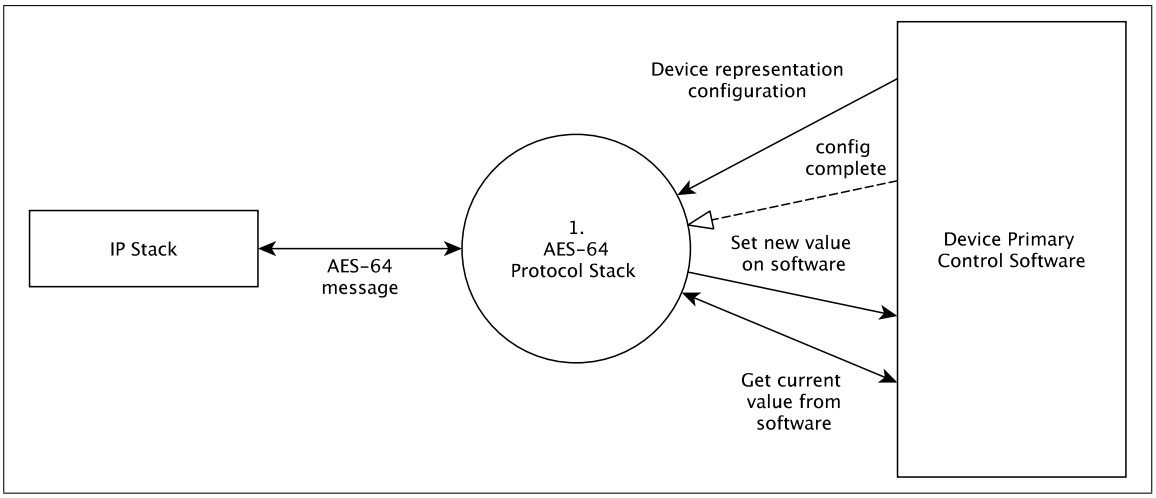


Figure B.1: Context diagram for the AES-64 protocol stack

B.3 Event list

The event list is as follows:

1. **AES-64 message**
   An AES-64 message is received from the IP stack. This event 'packages' all of the possible variations of AES-64 request and response. There are six different kinds of AES-64 request, each of which may convey a variety of commands.

2. **Device node configuration**
   The device primary control software (DPCS) configures the representation maintained on the AES-64 protocol stack. This involves the creation of at least one device node, level hierarchy entries and parameters.
   (Each parameter must have a corresponding level hierarchy entry.)

3. **Configuration complete**
   An event indicating that the DPCS has fully configured the AES-64 protocol stack and that it may now begin processing messages received from the IP stack.

The following interactions depicted on the context diagram are *not* external events:

1. **AES-64 message**
   An AES-64 message is transmitted through the IP stack.

2. **Set new value on software**
   The AES-64 protocol stack changes a value on the DPCS through a parameter's value function.

3. **Get current value from software**
   The AES-64 protocol stack retrieves a current value from the DPCS through a parameter's value function.

B.4 Transformation schema

The transformation schema diagrams are as follows:

Fig. B.2 shows the top-level organisation of the AES-64 protocol stack.

Fig. B.3 shows the organisation of the 'Handle Request' transformation.

Fig. B.4 shows the organisation of the 'Handle Device Request' transformation.

Fig. B.5 shows the organisation of the 'Handle Level Request' transformation.

Fig. B.6 shows the organisation of the 'Handle Parameter Request' transformation.

Fig. B.7 shows the organisation of the 'Receive Message' transformation.

All other transformations within the schema diagrams are specified by transformation specifications (Section B.5).

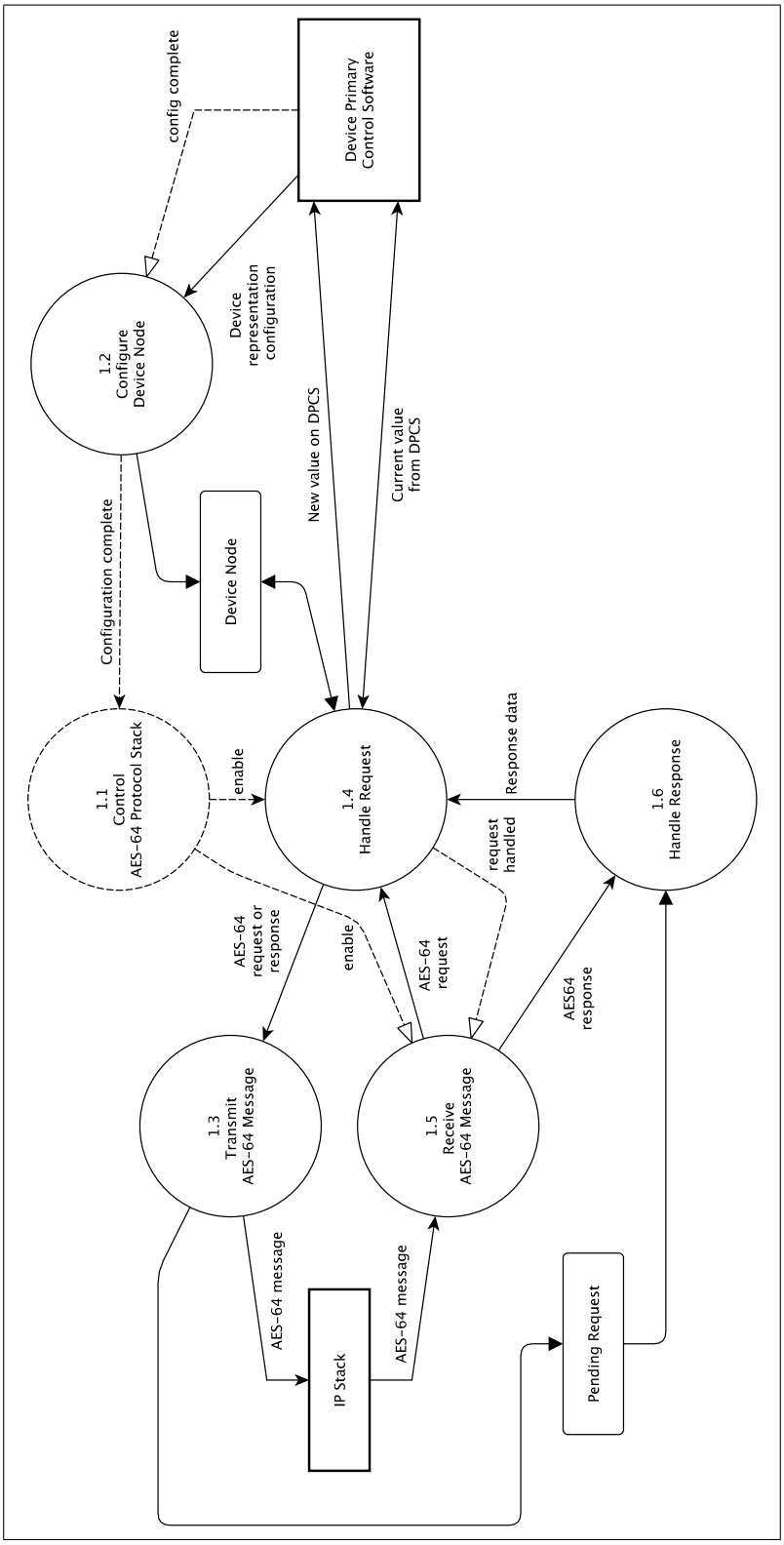


Figure B.2: Top-level transformation schema for the AES-64 protocol stack.

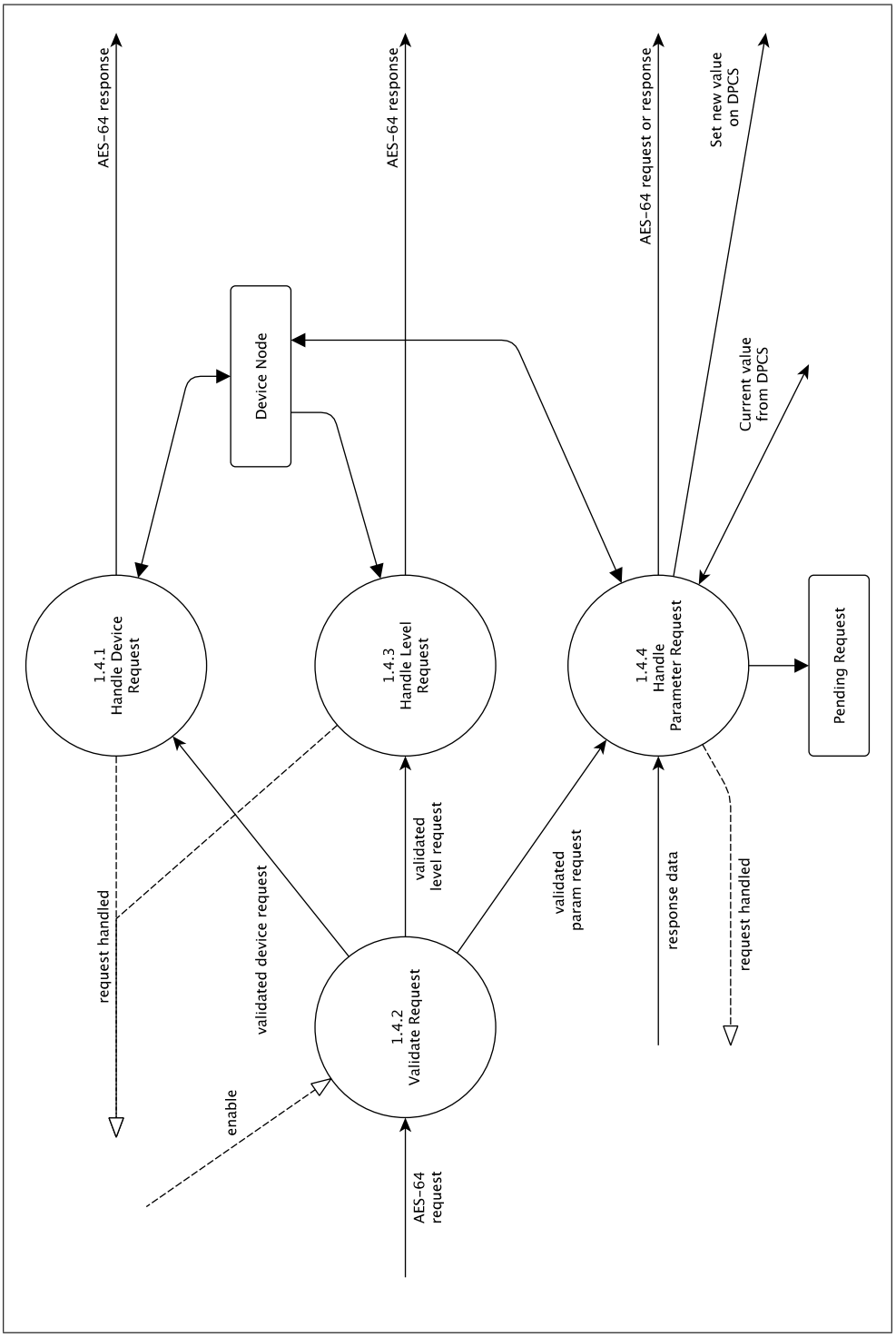


Figure B.3: The 'Handle Request' transformation schema.

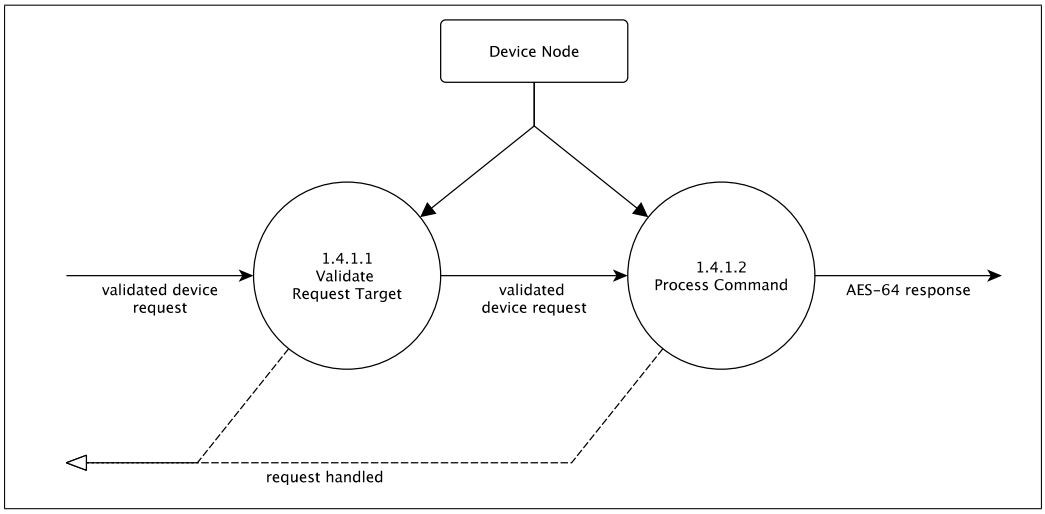


Figure B.4: The 'Handle Device Request' transformation schema.

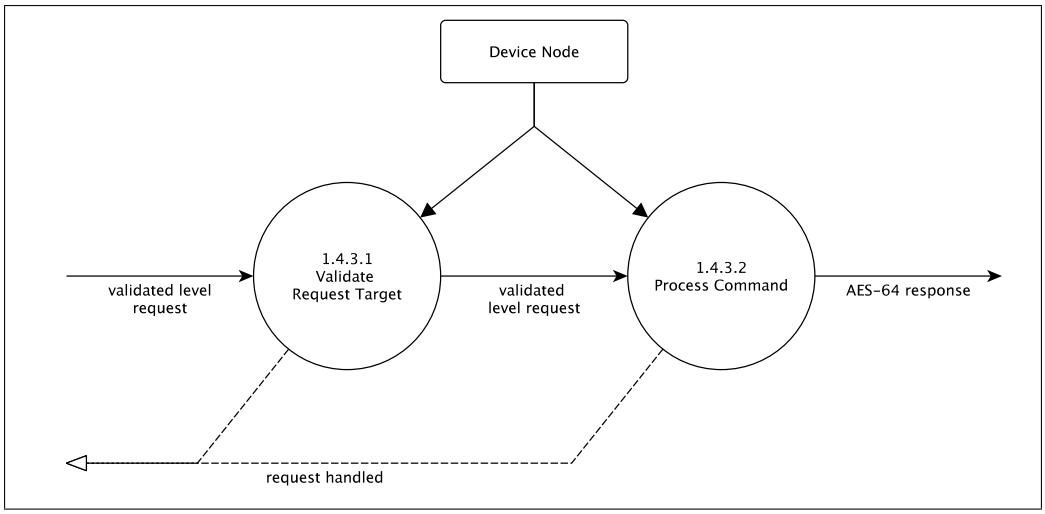


Figure B.5: The 'Handle Level Request' transformation schema.

