XMOS AVB-DC Design Guide

XMOS®

Table of Contents

1 Overview

1.1 Summary

The XMOS Audio Video Bridging Daisy Chain endpoint (AVB-DC) is a two-port Ethernet MAC relay implementation that provides time-synchronized, low latency streaming services through IEEE 802 networks.
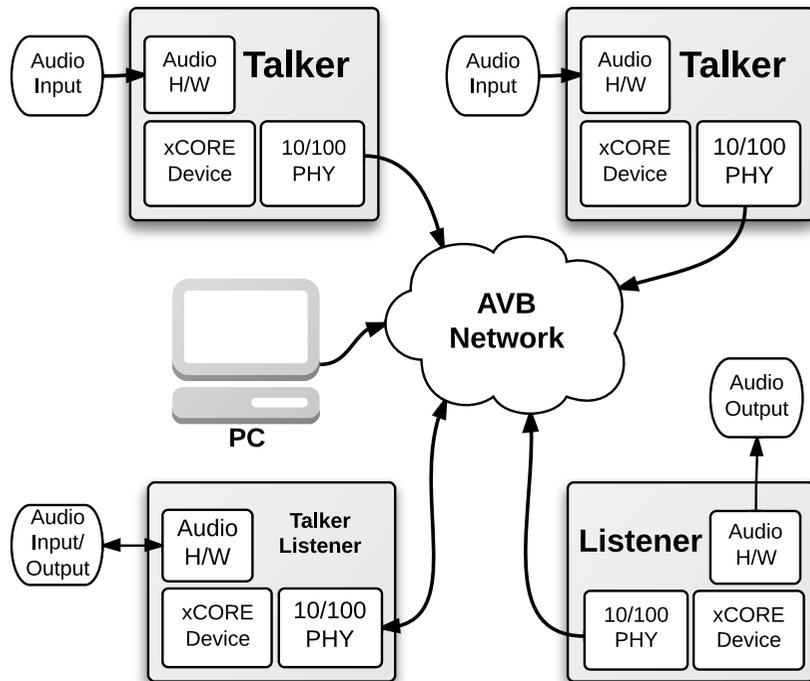
1.1.1 XMOS AVB-DC Key Features

▶ 2 x 100 Mbit/s full duplex Ethernet interface via MII

▶ Support for 1722.1 discovery, enumeration, command and control: ADP, AECP (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-DC 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
Requirements	
Development Tools	xTIMEComposer suite v13.0.2 or later
Ethernet	2 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

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 document[1] 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 document[2] 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 document[3].

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 document[4] 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 document[5] 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 document[6] is available for purchase from the IEEE website.

---

[3] http://standards.ieee.org/getieee802/download/802.1Q-2011.pdf
[4] http://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 evaluation and development of AVB-DC applications the following XMOS development platform is recommended:

▶ XK-SK-AVB-DC sliceKIT development platform[7]

This development kit consists of:

▶ 2x XP-SKC-L2 core board

▶ 2x XA-SK-AUDIO-PLL

▶ 2x XA-SK-E100

▶ 2x XA-XTAG2

▶ 2x XA-SK-XTAG2

▶ 2x PSU (12V 2.5A)

▶ 2x USB extension cable 1m

▶ 2x Ethernet cable 1m

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

---

[7] http://www.xmos.com/products/reference-designs/avb

5 System description

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.
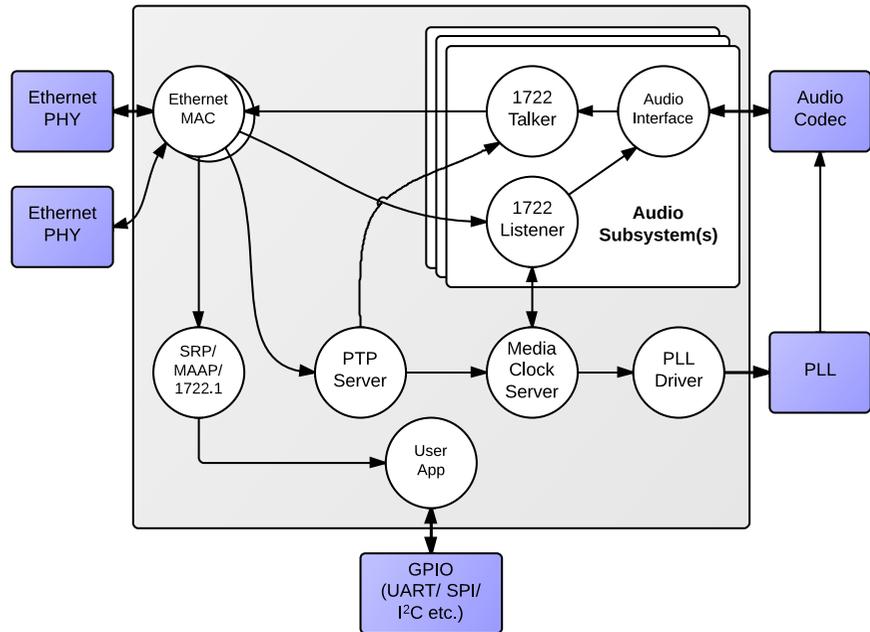
5.2 Ethernet MAC component

The MAC component provides two-port Ethernet connectivity to the AVB-DC solution. To use the component, two Ethernet PHYs must be attached to the xCORE ports via MII.

