

AVB Endpoint Design Guide

Document Number: XM000102A

Publication Date: 2014/11/24
XMOS © 2014, All Rights Reserved.



Table of Contents

1	Overview	3
1.1	Summary	3
1.1.1	XMOS AVB-LC Key Features	4
2	XMOS AVB-LC specification	5
3	Ethernet AVB standards	6
3.1	802.1AS	6
3.2	802.1Qav	6
3.3	802.1Qat	7
3.4	IEC 61883-6	7
3.5	IEEE 1722	7
3.6	IEEE 1722.1	7
4	Hardware development platforms	8
5	System description	9
5.1	High level system architecture	9
5.2	Ethernet MAC component	10
5.2.1	1722 packet routing	11
5.3	Precision Timing Protocol component	11
5.4	Audio components	12
5.4.1	AVB streams, channels, talkers and listeners	12
5.4.2	Internal routing, media FIFOs	13
5.4.3	Talker units	14
5.4.4	Listener units	15
5.4.5	Media FIFOs to XC channels	15
5.4.6	Audio hardware interfaces	15
5.5	Media clocks	16
5.5.1	Driving an external clock generator	16
5.6	Device Discovery, Connection Management and Control	17
5.6.1	The control task	17
5.6.2	1722.1	17
5.7	Resource usage	18
5.7.1	Available chip resources	18
5.7.2	Ethernet component	19
5.7.3	PTP component	19
5.7.4	Media clock server	19
5.7.5	Audio component(s)	19
5.7.6	Configuration/control	20
6	Programming guide	21
6.1	Getting started	21
6.1.1	Obtaining the latest firmware	21
6.1.2	Installing xTIMEcomposer Tools Suite	21
6.1.3	Importing and building the firmware	21
6.1.4	Installing the application onto flash memory	22
6.1.5	Using the Command Line Tools	22
6.1.6	Using Command Line Tools	22

6.2	Source code structure	23
6.2.1	Directory Structure	23
6.2.2	Key Files	25
7	API Reference	26
7.1	Configuration defines	26
7.1.1	Demo and hardware specific	26
7.1.2	Core AVB parameters	26
7.1.3	Ethernet	27
7.1.4	Audio subsystem	27
7.1.5	1722.1	28
7.2	Component tasks and functions	29
7.2.1	Core components	29
7.2.2	Audio components	41
7.2.3	Video components	45
7.3	AVB API	47
7.3.1	General control functions	47
7.3.2	Multicast Address Allocation commands	48
7.3.3	MAAP application hooks	49
7.3.4	AVB Control API	50
7.3.5	1722.1 Controller commands	68
7.3.6	1722.1 Discovery commands	70
7.3.7	1722.1 application hooks	71
7.4	1722.1 descriptors	75
7.4.1	Editing descriptors	75
7.4.2	Adding and removing descriptors	77
7.5	PTP client API	77
7.5.1	Time data structures	77
7.5.2	Getting PTP time information	77
7.5.3	Converting timestamps	80

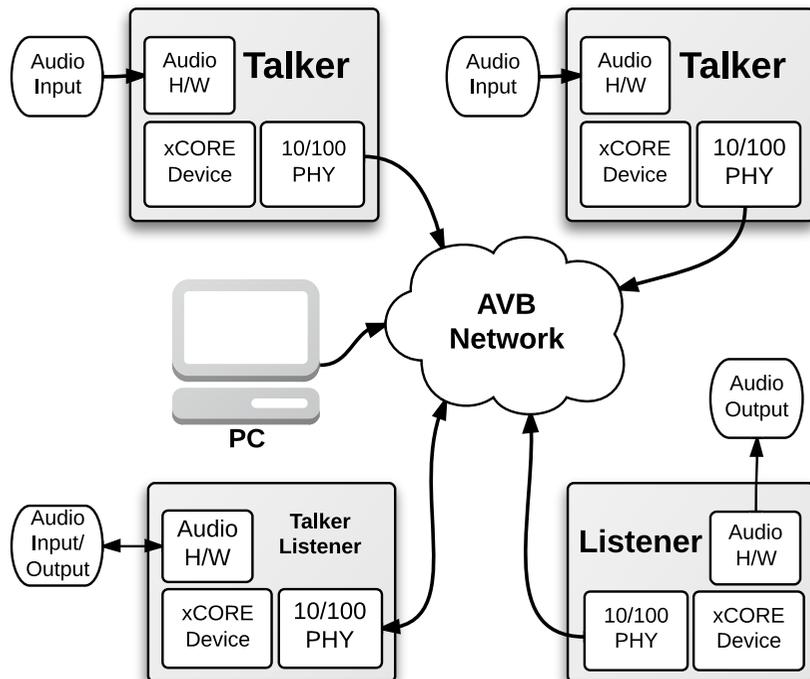
1 Overview

IN THIS CHAPTER

- Summary
-

1.1 Summary

The XMOS Audio Video Bridging (AVB) endpoint is a reference design that provides time-synchronized, low latency streaming services through IEEE 802 networks. The solution is firmware that is implemented on the XMOS xCORE architecture and can be deployed on a number of different xCORE parts depending on system requirements such as stream and channel count.



1.1.1 XMOS AVB-LC Key Features

- ▶ 100 Mbit/s full duplex Ethernet interface via MII
- ▶ Support for 1722.1 discovery, enumeration, command and control: ADP, AEC (AEM) and ACMP
- ▶ Simultaneous 1722 Talker and Listener support for sourcing and sinking audio
- ▶ 1722 MAAP support for Talker stream MAC address allocation
- ▶ 802.1Q Stream Reservation Protocols for QoS including MSRP and MVRP
- ▶ 802.1AS Precision Time Protocol server for synchronization
- ▶ I2S audio interface for connection to external codecs and DSPs
- ▶ Media clock recovery and interface to a PLL clock source for high quality audio clock reproduction

2 XMOS AVB-LC specification

Supported Standards

Ethernet	IEEE 802.3 (via MII)
AVB QoS	IEEE 802.1Qav, 802.1Qat
Precision Timing Protocol	IEEE 802.1AS-2011
Audio Stream Encapsulation	IEEE 1722-2011
Audio Format	IEC 61883-6 AM824
Enumeration and control	IEEE 1722.1-2013

Supported Devices

XMOS Devices	XS1-L16A-128-QF124-C10 XS1-L12A-128-QF124-C10
--------------	--

Requirements

Development Tools	xTIMEComposer suite v13.0.2 or later
Ethernet	1 x MII compatible 100Mbit PHY
Audio	Audio input/output device (e.g. Audio CODEC) Cirrus CS2100-CP PLL/Frequency synthesizer to generate CODEC master clock
Boot/Storage	Compatible SPI Flash Device

Licensing and Support

Reference code provided without charge under license from XMOS. Support via <http://www.xmos.com/secure/tickets>. Reference code is maintained by XMOS Limited.

3 Ethernet AVB standards

IN THIS CHAPTER

- ▶ 802.1AS
 - ▶ 802.1Qav
 - ▶ 802.1Qat
 - ▶ IEC 61883-6
 - ▶ IEEE 1722
 - ▶ IEEE 1722.1
-

Ethernet AVB consists of a collection of different standards that together allow audio and video to be streamed over Ethernet. The standards provide synchronized, uninterrupted streaming with multiple talkers and listeners on a switched network infrastructure.

3.1 802.1AS

802.1AS defines a Precision Timing Protocol based on the *IEEE 1558v2* protocol. It allows every device connected to the network to share a common global clock. The protocol allows devices to have a synchronized view of this clock to within microseconds of each other, aiding media stream clock recovery to phase align audio clocks.

The IEEE 802.1AS-2011 standard document¹ is available to download free of charge via the IEEE Get Program.

3.2 802.1Qav

802.1Qav defines a standard for buffering and forwarding of traffic through the network using particular flow control algorithms. It gives predictable latency control on media streams flowing through the network.

The XMOS AVB solution implements the requirements for endpoints defined by *802.1Qav*. This is done by traffic flow control in the transmit arbiter of the Ethernet MAC component.

The *802.1Qav* specification is available as a section in the IEEE 802.1Q-2011 standard document² and is available to download free of charge via the IEEE Get Program.

¹<http://standards.ieee.org/getieee802/download/802.1AS-2011.pdf>

²<http://standards.ieee.org/getieee802/download/802.1Q-2011.pdf>

3.3 802.1Qat

802.1Qat defines a stream reservation protocol that provides end-to-end reservation of bandwidth across an AVB network.

The 802.1Qat specification is available as a section in the IEEE 802.1Q-2011 standard document³.

3.4 IEC 61883-6

IEC 61883-6 defines an audio data format that is contained in *IEEE 1722* streams. The XMOS AVB solution uses *IEC 61883-6* to convey audio sample streams.

The IEC 61883-6:2005 standard document⁴ is available for purchase from the IEC website.

3.5 IEEE 1722

IEEE 1722 defines an encapsulation protocol to transport audio streams over Ethernet. It is complementary to the AVB standards and in particular allows timestamping of a stream based on the *802.1AS* global clock.

The XMOS AVB solution handles both transmission and receipt of audio streams using *IEEE 1722*. In addition it can use the *802.1AS* timestamps to accurately recover the audio master clock from an input stream.

The IEEE 1722-2011 standard document⁵ is available for purchase from the IEEE website.

3.6 IEEE 1722.1

IEEE 1722.1 is a system control protocol, used for device discovery, connection management and enumeration and control of parameters exposed by the AVB endpoints.

The IEEE 1722.1-2013 standard document⁶ is available for purchase from the IEEE website.

³<http://standards.ieee.org/getieee802/download/802.1Q-2011.pdf>

⁴http://webstore.iec.ch/webstore/webstore.nsf/ArtNum_PK/46793

⁵<http://standards.ieee.org/findstds/standard/1722-2011.html>

⁶<http://standards.ieee.org/findstds/standard/1722.1-2013.html>

4 Hardware development platforms

For initial development of single Ethernet port AVB applications the following XMOS development platform is recommended:

- ▶ XK-AVB-LC-SYS AVB Audio Endpoint⁷

It is recommended to have at least two boards for developing streaming audio applications. It is also recommended that an AVB compatible network switch be obtained and used while developing the system.

For developing an application specific board for AVB please refer to the hardware guides for the above boards which contain example schematics, BOMs, design guidelines etc.

⁷<http://www.xmos.com/products/reference-designs/avb>

5 System description

IN THIS CHAPTER

- ▶ High level system architecture
 - ▶ Ethernet MAC component
 - ▶ Precision Timing Protocol component
 - ▶ Audio components
 - ▶ Media clocks
 - ▶ Device Discovery, Connection Management and Control
 - ▶ Resource usage
-

The following sections describe the system architecture of the XMOS AVB software platform.

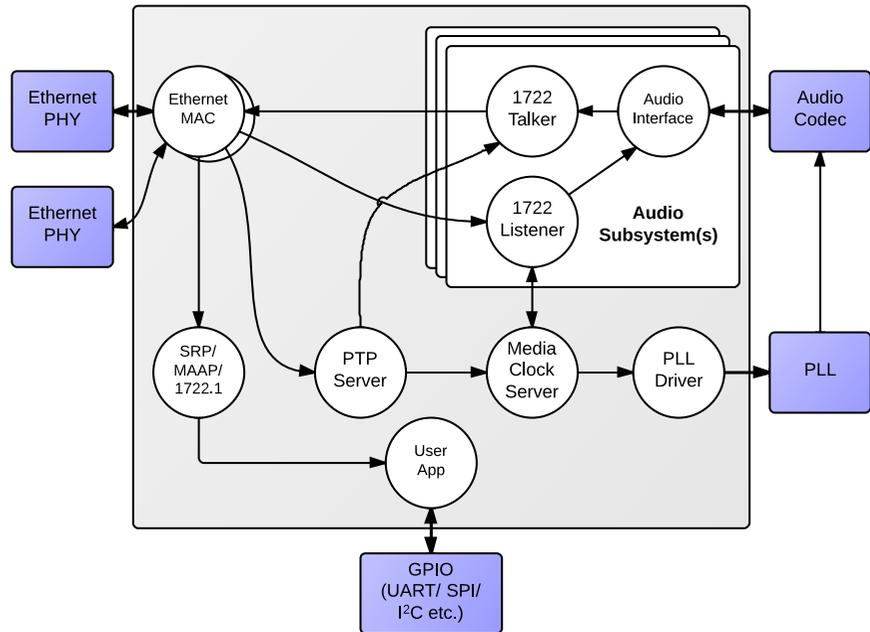
This software design guide assumes the reader is familiar with the XC language and XMOS XS1 devices.

5.1 High level system architecture

An endpoint consists of five main interacting components:

- ▶ The Ethernet MAC
- ▶ The Precision Timing Protocol (PTP) engine
- ▶ Audio streaming components
- ▶ The media clock server
- ▶ Configuration and other application components

The following diagram shows the overall structure of an XMOS AVB endpoint.