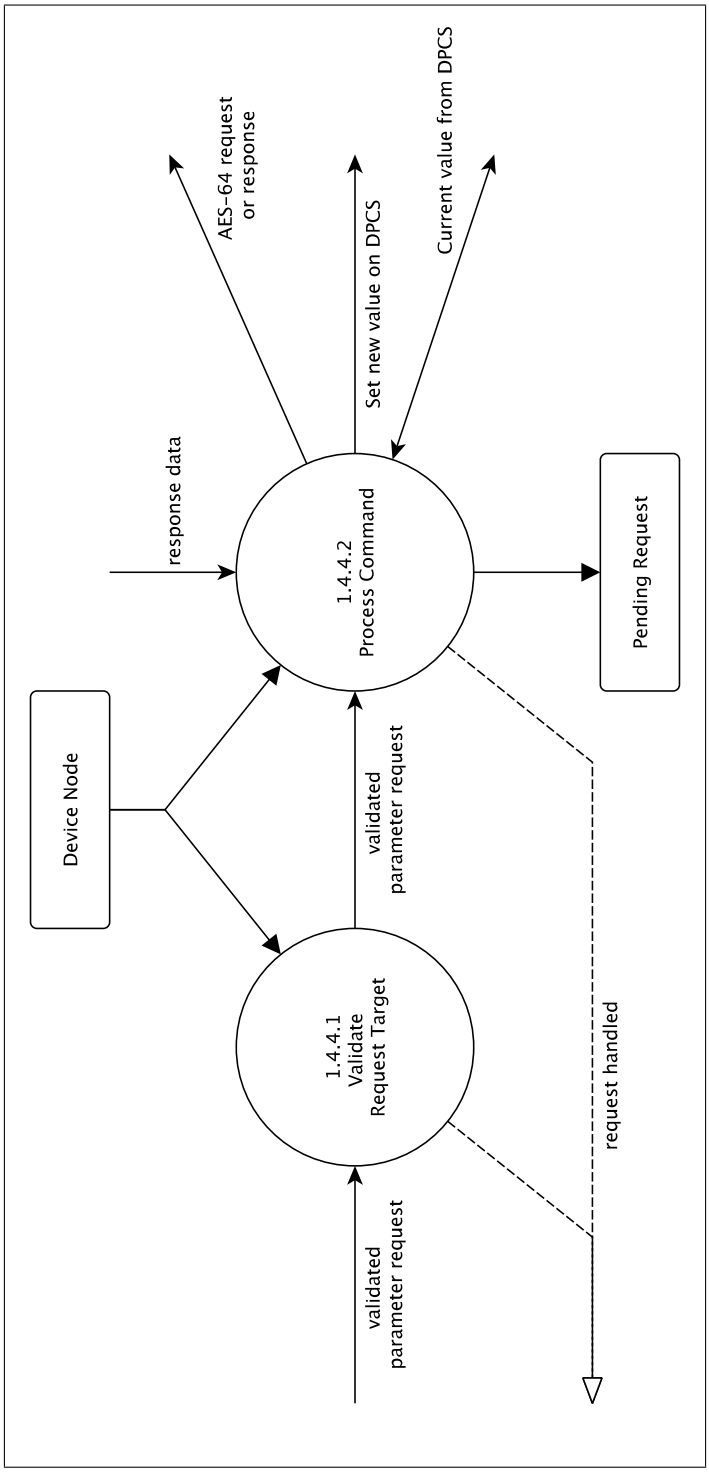


Figure B.6: The 'Handle Parameter Request' transformation schema.

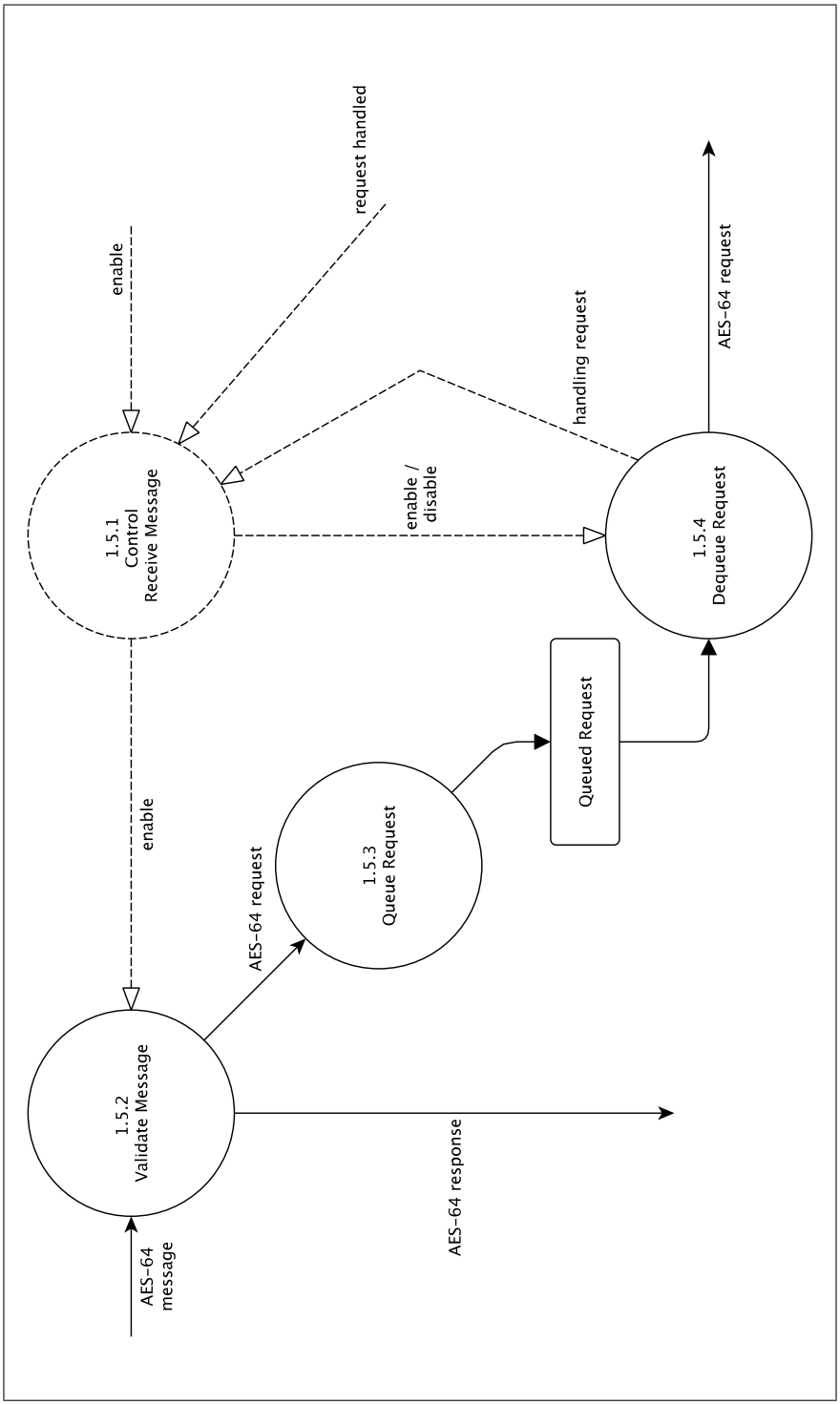


Figure B.7: The 'Receive Message' transformation schema.

B.5 Process specifications

1.1 Control AES-64 Protocol stack

```
CASE configuration complete
        send enable -> 1.5
```

1.2 Configure Device Node

```
CASE new device node
        create new device node and link to global device nodes
CASE new level hierarchy entry
        create new level hierarchy entry and add to last device node
CASE new parameter
        create parameter and add to last device node
CASE config complete
        send configuration complete -> 1.1
```

1.3 Transmit AES-64 Message

```
CASE AES-64 response:
        send AES-64 message
CASE AES-64 request:
        IF request requires a response:
                request sequence ID -> Pending Request store
        send AES-64 message
```

1.4.1.1 Validate Request Target

```
CASE validated device request:
        IF request target device node exists:
                send validated device request -> 1.4.1.2
        ELSE
                discard validated device request
                send request handled -> 1.5.1
```

1.4.1.2 Process Command

For command-specific process specifications, see Subsection B.5.1.

```
CASE validated device request:
        Process request command
        IF Device request requiring a response:
                send AES-64 response -> 1.3
        send request handled -> 1.5.1
```

1.4.2 Validate Request

```
CASE AES-64 request:
        SWITCH on AES-64 request Message type:
        CASE device request:
                send AES-64 request -> 1.4.1
        CASE parameter request:
                if AES-64 request Command qualifier == CLA or NAME:
                        send AES-64 request -> 1.4.3
                else
                        send AES-64 request -> 1.4.4
```

1.4.3.1 Validate Request Target

```
CASE validated level request:
        IF request target device node exists:
                send validated level request -> 1.4.3.2
        ELSE
                discard validated level request
                send request handled -> 1.5.1
```

1.4.3.2 Process Command

For command-specific process specifications, see Subsection B.5.2.

```
CASE validated level request:
        Process request command
        IF Parameter request with full address block target requiring a
           response:
                send AES-64 response -> 1.3
        send request handled -> 1.5.1
```

1.4.4.1 Validate Request Target

```
CASE validated parameter request:
        IF request target device node exists:
                send validated parameter request -> 1.4.4.2
        ELSE
                discard validated parameter request
                send request handled -> 1.5.1
```

1.4.4.2 Process Command

For command-specific process specifications, see Subsection B.5.3.

```
CASE validated parameter request:
        SWITCH parameter target type:
        CASE indexed:
                Process command
                IF request demands a response:
                        Send AES-64 response to 1.3
                Send request handled to 1.5.1
        CASE full address block:
                FOR EACH targeted parameter:
                        Process command
                        IF request demands a response:
                                Send AES-64 response to 1.3
                        Send request handled to 1.5.1
```

1.5.1 Control Receive Message

```
CASE enable:
        send enable -> 1.5.2
        send enable -> 1.5.3
CASE handling request:
        send disable -> 1.5.4
CASE request handled:
        send enable -> 1.5.4
```

1.5.2 Validate Message

```
CASE AES-64 message:
        IF AES-64 message type == RESPONSE:
                send AES-64 response -> 1.6
        ELSE:
                send AES-64 request -> 1.5.3
```

1.5.3 Queue Request

```
CASE AES-64 request:
        AES-64 request -> Queued Request store
```

1.5.4 Dequeue Request

```
CASE request <- Queued Request store:
        Send AES-64 request -> 1.4
        Send handling request -> 1.5.1
```

1.6 Handle Response

```
CASE AES-64 response:
        IF response sequence ID <- Pending Request:
                send AES-64 response data -> 1.4
                send response handled -> 1.5
        ELSE:
                discard AES-64 response
                send response handled -> 1.5
```

B.5.1 Device command handler specifications

IS ALIVE

```
Create response header
Response value field as per A.8.12
```

B.5.2 Level command handler specifications

GET CLA

```
Create response header
Create level alias list
Response value field as per A.8.3
cla resolved = FALSE
search range start = 0
search range end = level items table entry count
FOR EACH level identifier IN request full address block:
        IF level identifier == 0xEE || 0xEEEE || 0xEEEEEE:
                FOR EACH entry in the level items table within search
                    range
                that differs from the previous entry:
                        Create level alias entry
                                (length field, FAB, level alias string)
                        Add entry to level alias list
                cla resolved = TRUE
                EXIT for loop
        ELSE:
                FOR EACH entry in the level items table:
                        IF entry identifier == request level identifier:
                                IF search range start == -1:
                                        search range start = i
                                search range end = i
        IF search range end = -1:
                EXIT for loop
IF cla resolved:
        PACK level alias list into response value field
ELSE:
        DESTROY level alias list
        PACK '0' Entry count into response value field
```

GET NAME

```
Create response header
Response value field as per A.8.7
name resolved = FALSE
search range start = 0
search range end = level items table entry count
FOR EACH level identifier IN request full address block:
        IF level identifier == 0xEE || 0xEEEE || 0xEEEEEE:
                Copy parent level alias to response value field
                name resolved = TRUE
                EXIT for loop
        ELSE:
                FOR EACH entry in the level items table:
                        IF entry identifier == request level identifier:
                                IF search range start == -1:
                                        search range start = i
                                search range end = i
        IF search range end = -1:
                name resolved = FALSE
                EXIT for loop
IF name resolved == FALSE:
        Response value field = NACK
```

B.5.3 Parameter command handler specifications

GET ID

```
Create response header
Response value field as A.8.5
Response value field = target parameter ID
```

GET MSTGRP

```
Create response header
Response value field as A.8.6
Pack target parameter master list count
Pack target parameter current value
FOR EACH master on target parameter:
        Pack group entry
```

GET PTPGRP

```
Create response header
Response value field as A.8.8
Pack count of peers on target parameter
Pack target parameter current value
FOR EACH peer on target parameter:
        Pack group entry
```

GET SLVGRP

```
Create response header
Response value field as A.8.9
Pack count of slaves on target parameter
Pack target parameter current value
FOR EACH slave on target parameter:
        Pack group entry
```

GET VAL

```
Create response header
Retrieve target parameter value function identifiers
Call target parameter value function
Response value field as A.8.10
Response value data from target parameter value function
```

JOIN MSTSLV

```
Create response header
Unpack request slave entry
IF slave entry is on local device:
        Resolve slave entry full address block to single param ID
        Read slave parameter peer entries
        FOR EACH peer entry on slave parameter:
                Set request target parameter as a Master on peer
ELSE:
        Create FAB request header (GET PTPGRP)
        Pack new sequence ID in request header
        Send request
        Wait on response
        Unpack returned peers from response value field
        Create IDX request header (SET MASTER)
        FOR EACH returned peer:
                Send SET MASTER request supplying self as master entry
                Add peer to parameter slaves list
Response value field as A.8.13
```

SET GRPVAL

```
Create response header
Call target parameter value function with request value format and value
    field
/* No response of any kind to SET GRPVAL */
```

SET MASTER

```
Unpack request value field as master group entry
Add master group entry to target parameter Master list
Response as A.8.18
```

SET MASTER OFF

```
Unpack request value field as master group entry
Remove master group entry from target parameter master field
Response as A.8.19
```

SET MSTGRP

```
Unpack request value field as Group List Header and Group List Entries
Remove all master group entries from target parameter Master group list
For each supplied Master group entry:
        Add master group entry to target parameter Master group list
Response as A.8.20
```

SET PTPGRP

```
Unpack request value field as Group List Header and Group List Entries
Remove all entries from target parameter Peer group list
For each supplied Peer entry:
        Add entry to target parameter Peers list
Response as A.8.21
```

SET SLAVE

```
Unpack request value field as slave group entry
Add slave group entry to target parameter Slave group list
Response as A.8.22
```

SET VAL

```
Create target parameter value function with request value format and
    value field
Switch on value function return opcode:
CASE no error || broadcast:
        Send ACK
CASE send data:
        Create response header
        Response value format and data from value function
CASE no response:
        /* send nothing */
DEFAULT:
        Send ACK
/* Group updates */
IF target parameter has slaves:
        FOR EACH slave parameter:
                IF slave is local:
                        Call slave value function with request value
                            field and data
                ELSE:
                        Create IDX request (SET GRPVAL)
                        Pack request value field with original request
                            value field
```

UNJOIN MSTSLV

```
Create response header
Unpack request slave entry from request value field
IF slave is local to request target:
        Resolve request slave entry full address block
        FOR EACH peer on slave parameter:
                Remove this parameter as a Master from each peer
                Remove each peer from parameter slaves list
ELSE:
        Create new FAB request header (GET PTPGRP)
        Pack new sequence ID in request header
        Send request to slave
        Wait on response
        Unpack returned peers from response value field
        Create IDX request header (SET MASTER OFF)
        FOR EACH peer on slave parameter:
                Send SET MASTER OFF request supplying self as master
                    entry
                Remove each peer from parameter slaves list
Response value field as A.8.24
```

B.6 Data specification

There are two data specification tables for the essential model. The first specifies data items that are *directly referenced* in the transformation schema: the first entry is 'AES-64 message', the final entry is 'Pending request'. The second specifies data items that are referenced in the transformation specifications or that are components of items in the first table: the first entry is 'Device node configuration', the final entry is 'Value function return opcode'.