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 two-port 100 Mbps component consists of seven logcial cores, each running at 50 MIPS or more, that must be run on the same tile. These logcial cores handle both the receipt and transmission of Ethernet frames. The MAC component can be linked via channels to other components/logcial cores in the system. Each link can set a filter to control which packets are conveyed to it via that channel.
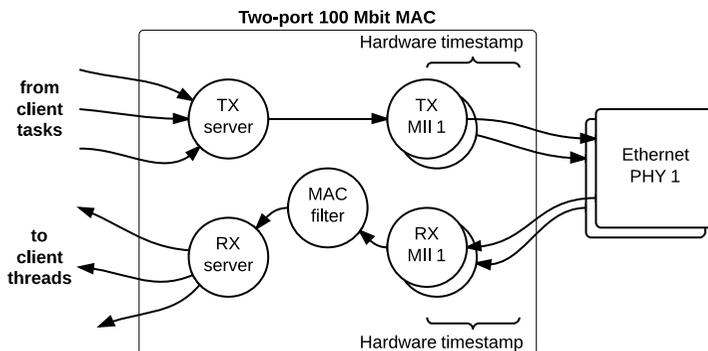
All configuration of the channel is managed by a client C/XC API, which configures and registers the filters. Details of the API used to configure MAC channels can be found in the Ethernet MAC component documentation[8]. This API is used for direct (layer-2) access to the MAC. For AVB applications it 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.

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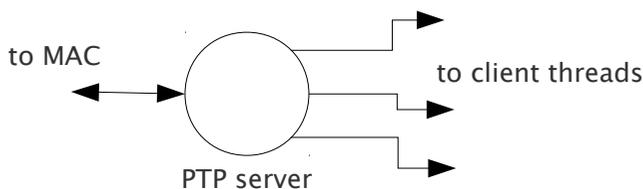
[8]https://www.xmos.com/resources/xsoftip?component=module_ethernet

The timing component consists of two logcial 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
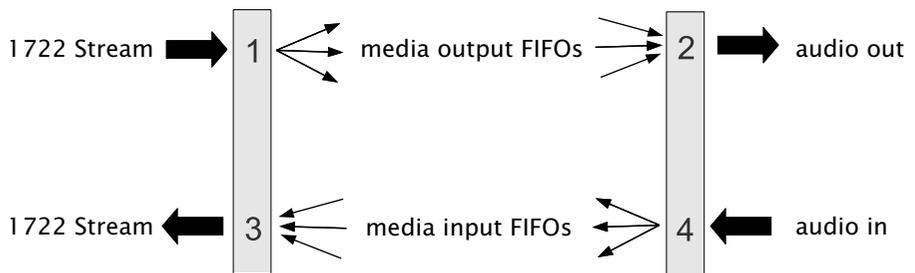
As described in the previous section, an IEEE 1722 audio stream may consist of many channels. These channels need to be routed to particular audio I/Os on the endpoint. To achieve maximum flexibility the XMOS design uses intermediate media FIFOs to route audio. Each FIFO contains a single channel of 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
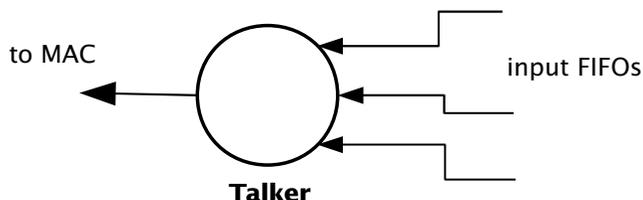
A talker unit consists of one logcial core which creates *IEEE 1722* packets and passes the audio samples onto the MAC. Audio samples are passed to this component via input media FIFOs. Samples are pushed into this FIFO from a different task implementing the audio hardware interface. The Talker task 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
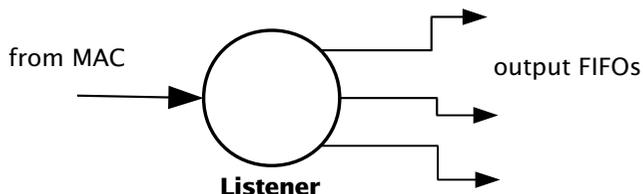
A Listener unit takes *IEEE 1722* packets from the MAC and converts them into a sample stream to be fed into a media FIFO. Each audio Listener component can listen to several *IEEE 1722* streams.

