XMOS AVB Documentation

Document Number: X6367A

Publication Date: 2013/2/28

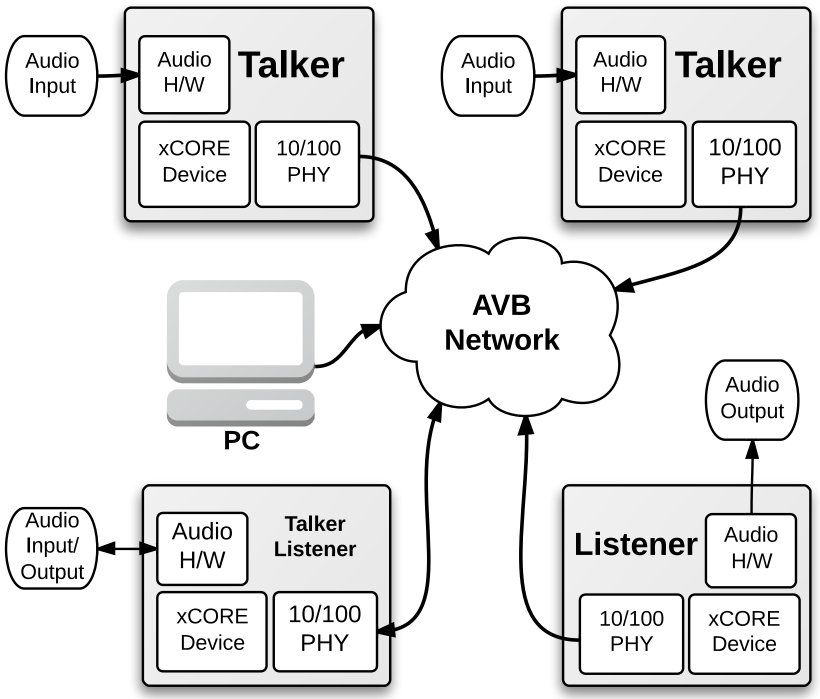
Table of Contents

1 Overview

IN THIS CHAPTER

1.1 Summary

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



1.1.1 XMOS AVB Key Features

- 100 Mbit/s full duplex Ethernet interface via MII
- Support for 1722.1 discovery, enumeration, command and control: ADP, AECP (AEM) and ACMP Draft 21
- 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, MMRP and MVRP
- 802.1AS Precision Time Protocol server for synchronization
- I2S, TDM and other audio interfaces for connection to external codecs and DSPs
- Media clock recovery and interface to a PLL clock source for high quality audio reproduction

1.2 XMOS AVB Specification

Supported Standards	
Ethernet	IEEE 802.3 (via MII)
AVB QoS	IEEE 802.1Qav, 802.1Qat
Precision Timing Protocol	IEEE 802.1AS
Audio Stream Encapsulation	IEEE 1722
Audio Format	IEC 61883-6 AM824
Enumeration and control	IEEE P1722.1 (Draft 21)
Supported Devices	
XMOS Devices	L16-128 L12-128
Requirements	
Development Tools	xTIMEComposer suite v12.0 or later
Ethernet	1 x MII compatible 100Mbit PHY
Audio	Audio input/output device (e.g. Audio CODEC) 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.	

1.3 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.

1.3.1 802.1AS

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.

1.3.2 802.1Qav

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.

1.3.3 802.1Qat

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].

1.3.4 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.

[1] http://standards.ieee.org/getieee802/download/802.1AS-2011.pdf
[2] http://standards.ieee.org/getieee802/download/802.1AS-2011.pdf
[3] http://standards.ieee.org/getieee802/download/802.1AS-2011.pdf

The IEC 61883-6:2005 standard document[4] is available for purchase from the IEC website.

1.3.5 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.

1.3.6 IEEE P1722.1

IEEE P1722.1 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 standard is currently in final draft and available to members of the 1722.1 Working Group[6]

[4] http://webstore.iec.ch/webstore/webstore.nsf/ArtNum_PK/46793
[5] http://standards.ieee.org/findstds/standard/1722-2011.html
[6] http://grouper.ieee.org/groups/1722/1/AVB-DECC/IEEE-1722.1_Working_Group.html

2 Hardware development platforms

For initial development of AVB applications the following XMOS development platforms are recommended:

- XK-AVB-LC-SYS AVB Audio Endpoint[7]

In addition the following kit can be used for development:

- SliceKIT Core Board[8] with 1 x Ethernet slice and 1 x Audio slice fitted with CS2100-CP PLL.