Data Item	Description	Composition	Type	Referenced by
AES-64 message	An AES-64 request or an AES-64 response.	[AES-64 request | AES-64 response]		0 (Context schema)
Device representation configuration	Configuration of the AES-64 protocol stack's device representation. A device node request, a level hierarchy entry, or a parameter.	[Device node configuration | Level hierarchy entry configuration | Parameter configuration]		0 (Context schema)
New value on DPCS	Contents of a parameter request value field to be processed by a parameter's value function.		Parameter-specific	0 (Context schema)
Current value from DPCS	Return value from a parameter's value function.		Parameter-specific	0 (Context schema)
AES-64 request	One of several forms of AES-64 request message.	[Parameter request | Device request]		1.1, AES-64 message
AES-64 response	A response message received in response to a previously issued request.	Response header + Response data		1.1, AES-64 message
Response data	Response-specific. See A.8 for response definitions.	Command-specific value format + response-specific value field	*defined by value format*	1.1
Validated device request	A device request with a confirmed target device node.		Device request	1.4
Validated level request	A level request with a confirmed target device node.		Parameter request	1.4
Validated param request	A parameter request with a confirmed target device node.		Parameter request	1.4
Device Node	Container for the protocol stack's representation of a functional device. The protocol stack may maintain more than one of these.	Device node ID + Parameter count + {Level hierarchy entry} + {Parameter}		1.1
Queued Request	AES-64 request held in a FIFO queue.	AES-64 request		1.5

Pending Request	Representation of an outbound request previously issued by the protocol stack. For resolving response data to previously issued requests and acting on the response data.	Sequence ID	1.1

Data Item	Description	Configuration	Type	Referenced by
Device node configuration	The information required to set up a device node on the AES-64 protocol stack	Maximum parameter count		Device representation configuration
Level hierarchy entry configuration	The information required to set up an entry in the level hierarchy for a single parameter.	Section block + Section block alias + Section type + Section type alias + Section number + Parameter block + Parameter block alias + Parameter block index + Parameter type + Parameter type alias + Parameter index		Device representation configuration
Parameter configuration	The information required to set up a single parameter on the device node.	Parameter name + value format + value function class + value function target + value function index + value function subindex		Device representation configuration
Parameter request	An AES-64 request that targets one or more parameters on a device node. It may or may not require the recipient to send a response.	[Parameter request with full address block target, requiring a response | Parameter request with full address block target, not requiring a response | Parameter request with indexed target, requiring a response | Parameter request with indexed target, not requiring a response]		AES-64 request
Device request	An AES-64 request that targets a device node. It may or may not require the recipient to send a response.	[Device request requiring a response | Device request not requiring a response]		AES-64 request

Data Item	Description	Configuration	Type	Referenced by
Response header	Specification-defined header of an AES-64 response message	AES-64 network information + User level + Message type + Sequence ID		AES-64 response
Parameter name	Name of a parameter. Typically identical with the corresponding Parameter Type alias.		string	Parameter configuration
Value format	Specification-defined enumerated type identifying the type of a value field or attribute.		*8-bit enumerated type*	Parameter configuration
Value function class	Identifies the type of control function that the parameter corresponds to		*32-bit enumerated type*	Parameter configuration
Value function target	Identifies the parameter's corresponding control within the value function class		*32-bit enumerated type*	Parameter configuration
Value function index	Identifies the instance of a target of a parameter's value function		*32-bit enumerated type*	Parameter configuration
Value function subindex	Identifies an index within an instance of a target of a parameter's value function.		*32-bit enumerated type*	Parameter configuration
Parameter request with full address block target, requiring a response	Parameter request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Sequence ID + Command executive + Command qualifier + Full address block + Command-specific value field		AES-64 request

Data Item	Description	Configuration	Type	Referenced by
Parameter request with full address block target, not requiring a response	Parameter request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Command executive + Command qualifier + Full address block + Command-specific value field		AES-64 request
Parameter request with indexed target, requiring a response	Parameter request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Sequence ID + Command executive + Command qualifier + Target parameter ID + Command-specific value field		AES-64 request
Parameter request with indexed target, not requiring a response	Parameter request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Command executive + Command qualifier + Target parameter ID + Command-specific value field		AES-64 request
Device request requiring a response	Device request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Sequence ID + Command executive + Command qualifier + Command-specific value field		AES-64 request
Device request not requiring a response	Device request combining a specification-defined header and a command-specific value field	AES-64 network information + User level + Message type + Command executive + Command qualifier + Command-specific value field		AES-64 request

Data Item	Description	Configuration	Type	Referenced by
AES-64 network information		Target IP address + Target device node ID + Source IP address + Source device node ID + Source param ID		
Target IP address			*unsigned 32-bit integer*	AES-64 network information
Target device node ID			*unsigned 32-bit integer*	AES-64 network information
Source IP address			*unsigned 32-bit integer*	AES-64 network information
Source device node ID			*unsigned 32-bit integer*	AES-64 network information
Source parameter ID			*unsigned 32-bit integer*	AES-64 network information
User level	Specification-defined security identifier.		*unsigned 8-bit integer*	AES-64 message
Message type	Specification-defined enumerated type at a fixed offset in all message headers, defines type of request or response		*8-bit enumerated type*	AES-64 message
Sequence ID	Arbitrary unique ID used to match Responses to Requests.		*unsigned 32-bit integer*	AES-64 response, AES-64 requests requiring a response

Data Item	Description	Configuration	Type	Referenced by
Full address block	Specification-defined data structure that identifies one or more parameters by their location(s) within the level hierarchy. May contain valid identifiers or wildcards.	Section block + Section type + Section number + Parameter block + Parameter block index + Parameter type + Parameter index		Parameter request with full address block target
Target parameter ID			*unsigned 32-bit integer*	Parameter request with indexed target
Command executive	Specification-defined		*8-bit enumerated type*	AES-64 request
Command qualifier	Specification-defined		*8-bit enumerated type*	AES-64 request

Data Item	Description	Configuration	Type	Referenced by
Command-specific value field	Value field specific to the command expressed by a request, as indicated by its Command executive and Command qualifier fields. See A.8 for definition of value fields.	Value format + [GET CLA request value field | GET ID request value field | GET MSTGRP request value field | GET NAME request value field | GET PTPGRP request value field | GET SLVGRP request value field | GET VAL request value field | GET VTBL request value field | IS ALIVE request value field | JOIN MSTSLV request value field | SET GRPVAL request value field | SET MASTER request value field | SET MASTER OFF request value field | SET MSTGRP request value field | SET PTPGRP request value field | SET SLAVE request value field | SET VAL request value field | UNJOIN MSTSLV request value field]		AES-64 request

Data Item	Description	Configuration	Type	Referenced by
Response-specific value field	Value field specific to the command that initiated an AES-64 response. See A.8 for definition of value fields.	Value format + [GET CLA response value field | GET ID response value field | GET MSTGRP response value field | GET NAME response value field | GET PTPGRP response value field | GET SLVGRP response value field | GET VAL response value field | GET VTBL response value field | IS ALIVE response value field | JOIN MSTSLV response value field | SET MASTER response value field | SET MASTER OFF response value field | SET MSTGRP response value field | SET PTPGRP response value field | SET SLAVE response value field | SET VAL response value field | UNJOIN MSTSLV response value field]		AES-64 response
Device node ID	Unique identifier for a Device node within the AES-64 protocol stack.		*unsigned 32-bit integer*	Device node
Parameter count	Count of the parameters contained within a Device node.		*unsigned 32-bit integer*	Device node

Data Item	Description	Configuration	Type	Referenced by
Level hierarchy entry	An entry in the level hierarchy which corresponds uniquely to one parameter.	Section block + Section block alias + Section type + Section type alias + Section number + Parameter block + Parameter block alias + Parameter block index + Parameter type + Parameter type alias + Parameter index		Device node
Parameter	A parameter which uniquely represents one functional attribute of the device primary control software. Belongs to a single Device node. Corresponds to one Level hierarchy entry in that Device node.	Parameter ID + Parameter name + Value format + Parameter value + Value function class + Value function target + Value function index + Value function subindex + Peer group list + Master group list + Slave group list		Device node
Section block	Specification-defined enumerated type.		*8-bit enumerated type*	Level hierarchy entry, Full address block
Section block alias	Level alias for section block identifier.		*string*	Level hierarchy entry
Section type	Specification-defined enumerated type.		*8-bit enumerated type*	Level hierarchy entry, Full address block
Section type alias	Level alias for section type identifier.		*string*	Level hierarchy entry
Section number	Identifies an instance within multiples of a section type.		*unsigned 24-bit integer*	Level hierarchy entry, Full address block
Parameter block	Specification-defined enumerated type.		*8-bit enumerated type*	Level hierarchy entry, Full address block

Data Item	Description	Configuration	Type	Referenced by
Parameter block alias	Level alias for parameter block identifer.		*string*	Level hierarchy entry
Parameter block index	Identifies an instance within multiples of a parameter block.		*unsigned 24-bit integer*	Level hierarchy entry, Full address block
Parameter type	Specification-defined enumerated type.		*16-bit enumerated type*	Level hierarchy entry, Full address block
Parameter type alias	Level alias for parameter type identifier.		*string*	Level hierarchy entry
Parameter index	Identifies an instance within multiples of a parameter type.		*unsigned 16-bit index*	Level hierarchy entry, Full address block
Maximum parameter count	Indicates the greatest number of parameters that may be held by a device node.		*unsigned 32-bit integer*	Device node configuration
Peer group list	List of entries, identifying parameters in a Peer-to-peer join with the parameter that owns the list.	Peer list count + {Peer group entry}		Parameter, Get PTPGRP
Master group list	List of entries, identifying parameters in a master-slave join with the parameter that owns the list, where the latter is a slave to the parameters in the list.	Master list count + {Master group entry}		Parameter, Get MSTGRP
Slave group list	List of entries, identifying parameters in a master-slave join with the parameter that owns the list, where the latter is a master to the parameters in the list.	Slave list count + {Slave group entry}		Parameter, Get SLVGRP

Data Item	Description	Configuration	Type	Referenced by
Level identifier	One of the identifiers within a level hierarchy entry.	[Section block | Section type | Section number | Parameter block | Parameter block index | Parameter type | Parameter index]		Get CLA
Level alias list	A list of level alias entries.	Entry count + { Level alias list entry }		Get CLA
Level alias list entry	Provides the alias of a level and a full address block giving its position in the level hierarchy.	Entry length in bytes + Full address block + Level alias		Get CLA
Master list count	Count of entries on a parameter's Master list.		*unsigned 32-bit integer*	Master group list, Get MSTGRP
Peer list count	Count of entries on a parameter's Peer list.		*unsigned 32-bit integer*	Peer group list, Get PTPGRP
Slave list count	Count of entries on a parameter's Slave list.		*unsigned 32-bit integer*	Slave group list, Get SLVGRP
Peer group entry	Entry providing information about a peer, its location, offset from the local parameter's value, and grouping attributes.	Target IP address + Target device node ID + Target Parameter ID + Value offset + Group status + Group join type + Group flags register		Peer group list, Get PTPGRP, Join MSTSLV
Master group entry	Entry providing information about a master, its location, offset from the local parameter's value, and grouping attributes.	Target IP address + Target device node ID + Target Parameter ID + Value offset + Group status + Group join type + Group flags register		Master group list, Get MSTGRP

Data Item	Description	Configuration	Type	Referenced by
Slave group entry	Entry providing information about a slave. Lacks the value offset and additional attributes since these details are not needed by a master.	Target IP address + Target device node ID + Target Parameter ID + Group status		Slave group list, Get SLVGRP
Value offset	Offset between a parameter's value and the value of a master or peer parameter it is joined to.		*unsigned 32-bit integer*	Peer group entry, Master group entry
Group status	Indicates the group relationship is active or inactive.		*8-bit enumerated type*	Peer group entry, Master group entry, Slave group entry
Group join type	Indicates whether the group value relationship is absolute or relative.		*8-bit enumerated type*	Peer group entry, Master group entry
Group flags register	Indicates miscellaneous group attributes.		*8-bit integer*	Peer group entry, Master group entry
Request slave entry	Details of a slave as supplied by a Join MSTSLV request. Differs from a Slave group entry.	Target IP address + Target device node ID + Full address block		Join MSTSLV
Group list header	Header preceding a list of peer group entries in a Set MSTGRP or PTPGRP request (see A.8.20 and A.8.21).	Entry count + Current param value		Set MSTGRP, Set PTPGRP
Value function return opcode	Opcode returned by a parameter's value function to report status of a set value operation.	[No error | Send data | No response]		Set VAL

C AES-64 task interfaces

C.1 Introduction

This appendix documents the client interfaces provided by each component task of the AES-64 protocol stack, which are used to conduct all inter-task communications over XC channels.

C.2 Stack control task client interface

```
/*
 * a64_control_api.h
 */

#ifndef A64_CONTROL_API_H_
#define A64_CONTROL_API_H_
#include <xccompat.h>
#include "a64_defines.h"
#include "a64_ctrl_cmd.h"
#include "a64_mp_cmd.h"
#include "a64_dnm_cmd.h"
#include "a64_value_function_classes.h"

void a64_ctrl_api_send_mp_cmd(chanend c_ctrl, enum A64MPCommand cmd,
        UInt32 nbytes);

/*
 * Notify stack control task that this message processing
 * task is now available to service requests
 */
void a64_ctrl_api_notify_mp_enabled(chanend c_ctrl, UInt32 mp_uid);

void a64_ctrl_api_notify_mp_completed(chanend c_ctrl, BOOL success);

/*
 * Retrieve a fresh sequence ID for an outbound request that
```

```
 * needs a response.
 */
void a64_ctrl_api_get_new_sequence_id(chanend c_ctrl,
        REFERENCE_PARAM(UInt32, sequence_id));

/*
 * Transmit a request or response message.
 */
void a64_ctrl_api_submit_tx_request(chanend c_ctrl,
        UInt8 tx_buffer[], UInt32 tx_length);

/*
 * Fire a 'get' value function on the stack control task.
 * Supply the parameter's value function identifiers,
 * retrieve the length and data returned from the value function.
 */
void a64_ctrl_api_process_get_value_function(chanend c_ctrl,
        UInt8 vf_class, UInt8 vf_target, UInt8 vf_idx, UInt8 vf_subidx,
        REFERENCE_PARAM(UInt8, value_length), UInt8 value[]);

/*
 * Fire a 'set' value function on the stack control task.
 * Supply the parameter's value function identifiers and
 * the 'set' request's value length and data. Retrieve the
 * value format, length and data of any response.
 */
UInt8 a64_ctrl_api_process_set_value_function(chanend c_ctrl,
        UInt8 vf_class, UInt8 vf_target, UInt8 vf_idx, UInt8 vf_subidx,
        UInt8 value_length, UInt8 value_data[],
        REFERENCE_PARAM(UInt8, response_valft),
        REFERENCE_PARAM(UInt8, response_length), UInt8 response_data[]);

/*
 * After sending an outbound request that demands a response (e.g.,
 * 'GET PTPGRP'), wait on the stack control task to signal that a
 * matching response has been received.
 */
void a64_ctrl_api_wait_for_response(chanend c_ctrl_app, UInt8 rsp_buf[],
        REFERENCE_PARAM(UInt32, rsp_numbytes));

#endif /* A64_CONTROL_API_H_ */
```

C.3 Message processing task client interface

```
/*
 * a64_msg_proc_client_api.h
 */

#ifndef A64_MSG_PROC_CLIENT_API_H_
#define A64_MSG_PROC_CLIENT_API_H_
#include <xccompat.h>
#include "a64_ctrl_cmd.h"
#include "a64_dnm_cmd.h"
#include "a64_mp_cmd.h"
#include "a64_defines.h"

void a64_msg_proc_api_send_ctrl_cmd(chanend c_msg_proc,
                enum A64CtrlCommand cmd);

/*
 * Inform message processing task:
 *      a) that it is enabled,
 *      b) what its unique ID is.
 */
void a64_msg_proc_api_enable_processing(chanend c_msg_proc, int mp_uid);

/*
 * Submit request to message processing task
 */
void a64_msg_proc_api_submit_message(chanend c_msg_proc,
        enum A64MsgType msg_type, unsigned char buf[],
        UInt32 nbytes);

/*
 * Advise message processing task that a transmission job
 * is now complete.
 */
void a64_msg_proc_api_notify_tx_completed(chanend c_msg_proc);

#endif /* A64_MSG_PROC_CLIENT_API_H_ */
```

