XMOS Specifications and Testplans

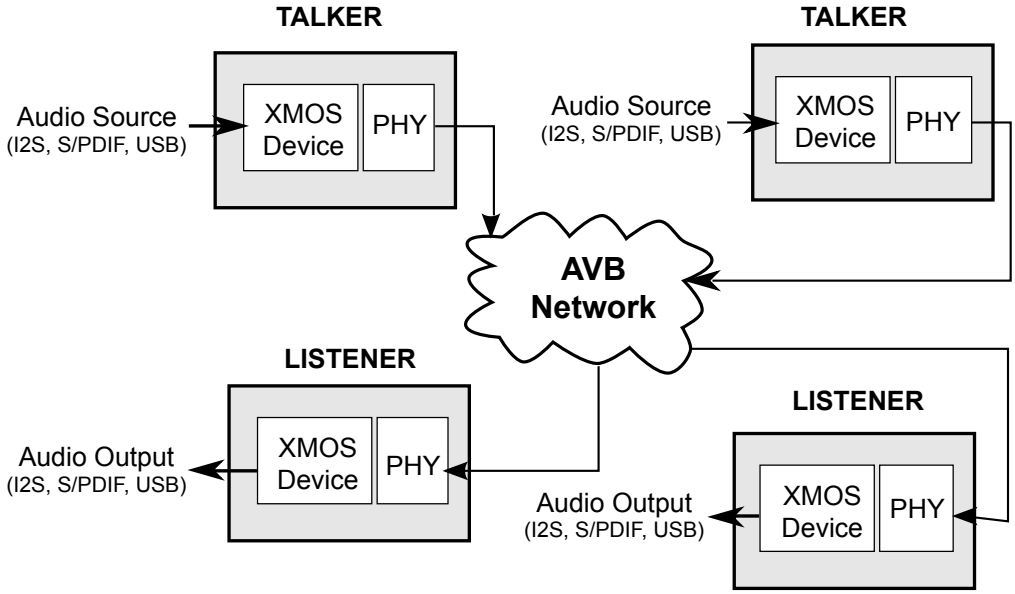
Version 0.1

XMOS

Copyright © 2010 XMOS Ltd. All Rights Reserved.

1 Summary

The XMOS Audio Visual Bridging (AVB) reference design can be used to stream synchronized audio over an ethernet network. The XMOS solution is based on event-driven programmable devices, which can transmit and receive multiple audio streams, and implement both talker and listener functionality.



1.1 XMOS AVB Features

- Supports all endpoint requirements: ethernet interface, packet processing, timing synchronisation, configuration/stream setup and media rate recovery can all be handled on device.
- Supports emerging AVB ethernet standards such as *IEEE 802.1as*, *IEEE 801.1Qav* and *IEEE P1722*.
- Flexible software based design implements both hardware interfaces and protocol layers in the same environment allowing, flexibility in system design and easy modification.
- Multiple audio hardware interfaces supported such as *I2S*, *TDM* and *S/PDIF*.

2 XMOS AVB Specification

Functionality	
Provides ethernet interface, audio transport, precise timing protocol clock synchronise and media clock recovery to streamed audio over ethernet.	
Supported Standards	
Ethernet	IEEE 802.3 (via MII)
AVB QoS	IEEE 802.1Qav
Precise Timing Protocol	IEEE 1588v2 or IEEE 802.1as
AVB Audio Over Ethernet	IEEE 1722
Audio Streaming	IEC 61883-6
Supported Devices	
XMOS Devices	XS1-G4 XS1-G2 XS1-L2
Requirements	
Development Tools	XMOS Desktop Tools v9.9.1 or later
Ethernet	1 $\times$ MII compatible 100Mbit PHY or 2 $\times$ MII compatible 100Mbit PHY or 1 $\times$ GMII compatible Gigabit PHY + programmable logic device
Audio	Audio input/output device (e.g. ADC/DAC audio CODEC) PLL/Frequency synthesizer chip to generate CODEC system clock
Boot/Storage	Compatible SPI Flash Device
Licensing and Support	
Reference code provided under license from XMOS. Contact support@xmos.com for details.	
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 allow synchronized, uninterrupted streaming with multiple talkers and listeners on a switched network infrastructure.

3.1 802.1as

802.1as defines a precise 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 and coordinated AVB traffic control.

This protocol is implemented in the XMOS AVB solution.

