Application Note: AN00202

Gigabit Ethernet AVB endpoint example using I2S master

This application note demonstrates a Gigabit Ethernet AVB endpoint that streams uncompressed audio over an Ethernet AVB network with guaranteed Quality of Service, low latency and time synchronization. It shows how to interface with a high performance audio codec via the I2S library.

The application is configured to provide a single Talker and Listener stream of 8 audio channels at up to 192kHz sampling rate.

The example also shows plug-and-play multichannel recording and playback with Apple Mac hardware running OS X 10.10.

Required tools and libraries

The code in this application note is known to work on version 14.1.0 of the xTIMEcomposer tools suite, it may work on other versions.

The application depends on the following libraries:

- lib_tsn (>=7.0.0)
- lib_i2s (>=2.0.1)

Required hardware

The application note is designed to run on the XMOS xCORE-200 Multichannel Audio platform version 2.

There is no dependency on this hardware and the firmware can be modified to run on any xCORE XE/XEF series device with the required external hardware.

The firmware was interoperability tested with a Late 2013 MacBook Pro running OS X version 10.10.3.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the IEEE AVB/TSN standards, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- The example uses various libraries, full details of the functionality of a library can be found in its user guide[2].

[1] http://www.xmos.com/published/glossary
[2] http://www.xmos.com/support/libraries

1 Overview

1.1 Introduction

The XMOS xCORE microcontroller family is an ideal platform for implementing Audio Video Bridging (AVB) endpoints. This application note describes how a standards-compliant endpoint can be instantiated and configured on the xCORE-200 Multichannel Audio platform using the AVB/TSN library and interfaced to audio codecs using an I2S master interface.

The example endpoint application features:

- 10/100/1000Mbps Ethernet MAC with AVB support
- 1722 61883-6 audio Talker and Listener (simultaneous) support
- I2S master interface supporting 8 input and 8 output channels up to 24 bit 192kHz
- 1722 MAAP support for Talker multicast MAC address acquisition
- 802.1Q MRP, MVRP, MSRP protocols
- gPTP server and protocol
- Audio clock recovery and interface to PLL clock generator
- Support for 1722.1 AVDECC: ADP, AECP (AEM) and ACMP
- Firmware update via 1722.1 EFU

1.2 Block diagram

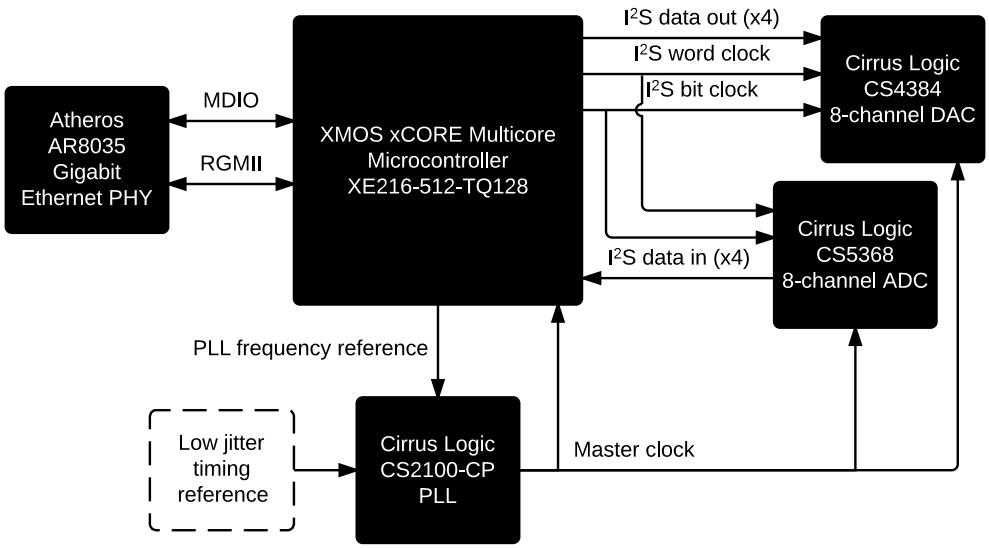


Figure 1: Block diagram of the xCORE-200 Multichannel audio hardware relevant to AVB

2 A Gigabit Ethernet AVB endpoint example

Figure 2 shows the high level task and communication structure for the application. The example consists of many tasks running in parallel.