C.4 Device representation task client interface

```
/*
 * a64_dnm_client_api.h
 */

#ifndef A64_DNM_CLIENT_API_H_
#define A64_DNM_CLIENT_API_H_
#include <xccompat.h>
#include "a64_defines.h"
#include "a64_command_token.h"
#include "a64_ctrl_cmd.h"
#include "a64_mp_cmd.h"
#include "a64_value_function_classes.h"

enum A64DNMParamAttribute {
        A64_DNM_PARAM_ATTRIBUTE_INVALID,
        A64_DNM_PARAM_ATTRIBUTE_NAME,
        A64_DNM_PARAM_ATTRIBUTE_VALFT,
        A64_DNM_PARAM_ATTRIBUTE_GU_VALUE,
        A64_DNM_PARAM_ATTRIBUTE_FLAGS,
        A64_DNM_PARAM_ATTRIBUTE_VALUE_CALLBACK_ID,
        A64_DNM_PARAM_ATTRIBUTE_PEERS_COUNT,
        A64_DNM_PARAM_ATTRIBUTE_PEER_ENTRY,
        A64_DNM_PARAM_ATTRIBUTE_MASTERS_COUNT,
        A64_DNM_PARAM_ATTRIBUTE_MASTER_ENTRY,
        A64_DNM_PARAM_ATTRIBUTE_SLAVES_COUNT,
        A64_DNM_PARAM_ATTRIBUTE_SLAVE_ENTRY,
};

/* Provided for future expansion */
enum A64DNMDNodeAttribute {
        A64_DNM_DNODE_ATTRIBUTE_INVALID
};

void a64_dnm_api_send_mp_cmd(chanend c_dnm, enum A64MPCommand cmd);

/*
 * Full address block resolution.
 * Returns success of the request.
 * Sends the device node ID and full address block to the DNM task.
 * If resolvable, the DNM task returns a bitmap of the selected
 * parameters, the size of the bitmap, and the number of
 * parameters located.
```

```
 */
UInt8 a64_dnm_api_resolve_fab(chanend c_dnm,
        UInt32 dnode_id,
        UInt8 sb, UInt8 st, UInt32 sn,
        UInt8 pb, UInt32 pbi, UInt32 pt, UInt32 pi,
        REFERENCE_PARAM(UInt32, size_param_select_bitmap),
        UInt32 param_select_bitmap[],
        REFERENCE_PARAM(UInt32, located_param_count));

/*
 * GET CLA resolution.
 * Returns success of the request.
 * Sends the device node ID and full address block to the DNM task.
 * The DNM task returns a list of child level aliases which can
 * be written directly to the MP task's outbound message buffer.
 * This requires the buffer to have been packed with the
 * response header already.
 */
UInt8 a64_dnm_api_get_cla(chanend c_dnm,
        UInt32 dnode_id,
        UInt8 sb, UInt8 st, UInt32 sn,
        UInt8 pb, UInt32 pbi, UInt32 pt, UInt32 pi,
        UInt8 buf[],
        REFERENCE_PARAM(UInt32, buf_length));

/*
 * GET NAME resolution.
 * Returns success of the request.
 * DNM task returns the length and alias of the identified level
 * which can be written directly to the MP task's outbound message
 * buffer. This requires the buffer to have been packed with the
 * response header already.
 */
UInt8 a64_dnm_api_get_name(chanend c_dnm,
        UInt32 dnode_id,
        UInt8 sb, UInt8 st, UInt32 sn,
        UInt8 pb, UInt32 pbi, UInt32 pt, UInt32 pi,
        UInt8 buf[],
        REFERENCE_PARAM(UInt32, buf_length));

/*
 * Validates single param target (i.e., whether it exists).
 * Returns success of the request.
 */
```

```
88 UInt8 a64_dnm_api_validate_param_id(chanend c_dnm,
        UInt32 dnode_id, UInt32 param_id);
90
   /*
92  * Validates device node target.
    * Returns success of the request.
94  */
   UInt8 a64_dnm_api_validate_dnode_id(chanend c_dnm,
96          UInt32 dnode_id);

98 /*
    * Read attributes from parameter target.
100 * Returns success of request and places attribute in
    * reference parameters.
102 */
   UInt8 a64_dnm_api_read_param_valft(chanend c_dnm,
104         UInt32 dnode_id, UInt32 param_id,
          REFERENCE_PARAM(UInt8, valft));
106
   UInt8 a64_dnm_api_read_param_gu_value(chanend c_dnm,
108         UInt32 dnode_id, UInt32 param_id,
          REFERENCE_PARAM(UInt32, gu_value));
110
   UInt8 a64_dnm_api_read_param_flags(chanend c_dnm, UInt32 dnode_id,
112         UInt32 param_id, UInt32 flags[]);

114 UInt8 a64_dnm_api_read_param_peer_count(chanend c_dnm,
          UInt32 dnode_id, UInt32 param_id, REFERENCE_PARAM(UInt8,
              peer_count));
116
   UInt8 a64_dnm_api_read_param_peer_entry(chanend c_dnm, UInt32 dnode_id,
118         UInt32 param_id, UInt8 entry_idx, UInt32 peer_entry[]);

120 UInt8 a64_dnm_api_read_param_master_count(chanend c_dnm, UInt32 dnode_id,
          UInt32 param_id, REFERENCE_PARAM(UInt8, master_count));
122
   UInt8 a64_dnm_api_read_param_master_entry(chanend c_dnm, UInt32 dnode_id,
124         UInt32 param_id, UInt8 entry_idx, UInt32 master_entry[]);

126 UInt8 a64_dnm_api_read_param_slave_count(chanend c_dnm, UInt32 dnode_id,
          UInt32 param_id, REFERENCE_PARAM(UInt8, slave_count));
128
   UInt8 a64_dnm_api_read_param_slave_entry(chanend c_dnm, UInt32 dnode_id,
```

```
130         UInt32 param_id, UInt8 entry_idx, UInt32 slave_entry[]);

132 UInt8 a64_dnm_api_read_param_value_function(chanend c_dnm,
        UInt32 dnode_id, UInt32 param_id, UInt8 value_function_info[]);
134
    UInt8 a64_dnm_api_read_dnode_count(chanend c_dnm,
136         REFERENCE_PARAM(UInt8, dnode_count));

138 UInt8 a64_dnm_api_read_dnode_param_count(chanend c_dnm, UInt32 dnode_id,
        REFERENCE_PARAM(UInt8, dnode_param_count));
140
    /*
142  * Write attributes to parameter target.
     * Returns success of operation.
144  */
    UInt8 a64_dnm_api_write_param_gu_value(chanend c_dnm, UInt32 dnode_id,
146         UInt32 param_id, UInt32 gu_value);

148 UInt8 a64_dnm_api_write_param_flags(chanend c_dnm, UInt32 dnode_id,
        UInt32 param_id, UInt32 flags[]);
150
    UInt8 a64_dnm_api_write_param_peer_entry(chanend c_dnm, UInt32 dnode_id,
152         UInt32 param_id, UInt32 peer_entry[]);

154 UInt8 a64_dnm_api_write_param_master_entry(chanend c_dnm,
        UInt32 dnode_id, UInt32 param_id, UInt32 master_entry[]);
156
    UInt8 a64_dnm_api_write_param_slave_entry(chanend c_dnm,
158         UInt32 dnode_id, UInt32 param_id, UInt32 slave_entry[]);

160 /*
     * Find attribute on parameter target.
162  * For group implementation.
     * Returns success of operation and data in reference parameter.
164  */
    UInt8 a64_dnm_api_find_param_peer_entry(chanend c_dnm, UInt32 dnode_id,
166         UInt32 param_id, UInt32 peer_entry[]);

168 UInt8 a64_dnm_api_find_param_master_entry(chanend c_dnm, UInt32 dnode_id,
        UInt32 param_id, UInt32 master_entry[]);
170
    UInt8 a64_dnm_api_find_param_slave_entry(chanend c_dnm, UInt32 dnode_id,
172         UInt32 param_id, UInt32 slave_entry[]);
```

```
174 /*
     * Remove attribute on parameter target.
176  * For group implementation.
     * Returns success of operation.
178  */
    UInt8 a64_dnm_api_remove_param_peer_entry(chanend c_dnm,
180         UInt32 dnode_id, UInt32 param_id, UInt32 peer_entry[]);

182 UInt8 a64_dnm_api_remove_param_master_entry(chanend c_dnm,
            UInt32 dnode_id, UInt32 param_id, UInt32 master_entry[]);
184
    UInt8 a64_dnm_api_remove_param_slave_entry(chanend c_dnm,
186         UInt32 dnode_id, UInt32 param_id, UInt32 slave_entry[]);

188 #endif /* A64_DNM_CLIENT_API_H_ */
```

D Device representation configuration for the XMOS Ethernet AVB streaming audio device

D.1 Introduction

This appendix presents the code that configures the device representation of the XMOS Ethernet AVB streaming audio device. This configures the AES-64 protocol stack to provide parameter control / status information for:

- the configuration of the control network interface;
- a configurable number of IEEE 1722 Talker units, where controls for connection management and control of other features are included;
- a configurable number of IEEE 1722 Listener units;
- input and output channels on the standard Cirrus Logic CS4272 audio codec.

Subsection 6.11.2 provides a more concise example of how device nodes, parameters and level hierarchy entries are created.

D.2 Approach

This code minimises its footprint in the executable binary by iterating over common commands ('create parameter', 'add parameter to device node') wherever possible (e.g., the loop on lines 117–123)[1].

Important function prototypes are shown in Listing D.1.

[1] This also makes the creation of many subtly distinguished parameters clearer and less vulnerable to programmer error.

D.3 Configuration

In this configuration file, the representation of an XMOS Ethernet AVB streaming audio device is carried out by five functions:

`a64_configure_device_info()` (l. 115) configures a device node to hold AES-64 network discovery information, including the device's IP address parameter and a 'device name'.

`a64_configure_audio_inputs()` (l. 142) configures the representation of the device's audio hardware input channels.

`a64_configure_avb_listeners()` (l. 159) configures the representation of the device's Ethernet AVB listener units, where the number of listener units and the number of channels are set by constant values in the XMOS Ethernet AVB subsystem's `avb_conf.h` file.

`a64_configure_audio_outputs()` (l. 188) configures the representation of the device's audio hardware output channels.

`a64_configure_avb_talkers()` (l. 209) configures the representation of the device's Ethernet AVB talker units, where the number of talker units and the number of channels are set by constant values in the XMOS Ethernet AVB subsystem's `avb_conf.h` file.

`a64_configure_avb_media_clocks()` (l. 239) configures the representation of the device's Ethernet AVB media clock server units, where the number of media clock server units are set by the `AVB_NUM_MEDIA_CLOCKS` constant in the XMOS Ethernet AVB subsystem's `avb_conf.h` file.