3.2 802.1Qav

802.1Qav defines a standard for buffering and forwarding of traffic through the network using particular flow control algorithms. It uses the global clock provided by *802.1as* to synchronize traffic forwarding and gives predictable latency control on media streams going 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.

3.3 802.1Qat

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

This protocol is not currently implemented in the AVB solution. It could be implemented as part of the user application on an XMOS device or by a separate host processor communicating with an XMOS device.

3.4 IEC 61883-6

IEC 61883-6 defines an audio data format that is contained in *IEEE P1722* streams.

The XMOS AVB solution uses *IEC 61883-6* to convey audio sample streams.

3.5 Emerging standards

3.5.1 IEEE P1722

IEEE P1722 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 P1722*. In addition it can use the *802.1as* timestamps to accurately recover the sample rate clock of the audio to match on the listener side.

4 XS1 Architecture

The XS1 architecture consists of one or more processing cores, called XCores, which have the following properties:

- Each XCore can execute up to eight threads concurrently, each at a speed of up to 100 MIPS. Each thread has a dedicated register set enabling it to operate as a logical core.
- The eight threads share a single 64 KByte unified memory with no access collisions.
- Integer and fixed point operations are provided for efficient DSP and cryptographic operations.
- I/O general purpose pins are provided, which can be programmed from software. Thread execution is deterministic and hence each thread can implement a hard real-time I/O task, regardless of the behavior of other threads.
- I/O pins are grouped into logical ports of width 1, 4, 8, 16 and 32 bits. Each port incorporates serialization/deserialization, synchronization with the external interface and precision timing.
- Each XCore incorporates eight timers that measure time relative to a 100 MHz reference clock.

The architecture is designed to support standard programming languages such as C. Extensions to standard languages, libraries, or the use of assembly language provide access to the full benefits of the instruction set.

4.1 XMOS XS1 Programmable Devices

The XS1 family is the first available implementation of the XS1 architecture. It includes the XS1-G4 device that integrates four XCore processors. Each device is connected to a high performance switch interconnect via four internal links called XMOS Links. Each link is capable of transferring data at 800 Mbits/second. The switch provides full connectivity between the cores on the programmable device, and also provides up to sixteen external XMOS Links. Each external link is capable of transferring data at up to 400 Mbits/second.

Other members of the family include the XS1-L1, XS1-G2 and XS1-L2. XS1 devices can be used standalone, or can be connected together using XMOS Links to create a network of XCore processors.

4.2 Software Libraries

The XS1 architecture implements hardware as software, so that hardware solutions are just software libraries. These software solutions are compiled and linked like ordinary C code into a binary file which can be loaded and executed on the device.

The XMOS AVB Reference Software is a library of source files that can be included in a larger application (templates and demos are provided with the reference design). The application can then be compiled and deployed onto a device to provide your AVB hardware solution.

4.3 XC Language

The XMOS originated XC language [xc87] is based on C. XC is a concurrent and real-time programming language designed to target the XS1 architecture. XC programs are easy to write and debug, leading to programs that are free from deadlocks, race conditions and memory violations and can be compiled to produce high performance multicore designs.

XC uses channels to provide high-speed bidirectional communication and synchronization between channel ends within threads on the same processor, a different processor on the same chip or a different chip.

5 XMOS AVB System Description

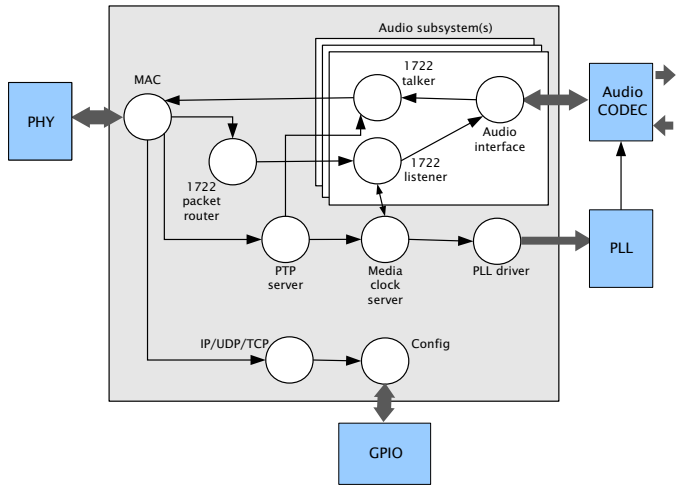


Figure ?? shows the overall structure of an XMOS AVB endpoint. An endpoint consists of five main components:

- The ethernet MAC
- The precise timing engine (PTP)
- Audio streaming components
- The media clock server
- Configuration and other application components

5.1 Ethernet MAC Component

The MAC component provides ethernet connectivity to the AVB solution. To use the component, a physical interface must be attached to the XCore ports to provide either an MII interface (for 100Mbps etherenet) or GMII interface (for Gigabit ethernet).