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. The diagram shows the supporting task that takes samples out of the media output FIFOs and serializes them over an XC channel:
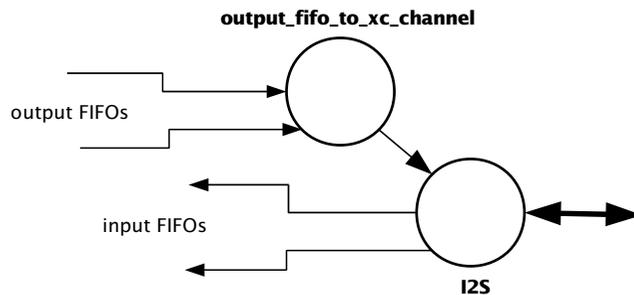
5.5 Media clocks

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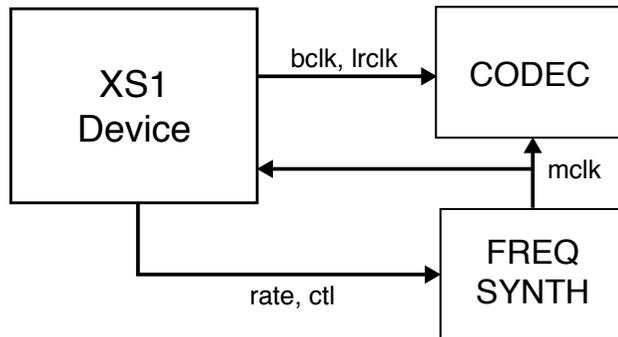
The XS1 device provides control to the frequency synthesizer and the frequency synthesizer provides the audio master clock to the CODEC and XS1 device. The sample bit and word clocks are then provided to the CODEC by the XS1 device.

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
Dual-port Ethernet with SMI	7	50	15 code, 1.5 per buffer	10 x 1bit, 4 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 1bit + 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 1bit 1 x 1bit 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 (n+m)
48	n in/m out	0.5 x (n+m)
88.2	n in/m out	1 x (n+m)
96	n in/m out	1 x (n+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

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 *XMOS AVB-DC Software Release* by clicking the *Request Access* link under *AVB DAISY-CHAIN KIT*. 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-DC software requires xTIMEcomposer version 13.0.2 or greater. It can be downloaded at the following URL

▶ https://www.xmos.com/en/support/downloads/xtimecomposer

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_daisy_chain` 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 sliceKIT core board.

2. Connect the xTAG-2 to the debug adapter.

3. Plug the xTAG-2 into your development system via USB.

4. Plug in the 12V power adapter and connect it to the sliceKIT core board.

5. In xTIMEcomposer, right-click on the binary within the *app_daisy_chain/bin* folder of the project.

6. Choose *Flash As ▶ Flash Configurations*.

7. Double click *xCORE Application* in the left panel.

8. Choose *hardware* in *Device options* and select the relevant xTAG-2 adapter.

9. Click on **Apply** if configuration has changed.

10. Click on **Flash**. Once completed, disconnect the power from the sliceKIT core board.

11. Repeat steps 1 through 8 for the second sliceKIT.

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-DC source package contains a simple demonstration application *app_daisy_chain*.

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 P1722.1 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
`avb_api.h`	Header file containing declarations for the core AVB control API.
`avb_1722_1_app_hooks.h`	Header file containing declarations for hooks into 1722.1
`ethernet_rx_client.h`	Header file for clients that require direct access to the ethernet MAC (RX).
`ethernet_tx_client.h`	Header file for clients that require direct access to the ethernet MAC (TX).
`gptp.h`	Header file for access to the PTP server.
`audio_i2s.h`	Header file containing the I2S audio component.

6.3 Entity Firmware Upgrade (EFU)

6.3.1 Introduction

The EFU loader is a flash device firmware upgrade mechanism for AVB endpoints.

The firmware upgrade implementation for XMOS AVB devices uses a subset of the Memory Object Upload mechanism described in Annex D of the 1722.1-2013 standard:

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

Supported functionality:

▶ Upload of new firmware to AVB device

▶ Reboot of device on firmware upgrade via the 1722.1 REBOOT command

xTIMEcomposer v13.0.2 or later is required to generate flash images compatible with the AVB-DC flash interface.

6.3.2   SPI Flash IC Requirements and Configuration

The current version of the AVB-DC EFU functionality supports boot flashes with the following properties only:

▶ A page size of 256 bytes

▶ Total flash size greater than or equal to the size required to store the boot loader, factory image and maximum sized upgrade image.

Other flash specific configuration parameters may be changed via `avb_flash_conf.h`:

FLASH_SECTOR_SIZE

FLASH_SPI_CMD_ERASE

FLASH_NUM_PAGES

FLASH_MAX_UPGRADE_IMAGE_SIZE

6.3.3   Installing the factory image to the device

Once the AVB-DC application has been built:

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

```
xflash --boot-partition-size 262144 <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> --boot-partition-size 262144 <binary>.xe
```

Ignore the following warning which is informative only:

```
Warning: F03098 Factory image and boot loader cannot be write-protected on flash device on nod
```

This programs the factory default firmware image into the flash device.

To use the firmware upgrade mechanism you need to build a firmware upgrade image:

1. Edit the `aem_entity_strings.h.in` file and increment the `AVB_1722_1_FIRMWARE_VERSION_STRING` and `AVB_1722_1_ADP_MODEL_ID` in `avb_conf.h`.

2. Rebuild the application

To generate the firmware upgrade image run the following command:

```
xflash --factory-version 13 --upgrade 1 <binary>.xe -o upgrade_image.bin
```

You should now have the firmware upgrade file upgrade_image.bin which can be transferred to the AVB end station.

6.3.4 Using the avdecc-lib CLI Controller to upgrade firmware

..note ::
> See the XMOS document *AVB System Requirements Guide* for installation details of the avdecccmdline tool.