Listing D.1: Important configuration function prototypes

```
struct DNode *dnode_create(UInt8 max_params);

void add_param_to_dnode_param_array(struct DNode *dnode, struct Param *
    add_this);

struct LevelItemsTable *init_level_items_table(UInt8 max_entries);

struct LevelItemsTableEntry *create_level_items_table_entry(enum
    SectionBlock sb, char *sb_alias, enum SectionType st, char *st_alias,
    UInt32 sn, enum ParamBlock pb, char *pb_alias, UInt32 pbi, enum
    ParamType pt, char *pt_alias, UInt32 pi);

int add_to_level_items_table(struct LevelItemsTable *level_items, struct
    LevelItemsTableEntry **li_entry);

int add_parameter_to_dnode(struct DNode *parent_dnode, char *name, UInt8
    valft, UInt32 value, UInt8 value_function_class, UInt8
    value_function_target, UInt8 value_function_target_index, UInt8
    value_function_target_subindex);
```

```
/*
 * a64_device_configuration.c
 */
#include <string.h>
#include <print.h>
#include "a64_defines.h"
#include "a64_identifiers.h"
#include "a64_conf.h"
#include "avb_conf.h"
#include "a64_device_configuration.h"
#include "a64_level_items_table.h"
#include "a64_parameter.h"
#include "a64_device_node.h"
#include "a64_value_function_classes.h"
#include "a64_network_vf.h"
#include "a64_device_vf.h"
#include "a64_experimental_vf.h"
#include "a64_ethernet_avb_source_vf.h"
#include "a64_ethernet_avb_sink_vf.h"
#include "a64_ethernet_avb_media_clock_vf.h"
#include "a64_cs4272_audio_vf.h"

/* Globals */
char *sb_alias_01 = "device␣config";
char *sb_alias_02 = "output␣signal";
char *sb_alias_03 = "input␣signal";
char *sb_alias_04 = "clock␣synchro";

char *st_alias_01 = "ip";
char *st_alias_02 = "i/face␣config";
char *st_alias_03 = "device";
char *st_alias_04 = "stream";
char *st_alias_05 = "audio";
char *st_alias_06 = "media";

char *pb_alias_01 = "config";
char *pb_alias_02 = "multicore";
char *pb_alias_03 = "digital␣gain";
char *pb_alias_04 = "clock␣synchro";

char *pt_alias_01 = "ip␣address";
char *pt_alias_02 = "subnet␣mask";
char *pt_alias_03 = "device␣type";
char *pt_alias_04 = "stream␣id";
```

```
char *pt_alias_05 = "advertise";
char *pt_alias_06 = "listen";
char *pt_alias_07 = "talker state";
char *pt_alias_08 = "listener state";
char *pt_alias_09 = "src maap addr";
char *pt_alias_10 = "dest maap addr";
char *pt_alias_11 = "format";
char *pt_alias_12 = "format sample rate";
char *pt_alias_13 = "vlan id";
char *pt_alias_14 = "pres time offset";
char *pt_alias_15 = "num channels";
char *pt_alias_16 = "channel map";
char *pt_alias_17 = "gain";
char *pt_alias_18 = "mute";
char *pt_alias_19 = "device name";
char *pt_alias_20 = "clock type";
char *pt_alias_21 = "clock sample rate";
char *pt_alias_22 = "clock source";
char *pt_alias_23 = "clock state";
char *pt_alias_24 = "sync source";

/* Prototypes for initialisation of grouped functions */
int a64_configure_device_info();
int a64_configure_audio_inputs();
int a64_configure_avb_listeners();
int a64_configure_audio_outputs();
int a64_configure_avb_talkers();
int a64_configure_avb_media_clocks();

#define NUM_CONFIG_PARAMS 4

int a64_device_configuration_init()
{
        int retval = 0;
        /* One device node for the device config information */
        struct DNode *new_node = dnode_create(NUM_CONFIG_PARAMS);
        if (new_node == NULL) return -1;
        append_to_global_dnodes(get_global_dnodes(), &new_node);

        retval = a64_configure_device_info();
#ifdef A64_VERBOSE_DEBUG
        if (retval == -1) printstrln("error configuring device info");
#endif
        /* Another new node for the rest of the device */
```

```
        struct DNode *new_node2 = dnode_create(40);
        if (new_node2 == NULL) return -1;
        append_to_global_dnodes(get_global_dnodes(), &new_node2);

        retval = a64_configure_audio_inputs();
#ifdef A64_VERBOSE_DEBUG
        if (retval == -1) printstrln("error␣configuring␣audio␣inputs");
#endif
        retval = a64_configure_avb_listeners();
#ifdef A64_VERBOSE_DEBUG
        if (retval == -1) printstrln("error␣configuring␣avb␣listeners");
#endif
        retval = a64_configure_audio_outputs();
#ifdef A64_VERBOSE_DEBUG
        if (retval == -1) printstrln("error␣configuring␣audio␣outputs");
#endif
        retval = a64_configure_avb_talkers();
#ifdef A64_VERBOSE_DEBUG
        if (retval == -1) printstrln("error␣configuring␣avb␣talkers");
#endif
    retval = a64_configure_avb_media_clocks();
#ifdef A64_VERBOSE_DEBUG
    if (retval == -1) printstrln("error␣configuring␣media␣clocks");
#endif
}

int a64_configure_device_info()
{
        /* Devices are expected to hold these parameters on
         * 'Device Node 0'. All other params should be on Device Node 1
         * and upwards
         ***/
        int retval = 0;
        struct DNode *last_dnode = get_last_dnode();
        enum A64ValueFunctionClass dc_vf_class_array[NUM_CONFIG_PARAMS] =
             {VFN_CLASS_NETWORK, VFN_CLASS_NETWORK,VFN_CLASS_DEVICE,
            VFN_CLASS_INVALID};
        UInt8 dc_st_array[NUM_CONFIG_PARAMS] = {ST_IP,
            ST_INTERFACE_CONFIG, ST_DEVICE, ST_DEVICE};
        char *dc_st_alias_array[NUM_CONFIG_PARAMS] = {st_alias_01,
            st_alias_02, st_alias_03, st_alias_03};
        UInt16 dc_pt_array[NUM_CONFIG_PARAMS] = {PT_IPADDRESS,
            PT_SUBNET_MASK, PT_DEVICE_NAME, PT_DEVICETYPE};
```

```
        char *dc_pt_alias_array[NUM_CONFIG_PARAMS] = {pt_alias_01,
            pt_alias_02, pt_alias_19, pt_alias_03};
128     enum A64Valft dc_valft_array[NUM_CONFIG_PARAMS] = {VALFT_INT_32,
            VALFT_INT_32, VALFT_STRING, VALFT_INT_32};
        UInt8 dc_vf_target_array[NUM_CONFIG_PARAMS] =
130             {NETWORK_IP_ADDRESS, NETWORK_SUBNET_MASK, DEVICE_NAME,
                    DEVICE_TYPE_AVB_DEVICE};

132     for (int i = 0; i < NUM_CONFIG_PARAMS; i++) {
                struct LevelItemsTableEntry *new_lite01 =
                    create_level_items_table_entry(SB_DEVICE_INFO_CONFIG,
                    sb_alias_01, dc_st_array[i], dc_st_alias_array[i], 0x1
                    , PB_CONFIG, pb_alias_01, 0x1, dc_pt_array[i],
                    dc_pt_alias_array[i], 0x1);
134             if (new_lite01 == NULL) return -1;
                add_to_level_items_table(last_dnode->level_items_table, &
                    new_lite01);
136             retval = add_parameter_to_dnode(last_dnode,
                    dc_pt_alias_array[i], dc_valft_array[i], 0, (UInt8)
                    dc_vf_class_array[i], dc_vf_target_array[i], 1, 0);
                if (retval == -1) return -2;
138     }
        return 0;
140 }

142 int a64_configure_audio_inputs()
    {
144     int retval = 0;
        enum A64ValueFunctionClass vf_class = VFN_CLASS_CS4272_AUDIO;
146     struct DNode *last_dnode = get_last_dnode();
        for (int ifacenum = 0; ifacenum < 2; ifacenum++) {
148             for (int i = 0; i < 1; i++) {
                        struct LevelItemsTableEntry *new_lite01 =
                            create_level_items_table_entry(SB_INPUT_SIGNAL,
                            sb_alias_03, ST_AUDIO, st_alias_05, (ifacenum + 1)
                            , PB_DIGITAL_GAIN, pb_alias_03, 0x1, PT_LEVEL_MUTE
                            , pt_alias_17, 0x1);
150                     if (new_lite01 == NULL) return -1;
                        add_to_level_items_table(last_dnode->
                            level_items_table, &new_lite01);
152                     retval = add_parameter_to_dnode(last_dnode,
                            pt_alias_17, VALFT_INT_32, 0, (UInt8) vf_class,
                            CS4272_AUDIO_INPUT_LEVEL_MUTE, (ifacenum + 1), (i
                            + 1));
```

```
                    if (retval == -1) return -2;
154             }
        }
156     return 0;
}
158
int a64_configure_avb_listeners()
160 {
        int retval = 0;
162     enum A64ValueFunctionClass vf_class = VFN_CLASS_ETHERNET_AVB_SINK
            ;
        struct DNode *last_dnode = get_last_dnode();
164     UInt16 listener_pt_array[6] = {PT_STREAM_ID, PT_STREAM_LISTEN,
            PT_MULTICORE_IS_RECEIVING_STREAM, PT_SRC_MAC_ADDRESS,
            PT_MULTICORE_VLAN_ID, PT_MULTICORE_NUM_OF_CHANNELS };
        char *listener_pt_alias_array[6] = {pt_alias_04, pt_alias_06,
            pt_alias_08, pt_alias_09, pt_alias_13, pt_alias_15};
166     enum A64Valft listener_valft_array[6] = {VALFT_INT_64,
            VALFT_INT_32, VALFT_STRING, VALFT_INT_48, VALFT_INT_32,
            VALFT_INT_32};
        UInt8 listener_vf_target_array[6] = {AVB_SINK_STREAM_ID,
            AVB_SINK_LISTEN, AVB_SINK_STATE, AVB_SINK_MAAP_ADDRESS,
            AVB_SINK_VLAN_ID, AVB_SINK_NO_OF_CHANNELS};
168     for (int sinknum = 0; sinknum < AVB_NUM_SINKS; sinknum++) {
                for (int i = 0; i < 6; i++) {
170                     struct LevelItemsTableEntry *new_lite01 =
                            create_level_items_table_entry(
                            SB_INPUT_SIGNAL, sb_alias_03, ST_STREAM,
                            st_alias_04, 0x1, PB_MULTICORE, pb_alias_02, (
                            sinknum + 1), /* C counts from zero, AES64
                            counts from 1 */ listener_pt_array[i],
                            listener_pt_alias_array[i], 0x1);
                        if (new_lite01 == NULL) return -1;
172                     add_to_level_items_table(last_dnode->
                            level_items_table, &new_lite01);
                        retval = add_parameter_to_dnode(last_dnode,
                            listener_pt_alias_array[i],
                            listener_valft_array[i], 0, (UInt8) vf_class,
                            listener_vf_target_array[i], (sinknum + 1), 0)
                            ;
174                     if (retval == -1) return -2;
                }
176             for (int channel_num = 0; channel_num <
                    AVB_MAX_CHANNELS_PER_STREAM; channel_num++) {
```

```
                        struct LevelItemsTableEntry *new_lite01 =
                            create_level_items_table_entry(SB_INPUT_SIGNAL
                            , sb_alias_03, ST_STREAM, st_alias_04, 0x1,
                            PB_MULTICORE, pb_alias_02, (sinknum + 1),
                            PT_MULTICORE_CHANNEL_MAP, pt_alias_16, (
                            channel_num + 1));
178                     if (new_lite01 == NULL) return -1;
                        add_to_level_items_table(last_dnode->
                            level_items_table, &new_lite01);
180                     retval = add_parameter_to_dnode(last_dnode,
                            pt_alias_16, VALFT_INT_32, 0, (UInt8) vf_class
                            , AVB_SINK_MAP, (sinknum + 1), (channel_num +
                            1));
                        if (retval == -1) return -2;
182                 }
            }
184         return 0;
}
186

188 int a64_configure_audio_outputs()
{
190     int retval = 0;
    enum A64ValueFunctionClass vf_class = VFN_CLASS_CS4272_AUDIO;
192     struct DNode *last_dnode = get_last_dnode();
    UInt16 audio_output_pt_array[2] = { PT_LEVEL_MUTE, PT_OUT_LEVEL };
194     char *audio_output_pt_alias_array[2] = { pt_alias_18, pt_alias_17 };
    enum A64Valft audio_output_valft_array[2] = {VALFT_INT_32,
        VALFT_INT_32};
196     UInt8 audio_output_vf_target_array[2] = {
        CS4272_AUDIO_OUTPUT_LEVEL_MUTE, CS4272_AUDIO_OUTPUT_LEVEL_GAIN };
    for (int ifacenum = 0; ifacenum < 2; ifacenum++) {
198         for (int i = 0; i < 2; i++) {
            struct LevelItemsTableEntry *new_lite01 =
                create_level_items_table_entry(SB_OUTPUT_SIGNAL,
                sb_alias_03, ST_AUDIO, st_alias_05, (ifacenum + 1),
                PB_DIGITAL_GAIN, pb_alias_03, 0x1, audio_output_pt_array[i
                ], audio_output_pt_alias_array[i], 0x1);
200             if (new_lite01 == NULL) return -1;
            add_to_level_items_table(last_dnode->level_items_table, &
                new_lite01);
202             retval = add_parameter_to_dnode(last_dnode,
                audio_output_pt_alias_array[i], audio_output_valft_array[i
                ], 0, (UInt8) vf_class, audio_output_vf_target_array[i], (
```

```
                ifacenum + 1), (i + 1));
            if (retval == -1) return -2;
        }
    }
    return 0;
}

int a64_configure_avb_talkers()
{
    int retval = 0;
    enum A64ValueFunctionClass vf_class = VFN_CLASS_ETHERNET_AVB_SOURCE;
    struct DNode *last_dnode = get_last_dnode();
    UInt16 talker_pt_array[9] = {PT_STREAM_ID, PT_STREAM_ADVERTISE,
        PT_MULTICORE_IS_TRANSMITTING_STREAM, PT_DST_MAC_ADDRESS,
        PT_MULTICORE_STREAM_FORMAT,
        PT_MULTICORE_STREAM_FORMAT_SAMPLING_RATE, PT_MULTICORE_VLAN_ID,
        PT_MULTICORE_PRESENTATION_TIME_OFFSET,
        PT_MULTICORE_NUM_OF_CHANNELS};
    char *talker_pt_alias_array[9] = {pt_alias_04, pt_alias_05,
        pt_alias_07, pt_alias_10, pt_alias_11, pt_alias_12, pt_alias_13,
        pt_alias_14, pt_alias_15};
    enum A64Valft talker_valft_array[9] = { VALFT_INT_64, VALFT_INT_32,
        VALFT_STRING, VALFT_INT_48, VALFT_STRING, VALFT_INT_32,
        VALFT_INT_32, VALFT_INT_32, VALFT_INT_32};
    UInt8 talker_vf_target_array[9] = {AVB_SOURCE_STREAM_ID,
        AVB_SOURCE_ADVERTISE, AVB_SOURCE_STATE, AVB_SOURCE_MAAP_ADDRESS,
        AVB_SOURCE_FORMAT, AVB_SOURCE_FORMAT_SAMPLE_RATE,
        AVB_SOURCE_VLAN_ID, AVB_SOURCE_PRESENTATION_TIME_OFFSET,
        AVB_SOURCE_NO_OF_CHANNELS};
    for (int sourcenum = 0; sourcenum < AVB_NUM_SOURCES; sourcenum++) {
        /* First the standard complement of AVB Talker parameters */
        for (int i = 0; i < 9; i++) {
            struct LevelItemsTableEntry *new_lite01 =
                create_level_items_table_entry(SB_OUTPUT_SIGNAL,
                sb_alias_02, ST_STREAM, st_alias_04, 0x1, PB_MULTICORE,
                pb_alias_02, (sourcenum + 1), /* C counts from zero, AES64
                 counts from 1 */ talker_pt_array[i],
                talker_pt_alias_array[i], 0x1);
            if (new_lite01 == NULL) return -1;
            add_to_level_items_table(last_dnode->level_items_table, &
                new_lite01);
            retval = add_parameter_to_dnode(last_dnode,
                talker_pt_alias_array[i], talker_valft_array[i], 0, (UInt8
                ) vf_class, talker_vf_target_array[i], (sourcenum + 1), 0)
```

```
                ;
            if (retval == -1) return -2;
        }
        /* Now the channel map parameters */
        for (int channel_num = 0; channel_num <
            AVB_MAX_CHANNELS_PER_STREAM; channel_num++) {
            struct LevelItemsTableEntry *new_lite01 =
                create_level_items_table_entry( SB_OUTPUT_SIGNAL,
                sb_alias_02, ST_STREAM, st_alias_04, 0x1, PB_MULTICORE,
                pb_alias_02, (sourcenum + 1), PT_MULTICORE_CHANNEL_MAP,
                pt_alias_16, (channel_num + 1));
            if (new_lite01 == NULL) return -1;
            add_to_level_items_table(last_dnode->level_items_table, &
                new_lite01);
            retval = add_parameter_to_dnode(last_dnode, pt_alias_16,
                VALFT_INT_32, 0, (UInt8) vf_class, AVB_SOURCE_MAP, (
                sourcenum + 1), (channel_num + 1));
            if (retval == -1) return -2;
        }
    }
    return 0;
}

int a64_configure_avb_media_clocks()
{
    int retval = 0;
    enum A64ValueFunctionClass vf_class =
        VFN_CLASS_ETHERNET_AVB_MEDIA_CLOCK;
    struct DNode *last_dnode = get_last_dnode();
    UInt16 media_clock_pt_array[4] = {PT_CLOCK_TYPE, PT_CLOCK_SAMPLE_RATE
        , PT_CLOCK_SOURCE, PT_CLOCK_STATE};
    char *media_clock_pt_alias_array[4] = {pt_alias_20, pt_alias_21,
        pt_alias_22, pt_alias_23};
    UInt8 media_clock_vf_target_array[4] = {MEDIA_CLOCK_TYPE,
        MEDIA_CLOCK_SAMPLE_RATE, MEDIA_CLOCK_SOURCE, MEDIA_CLOCK_STATE};
    for (int clock_num = 0; clock_num < AVB_NUM_MEDIA_CLOCKS; clock_num
        ++) {
        for (int i = 0; i < 4; i++) {
            struct LevelItemsTableEntry *new_lite01 =
                create_level_items_table_entry(
                SB_CLOCK_TIMING_SYNCHRONIZATION, sb_alias_04, ST_MEDIA,
                st_alias_06, 0x1, PB_CLOCK_AND_SYNCHRONIZATION,
                pb_alias_04, (clock_num + 1), media_clock_pt_array[i],
                media_clock_pt_alias_array[i], 0x1);
```

```
250             if (new_lite01 == NULL) return -1;
                add_to_level_items_table(last_dnode->level_items_table, &
                    new_lite01);
252           retval = add_parameter_to_dnode(last_dnode,
                  media_clock_pt_alias_array[i], VALFT_INT_32, 0, (UInt8)
                  vf_class, media_clock_vf_target_array[i], (clock_num + 1),
                  0);
              if (retval == -1) return -2;
254         }
      }
256   return 0;
}
```

E The AES-64 utilities

E.1 Introduction

This appendix documents the connection management and control utilities implemented to test and demonstrate the capabilities of the AES-64-enabled XMOS Ethernet AVB streaming audio device.