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

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

[7] http://www.xmos.com/products/reference-designs/avbl2

[8] http://www.xmos.com/discover/products/xkits/slicekit#tabs

3 System Description

IN THIS CHAPTER

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

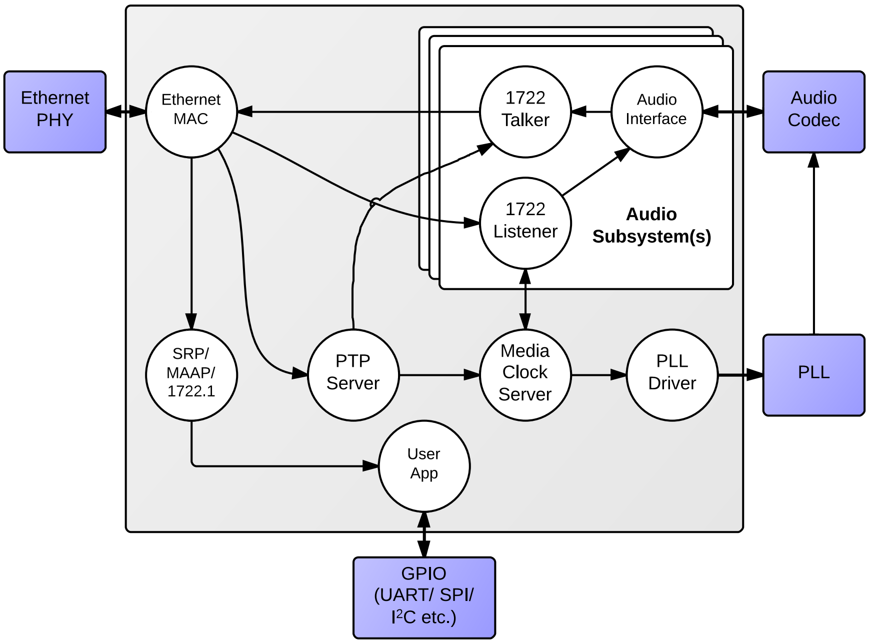
This software design guide assumes the reader is familiar with the XC language and XMOS XS1 devices.

3.1 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.



3.2 Ethernet MAC Component

The MAC component provides Ethernet connectivity to the AVB solution. To use the component, a Ethernet PHY must be attached to the XCore ports via MII. The XS1 device is also capable of implementing a dual 100 Mbps interface.[9].

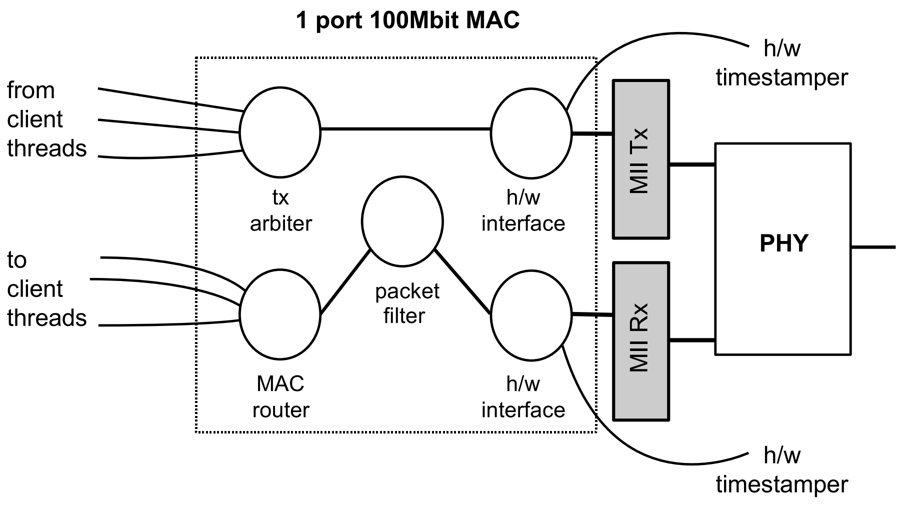
The XMOS Ethernet MAC component supports two features that are necessary to implement AVB standards with precise timing and quality constraints:

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

The single port 100 Mbps component consists of five logcial cores, each running at 50 MIPS or more, that must be run on the same tile. These logcial cores handle

[9]Dual MII is not included with the current software release. Contact XMOS for more information about the device capability and software availability.

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[10]. 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 §5.3).

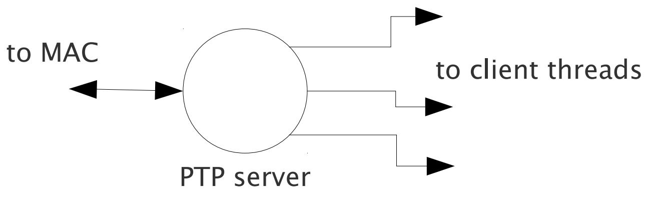
3.2.1 1722 Packet Routing

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 §5.3.

3.3 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.

[10]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 §5.5.

- 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 work 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.

3.4 Audio Components

3.4.1 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.