5.2 Ethernet MAC component

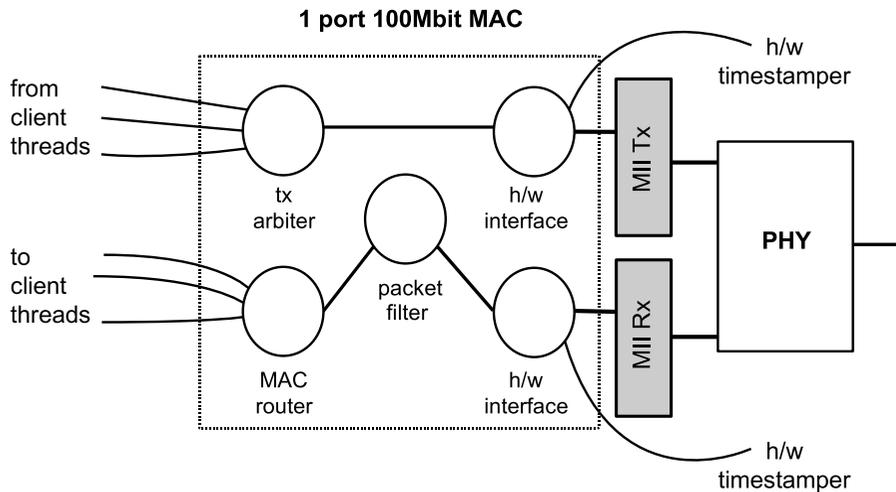
The MAC component provides Ethernet connectivity to the AVB solution. To use the component, a Ethernet PHY must be attached to the XCore ports via MII. The XS1 device is also capable of implementing a dual 100 Mbps interface - see the AVB Daisy Chain product for reference.

The XMOS Ethernet MAC component supports two features that are necessary to implement AVB standards with precise timing and quality constraints:

- ▶ *Timestamping* - allows receipt and transmission of Ethernet frames to be timestamped with respect to a clock (for example a 100 MHz reference clock can provide a resolution of 10 ns).
- ▶ *Time sensitive traffic shaping* - allows traffic bandwidth to be reserved and shaped on egress to provide a steady and guaranteed flow of outgoing media stream packets. The implementation provides flow control to satisfy the requirements of an AVB endpoint as specified in the IEEE 802.1Qav standard.

The single port 100 Mbps component consists of five logical cores, each running at 50 MIPS or more, that must be run on the same tile. These logical cores handle both the receipt and transmission of Ethernet frames. The MAC component can be

linked via channels to other components/logical cores in the system. Each link can set a filter to control which packets are conveyed to it via that channel.



All configuration of the channel is managed by a client C/XC API, which configures and registers the filters. Details of the API used to configure MAC channels can be found in the Ethernet MAC component documentation⁸. This API is used for direct (layer-2) access to the MAC. For AVB applications it is more likely that interaction with the Ethernet stack will be via the main AVB API (see Section §7.3).

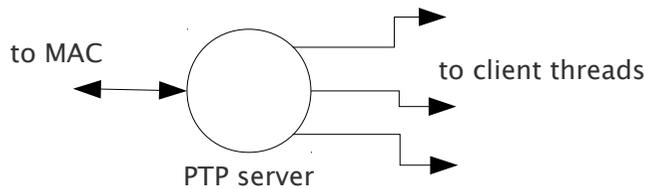
5.2.1 1722 packet routing

The AVB enabled Ethernet MAC also includes a *IEEE 1722* packet router that routes audio packets to the listener components in the system. It controls the routing by stream ID. This requires no configuration and is controlled implicitly via the AVB API described in Section §7.3.

5.3 Precision Timing Protocol component

The Precision Timing Protocol (PTP) component enables a system with a notion of global time on a network. The component implements the *IEEE 802.1AS* protocol. It allows synchronization of the presentation and playback rate of media streams across a network.

⁸https://www.xmos.com/resources/xsoftip?component=module_ethernet



The timing component consists of two logical cores. It connects to the Ethernet MAC component and provides channel ends for clients to query for timing information. The component interprets PTP packets from the MAC and maintains a notion of global time. The maintenance of global time requires no application interaction with the component.

The PTP component can be configured at runtime to be a potential *PTP grandmaster* or a *PTP slave* only. If the component is configured as a grandmaster, it supplies a clock source to the network. If the network has several grandmasters, the potential grandmasters negotiate between themselves to select a single grandmaster. Once a single grandmaster is selected, all units on the network synchronize a global time from this source and the other grandmasters stop providing timing information. Depending on the intermediate network, this synchronization can be to sub-microsecond level resolution.

Client tasks connect to the timing component via channels. The relationship between the local reference counter and global time is maintained across this channel, allowing a client to timestamp with a local timer very accurately and then convert it to global time, giving highly accurate global timestamps.

Client tasks can communicate with the server using the API described in Section §7.5.

- ▶ The PTP system in the endpoint is self-configuring, it runs automatically and gives each endpoint an accurate notion of a global clock.
- ▶ The global clock is *not* the same as the audio word clock, although it can be used to derive it. An audio stream may be at a rate that is independent of the PTP clock but will contain timestamps that use the global PTP clock domain as a reference domain.

5.4 Audio components

5.4.1 AVB streams, channels, talkers and listeners

Audio is transported in streams of data, where each stream may have multiple channels. Endpoints producing streams are called *Talkers* and those receiving them are called *Listeners*. Each stream on the network has a unique 64-bit stream ID.

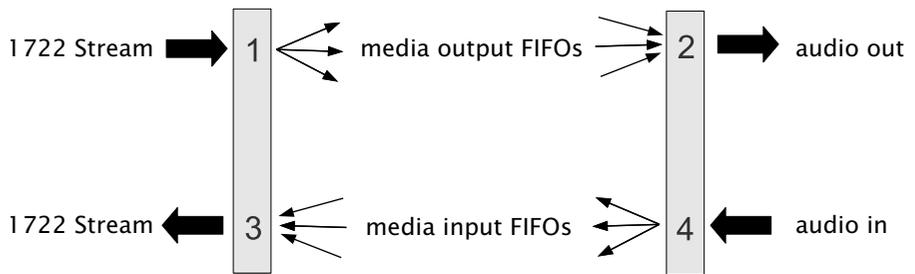
A single endpoint can be a Talker, a Listener or both. In general each endpoint will have a number of *sinks* with the capacity to receive a number of incoming streams and a number of *sources* with the capacity to transmit a number of streams.

Routing is done using layer 2 Ethernet addresses. Each stream is sent from a particular source MAC address to a particular destination MAC address. The destination MAC address is a multicast address so that several Listeners may receive it. In addition, AVB switches can reserve an end-to-end path with guaranteed bandwidth for a stream. This is done by the Talker endpoint advertising the stream to the switches and the Listener(s) registering to receive it. If sufficient bandwidth is not available, this registration will fail.

Streams carry their own *presentation time*, the time that samples are due to be output, allowing multiple Listeners that receive the same stream to output in sync.

- ▶ Streams are encoded using the 1722 AVB transport protocol.
- ▶ All channels in a stream must be synchronized to the same sample clock.
- ▶ All the channels in a stream must come from the same Talker.
- ▶ Routing of audio streams uses Ethernet layer 2 routing based on a multicast destination MAC address
- ▶ Routing of channels is done at the stream level. All channels within a stream must be routed to the same place. However, a stream can be multicast to several Listeners, each of which picks out different channels.
- ▶ A single end point can be both a Talker and Listener.
- ▶ Information such as stream ID and destination MAC address of a Talker stream should be communicated to Listeners via 1722.1. (see Section §5.6).

5.4.2 Internal routing, media FIFOs



As described in the previous section, an IEEE 1722 audio stream may consist of many channels. These channels need to be routed to particular audio I/Os on the endpoint. To achieve maximum flexibility the XMOS design uses intermediate media FIFOs to route audio. Each FIFO contains a single channel of audio.

The above figure shows the breakdown of 1722 streams into local FIFOs. The figure shows four points where transitions to and from media FIFOs occur. For audio being received by an endpoint:

1. When a 1722 stream is received, its channels are mapped to output media FIFOs. This mapping can be configured dynamically so that it can be changed at runtime by the configuration component.
2. The digital hardware interface maps media FIFOs to audio outputs. This mapping is fixed and is configured statically in the software.

For audio being transmitted by an endpoint:

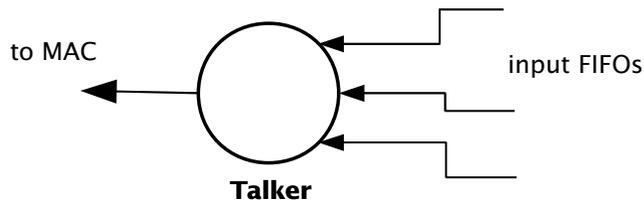
1. The digital hardware interface maps digital audio inputs to local media FIFOs. This mapping is fixed and cannot be changed at runtime.
2. Several input FIFOs can be combined into a 1722 stream. This mapping is dynamic.

The configuration of the mappings is handled through the API describe in §7.3.



Media FIFOs use shared memory to move data between tasks, thus the filling and emptying of the FIFO must be on the same tile.

5.4.3 Talker units

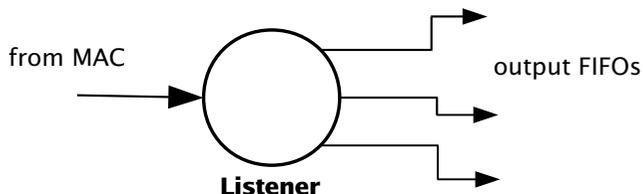


A talker unit consists of one logical core which creates *IEEE 1722* packets and passes the audio samples onto the MAC. Audio samples are passed to this component via input media FIFOs. Samples are pushed into this FIFO from a different task implementing the audio hardware interface. The Talker task removes the samples and combines them into *IEEE 1722* Ethernet packets to be transmitted via the MAC component.

When the packets are created the timestamps are converted to the time domain of the global clock provided by the PTP component, and a fixed offset is added to the timestamps to provide the *presentation time* of the samples (*i.e* the time at which the sample should be played by a Listener).

A system may have several Talker units. However, since samples are passed via a shared memory interface a talker can only combine input FIFOs that are created on the same tile as the talker. The instantiating of talker units is performed via the API described in Section §7.2. Once the talker unit starts, it registers with the main control task and is controlled via the main AVB API described in Section §7.3.

5.4.4 Listener units



A Listener unit takes *IEEE 1722* packets from the MAC and converts them into a sample stream to be fed into a media FIFO. Each audio Listener component can listen to several *IEEE 1722* streams.

A system may have several Listener units. The instantiating of Listener units is performed via the API described in Section §7.2. Once the Listener unit starts, it registers with the main control task and is controlled via the main AVB API described in Section §7.3.

5.4.5 Media FIFOs to XC channels

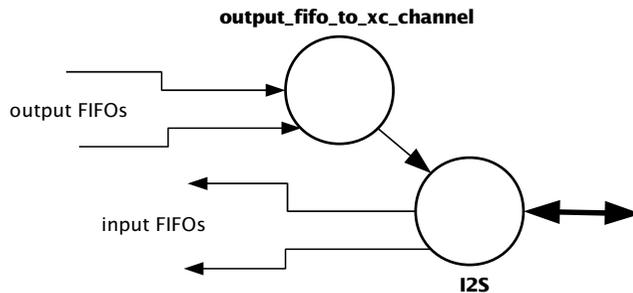
Sometimes it is useful to convert the audio stream in a media FIFO into a sample stream over an XC channel. This may be needed to move samples off tile or if the audio interface task requires samples over a channel. Several functions are provided to do this and are described in Section §7.2.

5.4.6 Audio hardware interfaces

The audio hardware interface components drive external audio hardware, pull audio out of media output FIFOs and push into media input FIFOs.

Different interfaces interact in different ways, some directly push and pull from the media FIFOs, whereas some for performance reasons require samples to be provided over an XC channel.

The following diagram shows one potential layout of the I2S component which pushes its input directly to media input FIFOs but takes output FIFOs from an XC channel.



5.5 Media clocks

A media clock controls the rate at which information is passed to an external media playing device. For example, an audio word clock that governs the rate at which samples should be passed to an audio CODEC. An XMOS AVB endpoint can keep track of several media clocks.

A media clock can be synchronized to one of two sources:

- ▶ An incoming clock signal on a port.
- ▶ The word clock of a remote endpoint, derived from an incoming *IEEE 1722* audio stream.

A hardware interface can be tied to a particular media clock, allowing the media output from the XMOS device to be synchronized with other devices on the network.

All media clocks are maintained by the media clock server component. This component maintains the current state of all the media clocks in the system. It then periodically updates other components with clock change information to keep the system synchronized. The set of media clocks is determined by an array passed to the server at startup.

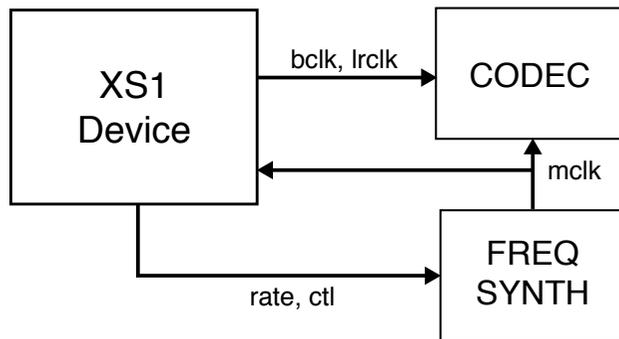
The media clock server component also receives information from the audio listener component to track timing information of incoming *IEEE 1722* streams. It then sends control information back to ensure the listening component honors the presentation time of the incoming stream.



Multiple media clocks require multiple hardware PLLs. AVB-LC hardware supports a single media clock.