Each utility is described in terms of its purpose, usage, and the AES-64 messaging that it carries out.

a64_discover provides device discovery and rudimentary device enumeration;

a64_connect performs stream connection between a nominated Talker and a nominated Listener;

a64_disconnect instructs a Talker to withdraw its Talker Advertise or a Listener to withdraw its Listener Ready, in each case ending a stream connection;

a64_query queries the status of the Talker and Listener components on the device;

a64_channel sets the mapping between a stream multicore's channels and a Talker's audio inputs or a Listener's audio outputs;

a64_join_setup_1 sets up a proof-of-concept parameter group between two parameters on one device, and **a64_join_teardown_1** tears down the parameter group;

a64_join_setup_2 sets up a proof-of-concept parameter group between two parameters on two different devices, and **a64_join_teardown_2** tears down the parameter group;

a64_get_mstgrp queries the 'master' group list of a parameter on the device to verify the operation of the join/unjoin implementation;

a64_get_slvgrp queries the 'slave' group list of a parameter on the device to verify the operation of the join/unjoin implementation;

a64_mute mutes an audio input or output channel on the device;

a64_volume sets the level of an audio output channel on the device.

a64_level_check gets the level of the audio output channels on the device.

a64_set_clock_type sets the media clock server clock type.

a64_set_clock_rate sets the sampling rate generated by the media clock server.

a64_set_clock_state disables or enables the media clock server's clock signal.

a64_set_clock_source sets the source channel used by the media clock server to generate one type of clock signal.

E.2 Utility documentation

Messaging diagrams. Most of the utilities described here implement straightforward AES-64 messaging to issue commands or queries (Fig. E.1). Where a utility's messaging is more complicated, an illustration is provided.

E.2.1 a64_discover

`a64_discover` (no arguments)
Discovers and partially enumerates any AES-64-capable devices on the network.

Messaging. This utility applies the discovery and enumeration procedures outlined in Subsections A.9.1 and A.9.2. After broadcasting a 'GET VAL' targeting the configuration device node's 'IP Address' parameter, it collects the network's responses and interrogates each responding device in turn, partially enumerating each by a series of 'GET CLA' requests and a final 'GET VAL' request targeting the configuration device node's 'Device Name' parameter (Fig. E.2). In addition to printing the table of discovered devices onscreen, it dumps a comma-separated-variable file that can be parsed by other utilities.

E.2.2 a64_connect

`a64_connect [Talker IP address] [Listener IP address]`
Sets up a stream connection between nominated Talker and Listener devices.

Messaging. See Fig. E.3.

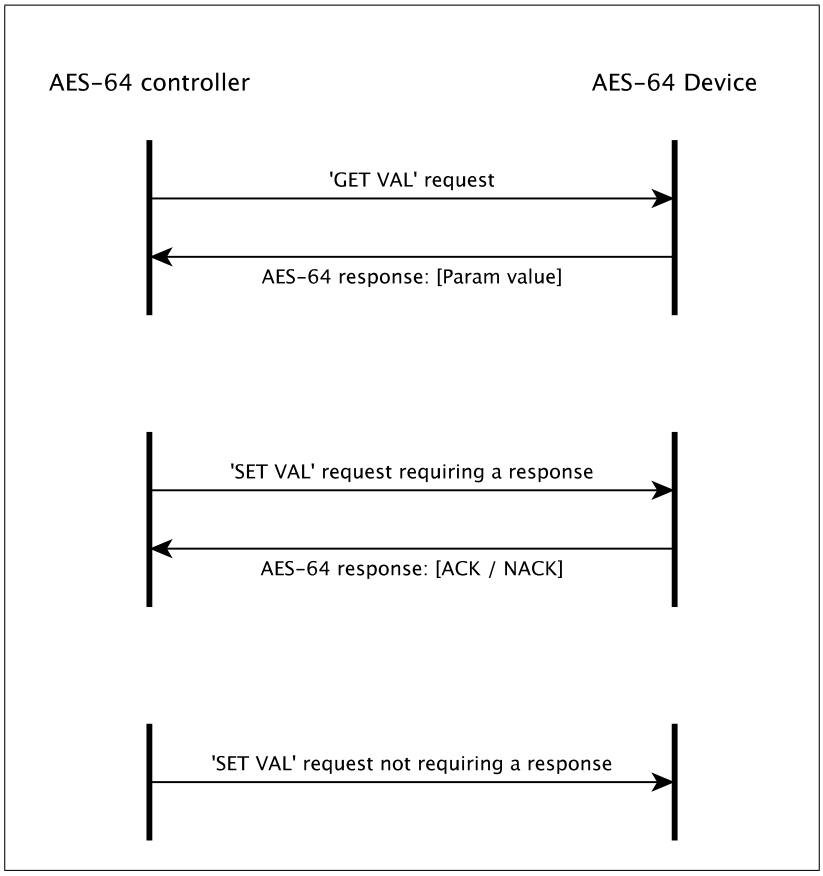


Figure E.1: Generic AES-64 messaging patterns: (a) 'Get Val' request;
(b) 'Set Val' request requiring a response;
(c) 'Set Val' request not requiring a response

E.2.3 a64_disconnect

`a64_disconnect [t=Talker | l=Listener] [IP address]`

Disconnects a stream, either by deactivating the stream Talker or a stream Listener.

Messaging. Sends a 'SET VAL' message with the value 'false' to either:

```
OUTPUT SIGNAL  STREAM  1   MULTICORE  1   ADVERTISE  1
```

or:

```
INPUT SIGNAL   STREAM  1   MULTICORE  1   LISTEN     1
```

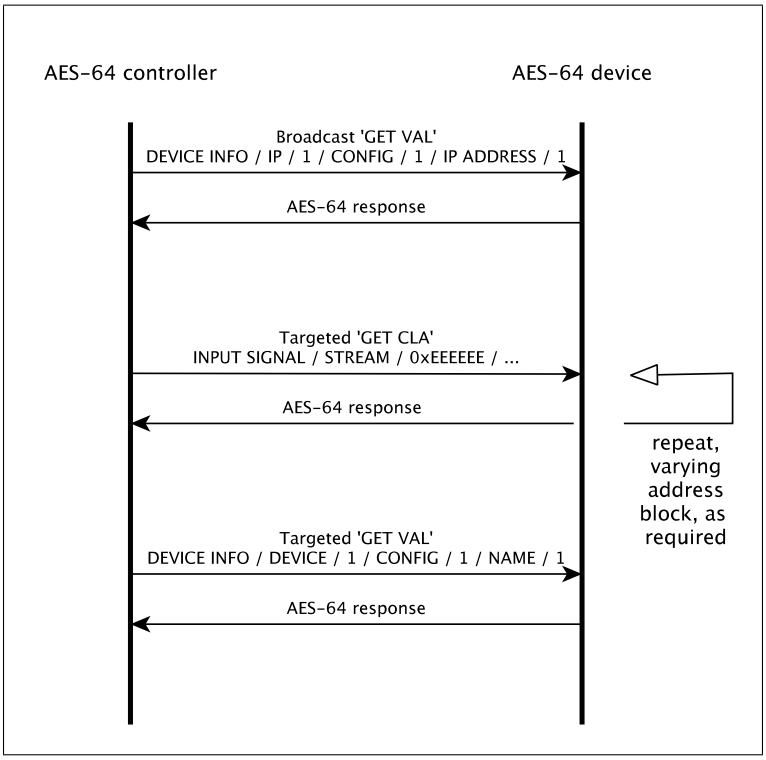


Figure E.2: AES-64 messaging for the a64_discover utility.

E.2.4 a64_query

`a64_query [IP address]`

Queries the status of the Talker and Listener units on a device.

Messaging. Sends a 'GET VAL' message to each of the following parameters:

```
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   STREAM ID         1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   DST MAC ADDRESS   1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   MULTICORE VLAN ID 1
OUTPUT SIGNAL   STREAM  1   MULTICORE  1   ADVERTISE         1
INPUT SIGNAL    STREAM  1   MULTICORE  1   STREAM ID         1
INPUT SIGNAL    STREAM  1   MULTICORE  1   SRC MAC ADDRESS   1
INPUT SIGNAL    STREAM  1   MULTICORE  1   MULTICORE VLAN ID 1
INPUT SIGNAL    STREAM  1   MULTICORE  1   LISTEN            1
```

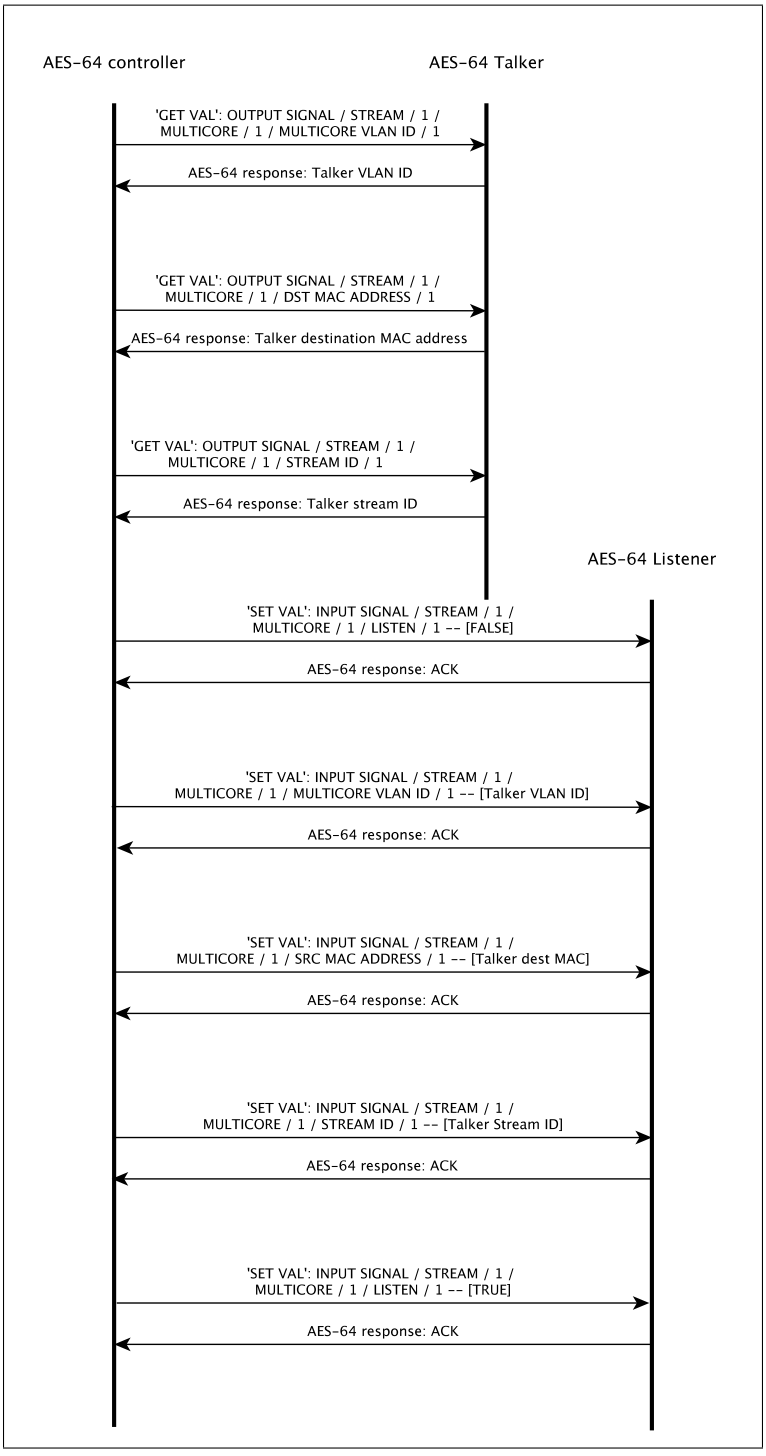


Figure E.3: AES-64 messaging for the a64_connect utility.

E.2.5 a64_channel

```
a64_channel [IP address] [t=Talker | l=Listener]
            [Multicore stream channel no.] [Media FIFO no.]
```

Sets a mapping between a multicore stream channel and a media FIFO (e.g., an audio input or output) within either the Talker or Listener unit of a device.

Messaging. Sends a 'SET VAL' request to either (for a Talker):

```
OUTPUT SIGNAL STREAM 1 MULTICORE 1 CHANNEL MAP [stream channel #]
```

or (for a Listener):

```
INPUT SIGNAL STREAM 1 MULTICORE 1 CHANNEL MAP [stream channel #]
```

The value field of the request contains the Media FIFO number.

E.2.6 a64_join_setup_1

```
a64_join_setup_1 [IP address]
```

Sets up a proof-of-concept parameter group between two parameters on the same device.

Messaging. Sends a 'JOIN MSTSLV' request requiring a response to:

```
OUTPUT SIGNAL AUDIO 1 DIGITAL GAIN 1 LEVEL MUTE 1
```

with a value field that nominates this parameter *on the same device* as a slave:

```
OUTPUT SIGNAL AUDIO 2 DIGITAL GAIN 1 LEVEL MUTE 1
```

The master parameter on device #1 must then interrogate the slave parameter to discover its peer affiliations before sending the 'SET MASTER' request (and, finally, an acknowledgment of the original 'JOIN MSTSLV' request, back to the controller). If the slave parameter has any peers, its peers will also become slaves to the master parameter (Fig. E.4).

The AES-64 response confirms that the join is established. Once this join is established, enabling the channel mute on output channel #1 will simultaneously mute output channel #2 as well.

`a64_join_teardown_1` tears down this join. The messaging pattern is unchanged, except that the 'JOIN MSTSLV' request becomes an 'UNJOIN MSTSLV' request and the 'SET MASTER' requests become 'SET MASTER OFF' requests.

E.2.7 a64_join_setup_2

`a64_join_setup_2 [Device #1 IP address] [Device #2 IP address]`
Sets up a proof-of-concept parameter group between two parameters on two different devices.

Messaging. Sends a 'JOIN MSTSLV' request requiring a response to this parameter on Device #1:

```
OUTPUT SIGNAL AUDIO  1  DIGITAL GAIN 1  LEVEL MUTE  1
```

with a value field that nominates this parameter on Device #2 as a slave:

```
OUTPUT SIGNAL AUDIO  1  DIGITAL GAIN 1  LEVEL MUTE  1
```

The parameter on device #1 must then interrogate the parameter on Device #2 to discover its peer affiliations before sending the 'SET MASTER' request (and, finally, an acknowledgment of the original 'JOIN MSTSLV' request, back to the controller). If the parameter on Device #2 has any peers, its peers will also become slaves to the parameter on Device #1 (Fig. E.4).

The AES-64 response confirms that the join is established. Once this join is established, enabling the channel mute on Device #1's output channel #1 will simultaneously mute output channel #1 on Device #2. `a64_join_teardown_2` tears down the join. The messaging pattern is unchanged, except that the 'JOIN MSTSLV' request becomes an 'UNJOIN MSTSLV' request and the 'SET MASTER' requests become 'SET MASTER OFF' requests.

E.2.8 a64_get_mstgrp

`a64_get_mstgrp [IP address]`
Verifies parameter group operations by returning the target parameter's Master group list.

Messaging. Sends a 'GET MSTGRP' request to either[1] `a64_join_setup_1`:

```
OUTPUT SIGNAL AUDIO  2  DIGITAL GAIN 1  LEVEL MUTE  1
```

or (for testing `a64_join_setup_2`, where the request must be sent to Device #2):

```
OUTPUT SIGNAL AUDIO  1  DIGITAL GAIN 1  LEVEL MUTE  1
```

[1] This is not yet implemented as a command-line option - the binary needs to be recompiled to change the request address block.