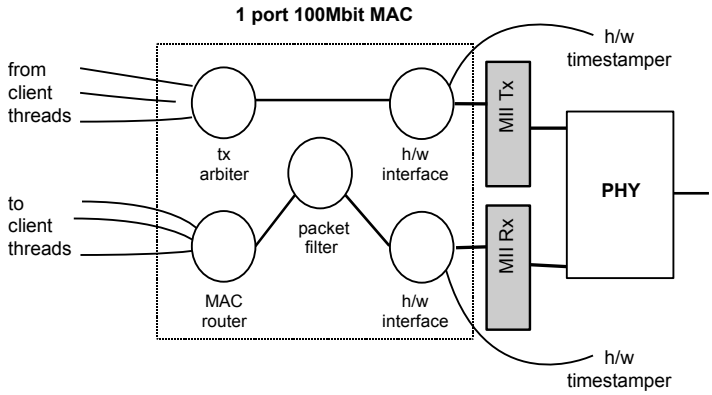
The MAC component supports two features that are necessary to implement AVB standards with precise timing and quality constraints.

- *Timestamping* - allows receipt and transmission of ethernet frames to be timestamped with respect to a clock (*i.e* a 100MHz reference clock can provide a resolution of 10ns).

- *Bandwidth control* - allows different channels to have different priorities and bandwidth restrictions to allow steady flow of outgoing media stream packets. The implementation provides flow control to satisfy the requirements of an AVB endpoint as specified in the upcoming *802.1Qav* standard.}

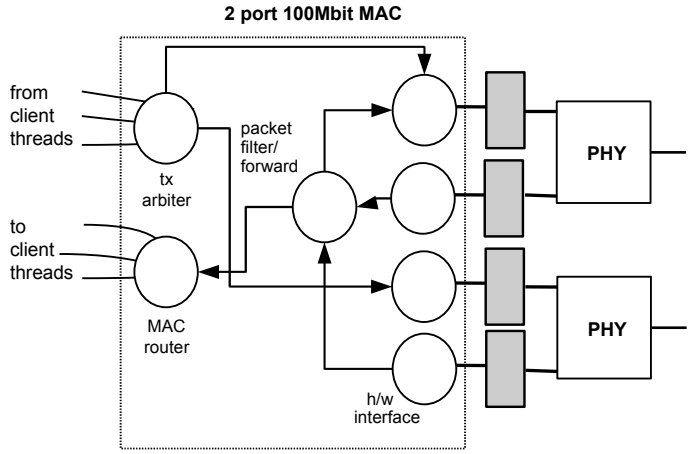
5.1.1 Single Port 100MBit interface

The single port 100MBit component consists of five threads (each running at 50MIPs or more) that must be run on the same core. These threads handle both the receiving and transmission of ethernet frames. The MAC component can be linked (via channels) to other components/threads 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 API, which configures and registers the filters. Details of the API used to configure MAC channels can be found in the header files in the src/mac/client/ directory within the reference design software.

5.1.2 Dual Port 100MBit interface



The dual port 100MBit component consists of seven threads. These threads handle the receiving, filtering, forwarding and transmission of ethernet frames. Traffic coming in on one port is forwarded to the other port unless it is specifically addressed to the mac address of the endpoint or a broadcast or multicast packet.

The client API for the dual port MAC is the same as the single port MAC. Each receive of a frame informs the application which source port the frame came from. Similarly each transmit needs to be directed to a port (or broadcast to both).

5.1.3 Single port Gigabit interface

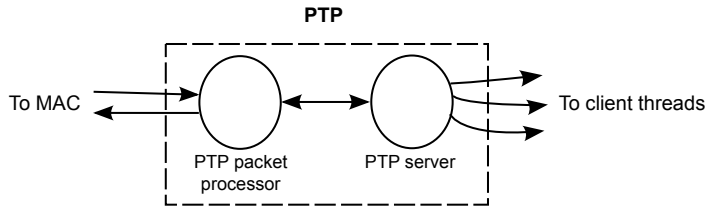
The gigabit interface communicates with an external programmable logic chip which interfaces to the GMII signal coming from a gigabit PHY. For details on the IP for this device and the hardware setup, please contact support@xmos.com.