1. To program the new firmware, first run avdecccmdline and select the interface number that represents the Ethernet interface that the AVB network is connected to:

```
Enter the interface number (1-7): 1
```

2. Use the list command to view all AVB end stations on the network:

```
$ list

End Station | Name        | Entity ID          | Firmware Version | MAC
------------------------------------------------------------------------------
C         0 | AVB 4in/4out | 0x002297fffe005279 |          1.0.0 | 002297005279
```

3. Select the end station that you wish to upgrade using the select command with the integer ID shown in the End Station column of the list output and two additional zeroes indicating the Entity and Configuration indices:

```
$ select 0 0 0
```

4. Begin the firmware upgrade process using the upgrade command with the full path of the upgrade_image.bin file:

```
$ upgrade /path/to/upgrade_image.bin
Erasing image...
Successfully erased.
Uploading image...
################################################################################
Successfully upgraded image.
Do you want to reboot the device? [y/n]: y
```

5. The device should now reboot and re-enumerate with an upgraded Firmware Version string. Test this using the list command:

```
$ list

End Station | Name        | Entity ID          | Firmware Version | MAC
------------------------------------------------------------------------------
C         0 | AVB 4in/4out | 0x002297fffe005279 |          1.1.0 | 002297005279
```

7 API Reference

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.

AVB_DEMO_ENABLE_TALKER

> Global switch to enable or disable AVB Talker functionality in the demo.

AVB_DEMO_ENABLE_LISTENER

> Global switch to enable or disable AVB Listener functionality in the demo.

AVB_DEMO_NUM_CHANNELS

> 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 .

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

AVB_MAX_AUDIO_SAMPLE_RATE

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

AVB_NUM_SOURCES

> The total number of AVB sources (streams that are to be transmitted).

AVB_NUM_TALKER_UNITS

> The total number or Talker components (typically the number of tasks running the :c:func: function).

AVB_MAX_CHANNELS_PER_TALKER_STREAM

> The maximum number of channels permitted per 1722 Talker stream.

AVB_NUM_MEDIA_INPUTS

> The total number of media inputs (typically number of I2S input channels).

AVB_NUM_SINKS

> The total number of AVB sinks (incoming streams that can be listened to)

AVB_NUM_LISTENER_UNITS

> The total number or listener components (typically the number of tasks running the :c:func: function)

AVB_MAX_CHANNELS_PER_LISTENER_STREAM

> The maximum number of channels permitted per 1722 Listener stream.

AVB_NUM_MEDIA_OUTPUTS

> The total number of media outputs (typically the number of I2S output channels).

AVB_NUM_MEDIA_UNITS

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

AVB_NUM_MEDIA_CLOCKS

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

AVB_ENABLE_1722_1

Enable 1722.1 AVDECC on the entity.

AVB_1722_1_TALKER_ENABLED

Enable the 1722.1 Talker functionality.

AVB_1722_1_LISTENER_ENABLED

Enable the 1722.1 Listener functionality.

AVB_1722_1_CONTROLLER_ENABLED

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. For details on the Ethernet component, see the Ethernet Component Guide[9].

---

[9]http://github.xcore.com/sc_ethernet/index.html

7.2.1   Core components

avb_manager()

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

Type

```
[[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)
```

Parameters

i_avb[]   array of avb_interface server interfaces connected to clients of
                  avb_manager

num_avb_clients
                  number of client interface connections to the server and the
                  number of elements of i_avb[]

i_srp   client interface of type srp_interface into an `srp_task()` task

c_media_ctl[]
                  array of chanends connected to components that register/control
                  media FIFOs

c_listener_ctl[]
                  array of chanends connected to components that register/control
                  IEEE 1722 sinks

c_talker_ctl[]
                  array of chanends connected to components that register/control
                  IEEE 1722 sources

c_mac_tx   chanend connection to the Ethernet TX server

i_media_clock_ctl
                  client interface of type media_clock_if connected to the media
                  clock server

c_ptp   chanend connection to the PTP server

avb_srp_info_t

Struct containing fields required for SRP reservations.

Fields

`unsigned stream_id`
> 64-bit Stream ID of the stream

`unsigned char dest_mac_addr`
> Stream destination MAC address.

`short vlan_id`
> VLAN ID for Stream.

`short tspec_max_frame_size`
> Maximum frame size sent by Talker.

`short tspec_max_interval`
> Maximum number of frames sent per class measurement interval.

`unsigned char tspec`
> Data Frame Priority and Rank fields.

`unsigned accumulated_latency`
> Latency at ingress port for Talker registrations, or latency at end of egress media for Listener Declarations.

srp_interface

register_stream_request()

Used by a Talker application entity to issue a request to the MSRP Participant to initiate the advertisement of an available Stream.

Type

    void 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

deregister_stream_request()

Used by a Talker application entity to request removal of the Talker's advertisement declaration, and thus remove the advertisement of a Stream, from the network.

Type

```
void deregister_stream_request(unsigned stream_id[2])
```

Parameters

stream_id   two int array containing the Stream ID of the stream to deregister

register_attach_request()

Used by a Listener application entity to issue a request to attach to the referenced Stream.