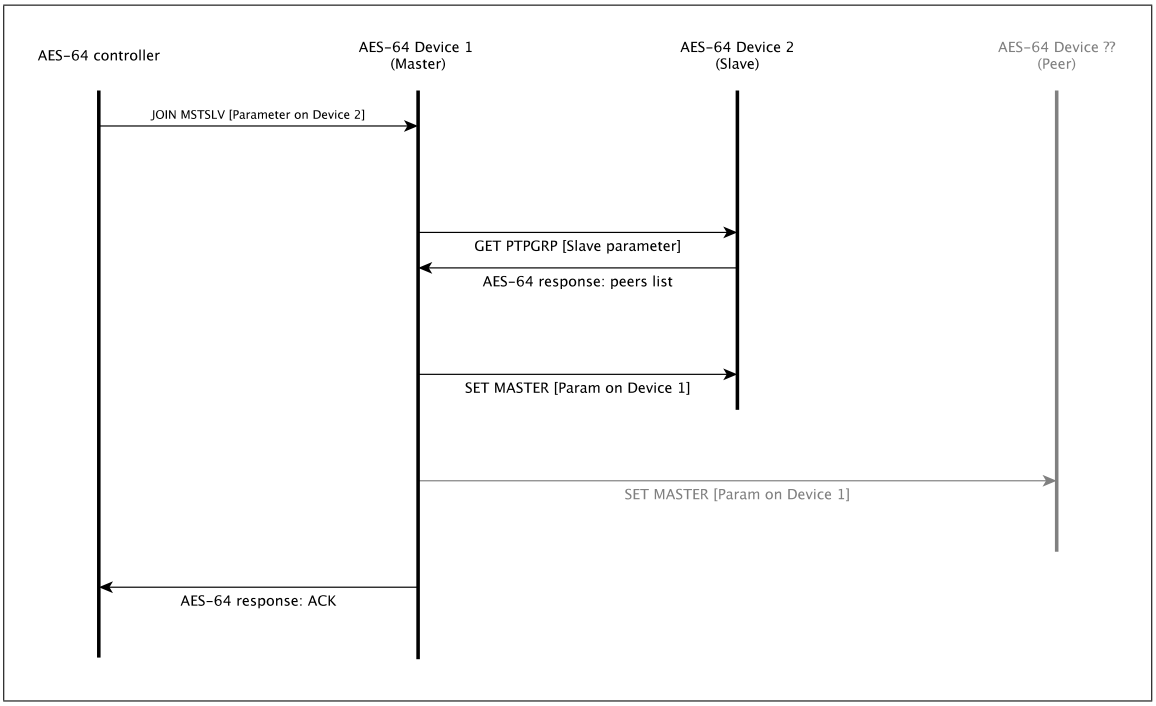


Figure E.4: AES-64 messaging for set up of a master-slave join. This diagram applies to both join setup utilities, although when a join is established by a64_join_setup_1, the slave parameter will be on the same device as the master parameter.

E.2.9 a64_get_slvgrp

`a64_get_slvgrp [IP address]`

Verifies parameter group operations by returning the target parameter's Slave group list.

Messaging. Sends a 'GET SLVGRP' request to:

```
OUTPUT SIGNAL AUDIO  1  DIGITAL GAIN 1  LEVEL MUTE  1
```

E.2.10 a64_mute

```
a64_mute [IP address] [i=Input | o=Output] [Audio port #]
         [e=Enable | d=Disable]
```

Mutes audio input or output channels on a device.

Messaging. Sends a 'SET VAL' request to one of these addresses on the target device:

```
OUTPUT SIGNAL AUDIO  [port #]  DIGITAL GAIN 1  LEVEL MUTE  1
INPUT SIGNAL  AUDIO  [port #]  DIGITAL GAIN 1  LEVEL MUTE  1
```

The value field of the request is 'True' or 'False' as per the enable / disable switch.

E.2.11 a64_volume

`a64_volume [IP address] [audio port \#]`
Changes the volume level on an audio output channel[2] of a device.

Messaging. Sends a 'SET VAL' request to one of these addresses on the target device:

```
OUTPUT SIGNAL AUDIO [port #] DIGITAL GAIN 1 OUT LEVEL 1
```

E.2.12 a64_level_check

`a64_level_check [IP address]`
Queries the audio output level on both channels of a device.

Messaging. Sends a 'GET VAL' request to these addresses on the target device:

```
OUTPUT SIGNAL AUDIO 1 DIGITAL GAIN 1 OUT LEVEL 1
OUTPUT SIGNAL AUDIO 2 DIGITAL GAIN 1 OUT LEVEL 1
```

E.2.13 a64_set_clock_type

`a64_set_clock_type [IP address] [Media clock #] [p=PTP | s=Stream ]`
Controls whether a media clock server unit generates a clock that is derived from the local IEEE 802.1AS server, or derived from an inbound IEEE 1722 stream. There is also a complementary `a64_get_clock_type` utility.

Messaging. Sends a 'SET VAL' request to this address on the target device:

```
CLOCK TIMING SYNC MEDIA 1 CLOCK AND SYNC [clock unit #] CLOCK TYPE 1
```

E.2.14 a64_set_clock_rate

`a64_set_clock_rate [IP address] [Media clock #] [Sampling rate in Hz]`
Sets the sampling rate signal generated by a media clock server unit. Sampling rate is specified in Hz. There is also a complementary `a64_get_clock_rate` utility.

[2]The CS4272 does not implement volume control on its input channels: there is merely a switchable 'mute', i.e., on/off.

Messaging. Sends a 'SET VAL' request to this address on the target device:

```
CLOCK TIMING SYNC MEDIA  1  CLOCK AND SYNC [clock unit #] CLOCK RATE  1
```

E.2.15 a64_set_clock_state

`a64_set_clock_state [IP address] [Media clock #] [e=enable | d=disable ]`
Enables or disables a media clock server unit. There is a complementary `a64_get_clock_state` utility.

Messaging. Sends a 'SET VAL' request to this address on the target device:

```
CLOCK TIMING SYNC MEDIA  1  CLOCK AND SYNC [clock unit #] CLOCK STATE  1
```

E.2.16 a64_set_clock_source

`a64_set_clock_source [IP address] [Media clock #] [e=enable | d=disable ]`
Selects a input channel source for a media clock server unit when its clock type is set to STREAM (Subsection E.2.13. There is a complementary `a64_get_clock_source` utility.

Messaging. Sends a 'SET VAL' request to this address on the target device:

```
CLOCK TIMING SYNC MEDIA  1  CLOCK AND SYNC [clock unit #] CLOCK SOURCE  1
```

F Latency timings for the AES-64 implementation

F.1 Introduction

This appendix presents the compiled timing measurements used in the quantitative evaluation of the AES-64 implementation (Section 7.3).

F.2 Timing measurements

As described in Section 7.3, these measurements were taken in ten blocks of five measurement cycles. Each block of measurement cycles was captured to a separate WireShark packet dump. The measurement methodology for each cycle is described again here:

1. Power on both Ethernet AVB devices;
2. Using the `a64_discover` utility, discover and enumerate both devices;
3. Using the `a64_connect` utility, set up an Ethernet AVB stream between the two DSP4YOU devices;
4. Using the `a64_channel` utility, set a multicore-channel-to-media-buffer mapping on the Listener;
5. Using the `a64_disconnect` utility, tear down the Ethernet AVB stream by detaching the Listener;
6. Power off both Ethernet AVB devices.

		Discovery		Get Talker VLAN ID		Get Talker MAC addr		Get Talker Stream ID		Set Listener VLAN ID		Set Listener MAC addr	
		Treq	dT	Trsp	dT	Trsp	dT	Trsp	dT	Trsp	dT	Trsp	dT
Block 1	1	0	0.000403	41.612567	0.000261	41.662982	0.000255	41.713496	0.000413	42.118216	0.000264	42.168457	0.000418
	2	150.946997	150.947426	156.040127	0.000287	156.09062	0.000238	156.141041	0.000405	156.553354	0.000263	156.603597	0.000403
	3	226.509377	226.509808	229.881108	0.000347	229.931648	0.000259	229.982206	0.000388	230.387172	0.000242	230.437267	0.000246
	4	320.373643	320.374163	324.435687	0.000363	324.486235	0.000254	324.536728	0.000375	324.941534	0.000253	324.991813	0.000419
	5	452.771276	452.77172	455.924364	0.000302	455.974851	0.00026	456.025404	0.000424	456.430378	0.000258	456.480619	0.000423
Block 2	1	0	0.000392	2.9607	0.000288	3.010994	0.000257	3.061379	0.000381	3.46641	0.000258	3.516653	0.000396
	2	65.322773	65.323259	68.761736	0.000309	68.81224	0.000254	68.862757	0.000382	69.267634	0.000259	69.317874	0.000397
	3	119.441364	119.441774	122.355515	0.000266	122.406062	0.00027	122.456621	0.000334	122.865992	0.000256	122.916107	0.000299
	4	209.276508	209.276967	211.763858	0.000364	211.814398	0.000278	211.864973	0.000411	212.269848	0.000254	212.320091	0.00044
	5	280.427454	280.427653	284.505763	0.00036	284.556318	0.000255	284.606692	0.000269	285.010643	0.000243	285.060865	0.000386
Block 3	1	0	0.000395	3.126834	0.000352	3.177372	0.000244	3.227885	0.000391	3.638675	0.000255	3.688867	0.000378
	2	143.40262	143.403081	145.961117	0.000372	146.011684	0.000261	146.062223	0.000394	146.466197	0.000271	146.516387	0.000338
	3	219.590529	219.591018	223.924976	0.000338	223.975501	0.000265	224.026029	0.000394	224.437719	0.000508	224.487847	0.000258
	4	273.1783	273.178705	276.135743	0.000359	276.186287	0.000251	276.23682	0.000411	276.64848	0.000255	276.698705	0.000409
	5	419.845611	419.846004	422.877197	0.000287	422.927676	0.000262	422.978135	0.000379	423.382774	0.000263	423.433011	0.000395
Block 4	1	0	0.000391	2.707348	0.000288	2.757833	0.000258	2.808401	0.000391	3.215895	0.000248	3.266099	0.00028
	2	55.954736	55.955522	58.313941	0.000297	58.364141	0.000254	58.414672	0.000422	58.819756	0.000261	58.869963	0.000369
	3	137.980981	137.981396	140.425283	0.000312	140.47579	0.000257	140.52628	0.000417	140.931428	0.000261	140.981507	0.00041
	4	228.777799	228.778272	231.473738	0.000288	231.523924	0.000255	231.574435	0.000383	231.979086	0.000274	232.029145	0.000331
	5	323.24714	323.247532	326.187004	0.00031	326.237485	0.000247	326.287975	0.000253	326.694916	0.000238	326.745149	0.000381
Block 5	1	0	0.000219	4.356728	0.000323	4.407841	0.000251	4.458234	0.000276	4.879678	0.000268	4.929918	0.000391
	2	76.477214	76.477606	79.556565	0.000362	79.607105	0.000275	79.657614	0.000372	80.062581	0.000279	80.11259	0.000406
	3	138.971402	138.971816	142.094699	0.000343	142.145172	0.000246	142.195603	0.000399	142.600229	0.000243	142.650447	0.000384
	4	200.550785	200.551273	217.119732	0.000294	217.170238	0.000269	217.220766	0.000387	217.6281	0.000252	217.678337	0.000378
	5	297.944315	297.944721	310.263188	0.000372	310.31376	0.000257	310.364312	0.000412	310.768951	0.00026	310.819152	0.000366
Block 6	1	0	0.000397	2.736934	0.000289	2.788232	0.000253	2.840712	0.000293	3.245102	0.000249	3.295346	0.000389
	2	61.219443	61.219867	63.586272	0.000282	63.636744	0.000259	63.687277	0.000382	64.095242	0.000267	64.145247	0.000356
	3	127.762177	127.762596	130.467057	0.000282	130.517563	0.000248	130.568099	0.000388	130.972531	0.000262	131.022771	0.000388
	4	204.200226	204.200604	206.151214	0.00026	206.201633	0.000258	206.252198	0.00039	206.661129	0.000252	206.71123	0.00026
	5	288.183011	288.1834	290.96353	0.000339	291.014032	0.000248	291.064394	0.000394	291.46936	0.000256	291.51957	0.000296
Block 7	1	0	0.00045	2.626869	0.000371	2.677424	0.000266	2.727966	0.000419	3.132725	0.000247	3.182964	0.000393
	2	61.597964	61.598437	64.188708	0.000365	64.239266	0.000275	64.289605	0.000273	64.694383	0.000265	64.744621	0.000418
	3	110.162126	110.162589	114.919237	0.000304	114.969729	0.000265	115.020234	0.000373	115.427624	0.000269	115.477864	0.000407
	4	184.119814	184.120292	186.494587	0.000304	186.545077	0.00026	186.595494	0.000318	187.000637	0.000265	187.050847	0.000406
	5	257.649933	257.650352	259.929313	0.000283	259.97981	0.000253	260.030345	0.000386	260.434928	0.00025	260.485153	0.000375
Block 8	1	0	0.000473	2.839422	0.000331	2.889938	0.000253	2.940089	0.000261	3.344915	0.000245	3.395063	0.000354
	2	67.230572	67.231037	70.213187	0.000354	70.263722	0.000259	70.314205	0.000421	70.719394	0.000262	70.769635	0.000408
	3	123.863996	123.864386	126.792287	0.000278	126.842726	0.000266	126.892983	0.000221	127.297286	0.000245	127.347488	0.000339
	4	175.89446	175.894914	178.468109	0.000377	178.518683	0.00026	178.569223	0.000409	178.973436	0.000262	179.023664	0.000409
	5	256.467336	256.46782	259.721727	0.000357	259.772282	0.000257	259.822654	0.000299	260.227184	0.000252	260.277384	0.000401
Block 9	1	0	0.000462	4.075373	0.000374	4.12593	0.000267	4.176476	0.000417	4.58157	0.000271	4.631756	0.000303
	2	132.802568	132.802979	145.164253	0.000361	145.214439	0.000274	145.264931	0.000326	145.675858	0.000271	145.72609	0.000405
	3	218.442601	218.443079	221.305269	0.000353	221.355622	0.000271	221.406038	0.00033	221.810225	0.000272	221.860438	0.000345
	4	295.497588	295.497993	297.879671	0.000373	297.930223	0.000271	297.980704	0.000374	298.385177	0.000266	298.435398	0.000351
	5	389.170316	389.17072	391.761317	0.000362	391.811882	0.000266	391.862337	0.000279	392.266988	0.000252	392.317083	0.000412
Block 10	1	0	0.000469	2.636496	0.000343	2.687029	0.000258	2.737561	0.000393	3.14268	0.000273	3.192735	0.000333
	2	74.753415	74.753838	78.055629	0.000322	78.106029	0.000246	78.156344	0.000397	78.560517	0.000249	78.610744	0.000377
	3	139.212397	139.212784	141.407808	0.00028	141.457963	0.000247	141.508379	0.000256	141.911414	0.000243	141.961542	0.000264
	4	189.566757	189.567216	192.597476	0.000279	192.647911	0.000252	192.698322	0.000285	193.101647	0.00024	193.152779	0.000261
	5	263.288166	263.288557	265.359499	0.000271	265.411902	0.000255	265.462305	0.000276	265.865859	0.000257	265.915978	0.000245
		Mean	0.0004247	Mean	0.00032276	Mean	0.00025838	Mean	0.00036046	Mean	0.00026256	Mean	0.0003635
		Median	0.000419	Median	0.0003225	Median	0.0002575	Median	0.000382	Median	0.000521	Median	0.0003825