The single port gigabit interface consists of four threads which must be run at 100 MIPS (i.e. for an XS1-G4 they must be the only threads on one core). In addition, two other threads are required on another core for buffering transmission of packets. The gigabit MAC filters out AVB audio packets and packets destined for the endpoint MAC address. These packets are then buffered in a 40k packet buffer which is routed to the rest of the chip at ~300MBit/s.

5.2 Precise Timing Protocol Component

The precise timing protocol component (PTP) provides a system with a notion of global time on a network. The component supports the *IEEE 1588v2* timing protocol

and the upcoming *AVB 802.1as* timing protocol. It allows synchronization of the presentation and playback rate of media streams across a network (see Figure ??).



The timing component, which consists of two threads, 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 maintainance of global time requires no application interaction with the component.

The PTP component can be configured at runtime to be a *PTP grandmaster* or a *PTP slave*. 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 threads 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 threads can communicate with the server using the API described in the file:

src/ptp/client/ptp_client.h

- 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 sample rate clock used to time sampling (though it can be used to create the sample clock)

5.3 Audio Components

5.3.1 AVB Streams, Channels, Talkers and Listeners

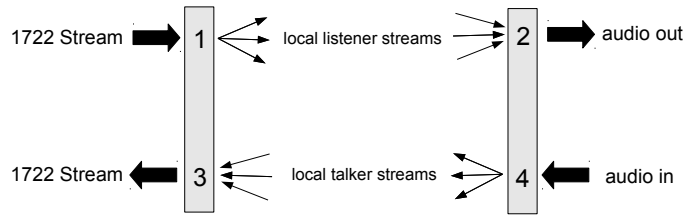
Audio is transported in streams of data, where each stream may have multiple channels. Endpoints producing the streams are called *Talkers and those receiving them are called *Listeners*. Each stream on the network has a unique 64-bit stream ID.

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 may be a multicast address (i.e. 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 registering to receive it. If sufficient bandwidth is not available, this registration may 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, which can be either unicast (one-to-one) or multicast (one-to-many).
- Routing 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.
- The stream ID is the only information you can obtain about a stream before registering to listen to it. Any other information about the stream must be communicated by a higher level protocol (see Section).

5.3.2 Internal Routing



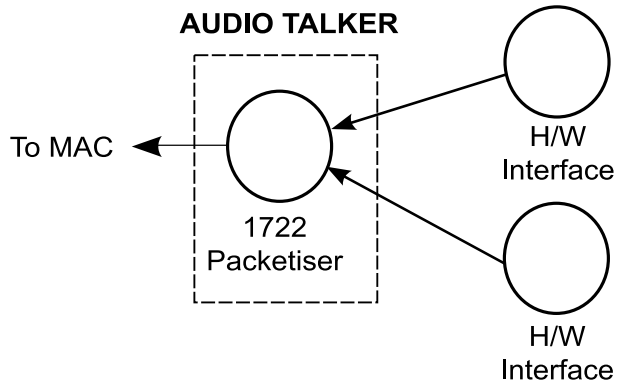
As described in the previous section, an IEEE P1722 audio stream may consist of many channels. These channels need to be routed to particular audio I/Os on the endpoint. To achieve maximum flexibility the XMOS design uses intermediate *local* audio streams to route audio.

The above figure shows the breakdown of 1722 streams into local streams. The Figure shows four points where transitions to and from local streams occur.

1. When a 1722 stream is received, its channels are mapped to local listener streams. The simplest (and most efficient) mapping is for one 1722 stream to map to one local stream. This mapping can be configured dynamically (can be changed at runtime by the configuration component).
2. The digital hardware interface maps local listener streams to audio outputs. This mapping is fixed and is configured statically in the software.
3. Several local talker streams can be combined into a 1722 stream. This mapping is dynamic.
4. The digital hardware interface maps digital audio inputs to local talker streams. This mapping is fixed (i.e. cannot be changed at runtime).

The configuration of the mappings is handled through the 1722 client API. This can be found in the file src/avb_1722/client/avb_1722.h.