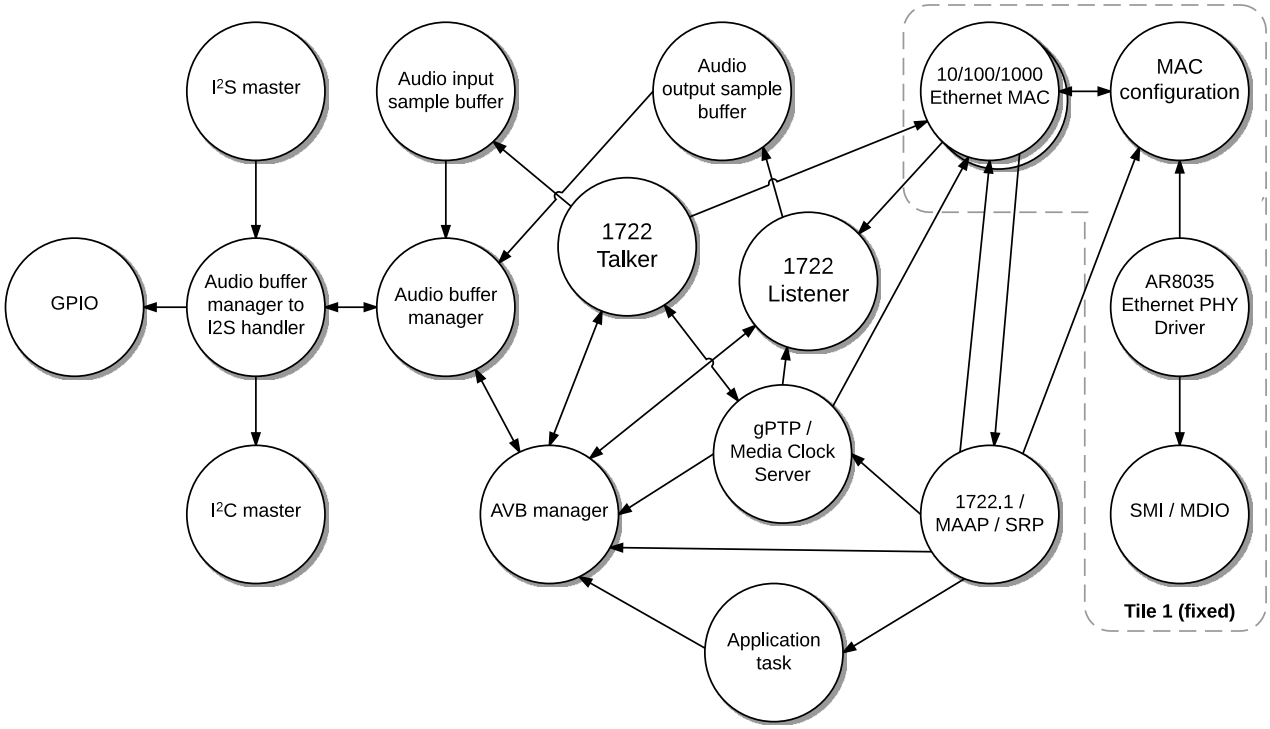


Figure 2: Task diagram of the AVB endpoint example

Specifically:

- The Gigabit Ethernet MAC which consists of 7 tasks and handles the RGMII interface and packet queuing.
- The Ethernet MAC configuration task which allows client tasks to configure MAC address filtering and other parameters.
- The AR8035 Ethernet PHY driver which configures PHY registers and periodically reads the link status.
- The SMI task which implements the MDIO register interface to the Ethernet PHY and communicates with the PHY driver.

The above tasks must be placed on Tile 1 of the XE216 microcontroller.

- The I2S master task from the XMOS I2S library which handles the I2S digital audio interface.
- The audio buffer manager to I2S handler which receives callbacks from the I2S task, initializes the audio codecs and PLL by communicating with the I2C master task (from the XMOS I2C library) and configures reset and clock frequency pins via a GPIO task.
- The audio input sample buffer task, which provides a handle to a double buffered audio buffer that shares memory between the audio buffer manager task and 1722 Talker task.
- The audio buffer manager task which performs buffering operations for input and output, then communicates free buffer locations with the I2S handler via a channel.
- The audio output sample buffer task, which provides a handle to sample FIFOs that share memory

between the audio buffer manager task and the 1722 Listener.