		Set Listener Stream ID			Set Listener State			Set Listener Channel			Set Listener State (Detach)		
		Treq	Trsp	dT	Treq	Trsp	dT	Treq	Trsp	dT	Treq	Trsp	dT
Block 1	1	42.218559	42.21881	0.000251	42.268729	42.269002	0.000273	73.810933	73.811308	0.000375	77.313552	77.313946	0.000394
	2	156.653821	156.654149	0.000328	156.704176	156.704502	0.000326	165.765041	165.765423	0.000382	169.267528	169.267783	0.000255
	3	230.487352	230.487604	0.000252	230.53757	230.537877	0.000307	237.030478	237.030913	0.000435	239.7897	239.789964	0.000264
	4	325.042008	325.042346	0.000338	325.09227	325.092601	0.000331	330.897081	330.897475	0.000394	334.168063	334.168501	0.000438
	5	456.530803	456.531102	0.000299	456.581117	456.581427	0.00031	465.329092	465.331915	0.002823	469.087475	469.08786	0.000385
Block 2	1	3.566806	3.567078	0.000272	3.617093	3.617423	0.00033	9.75845	9.758811	0.000361	12.260756	12.26106	0.000304
	2	69.367995	69.368238	0.000243	69.418341	69.418657	0.000316	75.471136	75.471559	0.000423	78.062095	78.062428	0.000333
	3	122.96611	122.966419	0.000309	123.016332	123.016661	0.000329	127.670214	127.672897	0.002683	130.068983	130.069356	0.000373
	4	212.370316	212.370648	0.000332	212.420631	212.42095	0.000319	227.82961	227.82999	0.00038	230.564494	230.564841	0.000347
	5	285.11097	285.111231	0.000261	285.161412	285.161744	0.000332	293.934044	293.936722	0.002678	297.860819	297.861125	0.000306
Block 3	1	3.738982	3.73926	0.000278	3.78926	3.78954	0.00028	8.340883	8.341572	0.00689	10.7716	10.772011	0.000411
	2	146.566646	146.56696	0.000314	146.617205	146.617488	0.000283	151.143371	151.143801	0.00043	156.277579	156.277986	0.000407
	3	224.537958	224.538222	0.000264	224.58831	224.588644	0.000334	229.634687	229.640099	0.005412	232.681558	232.681884	0.000326
	4	276.748864	276.749163	0.000299	276.799185	276.799517	0.000332	281.181807	281.185226	0.003419	283.579297	283.579647	0.00035
	5	423.484381	423.484643	0.000262	423.534705	423.534982	0.000277	426.763874	426.764468	0.000594	433.161397	433.161816	0.000419
Block 4	1	3.316315	3.316615	0.0003	3.36666	3.366991	0.000331	7.345428	7.345773	0.000345	10.87208	10.872391	0.000311
	2	58.920076	58.920359	0.000283	58.970289	58.970616	0.000327	66.518377	66.521882	0.003505	70.011571	70.011881	0.00031
	3	141.031493	141.031814	0.000321	141.081844	141.082177	0.000333	151.349056	151.349943	0.000887	153.828144	153.828562	0.000418
	4	232.079402	232.079736	0.000334	232.129729	232.130038	0.000309	241.117912	241.118372	0.00046	245.56418	245.564604	0.000424
	5	326.795374	326.795675	0.000301	326.845788	326.846143	0.000355	330.122939	330.123273	0.000334	332.778093	332.7785	0.000407
Block 5	1	4.98004	4.98029	0.00025	5.03038	5.030677	0.000297	9.842991	9.843419	0.000428	14.817067	14.817415	0.000348
	2	80.162628	80.162936	0.000308	80.212957	80.213284	0.000327	89.248794	89.249264	0.00047	94.055251	94.055561	0.00031
	3	142.700567	142.700837	0.00027	142.750881	142.75117	0.000289	149.245212	149.245613	0.000401	152.498708	152.499085	0.000377
	4	217.728573	217.72887	0.000297	217.778945	217.779279	0.000334	222.333837	222.334225	0.000388	238.286043	238.286448	0.000405
	5	310.869251	310.869513	0.000262	310.919546	310.919879	0.000333	317.420534	317.424494	0.00396	322.690641	322.690988	0.000347
Block 6	1	3.345608	3.345946	0.000338	3.395905	3.396237	0.000332	7.015809	7.019564	0.003755	10.758111	10.758448	0.000337
	2	64.195294	64.195671	0.000377	64.245554	64.245887	0.000333	66.753345	66.753713	0.000368	69.240379	69.240748	0.000369
	3	131.073033	131.073346	0.000313	131.123118	131.123426	0.000308	133.850096	133.850535	0.000439	136.848984	136.849328	0.000344
	4	206.761328	206.761592	0.000264	206.811739	206.812064	0.000325	209.237995	209.23837	0.000375	211.973261	211.973716	0.000455
	5	291.569816	291.57015	0.000334	291.620189	291.620523	0.000334	303.687866	303.688278	0.000412	309.981621	309.982043	0.000422
Block 7	1	3.23307	3.23333	0.00026	3.283321	3.283621	0.0003	11.359543	11.359931	0.000388	14.94233	14.942636	0.000306
	2	64.794797	64.7951	0.000303	64.845185	64.845518	0.000333	68.003287	68.003717	0.00043	70.802026	70.802422	0.000396
	3	115.528144	115.528473	0.000329	115.578467	115.578791	0.000324	123.32057	123.320883	0.000313	126.179275	126.17963	0.000355
	4	187.101017	187.101313	0.000296	187.151376	187.151717	0.000341	191.90083	191.901261	0.000431	198.602052	198.602373	0.000321
	5	260.535261	260.535516	0.000255	260.585566	260.585864	0.000298	265.799183	265.799623	0.00044	270.005427	270.005731	0.000304
Block 8	1	3.44517	3.445434	0.000264	3.495471	3.495773	0.000302	6.621296	6.621681	0.000385	10.371568	10.37198	0.000412
	2	70.819779	70.820088	0.000309	70.870079	70.870414	0.000335	74.291637	74.292032	0.000395	76.250932	76.251283	0.000351
	3	127.397671	127.397988	0.000317	127.447948	127.448228	0.00028	134.867256	134.87059	0.003334	137.540168	137.540523	0.000355
	4	179.073762	179.074028	0.000266	179.124106	179.124444	0.000338	184.402666	184.403033	0.000367	189.403059	189.40333	0.000271
	5	260.327477	260.327738	0.000261	260.377775	260.378094	0.000319	264.543832	264.54445	0.000618	268.014426	268.014849	0.000423
Block 9	1	4.682003	4.682353	0.00035	4.732301	4.732644	0.000343	10.320912	10.321441	0.000529	18.685768	18.686196	0.000428
	2	145.776255	145.776555	0.0003	145.826577	145.826904	0.000327	151.129688	151.130064	0.000376	153.608375	153.60869	0.000315
	3	221.910679	221.910999	0.00032	221.961004	221.961323	0.000319	226.183281	226.183696	0.000415	229.565972	229.566335	0.000363
	4	298.487335	298.487589	0.000254	298.537584	298.537851	0.000267	304.764811	304.765246	0.000435	307.228154	307.228527	0.000373
	5	392.367325	392.367654	0.000329	392.417637	392.417979	0.000342	398.221463	398.225574	0.004111	402.461455	402.46171	0.000255
Block 10	1	3.24295	3.243241	0.000291	3.293192	3.293508	0.000316	6.65906	6.65945	0.00039	10.057776	10.058098	0.000322
	2	78.660891	78.66116	0.000269	78.7112	78.711502	0.000302	82.933701	82.934028	0.000327	86.076562	86.076964	0.000402
	3	142.011702	142.011967	0.000265	142.061798	142.062061	0.000263	145.100918	145.101225	0.000307	148.540676	148.540969	0.000293
	4	193.202862	193.203114	0.000252	193.253002	193.253308	0.000306	198.867242	198.869459	0.002217	201.554091	201.554404	0.000313
	5	265.966081	265.96634	0.000259	266.016197	266.016471	0.000274	278.234309	278.23471	0.000401	288.342534	288.342828	0.000294
			Mean	0.00029206			0.0003155		Mean	0.00109228			0.00035496
			Median	0.000297			0.000325		Median	0.00043		Median	0.000351
				Mean time for stream connection setup			0.0021752						
				Median time for stream connection setup			0.0024875						

Bibliography

AES Standards Committee, "AES standards project report: Use cases for networks in professional audio," 2008.

Attero Tech. (2010, November) XMOS - AVB demo board schematics. [Online]. Available: http://www.xmos.com/node/14454?version=latest

Audio Engineering Society, "AES-3 standard for digital audio - digital input-output interfacing - serial transmission format for two-channel linearly represented digital audio data," 2003.

Audio Engineering Society, "Draft AES informative document for the standard of audio applications of networks - integrated control, monitoring and connection management for digital audio and other media networks," March 2011.

Audio Engineering Society, "AES-64 standard for audio applications of networks - command, control and connection management for integrated media," 2012.

R. Boatright. (2009) Understanding IEEE's new audio video bridging standards. [Online]. Available: http://eetimes.com/General/PrintView/4008284

Cirrus Logic. (2005, August) CS4272 24-bit, 192 kHz stereo audio codec. [Online]. Available: http://www.cirrus.com/en/pubs/proDatasheet/CS4272_F1.pdf

DSP4YOU. (2012, February) AVB streamer eval kit datasheet. [Online]. Available: http://www.dsp4you.com/downloads/AVB%20Streamer%20Eval%20Kit.pdf

C. Fencott, *Formal methods for concurrency.* Thomson Computer Press, 1995.

P. Foulkes, "An investigation into the control of audio streaming across networks having diverse quality of service mechanisms," Ph.D. dissertation, Rhodes University, September 2011.

H. Gomaa, "A software design method for real-time systems," *Communications of the ACM*, vol. 27, no. 9, pp. 938–949, 1984.

H. Gomaa, "Software design methods for real-time systems," Software Engineering Institute Carnegie Mellon University, December 1989.

K. Gross, "Audio networking: Applications and requirements," *Journal of the Audio Engineering Society*, vol. 54, no. 1/2, pp. 62–66, January/February 2006.

C. A. R. Hoare, *Communicating sequential processes.* Prentice Hall International, 1985.

D. C. Hyde. (1995, March) Introduction to the programming language Occam. [Online]. Available: http://www.eg.bucknell.edu/~cs366/occam.pdf

IEC, "IEC 61883-6 audio and music data transmission protocol," 2005.

IEEE 1722, "IEEE 1722 standard for layer 2 transport protocol for time sensitive applications in bridged local area networks," 2012.

IEEE 802.1AS, "IEEE Std 802.1AS timing and synchronization for time-sensitive applications in bridged local area networks," March 2011.

IEEE 802.1BA/D2.1, "IEEE 802.1BA/D2.1 draft standard for local and metropolitan area networks: Audio video bridging (AVB) systems," 2010, IEEE 802.1BA/D2.1.

IEEE 802.1Qat, "IEEE Std 802.1Qat standard for local and metropolitan area networks: Virtual bridged local area networks amendment 14: Stream reservation protocol," September 2010.

IEEE 802.1Qav, "IEEE Std 802.1Qav standard for local and metropolitan area networks: Virtual bridged local area networks amendment 12: Forwarding and queueing enhancements for time-sensitive streams," January 2010.

IEEE P1722.1/D21, "IEEE P1722.1/D21 draft standard for device discovery, connection management and control protocol for ieee 1722 based devices," July 2012, IEEE P1722.1/D21.

J. C. Kelly, "A comparison of four design methods for real-time systems," in *Proceedings of the 9th international conference on Software Engineering.* IEEE Computer Society Press, 1987, pp. 238–252.

D. May. (2009) Multicore architecture. [Online]. Available: http://www.cs.bris.ac.uk/~dave/iet2009.pdf

D. May. (2009, October) The XMOS XS1 architecture. [Online]. Available: http://www.xmos.com/node/14080?version=latest

N. Bouillot, et al, "AES white paper: Best practices in network audio," *Journal of the Audio Engineering Society*, vol. 57, no. 9, pp. 729–741, September 2009.

N. Chigwamba, et al, "Parameter relationships in high-speed audio networks," *Journal of the Audio Engineering Society*, vol. 60, no. 1/2, pp. 132–146, March 2012.

P. Foulkes et al., "Network neutral control over quality of service networks," *J. Audio Eng. Soc*, vol. 59, no. 11, pp. 835–844, 2011. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=16151

D. Pountain and D. May, *A tutorial introduction to Occam programming.* BSP Professional Books, 1988.

A. W. Roscoe, *The theory and practice of concurrency*, 3rd ed. Pearson, 2005. [Online]. Available: www.cs.ox.ac.uk/bill.roscoe/publications/68b.pdf

F. Rumsey, "Audio networking for the pros," *Journal of the Audio Engineering Society*, vol. 57, no. 4, pp. 271–275, April 2009.

B. Selic, G. Gullekson, and P. Ward, *Real-time object-oriented modeling.* John Wiley and Sons, 1994.

H. Simpson and K. Jackson, "Process synchronisation in MASCOT," *British Computer Society Journal*, vol. 22, no. 4, pp. 332–345, 1979.

S. Summit. (2005) comp.lang.c FAQ list question 10.4. [Online]. Available: http://c-faq.com/cpp/multistmt.html

L. Torvalds. (2001) Linux kernel coding style. [Online]. Available: https://www.kernel.org/doc/Documentation/CodingStyle

Universal Media Access Networks. (2012) UNOS overview. [Online]. Available: http://www.unosnet.com/unosnet/index.php/overview.html

Valgrind Developers. (2012, August) Valgrind user manual. [Online]. Available: http://valgrind.org/docs/manual/manual.html

P. Ward and S. Mellor, *Structured development for realtime systems, vols 1 – 3.* Yourdon Press, 1985.

P. Ward and S. Mellor, *Structured development for realtime systems, vol 1.* Yourdon Press, 1985a.

P. Ward and S. Mellor, *Structured development for realtime systems, vol 2.* Yourdon Press, 1985b.

J. Watkinson, *Introduction to digital audio.* Focal Press, 1994.

P. White. (2004) Yamaha O1X: Firewire interface. [Online]. Available: http://www.soundonsound.com/sos/mar04/articles/yamaha01x.htm

R. Williams, *Real-Time Systems Development.* Butterworth-Heinemann, 2005.

XMOS. (2011) XMOS AVB design guide. [Online]. Available: http://www.xmos.com/published/avb-reference-design-guide-0

XMOS. (2011a, November) XC programming guide. [Online]. Available: http://www.xmos.com/download/public/XC-Programming-Guide(X1009B).pdf

XMOS. (2011b, November) XC specification. [Online]. Available: https://www.xmos.com/download/public/XC-Specification(X5965C).pdf

XMOS. (2011c) AVB LCB reference design product brief. [Online]. Available: http://www.xmos.com/published/avbl2pb?version=latest

XMOS. (2011d, January) AVB audio endpoint kit schematics. [Online]. Available: https://www.xmos.com/en/download/public/AVB-Audio-Endpoint-Kit-Schematic%282.0%29.pdf

XMOS. (2012, March) XMOS architecture and XS1 devices. [Online]. Available: http://www.xmos.com/download/public/XS1-Family-Product-Brief(X3061D).pdf?support=1

XMOS. (2012) The AVB audio software reference design. [Online]. Available: http://www.xmos.com/applications/avb

XMOS. (2012, March) XMOS TCP/IP stack design guide. [Online]. Available: http://www.xmos.com/published/xmos-tcpip-stack-design-guide

XMOS. (2013) xCore architecture overview. [Online]. Available: http://www.xmos.com/download/public/xCORE-Architecture.pdf

Yamaha Corporation Pro Audio and Digital Musical Instrument Division, *Yamaha O3D digital mixing console: Owner's manual*, 1997.

Yamaha Corporation Pro Audio and Digital Musical Instrument Division. (2003) Digital mixing studio O1X owner's manual. [Online]. Available: http://www2.yamaha.co.jp/manual/pdf/emi/english/synth/01XE1.pdf

E. Yourdon. (1989) Just enough structured analysis. [Online]. Available: http://www.yourdon.info/jesa/jesa.php

R. Zvonar. (2006) A history of spatial music. [Online]. Available: http://www.zvonar.com/writing/spatial_music/History.html