5.3.3 Audio Talker



The talker component consists of one thread which creates *IEEE P1722* packets and passes the audio samples onto the MAC. Audio samples are passed to this component via a shared memory sample FIFO representing a local talker stream. Samples are pushed into this fifo from a different thread implementing the audio hardware interface. The packetizer thread removes the samples and combines them into *IEEE P1722* ethernet packets to be transmitted via the MAC component.

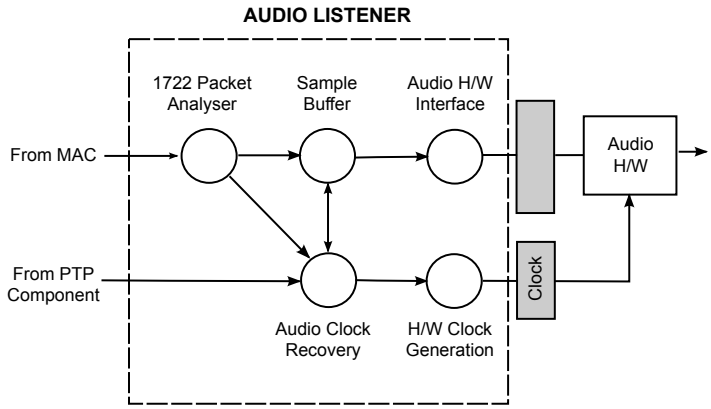
When the packets are created the timestamps are converted to the time domain of the global clock provided by the PTP component, and a fixed offset is added to the timestamps to provide the *presentation time* of the samples (*i.e* the time at which the sample should be played by a listener).

The audio talker can be configured on how to combine local audio streams to become *IEEE P1722* streams. However, since samples are passed via a shared memory interface a talker can only combine talker streams that are created on the same core as the talker.

5.3.4 The 1722 Packet Router

The *IEEE P1722* packet router component routes *IEEE P1722* audio packets to the listener components in the system. It controls the routing by stream ID. The router is configured over a shared memory interface using the same client functions that control the listener components. It is important that any configuration threads that affect the packet router are on the same core as the packet router.

5.3.5 Audio Listener



The audio listener component takes *IEEE P1722* packets from the packet router and converts them into a sample stream to be fed into a buffer thread. Each audio listener component can listen to several *IEEE P1722* streams. The configuration of which streams it listens to and how they are split into local listener streams is configured by a channel interface from a separate configuration thread.

5.3.6 Audio Hardware Interfaces

The audio hardware interface that the audio component connects to can be configured to specific hardware.

- A hardware interface component may have to option to connect to a listener component or a talker component.
- An interface may have a link with the media clock server to drive its timing.
- The mapping between inputs/outputs and local audio streams is usually fixed and can be configured by a parameter to the function implementing the audio h/w thread. See the specific interface header file for details.

5.4 Media Clocks

A media clock controls the rate at which information is passed to an external media playing device. For example, an audio sample 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 three sources:

- An incoming clock signal on a port.
- The rate of the incoming samples (provided by the *IEEE P1722* timestamps) on an incoming *IEEE P1722* audio stream.
- The global PTP clock.

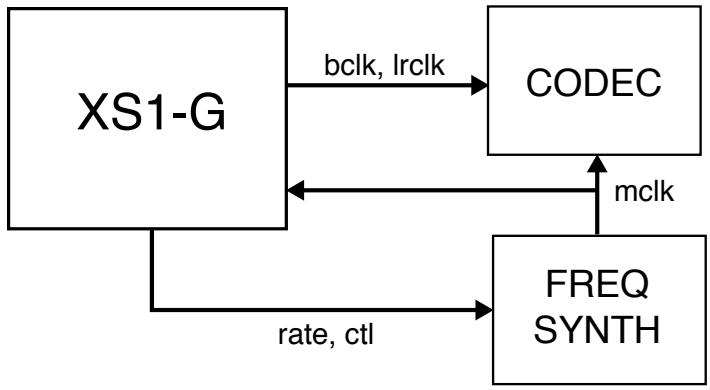
A hardware interface can be tied to a particular media clock, allowing the media output from the XMOS device to be synchronized with other devices on the network.

All media clocks are maintained by the media clock server component. This component takes one thread of processing and 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 P1722* streams. It then sends control information back to ensure the listening component honors the presentation time of the incoming stream.