5.5.1 Driving an external clock generator

A high quality, low jitter master clock is often required to drive an audio CODEC and must be synchronized with an AVB media clock. The XS1 chip cannot provide this clock directly but can provide a lower frequency source for a frequency synthesizer chip or external PLL chip. The frequency synthesizer chip must be able to generate a high frequency clock based on a lower frequency signal, such as the Cirrus Logic CS2100-CP. The recommended configuration is as in the block diagram below:



The XS1 device provides control to the frequency synthesizer and the frequency synthesizer provides the audio master clock to the CODEC and XS1 device. The sample bit and word clocks are then provided to the CODEC by the XS1 device.

5.6 Device Discovery, Connection Management and Control

5.6.1 The control task

In addition to components described in previous sections, an AVB endpoint application requires a task to control and configure the system. This control task varies across applications but the protocol to provide device discovery, connection management and control services has been standardised by the IEEE in 1722.1.

5.6.2 1722.1

The 1722.1 standard defines four independent steps that can be used to connect end stations that use 1722 streams to transport media across a LAN. The steps are:

1. Discovery
2. Enumeration
3. Connection Management
4. Control

These steps can be used together to form a system of end stations that interoperate with each other in a standards compliant way. The application that will use these individual steps is called a *Controller* and is the third member in the Talker, Listener and Controller device relationship.

A Controller may exist within a Talker, a Listener, or exist remotely within the network in a separate endpoint or general purpose computer.

The Controller can use the individual steps to find, connect and control entities on the network but it may choose to not use all of the steps if the Controller already

knows some of the information (e.g. hard coded values assigned by user/hardware switch or values from previous session establishment) that can be gained in using the steps. The only required step is connection management because this is the step that establishes the bandwidth usage and reservations across the AVB network.

The four steps are broken down as follows:

- ▶ Discovery is the process of finding AVB endpoints on the LAN that have services that are useful to the other AVB endpoints on the network. The discovery process also covers the termination of the publication of those services on the network.
- ▶ Enumeration is the process of the collection of information from the AVB endpoint that could help an 1722.1 Controller to use the capabilities of the AVB endpoint. This information can be used for connection management.
- ▶ Connection management is the process of connecting or disconnecting one or more streams between two or more AVB endpoint.
- ▶ Control is the process of adjusting a parameter on the endpoint from another endpoint. There are a number of standard types of controls used in media devices like volume control, mute control and so on. A framework of basic commands allows the control process to be extended by the endpoint.



The XMOS endpoint provides full support for Talker and Listener 1722.1 services. It is expected that Controller software will be available on the network for handling connection management and control.

To assist in this task a unified control API is presented in Section §7.3.

5.7 Resource usage

5.7.1 Available chip resources

Each XMOS device has a set of resources detailed in the following table. The resources are split amongst different tiles on the device which may affect how resources can be used:

Device	Logical Cores	MIPS/Core	Memory (KB)	Ports
XS1-L16A-128-QF124-C10	16	1000	128	32 x 1bit 12 x 4bit 7 x 8bit 3 x 16bit



Note that some ports overlap on the device so, for example, using a 16 bit port may make some 1 bit ports unavailable. See the device datasheets for details.

The following sections detail the resource required for each component. Please note that the memory requirements for code size should be taken as a rough

guide since exact memory usage depends on the integration of components (which components are on which tile etc.) in the final build of the application.

5.7.2 Ethernet component

Each endpoint requires an Ethernet MAC layer.

Component	Logical Cores	MIPS/Core	Memory (KB)	Ports
Ethernet	5	50	15 code, 1.5 per buffer	6 x 1 bit, 2 x 4bit

5.7.3 PTP component

Every AVB endpoint must include a PTP component.

Component	Logical Cores	MIPS/Core	Memory (KB)	Ports
PTP	1	50	7	None

5.7.4 Media clock server

Every AVB endpoint must include a media clock server.

Component	Logical Cores	MIPS/Core	Memory (KB)	Ports
Media Clock Server	1	50	1	None

If the endpoint drives an external PLL, a PLL driver component is required.

Component	Logical Cores	MIPS/Core	Memory (KB)	Ports
PLL driver	0 - 1	50	0.5	1 x 1 bit + ports to configure PLL



PTP, Media Clock Server and PLL driver components may be combined into a single logical core running at 100 MIPS if the number of channels is constrained to 2.

5.7.5 Audio component(s)

Each endpoint may have several listener and talker components. Each listener/talker component is capable of handling four IEEE 1722 streams and up to 12 channels of audio.

Component	Logical Cores	MIPS/Core	Memory (KB)	Ports
1722 listener unit	1	50	5	None
1722 talker unit	1	50	5	None



The Talker and Listener components may be combined into a single logical core running at 100 MIPS if the number of streams is 1 and the number of channels is ≤ 4 .

The amount of resource required for audio processing depends on the interface and the number of audio channels required. The overheads for the interface are:

Component	Logical Cores	MIPS/Core	Memory(KB)	Ports
I2S	1	50	0.5	3 x 1 bit 1 x 1 bit per stereo channel

The following table shows that number of channels an interface can handle per logical core:

Component	Sample Rate (kHz)	Channels
I2S	44.1/48	8 in and 8 out
I2S	88.2/96	4 in and 4 out

Note that several instances of the audio interface component can be made *e.g.* you could use 2 logical cores to handle 16 channels of I2S. The following table shows how much buffering memory is required depending on the number of audio channels.

Sample Rate (kHz)	Audio Channels	Memory (KB)
44.1	n in/m out	0.5 x (m)
48	n in/m out	0.5 x (m)
88.2	n in/m out	1 x (m)
96	n in/m out	1 x (m)

5.7.6 Configuration/control

In addition to the other components there are application dependant tasks that control other I/O. For general configuration and slow I/O a minimum of 1 logical core (50 MIPS) should be reserved.

6 Programming guide

IN THIS CHAPTER

- ▶ Getting started
 - ▶ Source code structure
-

6.1 Getting started

6.1.1 Obtaining the latest firmware

1. Log into xmos.com and access *My XMOS ▶ Reference Designs*
2. Request access to the *AVB Endpoint Software* by clicking the *Request Access* link under *AVB Audio Endpoint*. An email will be sent to your registered email address when access is granted.
3. A *Download* link will appear where the *Request Access* link previously appeared. Click and download the firmware zip.

6.1.2 Installing xTIMEcomposer Tools Suite

The AVB-LC software requires xTIMEcomposer version 13.0.2 or greater. It can be downloaded at the following URL

- ▶ <http://www.xmos.com/support/xtools>

6.1.3 Importing and building the firmware

To import and build the firmware, open xTIMEcomposer Studio and follow these steps:

1. Choose *File ▶ Import*.
2. Choose *General ▶ Existing Projects into Workspace* and click **Next**.
3. Click **Browse** next to '**Select archive file**' and select the firmware .zip file downloaded in section 1.
4. Make sure that all projects are ticked in the *Projects* list.
5. Click **Finish**.
6. Select the `app_avb_lc_demo` project in the Project Explorer and click the **Build** icon in the main toolbar.

6.1.4 Installing the application onto flash memory

1. Connect the xTAG-2 debug adapter (XA-SK-XTAG2) to the first AVB endpoint board.
2. Plug the xTAG-2 into your development system via USB.
3. Plug in the 5V power adapter and connect it to the AVB endpoint board.
4. In xTIMEcomposer, right-click on the binary within the *app_avb_lc_demo/bin* folder of the project.
5. Choose *Flash As ► Flash Configurations*.
6. Double click *xCORE Application* in the left panel.
7. Choose *hardware* in *Device options* and select the relevant xTAG-2 adapter.
8. Click on **Apply** if configuration has changed.
9. Click on **Flash**. Once completed, reset the AVB endpoint board using the reset button.
10. Repeat steps 1 through 8 for the second endpoint.

6.1.5 Using the Command Line Tools

1. Open the XMOS command line tools (Command Prompt) and execute the following command:

```
xrun --xscope <binary>.xe
```

2. If multiple xTAG-2s are connected, obtain the adapter ID integer by executing:

```
xrun -l
```

3. Execute the *xrun* command with the adapter ID flag

```
xrun --id <id> --xscope <binary>.xe
```

6.1.5.1 Installing the application onto flash via Command Line

1. Connect the xTAG-2 debug adapter to the relevant development board, then plug the xTAG-2 into your PC or Mac.

6.1.6 Using Command Line Tools

1. Open the XMOS command line tools (Command Prompt) and execute the following command:

```
xflash <binary>.xe
```

2. If multiple xTAG-2s are connected, obtain the adapter ID integer by executing:

```
xrun -l
```

3. Execute the *xflash* command with the adapter ID flag

```
xflash --id <id> <binary>.xe
```

6.2 Source code structure

6.2.1 Directory Structure

The source code is split into several top-level directories which are presented as separate projects in xTIMEcomposer Studio. These are split into modules and applications.

Applications build into a single executable using the source code from the modules. The modules used by an application are specified using the `USED_MODULES` variable in the application Makefile. For more details on this module structure please see the XMOS build system document *Using XMOS Makefiles (X6348)*.

The AVB-LC source package contains a simple demonstration application *app_avb_lc_demo*.

Core AVB modules are presented in the `sc_avb` repository. Some support modules originate in other repositories:

Directory	Description	Repository
module_ethernet	Ethernet MAC	sc_ethernet
module_ethernet_board_support	Hardware specific board configuration for Ethernet MAC	sc_ethernet
module_ethernet_smi	SMI interface for reading/writing registers to the Ethernet PHY	sc_ethernet
module_otp_board_info	Interface for reading serial number and MAC addresses from OTP memory	sc_otp
module_i2c_simple	Two wire configuration protocol code.	sc_i2c
module_random	Random number generator	sc_util
module_logging	Debug print library	sc_util
module_sliceKIT_support	sliceKIT core board support	sc_sliceKIT_support

The following modules in sc_avb contain the core AVB code and are needed by every application:

Directory	Description
module_avb	Main AVB code for control and configuration.
module_avb_1722	IEEE 1722 transport (listener and talker functionality).
module_avb_1722_1	IEEE 1722.1 AVB control protocol.
module_avb_1722_maap	IEEE 1722 MAAP - Multicast address allocation code.
module_avb_audio	Code for media FIFOs and audio hardware interfaces (I2S).
module_avb_flash	Flash access for firmware upgrade
module_avb_media_clock	Media clock server code for clock recovery.
module_avb_srp	802.1Qat stream reservation (SRP/MRP/MVRP) code.
module_avb_util	General utility functions used by all modules.
module_gptp	802.1AS Precision Time Protocol code.

6.2.2 Key Files

File	Description
<code>avb_api.h</code>	Header file containing declarations for the core AVB control API.
<code>avb_1722_1_app_hooks.h</code>	Header file containing declarations for hooks into 1722.1
<code>ethernet_rx_client.h</code>	Header file for clients that require direct access to the ethernet MAC (RX).
<code>ethernet_tx_client.h</code>	Header file for clients that require direct access to the ethernet MAC (TX).
<code>gptp.h</code>	Header file for access to the PTP server.
<code>audio_i2s.h</code>	Header file containing the I2S audio component.

7 API Reference

IN THIS CHAPTER

- ▶ Configuration defines
 - ▶ Component tasks and functions
 - ▶ AVB API
 - ▶ 1722.1 descriptors
 - ▶ PTP client API
-

7.1 Configuration defines

7.1.1 Demo and hardware specific

Demo parameters and hardware port definitions are set in a header configuration file named `app_config.h` within the `src/` directory of the application.

Macro	AVB_DEMO_ENABLE_TALKER
Description	Global switch to enable or disable AVB Talker functionality in the demo.

Macro	AVB_DEMO_ENABLE_LISTENER
Description	Global switch to enable or disable AVB Listener functionality in the demo.

Macro	AVB_DEMO_NUM_CHANNELS
Description	Number of input/output audio channels in the demo application For simplicity, input and output is identical in size but can be configured differently in <code>avb_conf.h</code> .

7.1.2 Core AVB parameters

Each application using the AVB modules must include a header configuration file named `avb_conf.h` within the `src/` directory of the application and this file must set the `#defines` in the following two sections.

See the demo application for a realistic example.

 Defaults for these #defines are assigned in their absence, but may cause compilation failure or unpredictable/erroneous behaviour.

7.1.3 Ethernet

See the Ethernet documentation for detailed information on its parameters:

<https://www.xmos.com/published/xmos-layer-2-ethernet-mac-component?version=latest>

7.1.4 Audio subsystem

Macro	AVB_MAX_AUDIO_SAMPLE_RATE
Description	The maximum sample rate in Hz of audio that is to be input or output.

Macro	AVB_NUM_SOURCES
Description	The total number of AVB sources (streams that are to be transmitted).

Macro	AVB_NUM_TALKER_UNITS
Description	The total number of Talker components (typically the number of tasks running the avb_1722_talker() function).

Macro	AVB_MAX_CHANNELS_PER_TALKER_STREAM
Description	The maximum number of channels permitted per 1722 Talker stream.

Macro	AVB_NUM_MEDIA_INPUTS
Description	The total number of media inputs (typically number of I2S input channels).

Macro	AVB_NUM_SINKS
Description	The total number of AVB sinks (incoming streams that can be listened to).

Macro	AVB_NUM_LISTENER_UNITS
Description	The total number of listener components (typically the number of tasks running the avb_1722_listener() function).

Macro	AVB_MAX_CHANNELS_PER_LISTENER_STREAM
Description	The maximum number of channels permitted per 1722 Listener stream.

Macro	AVB_NUM_MEDIA_OUTPUTS
Description	The total number of media outputs (typically the number of I2S output channels).