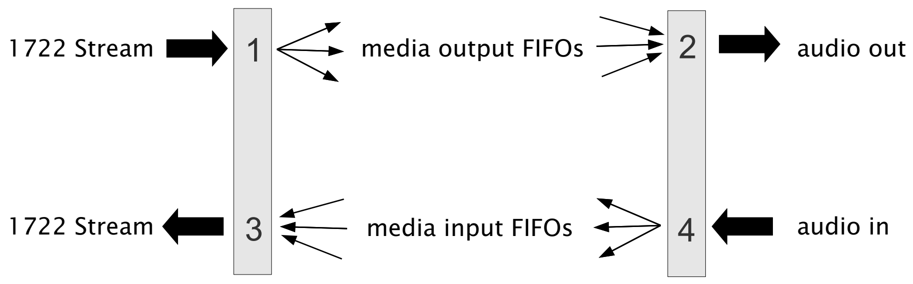
A single endpoint can be a Talker, a Listener or both. In general each endpoint will have a number of *sinks* with the capacity to receive a number of incoming streams and a number of *sources* with the capacity to transmit a number of streams.

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 §3.6).

3.4.2 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.

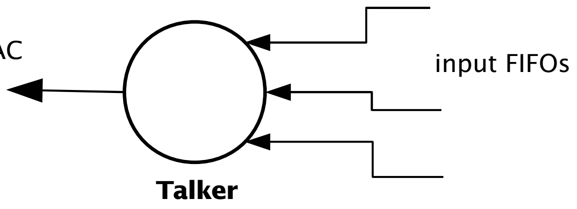
For audio being transmitted by an endpoint:

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 §5.3.

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

3.4.3 Talker Units

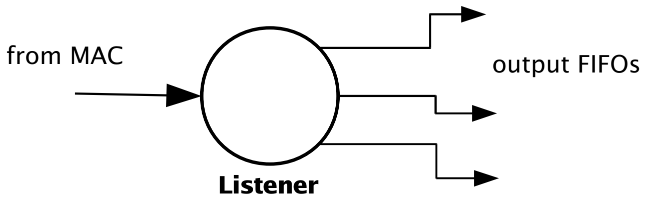


A talker unit consists of one logcial core which creates *IEEE 1722* packets and passes the audio samples onto the MAC. Audio samples are passed to this component via input media FIFOs. Samples are pushed into this FIFO from a different task implementing the audio hardware interface. The packetizer task 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 §5.2. Once the talker unit starts, it registers with the main control task and is control via the main AVB API described in Section §5.3.

3.4.4 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 FIFOs. 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 §5.2. Once the Listener unit starts, it registers with the main control task and is controlled via the main AVB API described in Section §5.3.

3.4.5 Media FIFOs to XC Channels

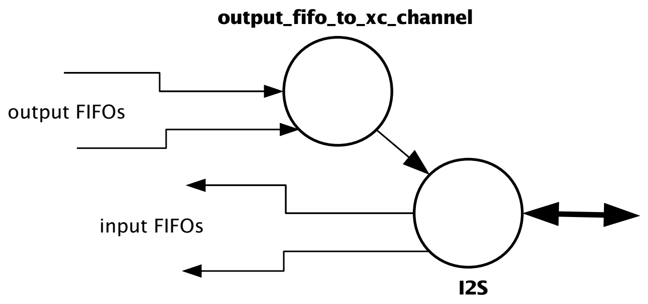
Sometimes it is useful to convert the audio stream in a media FIFO into a sample stream over an XC channel. This may be needed to move samples off tile or if the audio interface task requires samples over a channel. Several functions are provided to do this and are described in Section §5.2.

3.4.6 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 of 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:



3.5 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.

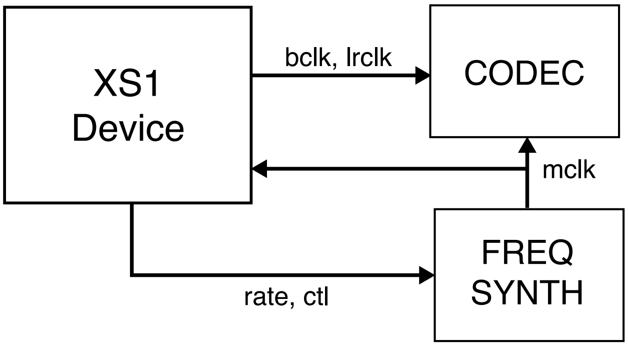
A hardware interface can be tied to a particular media clock, allowing the media output from the XMOS device to be synchronized with other devices on the network.

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.

3.5.1 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.

3.6 Device Discovery, Connection Management and Control

3.6.1 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.

3.6.2 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 cloud.

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 Entity. 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. Basic 1722.1 Controller functionality is available to allow 'plug and play' connection between two XMOS endpoints, however, it is expected that GUI Controller software will be available on the network for setting up larger topologies.