5.4.1 Driving an external clock generator

A high quality, low jitter master clock is required to drive an audio codec and this clock 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 (e.g a chip such as the Cirrus Logic CS2300 or similar). The recommended configuration is as in the block diagram below:



The XS1 device provides control to the frequency synthesizer (which is often a divided down clock to drive the higher required rate) and the frequency synthesizer provides

the master clock to the codec. The sample bit clocks and word clocks are then provided to the codec by the XS1 device.

To drive an external clock generator requires one thread connected to the media clock server. See src/audio/media_clocks/external for details.

5.5 Configuration and application threads

The AVB standard protocols control the time synchronization, streaming and routing of audio data. However, it is likely that a higher level configuration protocol will be required to configure an AVB endpoint.

Such a protocol needs two parts: discovery and control. The discovery part must determine higher level information about other endpoints on the network. For example:

- Discover which other Talkers/Listeners are on the network
- Discover which streams are available and meta-information about them (sample rate, description, global clock synchronization participation *etc.*)

The control part controls the device. For example it controls:

- The streams the Talker outputs/Listener inputs.
- Audio input/output (sample rate, gain, dsp *etc.*)
- Other non-audio aspects.

The AVB software solution includes a port of the uIP protocol stack which is a small memory footprint stack that can be used for UDP/TCP communication to aid implementation of an upper layer configuration protocol. Future software updates will include an implementation of Zeroconf standards (IP link local addressing, multicast DNS and DNS service discovery) to aid in initialisation and discovery. These libraries will aid in supporting configuration protocols such as the forth-coming 1722.1 AVB standard.

6 Xmos AVB API

6.1 The top-level main

6.2 Server functions

void media_clock_sever The media clock server

void ptp_server The ptp server

6.3 Main thread functions

6.4 Configuration functions

void avb_initchanend *media_ctl[]*, chanend *?listener_ctl[]*, chanend *?talker_ctl[]*, chanend *timebase_ctl*, chanend *c_mac_rx*, chanend *c_mac_tx*, unsigned char *mac_addr[]* Initialize the avb system.

Parameters

- **media_ctl** – The array of chanends that connect to the media units in the system.
- **listener_ctl** – The array of chanends that connect to the listener units in the system (possibly null)
- **talker_ctl** – The array of chanends that connect to the talker units in the system (possibly null)

7 Ethernet Client API

int mac_rxchanend *c_mac*, unsigned char *buffer[]*, unsigned int *&src_port* This function receives a complete frame (i.e. src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32.

Parameters

- **c_mac** – chanend connect to the ethernet server
- **buffer** – The buffer to fill with the incoming packet
- **src_port** – A reference parameter to be filled with the ethernet port the packet came from.

Returns:

The number of bytes in the frame.

Notes:

1. This function is a blocking call, (i.e. it will wait until a complete packet is received).
2. Only the packets whih pass CRC32 are processed.

int mac_rx_timedchanend *c_mac*, unsigned char *buffer[]*, REFERENCE_PARAM(unsigned *int*, time), REFERENCE_PARAM(unsigned *int*, src_port) This function receives a complete frame (i.e. src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32. It also timestamps the arrival of the frame

Parameters

- **c_mac** – chanend connect to the ethernet server
- **buffer** – The buffer to fill with the incoming packet
- **time** – A reference parameter to be filled with the timestamp of the packet
- **src_port** – A reference parameter to be filled with the ethernet port the packet came from.

Returns:

The number of bytes in the frame.

Notes:

1. This function is a blocking call, (i.e. it will wait until a complete packet is received).
2. Only the packets whih pass CRC32 are processed.

8 PTP Client API

void ptp_serverchanend *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.

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 (PTP_GRANDMASTER_CAPABLE or PTP_SLAVE_ONLY)

ptp_request_time_infochanend *ptp_server* This functions requests a **'ptp_time_info'_** structure from the ptp server. This is an asynchonous call so needs to be completed later with a call to ptp_get_requested_time_info.

Parameters

- **ptp_server** – chanend connecting to the ptp server

See Also:

ptp_get_requested_time_info **'ptp_get_time_info'_** **'ptp_request_time_info_mod64'_**

ptp_get_requested_time_infochanend *ptp_server* This functions receives a **'ptp_time_info'_** structure from the ptp server. This completes an asynchonous 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.

Parameters

- **ptp_server** – chanend connecting to the ptp server
- **ptp_time_info** – a reference parameter to be filled with the time information structure

See Also:

ptp_request_time_info **'ptp_get_time_info'_** **'ptp_request_time_info_mod64'_**

9 XTCP API