Macro	AVB_NUM_MEDIA_UNITS
Description	The number of components in the endpoint that will register and initialize media FIFOs (typically an audio interface component such as I2S).

Macro	AVB_NUM_MEDIA_CLOCKS
Description	The number of media clocks in the endpoint. Typically the number of clock domains, each with a separate PLL and master clock.

7.1.5 1722.1

Macro	AVB_ENABLE_1722_1
Description	Enable 1722.1 AVDECC on the entity.

Macro	AVB_1722_1_TALKER_ENABLED
Description	Enable the 1722.1 Talker functionality.

Macro	AVB_1722_1_LISTENER_ENABLED
Description	Enable the 1722.1 Listener functionality.

Macro	AVB_1722_1_CONTROLLER_ENABLED
Description	Enable 1722.1 Controller functionality on the entity.

Descriptor specific strings can be modified in a header configuration file named `aem_entity_strings.h.in` within the `src/` directory. It is post-processed by a script in the build stage to expand strings to 64 octet padded with zeros.

Define	Description
AVB_1722_1_ENTITY_NAME_STRING	A string (64 octet max) containing an Entity name
AVB_1722_1_FIRMWARE_VERSION_STRING	A string (64 octet max) containing the firmware version of the Entity
AVB_1722_1_GROUP_NAME_STRING	A string (64 octet max) containing the group name of the Entity
AVB_1722_1_SERIAL_NUMBER_STRING	A string (64 octet max) containing the serial number of the Entity
AVB_1722_1_VENDOR_NAME_STRING	A string (64 octet max) containing the vendor name of the Entity
AVB_1722_1_MODEL_NAME_STRING	A string (64 octet max) containing the model name of the Entity

7.2 Component tasks and functions

The following functions provide components that can be combined in the top-level main.

7.2.1 Core components

Function	avb_manager
Description	Core AVB API management task that can be combined with other AVB tasks such as SRP or 1722.1.

Continued on next page

Type	<pre> [[combinable]] void avb_manager(server interface avb_interface i_avb[num_avb_clients], unsigned num_avb_clients, client interface srp_interface i_srp, chanend c_media_ctl[], chanend(& ?c_listener_ctl) [], chanend(& ?c_talker_ctl) [], chanend c_mac_tx, client interface media_clock_if ?i_media_clock_ctl, chanend c_ptp) </pre>
Parameters	<p>i_avb[] array of avb_interface server interfaces connected to clients of avb_manager</p> <p>num_avb_clients number of client interface connections to the server and the number of elements of i_avb[]</p> <p>i_srp client interface of type srp_interface into an srp_task() task</p> <p>c_media_ctl[] array of chanends connected to components that register/control media FIFOs</p> <p>c_listener_ctl[] array of chanends connected to components that register/control IEEE 1722 sinks</p> <p>c_talker_ctl[] array of chanends connected to components that register/control IEEE 1722 sources</p> <p>c_mac_tx chanend connection to the Ethernet TX server</p> <p>i_media_clock_ctl client interface of type media_clock_if connected to the media clock server</p> <p>c_ptp chanend connection to the PTP server</p>

Type	avb_srp_info_t
Description	Struct containing fields required for SRP reservations.
Fields	<pre> unsigned stream_id 64-bit Stream ID of the stream </pre>

Continued on next page

	<p><code>unsigned char dest_mac_addr</code> Stream destination MAC address.</p> <p><code>short vlan_id</code> VLAN ID for Stream.</p> <p><code>short tspec_max_frame_size</code> Maximum frame size sent by Talker.</p> <p><code>short tspec_max_interval</code> Maximum number of frames sent per class measurement interval.</p> <p><code>unsigned char tspec</code> Data Frame Priority and Rank fields.</p> <p><code>unsigned accumulated_latency</code> Latency at ingress port for Talker registrations, or latency at end of egress media for Listener Declarations.</p> <p><code>unsigned char failure_bridge_id</code></p> <p><code>unsigned char failure_code</code></p>
--	--

Type	srp_interface	
Description		
Functions	Function	register_stream_request
	Description	Used by a Talker application entity to issue a request to the MSRP Participant to initiate the advertisement of an available Stream.
	Type	short register_stream_request(avb_srp_info_t stream_info)
	Parameters	stream_info Struct of type avb_srp_info_t containing parameters of the stream to register
	Returns	The VLAN ID actually joined

Continued on next page

Function	avb_srp_task	
Description	SRP task that implements MSRP and MVRP protocols. Can be combined with other combinable tasks.	
Type	[[combinable]] void avb_srp_task(client interface avb_interface i_avb, server interface srp_interface i_srp, chanend c_mac_rx, chanend c_mac_tx)	
Parameters	i_avb	client interface of type avb_interface into the avb_manager() for API control of the stack
	i_srp	server interface of type srp_interface that offers client tasks access to SRP reservation functionality
	c_mac_rx	chanend into the Ethernet RX server
	c_mac_tx	chanend into the Ethernet TX server

Type	avb_1722_1_aecp_aem_status_code	
Description		
Values	AECP_AEM_STATUS_SUCCESS	The AVDECC Entity successfully performed the command and has valid results.
	AECP_AEM_STATUS_NOT_IMPLEMENTED	The AVDECC Entity does not support the command type.
	AECP_AEM_STATUS_NO_SUCH_DESCRIPTOR	A descriptor with the descriptor_type and descriptor_index specified does not exist.
	AECP_AEM_STATUS_ENTITY_LOCKED	The AVDECC Entity has been locked by another AVDECC Controller.
	AECP_AEM_STATUS_ENTITY_ACQUIRED	The AVDECC Entity has been acquired by another AVDECC Controller.

Continued on next page

<p>AACP_AEM_STATUS_NOT_AUTHENTICATED The AVDECC Controller is not authenticated with the AVDECC Entity.</p> <p>AACP_AEM_STATUS_AUTHENTICATION_DISABLED The AVDECC Controller is trying to use an authentication command when authentication isn't enable on the AVDECC Entity.</p> <p>AACP_AEM_STATUS_BAD_ARGUMENTS One or more of the values in the fields of the frame were deemed to be bad by the AVDECC Entity (unsupported, incorrect combination, etc).</p> <p>AACP_AEM_STATUS_NO_RESOURCES The AVDECC Entity cannot complete the command because it does not have the resources to support it.</p> <p>AACP_AEM_STATUS_IN_PROGRESS The AVDECC Entity is processing the command and will send a second response at a later time with the result of the command.</p> <p>AACP_AEM_STATUS_ENTITY_MISBEHAVING The AVDECC Entity is generated an internal error while trying to process the command.</p> <p>AACP_AEM_STATUS_NOT_SUPPORTED The command is implemented but the target of the command is not supported. For example trying to set the value of a read-only Control.</p> <p>AACP_AEM_STATUS_STREAM_IS_RUNNING The Stream is currently streaming and the command is one which cannot be executed on an Active Stream.</p>
--

Type	avb_1722_1_control_callbacks
Description	

Continued on next page

Type	avb_1722_1_control_callbacks (continued)	
Functions	Function	get_control_value
	Description	This function events on a GET_CONTROL 1722.1 command received from a Controller.
	Type	unsigned char get_control_value(unsigned short control_index, unsigned short &values_length, char values[AEM_MAX_CONTROL_VALUES_LENGTH_BYTES])
	Parameters	control_index the index of the CONTROL descriptor values_length a reference to the length in bytes of the values array values an array of values to return to the Controller The contents of this field are dependent on the Control being fetched.
	Returns	an AEM status code of enum avb_1722_1_aecp_aem_status_code for the GET_CONTROL response

Continued on next page

Type	avb_1722_1_control_callbacks (continued)	
	Function	set_control_value
	Description	This function events on a SET_CONTROL 1722.1 command received from a Controller. The response should always contain the current value (i.e. it contains the new value if the commands succeeds, or the old value if it fails)
	Type	unsigned char set_control_value(unsigned short control_index, unsigned short values_length, char values[AEM_MAX_CONTROL_VALUES_LENGTH_BYTES])
	Parameters	control_index the index of the CONTROL descriptor values_length the length in bytes of the values array values an array of values to set from the Controller. The contents of this field are dependent on the Control being addressed.
	Returns	an AEM status code of enum avb_1722_1_aecp_aem_status_code for the SET_CONTROL response

Function	avb_1722_1_maap_task
Description	1722.1 task that runs ADP, ACMP and AECMP protocols and interacts with the rest of the AVB stack. Can be combined with other combinable tasks.
Type	[[combinable]] void avb_1722_1_maap_task(otp_ports_t &otp_ports, client interface avb_interface i_avb, client interface avb_1722_1_control_callbacks i_1722_1_entity, client interface spi_interface ?i_spi, chanend c_mac_rx, chanend c_mac_tx, chanend c_ptp)

Continued on next page

Parameters	<code>otp_ports</code>	reference to an OTP ports structure of type <code>otp_ports_t</code>
	<code>i_avb</code>	client interface of type <code>avb_interface</code> into <code>avb_manager()</code>
	<code>i_1722_1_entity</code>	client interface of type <code>avb_1722_1_control_callbacks</code>
	<code>i_spi</code>	client interface of type <code>spi_interface</code> into <code>avb_srp_task()</code>
	<code>c_mac_rx</code>	chanend into the Ethernet RX server
	<code>c_mac_tx</code>	chanend into the Ethernet TX server
	<code>c_ptp</code>	chanend into the PTP server

Type	fl_spi_ports
Description	Struct containing ports and clocks used to access a flash device.
Fields	<p><code>buffered in port:8 spiMISO</code> Master input, slave output (MISO) port.</p> <p><code>out port spiSS</code> Slave select (SS) port.</p> <p><code>buffered out port:32 spiCLK</code> Serial clock (SCLK) port.</p> <p><code>buffered out port:8 spiMOSI</code> Master output, slave input (MOSI) port.</p> <p><code>clock spiClkblk</code> Clock block for use with SPI ports.</p>

Type	spi_interface
Description	

Continued on next page

Type	spi_interface (continued)									
Functions	Function command_status									
	Description	This function issues a single command without parameters to the SPI, and reads up to 4 bytes status value from the device.								
	Type	int command_status(int cmd, unsigned returnBytes)								
	Parameters	<table border="0"> <tr> <td data-bbox="494 460 655 492">cmd</td> <td data-bbox="655 460 1174 492">command value - listed above</td> </tr> <tr> <td data-bbox="494 516 655 548">returnBytes</td> <td data-bbox="655 516 1174 624">The number of bytes that are to be read from the device after the command is issued. 0 means no bytes will be read.</td> </tr> </table>	cmd	command value - listed above	returnBytes	The number of bytes that are to be read from the device after the command is issued. 0 means no bytes will be read.				
	cmd	command value - listed above								
	returnBytes	The number of bytes that are to be read from the device after the command is issued. 0 means no bytes will be read.								
	Returns	the read bytes, or zero if no bytes were requested. If multiple bytes are requested, then the last byte read is in the least-significant byte of the return value.								
	Function command_address_status									
	Description	This function issues a single command with a 3-byte address parameter and an optional data-set to be output to or input from the device.								
	Type	void command_address_status(int cmd, unsigned int address, unsigned char data[], int returnBytes)								
	Parameters	<table border="0"> <tr> <td data-bbox="494 1060 655 1091">cmd</td> <td data-bbox="655 1060 1174 1091">command value - listed above</td> </tr> <tr> <td data-bbox="494 1116 655 1148">address</td> <td data-bbox="655 1116 1174 1148">the address to send to the SPI device. Only the least significant 24 bits are used.</td> </tr> <tr> <td data-bbox="494 1194 655 1225">data</td> <td data-bbox="655 1194 1174 1268">an array of data that contains either data to be written to the device, or which is used to store that that is read from the device.</td> </tr> <tr> <td data-bbox="494 1300 655 1331">returnBytes</td> <td data-bbox="655 1300 1174 1460">If positive, then this is the number of bytes that are to be read from the device, into data. If negative, then this is (minus) the number of bytes to be written to the device from data. 0 means no bytes will be read or written.</td> </tr> </table>	cmd	command value - listed above	address	the address to send to the SPI device. Only the least significant 24 bits are used.	data	an array of data that contains either data to be written to the device, or which is used to store that that is read from the device.	returnBytes	If positive, then this is the number of bytes that are to be read from the device, into data. If negative, then this is (minus) the number of bytes to be written to the device from data. 0 means no bytes will be read or written.
	cmd	command value - listed above								
address	the address to send to the SPI device. Only the least significant 24 bits are used.									
data	an array of data that contains either data to be written to the device, or which is used to store that that is read from the device.									
returnBytes	If positive, then this is the number of bytes that are to be read from the device, into data. If negative, then this is (minus) the number of bytes to be written to the device from data. 0 means no bytes will be read or written.									

Function	spi_task
Description	Task that implements a SPI interface to serial flash, typically the boot flash. Can be combined or distributed into other tasks.
Type	[[distributable]] void spi_task(server interface spi_interface i_spi, fl_spi_ports &spi_ports)
Parameters	i_spi server interface of type spi_interface spi_ports reference to a fl_spi_ports structure containing the SPI flash ports and clockblock

Function	ptp_server
Description	This function runs the PTP server. It requires one logical core and runs indefinitely
Type	void ptp_server(chanend mac_rx, chanend mac_tx, chanend ptp_clients[], int num_clients, enum ptp_server_type server_type)
Parameters	mac_rx chanend connected to the ethernet server (receive) mac_tx chanend connected to the ethernet server (transmit) ptp_clients an array of chanends to connect to clients of the ptp server num_clients The number of clients attached server_type The type of the server (PTP_GRANDMASTER_CAPABLE or PTP_SLAVE_ONLY)

Function	media_clock_server
Description	The media clock server.