Type

```
void register_attach_request(unsigned stream_id[2])
```

Parameters

stream_id   two int array containing the Stream ID of the stream to register

deregister_attach_request()

Used by a Listener application entity to remove the request to attach to the referenced Stream.

Type

```
void deregister_attach_request(unsigned stream_id[2])
```

Parameters

stream_id   two int array containing the Stream ID of the stream to deregister

avb_srp_task()

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

avb_1722_1_aecp_aem_status_code

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.

AECP_AEM_STATUS_NOT_AUTHENTICATED
            The AVDECC Controller is not authenticated with the AVDECC Entity.

-XMOS-

AECP_AEM_STATUS_AUTHENTICATION_DISABLED

The AVDECC Controller is trying to use an authentication command when authentication isn't enable on the AVDECC Entity.

AECP_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).

AECP_AEM_STATUS_NO_RESOURCES

The AVDECC Entity cannot complete the command because it does not have the resources to support it.

AECP_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.

AECP_AEM_STATUS_ENTITY_MISBEHAVING

The AVDECC Entity is generated an internal error while trying to process the command.

AECP_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.

AECP_AEM_STATUS_STREAM_IS_RUNNING

The Stream is currently streaming and the command is one which cannot be executed on an Active Stream.

avb_1722_1_control_callbacks

get_control_value()

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,
                                unsigned char values[AEM_MAX_CONTROL_VALUES
```

Parameters

control_index
> the index of the CONTROL descriptor

values_length
> a reference to the length in bytes of the 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 for the GET_CONTROL response

set_control_value()

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,
                                unsigned char values[AEM_MAX_CONTROL_VALUES
```

Parameters

control_index
> the index of the CONTROL descriptor

values_length
> the length in bytes of the 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 for the SET_CONTROL response

avb_1722_1_task()

1722.1 task that runs ADP, ACMP and AECP protocols and interacts with the rest of the AVB stack.

Can be combined with other combinable tasks.

Type

```
[[combinable]]
void avb_1722_1_task(otp_ports_t &otp_ports,
                     client interface avb_interface i_avb,
                     client interface avb_1722_1_control_callbacks i_1722_1_entit
                     client interface spi_interface i_spi,
                     chanend c_mac_rx,
                     chanend c_mac_tx,
                     chanend c_ptp)
```

Parameters

otp_ports   reference to an OTP ports structure of type otp_ports_t

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()

c_mac_rx   chanend into the Ethernet RX server

c_mac_tx   chanend into the Ethernet TX server

c_ptp   chanend into the PTP server

fl_spi_ports

Struct containing ports and clocks used to access a flash device.

Fields

buffered in port:8 spiMISO
                Master input, slave output (MISO) port.

out port spiSS
                Slave select (SS) port.

buffered out port:32 spiCLK
                Serial clock (SCLK) port.

buffered out port:8 spiMOSI
                Master output, slave input (MOSI) port.

```
clock spiClkblk
```
> Clock block for use with SPI ports.

spi_interface

command_status()

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

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.

command_address_status()

This function issues a single command with a 3-byte address parameter and an optional data-set to be output to or input form the device.

Type

```
void command_address_status(int cmd,
                            unsigned int address,
                            unsigned char data[],
                            int returnBytes)
```

Parameters

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 . If negative, then this is (minus) the number of bytes to be written to the device from . 0 means no bytes will be read or written.

spi_task()

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_ports   reference to a structure containing the SPI flash ports and clock-block

ptp_server()

This function runs the PTP server.

It takes one thread 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)
`client`	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 (or)

media_clock_server()

The media clock server.

Type

```
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[])
```

Parameters

media_clock_ctl
server interface of type media_clock_if connected to the avb_manager() task and passed into avb_init()

ptp_svr   chanend connected to the PTP server

buf_ctl[]   array of links to listener components requiring buffer management

num_buf_ctl   size of the buf_ctl array

p_fs   output port to drive PLL reference clock

c_rx   chanend connected to the ethernet server (receive)

c_tx   chanend connected to the ethernet server (transmit)

c_ptp[]   an array of chanends to connect to clients of the ptp server

num_ptp   The number of PTP clients attached

server_type   The type of the PTP server (or)

avb_1722_listener()

An AVB IEEE 1722 audio listener thread.

This thread 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 thread is dynamically configured using the AVB control API.

Type

```
void avb_1722_listener(chanend c_mac_rx,
                       chanend ?c_buf_ctl,
                       chanend ?c_ptp_ctl,
                       chanend c_listener_ctl,
                       int num_streams)
```

Parameters

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

avb_1722_talker()

An AVB IEEE 1722 audio talker thread.

This thread 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

```
void avb_1722_talker(chanend c_ptp,
                     chanend c_mac_tx,
                     chanend c_talker_ctl,
                     int num_streams)
```

Parameters

`c_ptp`	link to the PTP timing server
`c_mac_tx`	transmit link to the ethernet MAC
`c_talker_ctl`	channel to configure the talker (given to `avb_init()`)
`num_streams`	the number of streams the unit controls

7.2.2 Audio components

The following types are used by the AVB audio components:

media_output_fifo_t

> This type provides a handle to a media output FIFO.

media_output_fifo_data_t

> This type provides the data structure used by a media output FIFO.

media_input_fifo_t

> This type provides a handle to a media input fifo.

media_input_fifo_data_t

> This type provides the data structure used by a media input fifo.