- The 1722 Talker which packetizes audio samples from the audio input sample buffer into a 1722 stream with a presentation timestamp and stream ID, then forwards them to the Ethernet MAC for transmission.
- The 1722 Listener which receives 1722 stream packets from the Ethernet MAC, depacketizes audio samples from them, then buffers the samples in an audio output FIFO. The Listener also communicates with the media clock server to maintain the 1722 presentation time and audio clock recovery.
- The gPTP and media clock server task which maintains a global time reference and implements clock recovery to synchronize the frequency and phase of the audio clock to the audio clock master on the network. This task also generates the low frequency reference signal to the Cirrus Logic CS2100-CP PLL.
- The 1722.1, MAAP and SRP task which combines the protocol stacks necessary to provide stream reservation and endpoint control.
- The AVB manager task which co-ordinates the communication and setup of all the tasks above.
- Finally, the application task, which sets up and implements application specific configuration and behavior via communication with the AVB manager and 1722.1 tasks.

Note that the application consists of 23 tasks implemented on 14 logical cores. Combined or distributed tasks will be scheduled as required when they are communicated with by the other tasks.

2.1 Example directory structure description

File	Description
`Makefile`	An XMOS application makefile containing board target, used modules (libraries) and compiler flags
`app_build_info`	Application specific build step, used to pre-process 1722.1 descriptors
`src/1722_1_callbacks.xc`	Callback functions that are executed on 1722.1 connection events
`src/XR-AUDIO-216-MC.xn`	XN target file for the xCORE-200 multichannel audio board. Describes system frequency, oscillator values and QSPI flash type
`src/aem_descriptors.h.in`	Header file containing 1722.1 AEM descriptors and templates for the AVB endpoint. Pre-processed by a Python script to generate the actual header file.
`src/aem_entity_strings.h.in`	Header file containing 1722.1 AEM descriptor strings for the AVB endpoint. Pre-processed by a Python script to generate the actual header file.
`src/avb_conf.h`	Main configuration header file for application and TSN library
`src/config.xscope`	XScope configuration file containing probes and printing parameters
`src/debug_conf.h`	Configuration parameters for the lib_logging debug printing library
`src/main.xc`	Application code and multicore main() function

Table 1: Key application files

2.2 Makefile additions for this example

This example uses XMOS libraries which can be included in the project via the USED_MODULES variable in the Makefile:

```
USED_MODULES = lib_tsn(>=7.0.0) lib_i2s(>=2.0.1)
```

A version number requirement is specified with the libraries to guarantee compatibility.

The compiler flags include -g (to enable debug information) and -report (to give a resource usage report after compilation).

The -fxscope flag is used to enable xSCOPE output for real-time debug printing on the host connected to the xTAG adapter. A define -DRGMII=1 is passed to the Ethernet library to indicate RGMII is being used and -lquadflash is used to link the Quad SPI flash library, which is bundled with the tools:

```
XCC_FLAGS = -Os -save-temps -g -report -fxscope -DRGMII=1 -lquadflash
```

Compiler flags specific to a particular file can be specified in the Makefile as follows:

```
XCC_FLAGS_main.xc = $(XCC_FLAGS) -falways-inline -O3
```