To assist in this task a unified control API is presented in Section §5.3.

3.7 Resource Usage

3.7.1 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
XS1-L12A-128-QF124-C10	12	1000	128	32 x 1bit 12 x 4bit 7 x 8bit 3 x 16bit
XS1-L10A-128-QF124-C10	10	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.

3.7.2 Ethernet Component

Each endpoint requires an Ethernet MAC layer.

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

3.7.3 PTP Component

Every AVB endpoint must include a PTP component.

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

3.7.4 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.

3.7.5 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 per stream.

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

Component	Logical Cores	MIPS/Core	Memory(KB)	Ports
I2S	1	50	0.5	3 x 1 bit 1 x 1 bit per stereo channel
TDM	1	50	0.5	3 x 1 bit 1 x 1 bit per 8 channels

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
TDM	48	8 in and 8 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)

3.7.6 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.

3.8 Choosing the right chip

The number of audio channels the device can handle depends on the XMOS device used. The XS1 platform is very flexible and can provide other functions alongside audio (depending on how much audio is used) including DSP functionality, controlling inputs and displays on a device or controlling non-AVB ethernet communication.

The amount of audio available for the XS1-L devices is detailed in the following sections. See Section §3.7 for more information on chip resource usage and how these figures were determined.

Please note that in a final application the exact capability depends on the type of digital audio interface, the mapping between 1722 and local streams, the complexity of the routing etc. and these figures are meant only as a rough guide.

- The maximum channel count figures assume that more than two channels are used per AVB stream to maximize channel count.
- At 100MBit/s, the capability on the XS1 for higher bit-rates are bounded by bandwidth and buffering in line with the AVB standard.
- For greater than 8 in/ 8 out channels, it is assumed that a multi-channel multiplexed protocol such as TDM is used to reduce the required pin count.

3.8.1 XS1-L16 Device

Sample Rate (kHz)	AVB Streams	Audio Channels
48	4 in/4 out	*16 in/16 out (TDM)*
48	4 in/4 out	8 in/8 out
96	2 in/2 out	4 in/4 out

4 Programming Guide

IN THIS CHAPTER

4.1 Getting Started

The following instructions explain how to build the demo for the XR-AVB-LC-BRD endpoint.

To install the software, open the 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 file firmware ZIP file.
4. Make sure that all projects are ticked in the *Projects* list.
5. Click **Finish**.

To build, select the `app_simple_avb_demo` project in the Project Explorer and click the **Build** icon.

From the command line, you can follow these steps:

1. To install, unzip the pacakge zipfile
2. To build, change into the `app_simple_avb_demo` directory and execute the command:

```
xmake all
```

4.1.1 Makefiles

The main Makefile for the project is in the `app_simple_avb_demo` directory. This file specifies build options and used modules.

4.1.2 Running the application

To upgrade the firmware you must, firstly connect the XTAG-2 to the relevant development board and plug the XTAG-2 into your PC or Mac.

4.1.2.1 Using the XMOS xTIMEcomposer Studio

Using the 12.0.0 tools or later and AVB version 5.2.0 or later, from within the xTIMEcomposer Studio:

1. Right click on the binary within the bin folder of the project.
2. Choose *Run As* ▶ *Run Configurations*
3. Double click *xCORE Application* in the left panel
4. Choose *hardware* in *Device options* and select the relevant XTAG-2 adapter
5. Select the *Run XScope output server* check box.
6. Click on **Apply** if configuration has changed
7. Click on **Run**

4.1.2.2 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 XTAG2s 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
```

4.1.3 Installing the application onto flash

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

4.1.3.1 Using the XMOS xTIMEcomposer Studio

To upgrade the flash from the xTIMEcomposer Studio, follow these steps:

1. Start the xTIMEcomposer Studio and open the workspace created in **Running the application**.
2. Right click on the binary within the bin folder of the project.

3. Choose *Flash As* ▶ *Flash Configurations*
4. Double click *xCORE Application* in the left panel
5. Choose *hardware* in *Device options* and select the relevant XTAG-2 adapter
6. Click on **Apply** if configuration has changed
7. Click on **Flash**

4.1.3.2 Using Command Line Tools

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

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

2. If multiple XTAG2s 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
```

4.2 Source code structure

4.2.1 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 documentation.

The source package contains a simple demonstration application *app_simple_avb_demo* that can be run on different hardware targets.

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_i2c_simple	Two wire configuration protocol code.	sc_i2c
module_random	Random number generator	sc_util

The following modules 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/TDM etc).
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.

4.2.2 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.

5 API Reference

IN THIS CHAPTER

5.1 Configuration Defines

5.1.1 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 `avb_conf.h`.

5.1.2 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 following values with #defines.

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.

5.1.3 Ethernet

See the Ethernet documentation for detailed information on its parameters.