Continued on next page

Type	<pre>void media_clock_server(server interface media_clock_if media_clock_ctl, chanend ?ptp_svr, chanend(& ?buf_ctl)[num_buf_ctl], unsigned num_buf_ctl, out buffered port:32 p_fs[])</pre>
Parameters	<p>media_clock_ctl server interface of type <code>media_clock_if</code> connected to the <code>avb_manager()</code> task and passed into <code>avb_init()</code></p> <p>ptp_svr chanend connected to the PTP server</p> <p>buf_ctl[] array of links to listener components requiring buffer management</p> <p>num_buf_ctl size of the <code>buf_ctl</code> array</p> <p>p_fs output port to drive PLL reference clock</p> <p>c_rx chanend connected to the ethernet server (receive)</p> <p>c_tx chanend connected to the ethernet server (transmit)</p> <p>c_ptp[] an array of chanends to connect to clients of the ptp server</p> <p>num_ptp The number of PTP clients attached</p> <p>server_type The type of the PTP server (<code>PTP_GRANDMASTER_CAPABLE</code> or <code>PTP_SLAVE_ONLY</code>)</p>

Function	avb_1722_listener
Description	<p>An AVB IEEE 1722 audio listener task. This task implements a listener. It takes IEEE 1722 packets from the ethernet MAC and splits them into output FIFOs. The buffer fill level of these streams is set in conjunction with communication to the media clock server. This task is dynamically configured using the AVB control API.</p>
Type	<pre>void avb_1722_listener(chanend c_mac_rx, chanend ?c_buf_ctl, chanend ?c_ptp_ctl, chanend c_listener_ctl, int num_streams)</pre>

Continued on next page

Parameters	<code>c_mac_rx</code>	receive link to the ethernet MAC
	<code>c_buf_ctl</code>	buffer control link to the media clock server
	<code>c_ptp_ctl</code>	PTP server link for retrieving PTP time info
	<code>c_listener_ctl</code>	channel to configure the listener (given to <code>avb_init()</code>)
	<code>num_streams</code>	the number of streams the unit will handle

Function	avb_1722_talker	
Description	An AVB IEEE 1722 audio talker task. This task implements a talker, taking media input FIFOs and combining them into 1722 packets to be sent to the ethernet component. It is dynamically configured using the AVB control API.	
Type	<pre>void avb_1722_talker(chanend c_ptp, chanend c_mac_tx, chanend c_talker_ctl, int num_streams)</pre>	
Parameters	<code>c_ptp</code>	link to the PTP timing server
	<code>c_mac_tx</code>	transmit link to the ethernet MAC
	<code>c_talker_ctl</code>	channel to configure the talker (given to <code>avb_init()</code>)
	<code>num_streams</code>	the number of streams the unit controls

7.2.2 Audio components

The following types are used by the AVB audio components:

Type	media_output_fifo_t
Description	This type provides a handle to a media output FIFO.

Type	<code>media_output_fifo_data_t</code>
Description	This type provides the data structure used by a media output FIFO.

Type	<code>media_input_fifo_t</code>
Description	This type provides a handle to a media input fifo.

Type	<code>media_input_fifo_data_t</code>
Description	This type provides the data structure used by a media input fifo.

The following functions implement AVB audio components:

Function	<code>init_media_input_fifos</code>						
Description	Initialize media input fifos. This function initializes media input FIFOs and ties the handles to their associated data structures. It should be called before the main component function on a task to setup the FIFOs.						
Type	<pre>void init_media_input_fifos(media_input_fifo_t ififos[], media_input_fifo_data_t ififo_data[], int n)</pre>						
Parameters	<table> <tr> <td><code>ififos</code></td> <td>an array of media input FIFO handles to initialize</td> </tr> <tr> <td><code>ififo_data</code></td> <td>an array of associated data structures</td> </tr> <tr> <td><code>n</code></td> <td>the number of FIFOs to initialize</td> </tr> </table>	<code>ififos</code>	an array of media input FIFO handles to initialize	<code>ififo_data</code>	an array of associated data structures	<code>n</code>	the number of FIFOs to initialize
<code>ififos</code>	an array of media input FIFO handles to initialize						
<code>ififo_data</code>	an array of associated data structures						
<code>n</code>	the number of FIFOs to initialize						

Function	<code>init_media_output_fifos</code>
Description	Initialize media output FIFOs. This function initializes media output FIFOs and ties the handles to their associated data structures. It should be called before the main component function on a thread to setup the FIFOs.

Continued on next page

Type	<pre>void init_media_output_fifos(media_output_fifo_t ofifos[], media_output_fifo_data_t ofifo_data[], int n)</pre>
Parameters	<p>ofifos an array of media output FIFO handles to initialize</p> <p>ofifo_data an array of associated data structures</p> <p>n the number of FIFOs to initialize</p>

Function	i2s_master
Description	<p>Input and output audio data using I2S format with the XCore acting as master. This function implements a task that can handle several synchronous I2S interfaces. It inputs and outputs 24-bit data packed into 32 bits. This function can handle up to 8in and 8out at 48KHz.</p>
Type	<pre>static void i2s_master(i2s_ports_t &ports, in buffered port:32(& ?p_din) [], int num_in, out buffered port:32(& ?p_dout) [], int num_out, int master_to_word_clock_ratio, media_input_fifo_t(& ?input_fifos) [], media_output_fifo_t(& ?output_fifos) [], chanend media_ctl, int clk_ctl_index)</pre>

Continued on next page

Parameters	<code>ports</code>	a reference to a structure of type <code>i2s_ports_t</code> containing the I2S port definitions
	<code>p_din</code>	array of ports to input data from
	<code>num_in</code>	number of input ports
	<code>p_dout</code>	array of ports to output data to
	<code>num_out</code>	number of output ports
	<code>master_to_word_clock_ratio</code>	the ratio of the master clock to the word clock; must be one of 128, 256 or 512
	<code>input_fifos</code>	a map from the inputs to local talker streams. The channels of the inputs are interleaved, for example, if you have two input ports, the map <code>{0,1,0,1}</code> would map to the two stereo local talker streams 0 and 1.
	<code>output_fifos</code>	a map from the outputs to local Listener streams
	<code>media_ctl</code>	a media clock control chanend connected to an <code>avb_manager()</code> task
	<code>clk_ctl_index</code>	the index of the clock control, default 0

Function	<code>media_output_fifo_to_xc_channel</code>
Description	Output audio streams from media fifos to an XC channel. This function outputs samples from several media output FIFOs over an XC channel over the streaming chanend <code>samples_out</code> . The protocol over the channel is that the thread expects a timestamp to be sent to it and then it will output <code>num_channels</code> samples, pulling from the <code>ofifos</code> array. It will then expect another timestamp before the next set of samples.
Type	<code>void</code> <code>media_output_fifo_to_xc_channel(streaming chanend samples_out, media_output_fifo_t ofifos[], int num_channels)</code>

Continued on next page

Parameters	<p><code>samples_out</code> the chanend on which samples are output</p> <p><code>ofifos</code> array of media output FIFOs to pull from</p> <p><code>num_channels</code> the number of channels (or FIFOs)</p>
-------------------	--

Function	media_output_fifo_to_xc_channel_split_lr
Description	<p>Output audio streams from media FIFOs to an XC channel, splitting left and right pairs.</p> <p>This function outputs samples from several media output FIFOs over an XC channel over the streaming chanend <code>samples_out</code>. The media FIFOs are assumed to be grouped in left/right stereo pairs which are then split.</p> <p>The protocol over the channel is that the thread expects a timestamp to be sent to it and then it will first output <code>num_channels/2</code> samples, pulling from all the even indexed elements of the <code>ofifos</code> array and then output all the odd elements. It will then expect another timestamp before the next set of samples.</p>
Type	<pre>void media_output_fifo_to_xc_channel_split_lr(streaming chanend samples_out, media_output_fifo_t output_fifos[], int num_channels)</pre>
Parameters	<p><code>samples_out</code> the chanend on which samples are output</p> <p><code>output_fifos</code> array of media output fifos to pull from</p> <p><code>num_channels</code> the number of channels (or FIFOs)</p>

7.2.3 Video components

The following functions implement AVB video components:

Function	tsi_output
Description	Hardware interface for the MPEG synchronous parallel interface.

Continued on next page

Type	<pre>void tsi_output(clock clk, out buffered port:32 p_data, in port p_clk, out buffered port:4 p_sync, out port p_valid, media_output_fifo_data_t &output_fifo)</pre>	
Parameters	clk	the clock block for the output
	p_data	MPEG data, an 8 bit port to be driven as a 32 bit buffered port
	p_clk	a one bit clock input with the 13.5 MHz MPEG SPI clock
	p_sync	MPEG start of packet but, a one bit port to be driven as a 4 bit buffered port
	p_valid	Valid bit for the data and sync bits. A one bit pin
	output_fifo	The media FIFO from which to draw samples

Function	tsi_input	
Description	Hardware interface for the MPEG synchronous parallel interface.	
Type	<pre>void tsi_input(clock clk, in buffered port:32 p_data, in port p_clk, in buffered port:4 p_sync, in port p_valid, ififo_t &ififo)</pre>	
Parameters	clk	a clock block for controlling the input
	p_data	MPEG data, an 8 bit port to be driven as a 32 bit buffered port
	p_clk	a one bit clock input with the 13.5 MHz MPEG SPI clock
	p_sync	MPEG start of packet but, a one bit port
	p_valid	Valid bit for the data and sync bits. A one bit pin
	ififo	The media FIFO from which to draw samples

7.3 AVB API

7.3.1 General control functions

Function	avb_get_control_packet	
Description	Receives an 802.1Qat SRP packet or an IEEE 1722 control packet. This function receives an AVB control packet from the ethernet MAC. It is selectable so can be used in a select statement as a case.	
Type	<pre>void avb_get_control_packet(chanend c_rx, unsigned int buf[], unsigned int &nbytes, unsigned int &port_num)</pre>	
Parameters	c_rx	chanend connected to the ethernet component
	buf	buffer to retrieve the packet into; buffer must have length at least MAX_AVB_CONTROL_PACKET_SIZE bytes
	nbytes	a reference parameter that is filled with the length of the received packet

Function	avb_process_srp_control_packet	
Description	Process an AVB SRP control packet. This function processes an 802.1Qat ethernet packet. This function should always be called on the buffer filled by avb_get_control_packet() .	
Type	<pre>void avb_process_srp_control_packet(client interface avb_interface i_avb, unsigned int buf[], unsigned len, chanend c_tx, unsigned int port_num)</pre>	
Parameters	i_avb	client interface of type avb_interface into avb_manager()
	buf	the incoming message buffer
	len	the length (in bytes) of the incoming buffer
	c_tx	chanend connected to the ethernet mac (TX)
	port_num	the id of the Ethernet interface the packet was received

Function	avb_process_1722_control_packet	
Description	Process an AVB 1722 control packet. This function processes a 1722 ethernet packet with the control data bit set. This function should always be called on the buffer filled by avb_get_control_packet() .	
Type	<pre>void avb_process_1722_control_packet(unsigned int buf[], unsigned nbytes, chanend c_tx, client interface avb_interface i_avb, client interface avb_1722_1_control_callbacks i_1722_1_entity, client interface spi_interface ?i_spi)</pre>	
Parameters	buf	the incoming message buffer
	nbytes	the length (in bytes) of the incoming buffer
	c_tx	chanend connected to the ethernet mac (TX)
	i_avb	client interface of type avb_interface into avb_manager()
	i_1722_1_entity	client interface of type avb_1722_1_control_callbacks
	i_spi	client interface of type spi_interface into avb_srp_task()

7.3.2 Multicast Address Allocation commands

Function	avb_1722_maap_request_addresses	
Description	Request a range of multicast addresses. This function requests a range of multicast addresses to use as destination addresses for IEEE 1722 streams. It starts the reservation process according to the 1722 MAAP protocol. If the reservation is successful it is reported via the status return value of avb_periodic() .	
Type	<pre>void avb_1722_maap_request_addresses(int num_addresses, char(& ?start_address) [])</pre>	

Continued on next page

Parameters	<p><code>num_addresses</code> number of addresses to try and reserve; will be reserved in a contiguous range</p> <p><code>start_address</code> an optional six byte array specifying the required start address of the range NOTE: must be within the MAAP reserved pool; if argument is null then the start address will be picked at random from the MAAP reserved pool</p>
-------------------	---

Function	avb_1722_maap_rerequest_addresses
Description	Re-request a claim on the existing address range. If there is a current address reservation, this will reset the state machine into the PROBE state, in order to cause the protocol to re-probe and re-allocate the addresses.
Type	<code>void avb_1722_maap_rerequest_addresses()</code>

Function	avb_1722_maap_relinquish_addresses
Description	Relinquish the reserved MAAP address range. This function abandons the claim to the reserved address range
Type	<code>void avb_1722_maap_relinquish_addresses()</code>

7.3.3 MAAP application hooks

Function	avb_talker_on_source_address_reserved
Description	MAAP has indicated that a multicast address has been successfully reserved for this Talker stream.
Type	<pre>void avb_talker_on_source_address_reserved(client interface avb_interface i_avb, int source_num, unsigned char mac_addr[6])</pre>