The following functions implement AVB audio components:

init_media_input_fifos()

> Initialize media input fifos.
>
> This function intializes media input 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.
>
> **Type**
>
> ```
> void init_media_input_fifos(media_input_fifo_t ififos[],
>                             media_input_fifo_data_t ififo_data[],
>                             int n)
> ```
>
> **Parameters**
>
> | | |
> |---|---|
> | ififos | an array of media input FIFO handles to initialize |
> | ififo_data | an array of associated data structures |
> | n | the number of FIFOs to initialize |

init_media_output_fifos()

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.

Type

```
void init_media_output_fifos(media_output_fifo_t ofifos[],
                             media_output_fifo_data_t ofifo_data[],
                             int n)
```

Parameters

ofifos	an array of media output FIFO handles to initialize
ofifo_data	an array of associated data structures
n	the number of FIFOs to initialize

i2s_master()

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.

Type

```
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)
```

Parameters

ports	a reference to a structure of type containing the I2S port definitions
p_din	array of ports to input data from
num_in	number of input ports
p_dout	array of ports to output data to
num_out	number of output ports
master_to_word_clock_ratio	the ratio of the master clock to the word clock; must be one of 128, 256 or 512
input_fifos	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 {0,1,0,1} w would map to the two stereo local talker streams 0 and 1.
output_fifos	a map from the outputs to local Listener streams
media_ctl	a media clock control chanend connected to an avb_manager() task
clk_ctl_index	the index of the clock control, default 0

media_output_fifo_to_xc_channel()

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 .

The protocol over the channel is that the thread expects a timestamp to be sent to it and then it will output samples, pulling from the array. It will then expect another timestamp before the next set of samples.

Type

```
void media_output_fifo_to_xc_channel(streaming chanend samples_out,
                                     media_output_fifo_t ofifos[],
                                     int num_channels)
```

Parameters

samples_out   the chanend on which samples are output

ofifos   array of media output FIFOs to pull from

num_channels
               the number of channels (or FIFOs)

media_output_fifo_to_xc_channel_split_lr()

Output audio streams from media FIFOs to an XC channel, splitting left and right pairs.

This function outputs samples from several media output FIFOs over an XC channel over the streaming chanend . The media FIFOs are assumed to be grouped in left/right stereo pairs which are then split.

The protocol over the channel is that the thread expects a timestamp to be sent to it and then it will first output samples, pulling from all the even indexed elements of the array and then output all the odd elements. It will then expect another timestamp before the next set of samples.

Type

```
void media_output_fifo_to_xc_channel_split_lr(streaming chanend samples_out,
                                              media_output_fifo_t output_fifos[],
                                              int num_channels)
```

Parameters

samples_out   the chanend on which samples are output

output_fifos
               array of media output fifos to pull from

num_channels
               the number of channels (or FIFOs)

7.3   AVB API

7.3.1   General control functions

avb_get_control_packet()

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

```
void avb_get_control_packet(chanend c_rx,
                            unsigned int buf[],
                            unsigned int &nbytes,
                            unsigned int &port_num)
```

Parameters

`c_rx`	chanend connected to the ethernet component
`buf`	buffer to retrieve the packet into; buffer must have length at least bytes
`nbytes`	a reference parameter that is filled with the length of the received packet

avb_process_srp_control_packet()

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

```
void avb_process_srp_control_packet(client interface avb_interface i_avb,
                                    unsigned int buf[],
                                    unsigned len,
                                    chanend c_tx,
                                    unsigned int port_num)
```

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

avb_process_1722_control_packet()

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

```
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_callback
                                      client interface spi_interface i_spi)
```

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

avb_1722_maap_request_addresses()

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

```
void avb_1722_maap_request_addresses(int num_addresses,
                                     char(& ?start_address)[])
```

Parameters

num_addresses

> number of addresses to try and reserve; will be reserved in a contiguous range

start_address

> 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

avb_1722_maap_rerequest_addresses()

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

```
void avb_1722_maap_rerequest_addresses()
```

avb_1722_maap_relinquish_addresses()

Relinquish the reserved MAAP address range.

This function abandons the claim to the reserved address range

Type

```
void avb_1722_maap_relinquish_addresses()
```

7.3.3   MAAP application hooks

avb_talker_on_source_address_reserved()

MAAP has indicated that a multicast address has been successfully reserved for this Talker stream.

Type

```
void avb_talker_on_source_address_reserved(client interface avb_interface i_avb,
                                           int source_num,
                                           unsigned char mac_addr[6])
```

Parameters

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

7.3.4   AVB Control API

device_media_clock_type_t

Values

DEVICE_MEDIA_CLOCK_INPUT_STREAM_DERIVED

DEVICE_MEDIA_CLOCK_LOCAL_CLOCK

device_media_clock_state_t

Values

DEVICE_MEDIA_CLOCK_STATE_DISABLED

DEVICE_MEDIA_CLOCK_STATE_ENABLED

avb_interface

initialise()

Type

```
void initialise(void)
```

_get_source_info()

Intended for internal use within client interface get and set extensions only.