5.1.4 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 avb_1722_talker() 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 avb_1722_listener() 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

5.1.5 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

5.2 Component functions

The following functions provide components that can be combined in the top-level main. For details on the ethernet and TCP/IP components see the Ethernet Component Guide[11] and the XTCP Component Guide[12].

5.2.1 Core Components

```
void ptp_server(chanend mac_rx,
                chanend mac_tx,
                chanend client[],
                int num_clients,
                enum ptp_server_type server_type)
```

This function runs the PTP server.

It takes one thread and runs indefinitely

This function has the following 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

[11] http://github.xcore.com/sc_ethernet/index.html
[12] http://github.xcore.com/sc_xtcp/index.html

`num_clients`	The number of clients attached
`server_type`	The type of the server (`PTP_GRANDMASTER_CAPABLE` or `PTP_SLAVE_ONLY`)

```
void media_clock_server(chanend media_clock_ctl,
                        chanend ?ptp_svr,
                        chanend ?buf_ctl[],
                        int buf_ctl_size,
                        out port p_fs[])
```

The media clock server.

This function has the following parameters:

`media_clock_ctl`	chanend connected to the main control thread and passed into avb_init()
`ptp_svr`	chanend connected to the PTP server
`buf_ctl[]`	array of links to listener components requiring buffer management
`buf_ctl_size`	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 (`PTP_GRANDMASTER_CAPABLE` or `PTP_SLAVE_ONLY`)

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

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.

This function has the following 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 retreiving 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

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

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.

This function has the following 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

5.2.2 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:

```
void init_media_input_fifos(media_input_fifo_t ififos[],
                            media_input_fifo_data_t ififo_data[],
                            int n)
```

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.

This function has the following 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

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

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.

This function has the following 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

```
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)
```

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

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 `samples_out`.

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

This function has the following 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)

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

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 `samples_out`. 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 `num_channels/2` samples, pulling from all the even indexed elements of the `ofifos` array and then output all the odd elements. It will then expect another timestamp before the next set of samples.

This function has the following 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)

5.3 AVB API

5.3.1 General Control Functions

```
void avb_init(chanend media_ctl[],
              chanend ?listener_ctl[],
              chanend ?talker_ctl[],
              chanend ?media_clock_ctl,
              chanend c_mac_rx,
              chanend c_mac_tx,
              chanend c_ptp)
```

Initialize the AVB control thread.

This function initializes the AVB system. It needs to be called in the main user control thread before any other AVB control call. The function takes chanends connected to other parts of the system and registers all of these components.

At this point the sinks, sources and media FIFOs are allocated numbers. The allocation is performed by registering numbers from 0 upwards working through the listener_ctl/talker_ctl/media_ctl arrays. Each component in this array may register several sink/sources/FIFOs. For example, if the listener_ctl array connects to two listener units each registering 3 sinks then the first unit will be allocated sink numbers 0,1,2 and the second 3,4,5.

Note that this call does not start any protocols communicating over the network (e.g. advertising talkers via IEEE 802.1Qat). That is deferred until the call to avb_start().

This function has the following parameters:

`media_ctl`	array of chanends connected to components that register/control media FIFOs
`listener_ctl`	array of chanends connected to components that register/control IEEE 1722 sinks
`talker_ctl`	array of chanends connected to components that register/control IEEE 1722 sources
`media_clock_ctl`	chanend connected to the media clock server
`c_mac_rx`	chanend connected to the ethernet server (RX)
`c_mac_tx`	chanend connected to the ethernet server (TX)
`c_ptp`	chanend connected to the ptp server

```
void avb_start(void)
```

Start any AVB protocol state machines.

This call starts any AVB protocol state machines running. It should be called after the ethernet link goes up.

```
void avb_periodic(unsigned int time_now)
```

Perform AVB periodic processing.

This function performs AVB periodic processing. It should be called from the main control thread at least once each ms.

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

Receives an 802.1Qat SRP packet or an IEEE P1722 MAAP 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.

This function has the following parameters:

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

```
void avb_process_control_packet(unsigned int buf[], int len, chanend c_tx)
```

Process an AVB control packet.

This function processes an ethernet packet and if it is a 802.1Qat or IEEE 1722 MAAP packet will handle it.

This function should always be called on the buffer filled by avb_get_control_packet().

This function has the following parameters:

`buf`	the incoming message buffer
`len`	the length (in bytes) of the incoming buffer
`c_tx`	chanend connected to the ethernet mac (TX)

5.3.2 Multicast Address Allocation Commands

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

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().

This function has the following 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

```
void 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.