Continued on next page

Parameters	<code>i_avb</code>	client interface of type <code>avb_interface</code> into avb_manager()
	<code>source_num</code>	The local source ID of the Talker
	<code>mac_addr</code>	The destination MAC address reserved for this Talker

7.3.4 AVB Control API

Type	<code>device_media_clock_type_t</code>
Description	
Values	<code>DEVICE_MEDIA_CLOCK_INPUT_STREAM_DERIVED</code> <code>DEVICE_MEDIA_CLOCK_LOCAL_CLOCK</code>

Type	<code>device_media_clock_state_t</code>
Description	
Values	<code>DEVICE_MEDIA_CLOCK_STATE_DISABLED</code> <code>DEVICE_MEDIA_CLOCK_STATE_ENABLED</code>

Type	<code>avb_interface</code>						
Description							
Functions	<table border="1"> <tr> <td>Function</td> <td><code>initialise</code></td> </tr> <tr> <td>Description</td> <td></td> </tr> <tr> <td>Type</td> <td><code>void initialise(void)</code></td> </tr> </table>	Function	<code>initialise</code>	Description		Type	<code>void initialise(void)</code>
Function	<code>initialise</code>						
Description							
Type	<code>void initialise(void)</code>						

Continued on next page

Type	avb_interface (continued)	
	Function	_get_source_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	avb_source_info_t _get_source_info(unsigned source_num)
	Function	_set_source_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	void _set_source_info(unsigned source_num, avb_source_info_t info)
	Function	_get_sink_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	avb_sink_info_t _get_sink_info(unsigned sink_num)
	Function	_set_sink_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	void _set_sink_info(unsigned sink_num, avb_sink_info_t info)
	Function	_get_media_clock_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	media_clock_info_t _get_media_clock_info(unsigned clock_num)

Continued on next page

Type	avb_interface (continued)	
	Function	_set_media_clock_info
	Description	Intended for internal use within client interface get and set extensions only.
	Type	void _set_media_clock_info(unsigned clock_num, media_clock_info_t info)
	Function	get_source_format
	Description	Get the format of an AVB source.
	Type	int get_source_format(unsigned source_num, enum avb_stream_format_t &format, int &rate)
	Parameters	source_num the local source number format the format of the stream rate the sample rate of the stream in Hz
	Function	set_source_format
	Description	Set the format of an AVB source. The AVB source format covers the encoding and sample rate of the source. Currently the format is limited to a single encoding MBLA 24 bit signed integers. This setting will not take effect until the next time the source state moves from disabled to potential.
	Type	int set_source_format(unsigned source_num, enum avb_stream_format_t format, int rate)
	Parameters	source_num the local source number format the format of the stream rate the sample rate of the stream in Hz

Continued on next page

Type	avb_interface (continued)					
	Function	get_source_channels				
	Description	Get the channel count of an AVB source.				
	Type	int get_source_channels(unsigned source_num, int &channels)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 460 655 492">source_num</td> <td data-bbox="655 460 1192 492">the local source number</td> </tr> <tr> <td data-bbox="494 513 655 545">channels</td> <td data-bbox="655 513 1192 545">the number of channels</td> </tr> </table>	source_num	the local source number	channels	the number of channels
source_num	the local source number					
channels	the number of channels					
	Function	set_source_channels				
	Description	Set the channel count of an AVB source. Sets the number of channels in the stream. This setting will not take effect until the next time the source state moves from disabled to potential.				
	Type	int set_source_channels(unsigned source_num, int channels)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 890 655 922">source_num</td> <td data-bbox="655 890 1192 922">the local source number</td> </tr> <tr> <td data-bbox="494 943 655 975">channels</td> <td data-bbox="655 943 1192 975">the number of channels</td> </tr> </table>	source_num	the local source number	channels	the number of channels
source_num	the local source number					
channels	the number of channels					
	Function	get_source_sync				
	Description	Get the media clock of an AVB source.				
	Type	int get_source_sync(unsigned source_num, int &sync)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 1190 655 1222">source_num</td> <td data-bbox="655 1190 1192 1222">the local source number</td> </tr> <tr> <td data-bbox="494 1243 655 1275">sync</td> <td data-bbox="655 1243 1192 1275">the media clock number</td> </tr> </table>	source_num	the local source number	sync	the media clock number
source_num	the local source number					
sync	the media clock number					

Continued on next page

Type	avb_interface (continued)		
	Function	get_source_vlan	
	Description	Get the destination vlan of an AVB source.	
	Type	int get_source_vlan(unsigned source_num, int &vlan)	
	Parameters	source_num	the local source number
		vlan	the destination vlan id
	Function	set_source_vlan	
	Description	Set the destination vlan of an AVB source. Sets the vlan that the source will transmit on. This defaults to 2. This setting will not take effect until the next time the source state moves from disabled to potential.	
	Type	int set_source_vlan(unsigned source_num, int vlan)	
	Parameters	source_num	the local source number
		vlan	the destination vlan id
	Function	get_source_state	
	Description	Get the current state of an AVB source.	
Type	int get_source_state(unsigned source_num, enum avb_source_state_t &state)		
Parameters	source_num	the local source number	
	state	the state of the source	

Continued on next page

Type	avb_interface (continued)		
	Function	set_source_state	
	Description	Set the current state of an AVB source. Sets the current state of an AVB source. You cannot set the state to <code>ENABLED</code> . Changing the state to <code>AVB_SOURCE_STATE_POTENTIAL</code> turns the stream on and it will automatically change to <code>ENABLED</code> when connected to a listener and streaming.	
	Type	int set_source_state(unsigned source_num, enum avb_source_state_t state)	
	Parameters	source_num	the local source number
		state	the state of the source
	Function	get_source_map	
	Description	Get the channel map of an avb source.	
	Type	int get_source_map(unsigned source_num, int map[], int &len)	
Parameters	source_num	the local source number to set	
	map	the map, an array of integers giving the input FIFOs that make up the stream	
	len	the length of the map; should be equal to the number of channels in the stream	

Continued on next page

Type	avb_interface (continued)							
	Function	set_source_map						
	Description	Set the channel map of an avb source. Sets the channel map of a source i.e. the list of input FIFOs that constitute the stream. This setting will not take effect until the next time the source state moves from disabled to potential.						
	Type	int set_source_map(unsigned source_num, int map[len], unsigned len)						
	Parameters	<table border="0"> <tr> <td data-bbox="494 566 624 596">source_num</td> <td data-bbox="659 566 1194 596">the local source number to set</td> </tr> <tr> <td data-bbox="494 622 534 649">map</td> <td data-bbox="659 622 1194 675">the map, an array of integers giving the input FIFOs that make up the stream</td> </tr> <tr> <td data-bbox="494 702 534 728">len</td> <td data-bbox="659 702 1194 754">the length of the map; should be equal to the number of channels in the stream</td> </tr> </table>	source_num	the local source number to set	map	the map, an array of integers giving the input FIFOs that make up the stream	len	the length of the map; should be equal to the number of channels in the stream
source_num	the local source number to set							
map	the map, an array of integers giving the input FIFOs that make up the stream							
len	the length of the map; should be equal to the number of channels in the stream							
	Function	get_source_dest						
	Description	Get the destination address of an avb source.						
	Type	int get_source_dest(unsigned source_num, unsigned char addr[], int &len)						
	Parameters	<table border="0"> <tr> <td data-bbox="494 1024 624 1054">source_num</td> <td data-bbox="659 1024 1194 1054">the local source number</td> </tr> <tr> <td data-bbox="494 1081 544 1107">addr</td> <td data-bbox="659 1081 1194 1107">the destination address as an array of 6 bytes</td> </tr> <tr> <td data-bbox="494 1134 534 1160">len</td> <td data-bbox="659 1134 1194 1187">the length of the address, should always be equal to 6</td> </tr> </table>	source_num	the local source number	addr	the destination address as an array of 6 bytes	len	the length of the address, should always be equal to 6
source_num	the local source number							
addr	the destination address as an array of 6 bytes							
len	the length of the address, should always be equal to 6							

Continued on next page

Type	avb_interface (continued)							
	Function	set_source_dest						
	Description	Set the destination address of an avb source. Sets the destination MAC address of a source. This setting will not take effect until the next time the source state moves from disabled to potential.						
	Type	int set_source_dest(unsigned source_num, unsigned char addr[len], unsigned len)						
	Parameters	<table border="0"> <tr> <td data-bbox="490 539 645 569">source_num</td> <td data-bbox="645 539 1189 569">the local source number</td> </tr> <tr> <td data-bbox="490 592 551 622">addr</td> <td data-bbox="645 592 1189 622">the destination address as an array of 6 bytes</td> </tr> <tr> <td data-bbox="490 645 537 675">len</td> <td data-bbox="645 645 1189 675">the length of the address, should always be equal to 6</td> </tr> </table>	source_num	the local source number	addr	the destination address as an array of 6 bytes	len	the length of the address, should always be equal to 6
source_num	the local source number							
addr	the destination address as an array of 6 bytes							
len	the length of the address, should always be equal to 6							
	Function	get_source_id						
	Description	Get the stream id that an AVB source is using.						
	Type	int get_source_id(unsigned source_num, unsigned int stream_id[2])						
	Parameters	<table border="0"> <tr> <td data-bbox="490 941 645 971">source_num</td> <td data-bbox="645 941 1189 971">the number of the source</td> </tr> <tr> <td data-bbox="490 994 604 1024">stream_id</td> <td data-bbox="645 994 1189 1024">int array containing the 64-bit ID of the stream</td> </tr> </table>	source_num	the number of the source	stream_id	int array containing the 64-bit ID of the stream		
source_num	the number of the source							
stream_id	int array containing the 64-bit ID of the stream							
	Function	get_sink_id						
	Description	Get the stream id that an AVB sink listens to.						
	Type	int get_sink_id(unsigned sink_num, unsigned int stream_id[2])						
	Parameters	<table border="0"> <tr> <td data-bbox="490 1294 645 1324">sink_num</td> <td data-bbox="645 1294 1189 1324">the number of the sink</td> </tr> <tr> <td data-bbox="490 1347 604 1377">stream_id</td> <td data-bbox="645 1347 1189 1377">int array containing the 64-bit ID of the stream</td> </tr> </table>	sink_num	the number of the sink	stream_id	int array containing the 64-bit ID of the stream		
sink_num	the number of the sink							
stream_id	int array containing the 64-bit ID of the stream							

Continued on next page

Type	avb_interface (continued)							
	Function	set_sink_id						
	Description	Set the stream id that an AVB sink listens to. Sets the stream id that an AVB sink listens to. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_id(unsigned sink_num, unsigned int stream_id[2])						
	Parameters	<table border="0"> <tr> <td style="padding-right: 20px;">sink_num</td> <td>the number of the sink</td> </tr> <tr> <td>stream_id</td> <td>int array containing the 64-bit of the stream</td> </tr> </table>	sink_num	the number of the sink	stream_id	int array containing the 64-bit of the stream		
sink_num	the number of the sink							
stream_id	int array containing the 64-bit of the stream							
	Function	get_sink_format						
	Description	Get the format of an AVB sink.						
	Type	int get_sink_format(unsigned sink_num, enum avb_stream_format_t &format, int &rate)						
	Parameters	<table border="0"> <tr> <td style="padding-right: 20px;">sink_num</td> <td>the local sink number</td> </tr> <tr> <td>format</td> <td>the format of the stream</td> </tr> <tr> <td>rate</td> <td>the sample rate of the stream in Hz</td> </tr> </table>	sink_num	the local sink number	format	the format of the stream	rate	the sample rate of the stream in Hz
sink_num	the local sink number							
format	the format of the stream							
rate	the sample rate of the stream in Hz							

Continued on next page

Type	avb_interface (continued)							
	Function	set_sink_format						
	Description	Set the format of an AVB sink. The AVB sink format covers the encoding and sample rate of the sink. Currently the format is limited to a single encoding MBLA 24 bit signed integers. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_format(unsigned sink_num, enum avb_stream_format_t format, int rate)						
	Parameters	<table border="0"> <tr> <td>sink_num</td> <td>the local sink number</td> </tr> <tr> <td>format</td> <td>the format of the stream</td> </tr> <tr> <td>rate</td> <td>the sample rate of the stream in Hz</td> </tr> </table>	sink_num	the local sink number	format	the format of the stream	rate	the sample rate of the stream in Hz
sink_num	the local sink number							
format	the format of the stream							
rate	the sample rate of the stream in Hz							
	Function	get_sink_channels						
	Description	Get the channel count of an AVB sink.						
	Type	int get_sink_channels(unsigned sink_num, int &channels)						
	Parameters	<table border="0"> <tr> <td>sink_num</td> <td>the local sink number</td> </tr> <tr> <td>channels</td> <td>the number of channels</td> </tr> </table>	sink_num	the local sink number	channels	the number of channels		
sink_num	the local sink number							
channels	the number of channels							
	Function	set_sink_channels						
	Description	Set the channel count of an AVB sink. Sets the number of channels in the stream. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_channels(unsigned sink_num, int channels)						
	Parameters	<table border="0"> <tr> <td>sink_num</td> <td>the local sink number</td> </tr> <tr> <td>channels</td> <td>the number of channels</td> </tr> </table>	sink_num	the local sink number	channels	the number of channels		
sink_num	the local sink number							
channels	the number of channels							