Type

```
avb_source_info_t _get_source_info(unsigned source_num)
```

_set_source_info()

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)
```

_get_sink_info()

Intended for internal use within client interface get and set extensions only.

Type

```
avb_sink_info_t _get_sink_info(unsigned sink_num)
```

_set_sink_info()

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)
```

_get_media_clock_info()

Intended for internal use within client interface get and set extensions only.

Type

```
media_clock_info_t _get_media_clock_info(unsigned clock_num)
```

_set_media_clock_info()

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)
```

get_source_format()

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

set_source_format()

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

get_source_channels()

Get the channel count of an AVB source.

Type

```
int get_source_channels(unsigned source_num, int &channels)
```

Parameters

source_num   the local source number

channels   the number of channels

set_source_channels()

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

source_num   the local source number

channels   the number of channels

get_source_sync()

Get the media clock of an AVB source.

Type

```
int get_source_sync(unsigned source_num, int &sync)
```

Parameters

source_num   the local source number

sync   the media clock number

set_source_sync()

Set the media clock of an AVB source.

Sets the media clock of the stream.

Type

```
int set_source_sync(unsigned source_num, int sync)
```

Parameters

source_num   the local source number

sync   the media clock number

get_source_presentation()

Get the presentation time offset of an AVB source.

Type

```
int get_source_presentation(unsigned source_num, int &presentation)
```

Parameters

source_num   the local source number to set

presentation
              the presentation offset in ms

set_source_presentation()

Set the presentation time offset of an AVB source.

Sets the presentation time offset of a source i.e. the time after sampling that the stream should be played. The default value for this is 2ms.

This setting will not take effect until the next time the source state moves from disabled to potential.

Type

```
int set_source_presentation(unsigned source_num, int presentation)
```

Parameters

source_num   the local source number to set

presentation
              the presentation offset in ms

get_source_vlan()

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, The media clock number

set_source_vlan()

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, The media clock number

get_source_state()

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

set_source_state()

Set the current state of an AVB source.

Sets the current state of an AVB source. You cannot set the state to . Changing the state to turns the stream on and it will automatically change to 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

get_source_map()

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

set_source_map()

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

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

get_source_dest()

Get the destination address of an avb source.

Type

```
int get_source_dest(unsigned source_num, unsigned char addr[], int &len)
```

Parameters

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

set_source_dest()

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

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

get_source_id()

Type

```
int get_source_id(unsigned source_num, unsigned int id[2])
```

get_sink_id()

Get the stream id that an AVB sink listens to.

Type

```
int get_sink_id(unsigned sink_num, unsigned int stream_id[2])
```

Parameters

sink_num   the number of the sink

stream_id   int array containing the 64-bit of the stream

set_sink_id()

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

sink_num   the number of the sink

stream_id   int array containing the 64-bit of the stream

get_sink_format()

Get the format of an AVB sink.

Type

```
int get_sink_format(unsigned sink_num,
                    enum avb_stream_format_t &format,
                    int &rate)
```

Parameters

sink_num   the local sink number

format   the format of the stream

rate   the sample rate of the stream in Hz

set_sink_format()

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

sink_num   the local sink number

format   the format of the stream

rate   the sample rate of the stream in Hz

get_sink_channels()

Get the channel count of an AVB sink.

Type

```
int get_sink_channels(unsigned sink_num, int &channels)
```

Parameters

sink_num   the local sink number

channels   the number of channels

set_sink_channels()

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

sink_num   the local sink number

channels   the number of channels

get_sink_sync()

Get the media clock of an AVB sink.

Type

```
int get_sink_sync(unsigned sink_num, int &sync)
```

Parameters

sink_num   the local sink number

sync   the media clock number

set_sink_sync()

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

sink_num   the local sink number

sync   the media clock number

get_sink_vlan()

Get the virtual lan id of an AVB sink.

Type

```
int get_sink_vlan(unsigned sink_num, int &vlan)
```

Parameters

sink_num   the number of the sink

vlan   the vlan id of the sink

set_sink_vlan()

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

sink_num   the number of the sink

vlan   the vlan id of the sink

get_sink_addr()

Get the incoming destination mac address of an avb sink.

Type

```
int get_sink_addr(unsigned sink_num, unsigned char addr[], int &len)
```

Parameters

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.

set_sink_addr()

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

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.

get_sink_state()

Get the state of an AVB sink.

Type

```
int get_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

set_sink_state()

Set the state of an AVB sink.

Sets the current state of an AVB sink. You cannot set the state to . Changing the state to turns the stream on and it will automatically change to 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

get_sink_map()

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

set_sink_map()

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

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

get_device_media_clock_rate()

Get the rate of a media clock.