```
void avb_1722_maap_relinquish_addresses()
```

Relinquish the reserved MAAP address range.

This function abandons the claim to the reserved address range

5.3.3 MAAP Application Hooks

```
void avb_talker_on_source_address_reserved(int source_num,
                                           unsigned char mac_addr[6])
```

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

This function has the following parameters:

`source_num`	The local source ID of the Talker
`mac_addr`	The destination MAC address reserved for this Talker

5.3.4 Media Clock Control

```
device_media_clock_type_t
```

This type has the following values:

`DEVICE_MEDIA_CLOCK_INPUT_STREAM_DERIVED`

`DEVICE_MEDIA_CLOCK_LOCAL_CLOCK`

```
int get_device_media_clock_type(int clock_num,
                                enum device_media_clock_type_t &clock_type)
```

Get the type of a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`clock_type`	the type of the clock

```
int set_device_media_clock_type(int clock_num,
                                enum device_media_clock_type_t clock_type)
```

Set the type of a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`clock_type`	the type of the clock

```
int get_device_media_clock_rate(int clock_num, int &rate)
```

Get the rate of a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`rate`	the rate of the clock in Hz

```
int set_device_media_clock_rate(int clock_num, int rate)
```

Set the rate of a media clock.

Sets the rate of the media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`rate`	the rate of the clock in Hz

```
int get_device_media_clock_source(int clock_num, int &source)
```

Get the source of a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock

`source`	the output FIFO number to base the clock on

```
int set_device_media_clock_source(int clock_num, int 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.

This function has the following parameters:

`clock_num`	the number of the media clock
`source`	the output FIFO number to base the clock on

```
device_media_clock_state_t
```

This type has the following values:

`DEVICE_MEDIA_CLOCK_STATE_DISABLED`

`DEVICE_MEDIA_CLOCK_STATE_ENABLED`

```
int get_device_media_clock_state(int clock_num,
                                 enum device_media_clock_state_t &state)
```

Get the state of a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`state`	the state of the clock

```
int set_device_media_clock_state(int clock_num,
                                 enum device_media_clock_state_t state)
```

Set the state of a media clock.

This function can be used to enabled/disable a media clock.

This function has the following parameters:

`clock_num`	the number of the media clock
`state`	the state of the clock

5.3.5 AVB Source Control

`avb_stream_format_t`

This type has the following values:

`AVB_SOURCE_FORMAT_MBLA_24BIT`

```
int get_avb_source_format(int source_num,
                          enum avb_stream_format_t &format,
                          int &rate)
```

Get the format of an AVB source.

This function has the following parameters:

`source_num`	the local source number
`format`	the format of the stream
`rate`	the sample rate of the stream in Hz

```
int set_avb_source_format(int source_num,
                          enum avb_stream_format_t format,
                          int rate)
```

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.

This function has the following parameters:

`source_num`	the local source number
`format`	the format of the stream
`rate`	the sample rate of the stream in Hz

```
int get_avb_source_channels(int source_num, int &channels)
```

Get the channel count of an AVB source.

This function has the following parameters:

`source_num`	the local source number
`channels`	the number of channels

`int set_avb_source_channels(int source_num, int 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.

This function has the following parameters:

`source_num`	the local source number
`channels`	the number of channels

`int get_avb_source_map(int source_num, int map[], int &len)`

Get the channel map of an avb source.

This function has the following 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

`int set_avb_source_map(int source_num, int map[], int len)`

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.

This function has the following 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

`int get_avb_source_sync(int source_num, int &sync)`

Get the media clock of an AVB source.

This function has the following parameters:

`source_num`	the local source number

`sync`	the media clock number

`int set_avb_source_sync(int source_num, int sync)`

Set the media clock of an AVB source.

Sets the media clock of the stream.

This function has the following parameters:

`source_num`	the local source number
`sync`	the media clock number

`int get_avb_source_presentation(int source_num, int &presentation)`

Get the presentation time offset of an AVB source.

This function has the following parameters:

`source_num`	the local source number to set
`presentation`	the presentation offset in ms

`int set_avb_source_presentation(int source_num, int 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.

This function has the following parameters:

`source_num`	the local source number to set
`presentation`	the presentation offset in ms

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

Get the destination address of an avb source.

This function has the following 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

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

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.

This function has the following 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

`int get_avb_source_vlan(int source_num, int &vlan)`

Get the destination vlan of an AVB source.

This function has the following parameters:

`source_num`	the local source number
`vlan`	the destination vlan id, The media clock number

`int set_avb_source_vlan(int source_num, int 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.

This function has the following parameters:

`source_num`	the local source number
`vlan`	the destination vlan id, The media clock number

`avb_source_state_t`

The state of an AVB source.

This type has the following values:

`AVB_SOURCE_STATE_DISABLED`	The source is disabled and will not transmit.
`AVB_SOURCE_STATE_POTENTIAL`	The source is enabled and will transmit if a listener requests it.
`AVB_SOURCE_STATE_ENABLED`	The source is enabled and transmitting.

`int get_avb_source_state(int source_num, enum avb_source_state_t &state)`

Get the current state of an AVB source.