Continued on next page

Type	avb_interface (continued)					
	Function	get_sink_sync				
	Description	Get the media clock of an AVB sink.				
	Type	int get_sink_sync(unsigned sink_num, int &sync)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 407 645 439">sink_num</td> <td data-bbox="658 407 1194 439">the local sink number</td> </tr> <tr> <td data-bbox="494 456 645 488">sync</td> <td data-bbox="658 456 1194 488">the media clock number</td> </tr> </table>	sink_num	the local sink number	sync	the media clock number
sink_num	the local sink number					
sync	the media clock number					
	Function	set_sink_sync				
	Description	Set the media clock of an AVB sink. Sets the media clock of the stream.				
	Type	int set_sink_sync(unsigned sink_num, int sync)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 732 645 763">sink_num</td> <td data-bbox="658 732 1194 763">the local sink number</td> </tr> <tr> <td data-bbox="494 781 645 813">sync</td> <td data-bbox="658 781 1194 813">the media clock number</td> </tr> </table>	sink_num	the local sink number	sync	the media clock number
sink_num	the local sink number					
sync	the media clock number					
	Function	get_sink_vlan				
	Description	Get the virtual lan id of an AVB sink.				
	Type	int get_sink_vlan(unsigned sink_num, int &vlan)				
	Parameters	<table border="0"> <tr> <td data-bbox="494 1031 645 1063">sink_num</td> <td data-bbox="658 1031 1194 1063">the number of the sink</td> </tr> <tr> <td data-bbox="494 1081 645 1113">vlan</td> <td data-bbox="658 1081 1194 1113">the vlan id of the sink</td> </tr> </table>	sink_num	the number of the sink	vlan	the vlan id of the sink
sink_num	the number of the sink					
vlan	the vlan id of the sink					

Continued on next page

Type	avb_interface (continued)							
	Function	set_sink_vlan						
	Description	Set the virtual lan id of an AVB sink. Sets the vlan id of the incoming stream. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_vlan(unsigned sink_num, int vlan)						
	Parameters	<table border="0"> <tr> <td style="padding-right: 20px;">sink_num</td> <td>the number of the sink</td> </tr> <tr> <td style="padding-right: 20px;">vlan</td> <td>the vlan id of the sink</td> </tr> </table>	sink_num	the number of the sink	vlan	the vlan id of the sink		
sink_num	the number of the sink							
vlan	the vlan id of the sink							
	Function	get_sink_addr						
	Description	Get the incoming destination mac address of an avb sink.						
	Type	int get_sink_addr(unsigned sink_num, unsigned char addr[], int &len)						
	Parameters	<table border="0"> <tr> <td style="padding-right: 20px;">sink_num</td> <td>The local sink number</td> </tr> <tr> <td style="padding-right: 20px;">addr</td> <td>The mac address as an array of 6 bytes.</td> </tr> <tr> <td style="padding-right: 20px;">len</td> <td>The length of the address, should always be equal to 6.</td> </tr> </table>	sink_num	The local sink number	addr	The mac address as an array of 6 bytes.	len	The length of the address, should always be equal to 6.
sink_num	The local sink number							
addr	The mac address as an array of 6 bytes.							
len	The length of the address, should always be equal to 6.							

Continued on next page

Type	avb_interface (continued)							
	Function	set_sink_addr						
	Description	Set the incoming destination mac address of an avb sink. Set the incoming destination mac address of a sink. This needs to be set if the address is a multicast address so the endpoint can register for that multicast group with the switch. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_addr(unsigned sink_num, unsigned char addr [len], unsigned len)						
	Parameters	<table border="0"> <tr> <td data-bbox="494 619 602 650">sink_num</td> <td data-bbox="659 619 911 650">The local sink number</td> </tr> <tr> <td data-bbox="494 672 548 703">addr</td> <td data-bbox="659 672 1107 703">The mac address as an array of 6 bytes.</td> </tr> <tr> <td data-bbox="494 725 534 756">len</td> <td data-bbox="659 725 1161 781">The length of the address, should always be equal to 6.</td> </tr> </table>	sink_num	The local sink number	addr	The mac address as an array of 6 bytes.	len	The length of the address, should always be equal to 6.
	sink_num	The local sink number						
	addr	The mac address as an array of 6 bytes.						
	len	The length of the address, should always be equal to 6.						
	Function	get_sink_state						
Description	Get the state of an AVB sink.							
Type	int get_sink_state(unsigned sink_num, enum avb_sink_state_t &state)							
Parameters	<table border="0"> <tr> <td data-bbox="494 1023 602 1054">sink_num</td> <td data-bbox="659 1023 919 1054">the number of the sink</td> </tr> <tr> <td data-bbox="494 1076 561 1107">state</td> <td data-bbox="659 1076 888 1107">the state of the sink</td> </tr> </table>	sink_num	the number of the sink	state	the state of the sink			
sink_num	the number of the sink							
state	the state of the sink							

Continued on next page

Type	avb_interface (continued)	
	Function	set_sink_state
	Description	Set the state of an AVB sink. Sets the current state of an AVB sink. You cannot set the state to <i>ENABLED</i> . Changing the state to <i>POTENTIAL</i> turns the stream on and it will automatically change to <i>ENABLED</i> when connected to a talker and receiving samples.
	Type	int set_sink_state(unsigned sink_num, enum avb_sink_state_t state)
	Parameters	sink_num the number of the sink state the state of the sink
	Function	get_sink_map
	Description	Get the map of an AVB sink.
	Type	int get_sink_map(unsigned sink_num, int map[], int &len)
	Parameters	sink_num the number of the sink map array containing the media output FIFOs that the stream will be split into len the length of the map; should equal to the number of channels in the stream

Continued on next page

Type	avb_interface (continued)							
	Function	set_sink_map						
	Description	Set the map of an AVB sink. Sets the map i.e. the mapping from the 1722 stream to output FIFOs. This setting will not take effect until the next time the sink state moves from disabled to potential.						
	Type	int set_sink_map(unsigned sink_num, int map[len], unsigned len)						
	Parameters	<table border="0"> <tr> <td data-bbox="494 566 655 599">sink_num</td> <td data-bbox="655 566 1192 599">the number of the sink</td> </tr> <tr> <td data-bbox="494 622 655 656">map</td> <td data-bbox="655 622 1192 656">array containing the media output FIFOs that the stream will be split into</td> </tr> <tr> <td data-bbox="494 702 655 735">len</td> <td data-bbox="655 702 1192 735">the length of the map; should equal to the number of channels in the stream</td> </tr> </table>	sink_num	the number of the sink	map	array containing the media output FIFOs that the stream will be split into	len	the length of the map; should equal to the number of channels in the stream
sink_num	the number of the sink							
map	array containing the media output FIFOs that the stream will be split into							
len	the length of the map; should equal to the number of channels in the stream							
	Function	get_device_media_clock_rate						
	Description	Get the rate of a media clock.						
	Type	int get_device_media_clock_rate(int clock_num, int &rate)						
	Parameters	<table border="0"> <tr> <td data-bbox="494 1023 655 1056">clock_num</td> <td data-bbox="655 1023 1192 1056">the number of the media clock</td> </tr> <tr> <td data-bbox="494 1079 655 1113">rate</td> <td data-bbox="655 1079 1192 1113">the rate of the clock in Hz</td> </tr> </table>	clock_num	the number of the media clock	rate	the rate of the clock in Hz		
clock_num	the number of the media clock							
rate	the rate of the clock in Hz							

Continued on next page

Type	avb_interface (continued)	
	Function	set_device_media_clock_rate
	Description	Set the rate of a media clock. Sets the rate of the media clock.
	Type	int set_device_media_clock_rate(int clock_num, int rate)
	Parameters	clock_num the number of the media clock rate the rate of the clock in Hz
	Function	get_device_media_clock_state
	Description	Get the state of a media clock.
	Type	int get_device_media_clock_state(int clock_num, enum device_media_clock_state_t &state)
	Parameters	clock_num the number of the media clock state the state of the clock
	Function	set_device_media_clock_state
	Description	Set the state of a media clock. This function can be used to enabled/disable a media clock.
	Type	int set_device_media_clock_state(int clock_num, enum device_media_clock_state_t state)
	Parameters	clock_num the number of the media clock state the state of the clock

Continued on next page

Type	avb_interface (continued)	
	Function	get_device_media_clock_source
	Description	Get the source of a media clock.
	Type	int get_device_media_clock_source(int clock_num, int &source)
	Parameters	clock_num the number of the media clock source the output FIFO number to base the clock on
	Function	set_device_media_clock_source
	Description	Set the source of a media clock. For clocks that are derived from an output FIFO. This function gets/sets which FIFO the clock should be derived from.
	Type	int set_device_media_clock_source(int clock_num, int source)
	Parameters	clock_num the number of the media clock source the output FIFO number to base the clock on
	Function	get_device_media_clock_type
	Description	Get the type of a media clock.
	Type	int get_device_media_clock_type(int clock_num, enum device_media_clock_type_t &clock_type)
	Parameters	clock_num the number of the media clock clock_type the type of the clock

Continued on next page

Type	avb_interface (continued)	
	Function	set_device_media_clock_type
	Description	Set the type of a media clock.
	Type	int set_device_media_clock_type(int clock_num, enum device_media_clock_type_t clock_type)
	Parameters	clock_num the number of the media clock clock_type the type of the clock

7.3.5 1722.1 Controller commands

Function	avb_1722_1_controller_connect	
Description	Setup a new stream connection between a Talker and Listener entity. The Controller shall send a CONNECT_RX_COMMAND to the Listener Entity. The Listener Entity shall then send a CONNECT_TX_COMMAND to the Talker Entity.	
Type	void avb_1722_1_controller_connect(const_guid_ref_t talker_guid, const_guid_ref_t listener_guid, int talker_id, int listener_id, chanend c_tx)	
Parameters	talker_guid	the GUID of the Talker being targeted by the command
	listener_guid	the GUID of the Listener being targeted by the command
	talker_id	the unique id of the Talker stream source to connect. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor
	listener_id	the unique id of the Listener stream source to connect. For entities using AEM, this corresponds to the id of the STREAM_INPUT descriptor
	c_tx	a transmit chanend to the Ethernet server

Function	avb_1722_1_controller_disconnect										
Description	Disconnect an existing stream connection between a Talker and Listener entity. The Controller shall send a DISCONNECT_RX_COMMAND to the Listener Entity. The Listener Entity shall then send a DISCONNECT_TX_COMMAND to the Talker Entity.										
Type	void avb_1722_1_controller_disconnect(const_guid_ref_t talker_guid, const_guid_ref_t listener_guid, int talker_id, int listener_id, chanend c_tx)										
Parameters	<table border="0"> <tr> <td>talker_guid</td> <td>the GUID of the Talker being targeted by the command</td> </tr> <tr> <td>listener_guid</td> <td>the GUID of the Listener being targeted by the command</td> </tr> <tr> <td>talker_id</td> <td>the unique id of the Talker stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor</td> </tr> <tr> <td>listener_id</td> <td>the unique id of the Listener stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_INPUT descriptor</td> </tr> <tr> <td>c_tx</td> <td>a transmit chanend to the Ethernet server</td> </tr> </table>	talker_guid	the GUID of the Talker being targeted by the command	listener_guid	the GUID of the Listener being targeted by the command	talker_id	the unique id of the Talker stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor	listener_id	the unique id of the Listener stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_INPUT descriptor	c_tx	a transmit chanend to the Ethernet server
talker_guid	the GUID of the Talker being targeted by the command										
listener_guid	the GUID of the Listener being targeted by the command										
talker_id	the unique id of the Talker stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor										
listener_id	the unique id of the Listener stream source to disconnect. For entities using AEM, this corresponds to the id of the STREAM_INPUT descriptor										
c_tx	a transmit chanend to the Ethernet server										

Function	avb_1722_1_controller_disconnect_all_listeners				
Description	Disconnect all Listener sinks currently connected to the Talker stream source with talker_id.				
Type	void avb_1722_1_controller_disconnect_all_listeners(int talker_id, chanend c_tx)				
Parameters	<table border="0"> <tr> <td>talker_id</td> <td>the unique id of the Talker stream source to disconnect its listeners. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor</td> </tr> <tr> <td>c_tx</td> <td>a transmit chanend to the Ethernet server</td> </tr> </table>	talker_id	the unique id of the Talker stream source to disconnect its listeners. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor	c_tx	a transmit chanend to the Ethernet server
talker_id	the unique id of the Talker stream source to disconnect its listeners. For entities using AEM, this corresponds to the id of the STREAM_OUTPUT descriptor				
c_tx	a transmit chanend to the Ethernet server				

Function	avb_1722_1_controller_disconnect_talker
Description	Disconnect the Talker source currently connected to the Listener stream sink with <code>listener_id</code> .
Type	void avb_1722_1_controller_disconnect_talker(int listener_id, chanend c_tx)
Parameters	<code>listener_id</code> the unique id of the Listener stream source to disconnect its Talker. For entities using AEM, this corresponds to the id of the STREAM_INPUT descriptor <code>c_tx</code> a transmit chanend to the Ethernet server

7.3.6 1722.1 Discovery commands

Function	avb_1722_1_adp_announce
Description	Start advertising information about this entity via ADP.
Type	void avb_1722_1_adp_announce(void)

Function	avb_1722_1_adp_depart
Description	Stop advertising information about this entity via ADP.
Type	void avb_1722_1_adp_depart(void)

Function	avb_1722_1_adp_discover
Description	Ask to discover the information for a specific entity GUID.
Type	void avb_1722_1_adp_discover(const_guid_ref_t guid)
Parameters	<code>guid</code> The GUID of the entity to discover

Function	avb_1722_1_adp_discover_all
Description	Ask to discover all available entities via ADP.
Type	void avb_1722_1_adp_discover_all(void)

Function	avb_1722_1_entity_database_flush
Description	Remove all discovered entities from the database.
Type	void avb_1722_1_entity_database_flush(void)

7.3.7 1722.1 application hooks

These hooks are called on events that can be acted upon by the application. They can be overridden by user defined hooks of the same name to perform custom functionality not present in the core stack.