Type

```
int get_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

set_device_media_clock_rate()

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

get_device_media_clock_state()

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

set_device_media_clock_state()

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

get_device_media_clock_source()

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

set_device_media_clock_source()

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

get_device_media_clock_type()

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

set_device_media_clock_type()

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

avb_1722_1_controller_connect()

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

avb_1722_1_controller_disconnect()

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

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

avb_1722_1_controller_disconnect_all_listeners()

Disconnect all Listener sinks currently connected to the Talker stream source with

Type

```
void avb_1722_1_controller_disconnect_all_listeners(int talker_id,
                                                    chanend c_tx)
```

Parameters

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

-XMOS-

avb_1722_1_controller_disconnect_talker()

Disconnect the Talker source currently connected to the Listener stream sink with

Type

```
void avb_1722_1_controller_disconnect_talker(int listener_id,
                                             chanend c_tx)
```

Parameters

listener_id   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

c_tx   a transmit chanend to the Ethernet server

7.3.6   1722.1 Discovery commands

avb_1722_1_adp_announce()

Start advertising information about this entity via ADP.

Type

```
void avb_1722_1_adp_announce(void)
```

avb_1722_1_adp_depart()

Stop advertising information about this entity via ADP.

Type

```
void avb_1722_1_adp_depart(void)
```

avb_1722_1_adp_discover()

Ask to discover the information for a specific entity GUID.

Type

```
void avb_1722_1_adp_discover(const_guid_ref_t guid)
```

Parameters

guid   The GUID of the entity to discover

avb_1722_1_adp_discover_all()

Ask to discover all available entities via ADP.

Type

```
void avb_1722_1_adp_discover_all(void)
```

avb_1722_1_entity_database_flush()

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.

avb_1722_1_entity_record

Fields

```
guid_t guid
```

```
unsigned int vendor_id
```

```
unsigned int entity_model_id
```

```
unsigned int capabilities
```

```
unsigned short talker_stream_sources
```

```
unsigned short talker_capabilities
```

```
unsigned short listener_stream_sinks
```

```
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
```

avb_entity_on_new_entity_available()

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

```
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)
```

Parameters

i_avb	client interface of type 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

avb_talker_on_listener_connect()

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 into avb_manager()
source_num	The local id of the Talker stream source
listener_guid	
	The GUID of the Listener entity that is connecting

avb_talker_on_listener_disconnect()

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 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.

avb_listener_on_talker_connect()

A Controller has indicated to connect this Listener sink to a Talker stream.

Type

```
avb_1722_1_acmp_status_t avb_listener_on_talker_connect(client interface avb_inte
                                                        int sink_num,
                                                        const_guid_ref_t talker_g
                                                        unsigned char dest_addr[6
                                                        unsigned int stream_id[2]
                                                        const_guid_ref_t my_guid)
```

Parameters

i_avb	client interface of type 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
my_guid	The GUID of this entity

avb_listener_on_talker_disconnect()

A Controller has indicated to disconnect this Listener sink from a Talker stream.

Type

```
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)
```

Parameters

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

![XMOS]

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.in` 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

ptp_timestamp

This type represents a timestamp in the gptp clock domain.

Fields

`unsigned int seconds`

`unsigned int nanoseconds`

7.5.2   Getting PTP time information

ptp_time_info

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.

ptp_time_info_mod64

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.

ptp_get_time_info()

Retrieve port progatation 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

ptp_get_time_info_mod64()

Retrieve time information from the ptp server.

This function gets an up-to-date structure of type 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

ptp_request_time_info()

This function requests a 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)
```

Parameters

ptp_server   chanend connecting to the ptp server

ptp_request_time_info_mod64()

This function requests a 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_mod64().

Type

```
void ptp_request_time_info_mod64(chanend ptp_server)
```

Parameters

ptp_server   chanend connecting to the PTP server

ptp_get_requested_time_info()

This function receives a structure from the PTP server.

This completes an asynchronous transaction initiated with a call to ptp_request_time_info(). The function can be placed in a select case which will activate when the PTP server is ready to send.

Type

```
void ptp_get_requested_time_info(chanend ptp_server, ptp_time_info &info)
```

Parameters

ptp_server   chanend connecting to the PTP server

info   a reference parameter to be filled with the time information structure

ptp_get_requested_time_info_mod64()

This function receives a structure from the PTP server.

This completes an asynchronous transaction initiated with a call to ptp_request_time_info_mod64(). The function can be placed in a select case which will activate when the PTP server is ready to send.

Type

```
void ptp_get_requested_time_info_mod64(chanend ptp_server,
                                       ptp_time_info_mod64 &info)
```

Parameters

ptp_server   chanend connecting to the PTP server

info   a reference parameter to be filled with the time information structure

7.5.3   Converting timestamps

local_timestamp_to_ptp()

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

```
void local_timestamp_to_ptp(ptp_timestamp &ptp_ts,
                            unsigned local_ts,
                            ptp_time_info &info)
```

Parameters

ptp_ts   the PTP timestamp structure to be filled with the converted time

local_ts   the local timestamp to be converted

info   a time information structure retrieved from the ptp server

local_timestamp_to_ptp_mod32()

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

```
unsigned local_timestamp_to_ptp_mod32(unsigned local_ts,
                                      ptp_time_info_mod64 &info)
```

Parameters

local_ts   the local timestamp to be converted

info   a time information structure retrieved from the PTP server

Returns

the least significant 32-bits of ptp time in nanoseconds

ptp_timestamp_to_local()

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 ptp_timestamp_to_local(ptp_timestamp &ts, ptp_time_info &info)
```

Parameters

ts   the PTP timestamp to convert

info   a time information structure retrieved from the PTP server.

Returns

the local timestamp

XMOS®

Copyright © 2014, All Rights Reserved.