This function has the following parameters:

`source_num`	the local source number
`state`	the state of the source

`int set_avb_source_state(int source_num, enum avb_source_state_t state)`

Set the current state of an AVB source.

Sets the current state of an AVB source. You cannot set the state to `ENABLED`. Changing the state to `AVB_SOURCE_STATE_POTENTIAL` turns the stream on and it will automatically change to `ENABLED` when connected to a listener and streaming.

This function has the following parameters:

`source_num`	the local source number
`state`	the state of the source

5.3.6 AVB Sink Control

`int get_avb_sink_channels(int sink_num, int &channels)`

Get the channel count of an AVB sink.

This function has the following parameters:

`sink_num`	the local sink number
`channels`	the number of channels

`int set_avb_sink_channels(int sink_num, int 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.

This function has the following parameters:

`sink_num`	the local sink number
`channels`	the number of channels

`int get_avb_sink_map(int sink_num, int map[], int &len)`

Get the map of an AVB sink.

This function has the following 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

`int set_avb_sink_map(int sink_num, int map[], int len)`

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.

This function has the following 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

`int get_avb_sink_sync(int sink_num, int &sync)`

Get the media clock of an AVB sink.

This function has the following parameters:

`sink_num`	the local sink number
`sync`	the media clock number

`int set_avb_sink_sync(int sink_num, int sync)`

Set the media clock of an AVB sink.

Sets the media clock of the stream.

This function has the following parameters:

`sink_num`	the local sink number
`sync`	the media clock number

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

Get the stream id that an AVB sink listens to.

This function has the following parameters:

`sink_num`	the number of the sink
`stream_id`	int array containing the 64-bit of the stream

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

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.

This function has the following parameters:

`sink_num`	the number of the sink
`stream_id`	int array containing the 64-bit of the stream

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

Get the incoming destination mac address of an avb sink.

This function has the following 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.

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

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.

This function has the following 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.

`int get_avb_sink_vlan(int sink_num, int &vlan)`

Get the virtual lan id of an AVB sink.

This function has the following parameters:

`sink_num`	the number of the sink
`vlan`	the vlan id of the sink

```
int set_avb_sink_vlan(int sink_num, int 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.

This function has the following parameters:

`sink_num`	the number of the sink
`vlan`	the vlan id of the sink

```
avb_sink_state_t
```

This type has the following values:

`AVB_SINK_STATE_DISABLED`

`AVB_SINK_STATE_POTENTIAL`

`AVB_SINK_STATE_ENABLED`

```
int get_avb_sink_state(int sink_num, enum avb_sink_state_t &state)
```

Get the state of an AVB sink.

This function has the following parameters:

`sink_num`	the number of the sink
`state`	the state of the sink

```
int set_avb_sink_state(int sink_num, enum avb_sink_state_t state)
```

Set the state of an AVB sink.

Sets the current state of an AVB sink. You cannot set the state to `ENABLED`. Changing the state to `POTENTIAL` turns the stream on and it will automatically change to `ENABLED` when connected to a talker and receiving samples.

This function has the following parameters:

`sink_num`	the number of the sink
`state`	the state of the sink

5.3.7 1722.1 Controller Commands

```
void avb_1722_1_controller_connect(guid_t &talker_guid,
                                   guid_t &listener_guid,
                                   int talker_id,
                                   int listener_id,
                                   chanend c_tx)
```

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.

This function has the following 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

```
void avb_1722_1_controller_disconnect(guid_t &talker_guid,
                                      guid_t &listener_guid,
                                      int talker_id,
                                      int listener_id,
                                      chanend c_tx)
```

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.

This function has the following 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

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

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

This function has the following 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

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

Disconnect the Talker source currently connected to the Listener stream sink with `listener_id`.

This function has the following 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

5.3.8 1722.1 Discovery Commands

`void avb_1722_1_adp_announce(void)`

Start advertising information about this entity via ADP.

`void avb_1722_1_adp_depart(void)`

Stop advertising information about this entity via ADP.

`void avb_1722_1_adp_discover(guid_t &guid)`

Ask to discover the information for a specific entity GUID.

This function has the following parameters:

`guid`	The GUID of the entity to discover

`void avb_1722_1_adp_discover_all(void)`

Ask to discover all available entities via ADP.

`void avb_1722_1_entity_database_flush(void)`

Remove all discovered entities from the database.

5.3.9 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`

This structure has the following members:

`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 as_grandmaster_id`

`unsigned int association_id`

`unsigned timeout`

```
void avb_entity_on_new_entity_available(guid_t &my_guid,
                                        avb_1722_1_entity_record &entity,
                                        chanend c_tx)
```

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.

This function has the following parameters:

`my_guid`	The GUID of this entity
`entity`	The information advertised by the remote entity

`c_tc`	A transmit channel end to the Ethernet server

```
void avb_talker_on_listener_connect(int source_num, guid_t &listener_guid)
```

A Controller has indicated that a Listener is connecting to this Talker stream source.

This function has the following parameters:

`source_num`	The local id of the Talker stream source
`listener_guid`	The GUID of the Listener entity that is connecting

```
void avb_talker_on_listener_disconnect(int source_num,
                                       guid_t &listener_guid,
                                       int connection_count)
```

A Controller has indicated that a Listener is disconnecting from this Talker stream source.

This function has the following parameters:

`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.