Type	avb_1722_1_entity_record
Description	
Fields	<p>guid_t guid</p> <p>unsigned int vendor_id</p> <p>unsigned int entity_model_id</p> <p>unsigned int capabilities</p> <p>unsigned short talker_stream_sources</p> <p>unsigned short talker_capabilities</p> <p>unsigned short listener_stream_sinks</p>

Continued on next page

	<pre> unsigned short listener_capabilities unsigned int controller_capabilities unsigned int available_index gmid_t gptp_grandmaster_id unsigned char gptp_domain_number unsigned short identify_control_index unsigned int association_id unsigned timeout </pre>
--	---

Function	avb_entity_on_new_entity_available	
Description	A new AVDECC entity has advertised itself as available. It may be an entity starting up or a previously seen entity that had timed out.	
Type	<pre> void avb_entity_on_new_entity_available(client interface avb_interface i_avb, const_guid_ref_t my_guid, avb_1722_1_entity_record *entity, chanend c_tx) </pre>	
Parameters	i_avb	client interface of type avb_interface into avb_manager()
	my_guid	The GUID of this entity
	entity	The information advertised by the remote entity
	c_tx	A transmit channel end to the Ethernet server

Function	avb_talker_on_listener_connect
Description	A Controller has indicated that a Listener is connecting to this Talker stream source.
Type	void avb_talker_on_listener_connect(client interface avb_interface i_avb, int source_num, const_guid_ref_t listener_guid)
Parameters	i_avb client interface of type avb_interface into avb_manager() source_num The local id of the Talker stream source listener_guid The GUID of the Listener entity that is connecting

Function	avb_talker_on_listener_disconnect
Description	A Controller has indicated that a Listener is disconnecting from this Talker stream source.
Type	void avb_talker_on_listener_disconnect(client interface avb_interface i_avb, int source_num, const_guid_ref_t listener_guid, int connection_count)
Parameters	i_avb client interface of type avb_interface into avb_manager() source_num The local id of the Talker stream source listener_guid The GUID of the Listener entity that is disconnecting connection_count The number of connections a Talker thinks it has on it's stream source, i.e. the number of connect TX stream commands it has received less the number of disconnect TX stream commands it has received. This number may not be accurate since an AVDECC Entity may not have sent a disconnect command if the cable was disconnected or the AVDECC Entity abruptly powered down.

Function	avb_listener_on_talker_connect	
Description	A Controller has indicated to connect this Listener sink to a Talker stream.	
Type	<pre>avb_1722_1_acmp_status_t avb_listener_on_talker_connect(client interface avb_interface i_avb, int sink_num, const_guid_ref_t talker_guid, unsigned char dest_addr[6], unsigned int stream_id[2], unsigned short vlan_id, const_guid_ref_t my_guid)</pre>	
Parameters	i_avb	client interface of type avb_interface into avb_manager()
	sink_num	The local id of the Listener stream sink
	talker_guid	The GUID of the Talker entity that is connecting
	dest_addr	The destination MAC address of the Talker stream
	stream_id	The 64 bit Stream ID of the Talker stream
	vlan_id	The VLAN ID of the Talker stream
	my_guid	The GUID of this entity

Function	avb_listener_on_talker_disconnect	
Description	A Controller has indicated to disconnect this Listener sink from a Talker stream.	
Type	<pre>void avb_listener_on_talker_disconnect(client interface avb_interface i_avb, int sink_num, const_guid_ref_t talker_guid, unsigned char dest_addr[6], unsigned int stream_id[2], const_guid_ref_t my_guid)</pre>	

Continued on next page

Parameters		
	<code>i_avb</code>	client interface of type <code>avb_interface</code> into <code>avb_manager()</code>
	<code>sink_num</code>	The local id of the Listener stream sink
	<code>talker_guid</code>	The GUID of the Talker entity that is disconnecting
	<code>dest_addr</code>	The destination MAC address of the Talker stream
	<code>stream_id</code>	The 64 bit Stream ID of the Talker stream
	<code>my_guid</code>	The GUID of this entity

7.4 1722.1 descriptors

The XMOS AVB reference design provides an AVDECC Entity Model (AEM) consisting of descriptors to describe the internal components of the Entity. For a complete overview of AEM, see section 7 of the 1722.1 specification.

An AEM descriptor is a fixed field structure followed by variable length data which describes an object in the AEM Entity model. The maximum length of a descriptor is 508 octets.

All descriptors share two common fields which are used to uniquely identify a descriptor by a type and an index. AEM defines a number of descriptors for specific parts of the Entity model. The descriptor types that XMOS currently provide in the reference design are listed in the table below.

7.4.1 Editing descriptors

The descriptors are declared in the a header configuration file named `aem_descriptors.h.in` within the `src/` directory of the application. The XMOS Reference column in the table refers to the array names of the descriptors in this file.

This file is post-processed by a script in the build stage to expand strings to 64 octet padded with zeros.

Name	Description	XMOS Reference
ENTITY	This is the top level descriptor defining the Entity.	desc_entity
CONFIGURATION	This is the descriptor defining a configuration of the Entity.	desc_configuration_0
AUDIO_UNIT	This is the descriptor defining an audio unit.	desc_audio_unit_0
STREAM_INPUT	This is the descriptor defining an input stream to the Entity.	desc_stream_input_0
STREAM_OUTPUT	This is the descriptor defining an output stream from the Entity.	desc_stream_output_0
JACK_INPUT	This is the descriptor defining an input jack on the Entity.	desc_jack_input_0
JACK_OUTPUT	This is the descriptor defining an output jack on the Entity.	desc_jack_output_0
AVB_INTERFACE	This is the descriptor defining an AVB interface.	desc_avb_interface_0
CLOCK_SOURCE	This is the descriptor describing a clock source.	desc_clock_source_0..1
LOCALE	This is the descriptor defining a locale.	desc_locale_0
STRINGS	This is the descriptor defining localized strings.	desc_strings_0
STREAM_PORT_INPUT	This is the descriptor defining an input stream port on a unit.	desc_stream_port_input_0
STREAM_PORT_OUTPUT	This is the descriptor defining an output stream port on a unit.	desc_stream_port_output_0
EXTERNAL_PORT_INPUT	This is the descriptor defining an input external port on a unit.	desc_external_input_port_0
EXTERNAL_PORT_OUTPUT	This is the descriptor defining an output external port on a unit.	desc_external_output_port_0
AUDIO_CLUSTER	This is the descriptor defining a cluster of channels within an audio stream.	desc_audio_cluster_0..N
AUDIO_MAP	This is the descriptor defining the mapping between the channels of an audio stream and the channels of the audio port.	desc_audio_map_0..N
CLOCK_DOMAIN	This is the descriptor describing a clock domain.	desc_clock_domain_0

7.4.2 Adding and removing descriptors

Descriptors are indexed by a descriptor list named `aem_descriptor_list` in the `aem_descriptors.h` file.

The format for this list is as follows:

```
Descriptor type
Number of descriptors of type (N)
Size of descriptor 0 (bytes)
Address of descriptor 0
...
Size of descriptor N (bytes)
Address of descriptor N
```

For example:

```
AEM_ENTITY_TYPE, 1, sizeof(desc_entity), (unsigned)desc_entity
```

7.5 PTP client API

The PTP client API can be used if you want extra information about the PTP time domain. An application does not need to directly use this to control the AVB endpoint since the talker, listener and media clock server units communicate with the PTP server directly.

7.5.1 Time data structures

Type	ptp_timestamp
Description	This type represents a timestamp in the gptp clock domain.
Fields	<pre>unsigned int seconds unsigned int nanoseconds</pre>

7.5.2 Getting PTP time information

Type	ptp_time_info
Description	This type is used to relate local XCore time with gptp time. It can be retrieved from the PTP server using the ptp_get_time_info() function.

Type	ptp_time_info_mod64
Description	This structure is used to relate local XCore time with the least significant 64 bits of gptp time. The 64 bits of time is the PTP time in nanoseconds from the epoch. It can be retrieved from the PTP server using the ptp_get_time_info_mod64() function.

Function	ptp_get_time_info
Description	Retrieve port propagation delay from the ptp server.
Type	void ptp_get_time_info(chanend ptp_server, ptp_time_info &info)
Parameters	ptp_server chanend connected to the ptp_server pdelay unsigned int with delay in ns

Function	ptp_get_time_info_mod64
Description	Retrieve time information from the ptp server. This function gets an up-to-date structure of type ptp_time_info_mod64 to use to convert local time to ptp time (modulo 64 bits).
Type	void ptp_get_time_info_mod64(chanend ?ptp_server, ptp_time_info_mod64 &info)
Parameters	ptp_server chanend connected to the ptp_server info structure to be filled with time information

Function	ptp_request_time_info
Description	This function requests a ptp_time_info structure from the PTP server. This is an asynchronous call so needs to be completed later with a call to ptp_get_requested_time_info() .
Type	void ptp_request_time_info(chanend ptp_server)

Continued on next page

Parameters	<code>ptp_server</code> <code>chanend</code> connecting to the ptp server
-------------------	---

Function	<code>ptp_request_time_info_mod64</code>
Description	This function requests a <code>ptp_time_info_mod64</code> structure from the PTP server. This is an asynchronous call so needs to be completed later with a call to <code>ptp_get_requested_time_info_mod64()</code> .
Type	<code>void</code> <code>ptp_request_time_info_mod64(chanend ptp_server)</code>
Parameters	<code>ptp_server</code> <code>chanend</code> connecting to the PTP server

Function	<code>ptp_get_requested_time_info</code>
Description	This function receives a <code>ptp_time_info</code> structure from the PTP server. This completes an asynchronous transaction initiated with a call to <code>ptp_request_time_info()</code> . The function can be placed in a select case which will activate when the PTP server is ready to send.
Type	<code>void</code> <code>ptp_get_requested_time_info(chanend ptp_server, ptp_time_info &info)</code>
Parameters	<code>ptp_server</code> <code>chanend</code> connecting to the PTP server <code>info</code> a reference parameter to be filled with the time information structure

Function	<code>ptp_get_requested_time_info_mod64</code>
Description	This function receives a <code>ptp_time_info_mod64</code> structure from the PTP server. This completes an asynchronous transaction initiated with a call to <code>ptp_request_time_info_mod64()</code> . The function can be placed in a select case which will activate when the PTP server is ready to send.
Type	<code>void</code> <code>ptp_get_requested_time_info_mod64(chanend ptp_server, ptp_time_info_mod64 &info)</code>

Continued on next page

Parameters	<code>ptp_server</code>	chanend connecting to the PTP server
	<code>info</code>	a reference parameter to be filled with the time information structure

7.5.3 Converting timestamps

Function	local_timestamp_to_ptp	
Description	Convert a timestamp from the local XCore timer to PTP time. This function takes a 32-bit timestamp taken from an XCore timer and converts it to PTP time.	
Type	<pre>void local_timestamp_to_ptp(ptp_timestamp &ptp_ts, unsigned local_ts, ptp_time_info &info)</pre>	
Parameters	<code>ptp_ts</code>	the PTP timestamp structure to be filled with the converted time
	<code>local_ts</code>	the local timestamp to be converted
	<code>info</code>	a time information structure retrieved from the ptp server

Function	local_timestamp_to_ptp_mod32	
Description	Convert a timestamp from the local XCore timer to the least significant 32 bits of PTP time. This function takes a 32-bit timestamp taken from an XCore timer and converts it to the least significant 32 bits of global PTP time.	
Type	<pre>unsigned local_timestamp_to_ptp_mod32(unsigned local_ts, ptp_time_info_mod64 &info)</pre>	
Parameters	<code>local_ts</code>	the local timestamp to be converted
	<code>info</code>	a time information structure retrieved from the PTP server
Returns	the least significant 32-bits of ptp time in nanoseconds	

Function	ptp_timestamp_to_local				
Description	Convert a PTP timestamp to a local XCore timestamp. This function takes a PTP timestamp and converts it to a local 32-bit timestamp that is related to the XCore timer.				
Type	unsigned <code>ptp_timestamp_to_local(ptp_timestamp &ts, ptp_time_info &info)</code>				
Parameters	<table><tr><td><code>ts</code></td><td>the PTP timestamp to convert</td></tr><tr><td><code>info</code></td><td>a time information structure retrieved from the PTP server.</td></tr></table>	<code>ts</code>	the PTP timestamp to convert	<code>info</code>	a time information structure retrieved from the PTP server.
<code>ts</code>	the PTP timestamp to convert				
<code>info</code>	a time information structure retrieved from the PTP server.				
Returns	the local timestamp				



Copyright © 2014, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS and the XMOS logo are registered trademarks of Xmos Ltd. in the United Kingdom and other countries, and may not be used without written permission. All other trademarks are property of their respective owners. Where those designations appear in this book, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.