```
void avb_listener_on_talker_connect(int sink_num,
                                    guid_t &talker_guid,
                                    unsigned char dest_addr[6],
                                    unsigned int stream_id[2],
                                    guid_t &my_guid)
```

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

This function has the following parameters:

`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

```
void avb_listener_on_talker_disconnect(int sink_num,
                                       guid_t &talker_guid,
                                       unsigned char dest_addr[6],
                                       unsigned int stream_id[2],
                                       guid_t &my_guid)
```

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

This function has the following parameters:

`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

5.4 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.

5.4.1 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`

5.4.2 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
```

5.5 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.

5.5.1 Time Data Structures

`ptp_timestamp`

This type represents a timestamp in the gptp clock domain.

This structure has the following members:

`unsigned int seconds`

`unsigned int nanoseconds`

5.5.2 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.

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

Retrieve port progatation delay from the ptp server.

This function has the following parameters:

`ptp_server`	chanend connected to the ptp_server
`pdelay`	unsigned int with delay in ns

```
void ptp_get_time_info_mod64(chanend ?ptp_server,
                             ptp_time_info_mod64 &info)
```

Retrieve time information from the ptp server.

This function gets an up-to-date structure of type *ptp_time_info_mod64* to use to convert local time to ptp time (modulo 64 bits).

This function has the following parameters:

`ptp_server`	chanend connected to the ptp_server
`info`	structure to be filled with time information

`void ptp_request_time_info(chanend ptp_server)`

This function requests a *ptp_time_info* structure from the PTP server.

This is an asynchronous call so needs to be completed later with a call to ptp_get_requested_time_info().

This function has the following parameters:

`ptp_server`	chanend connecting to the ptp server

`void ptp_request_time_info_mod64(chanend ptp_server)`

This function requests a *ptp_time_info_mod64* 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().

This function has the following parameters:

`ptp_server`	chanend connecting to the PTP server

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

This function receives a *ptp_time_info* 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.

This function has the following parameters:

`ptp_server`	chanend connecting to the PTP server
`info`	a reference parameter to be filled with the time information structure

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

This function receives a *ptp_time_info_mod64* 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.

This function has the following parameters:

`ptp_server`	chanend connecting to the PTP server
`info`	a reference parameter to be filled with the time information structure

5.5.3 Converting Timestamps

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

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.

This function has the following 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

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

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.

This function has the following parameters:

`local_ts`	the local timestamp to be converted
`info`	a time information structure retrieved from the PTP server

This function returns:

the least significant 32-bits of ptp time in nanoseconds

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

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.

This function has the following parameters:

`ts`	the PTP timestamp to convert
`info`	a time information structure retrieved from the PTP server.

This function returns:

the local timestamp

6 IEEE 1722 Bandwidth Usage

The AVB standard requires audio data to be split into packets to be transmitted over ethernet along with meta-information specified by the IEEE 1722 transport protocol. This meta-information incurs an overhead on the bandwidth of each stream of data.

The protocol overhead is detailed in the following table:

Protocol	Overhead (bytes)
Interframe gap	20
Ethernet header	18
IEEE 1722 header	24
61883-6 header	8
CRC	4

Each stream of audio data can contain several multiplexed channels of audio. The higher the number of channels per stream, the more efficient the audio transport is in terms of bandwidth. Note that the IEC 61883-6 standard recommends transmitting single channel streams as stereo with the right channel blank.

After the header in each packet, audio data is stored in AM824 format which pads 24-bit data to 4 byte quadlets. The frame rate is 8kHz so a 48kHz stream has 6 samples per packet, a 96kHz has 12 samples per packet and so on.

The following table shows the bandwidth for streams at different sample rates with different number of channels per stream.

Sample Rate (kHz)	Channels/Stream	Mbps
48	1	7.81
48	2	7.81
48	4	10.88
48	8	17.02
48	16	29.31
48	32	53.89
96	1	10.88
96	2	10.88
96	4	17.02
96	8	29.31
96	16	53.89

Note that the higher the number of channels per stream, the better the bandwidth usage (e.g. 4 x 8 channels streams uses less bandwidth than 16 x 2 channel streams). The IEEE 1722 standard also specifies that only 75% of available bandwidth can be used for AVB traffic.