2.3 AVB endpoint configuration defines

The avb_conf.h file provides configuration #defines for the AVB endpoint and overrides defaults in the lib_tsn library. These can be altered to suit the particular endpoint configuration required.

```
/******** Endpoint audio and clocking parameters ************************************************/

/* Talker configuration */

/** The total number of AVB sources (streams that are to be transmitted). */
#define AVB_NUM_SOURCES 1
/** The total number or Talker components (typically the number of
  * tasks running the `avb_1722_talker` function). */
#define AVB_NUM_TALKER_UNITS 1
/** The total number of media inputs (typically number of I2S input channels). */
#define AVB_NUM_MEDIA_INPUTS 8
/** Enable the 1722.1 Talker functionality */
#define AVB_1722_1_TALKER_ENABLED 1

/* Listener configuration */

/** The total number of AVB sinks (incoming streams that can be listened to) */
#define AVB_NUM_SINKS 1
/** The total number or listener components
  * (typically the number of tasks running the `avb_1722_listener` function) */
#define AVB_NUM_LISTENER_UNITS 1
/** The total number of media outputs (typically the number of I2S output channels). */
#define AVB_NUM_MEDIA_OUTPUTS 8
/** Enable the 1722.1 Listener functionality */
#define AVB_1722_1_LISTENER_ENABLED 1
```

```
/** The maximum number of channels permitted per 1722 Talker stream */
#define AVB_MAX_CHANNELS_PER_TALKER_STREAM 8
/** The maximum number of channels permitted per 1722 Listener stream */
#define AVB_MAX_CHANNELS_PER_LISTENER_STREAM 8

/** Use 61883-6 audio format for 1722 streams */
#define AVB_1722_FORMAT_61883_6 1

/** The number of components in the endpoint that will register and initialize media FIFOs
    (typically an audio interface component such as I2S). */
#define AVB_NUM_MEDIA_UNITS 1

/** The number of media clocks in the endpoint. Typically the number of clock domains, each with a
  * separate PLL and master clock. */
#define AVB_NUM_MEDIA_CLOCKS 1

/** The maximum sample rate in Hz of audio that is to be input or output */
#define AVB_MAX_AUDIO_SAMPLE_RATE 192000

/** Enable 1722 MAAP on the device, required for Talkers */
#define AVB_ENABLE_1722_MAAP 1

/******** 1722.1 parameters ********************************************************************/

/** Enable 1722.1 AVDECC on the entity */
#define AVB_ENABLE_1722_1 1
/** The entity capability flags as reported by 1722.1 ADP */
#define AVB_1722_1_ADP_ENTITY_CAPABILITIES (AVB_1722_1_ADP_ENTITY_CAPABILITIES_AEM_SUPPORTED| \
                                            AVB_1722_1_ADP_ENTITY_CAPABILITIES_CLASS_A_SUPPORTED| \
                                            AVB_1722_1_ADP_ENTITY_CAPABILITIES_GPTP_SUPPORTED| \
                                            AVB_1722_1_ADP_ENTITY_CAPABILITIES_EFU_MODE| \
                                            AVB_1722_1_ADP_ENTITY_CAPABILITIES_ADDRESS_ACCESS_SUPPORTED| \
                                            AVB_1722_1_ADP_ENTITY_CAPABILITIES_AEM_IDENTIFY_CONTROL_INDEX_VALID)

/** The model ID of the device as reported by 1722.1 ADP and AEM */
#define AVB_1722_1_ADP_MODEL_ID 0x1234

/** The list of AEM control descriptor IDs */
enum aem_control_indices {
    DESCRIPTOR_INDEX_CONTROL_IDENTIFY = 0,
};
/** Enable 1722.1 Entity Firmware Update functionality on the entity. */
#define AVB_1722_1_FIRMWARE_UPGRADE_ENABLED 1
/** Enable 1722.1 ACMP fast connect functionality on the entity. */
#define AVB_1722_1_FAST_CONNECT_ENABLED 0
/** Enable 1722.1 Controller functionality on the entity. */
#define AVB_1722_1_CONTROLLER_ENABLED 0

/******** Flash parameters *********************************************************************/

/** The maximum size in bytes of an XCore program image */
#define FLASH_MAX_UPGRADE_IMAGE_SIZE (128 * 1024)
/** The page size of the flash used */
#define FLASH_PAGE_SIZE (256)
```

2.4 Declaring resources used by the application

The example uses several hardware ports and clocks to drive and read I/O. These are declared at the beginning of `main.xc`

```
// Ports and clocks used by the application
on tile[0]: otp_ports_t otp_ports0 = OTP_PORTS_INITIALIZER; // Ports are hardwired to internal OTP for reading
                                                           // MAC address and serial number
// Fixed QSPI flash ports that are used for firmware upgrade and persistent data storage
on tile[0]: fl_QSPIPorts qspi_ports =
{
  XS1_PORT_1B,
  XS1_PORT_1C,
  XS1_PORT_4B,
  XS1_CLKBLK_1
};

on tile[1]: rgmii_ports_t rgmii_ports = RGMII_PORTS_INITIALIZER; // Fixed RGMII ports on Tile 1
on tile[1]: port p_smi_mdio = XS1_PORT_1C;
on tile[1]: port p_smi_mdc = XS1_PORT_1D;
on tile[1]: port p_eth_reset = XS1_PORT_4A;

on tile[1]: out port p_leds_row = XS1_PORT_4C;
on tile[1]: out port p_leds_column = XS1_PORT_4D;
on tile[0]: port p_i2c = XS1_PORT_4A;

// I2S ports and clocks
on tile[0]: out buffered port:32 p_fs[1] = { XS1_PORT_1A }; // Low frequency PLL frequency reference
on tile[0]: out buffered port:32 p_i2s_lrclk = XS1_PORT_1G;
on tile[0]: out buffered port:32 p_i2s_bclk = XS1_PORT_1H;
on tile[0]: in port p_i2s_mclk = XS1_PORT_1F;
on tile[0]: out buffered port:32 p_aud_dout[4] = {XS1_PORT_1M, XS1_PORT_1N, XS1_PORT_1O, XS1_PORT_1P};
on tile[0]: in buffered port:32 p_aud_din[4] = {XS1_PORT_1I, XS1_PORT_1J, XS1_PORT_1K, XS1_PORT_1L};
on tile[0]: clock clk_i2s_bclk = XS1_CLKBLK_3;
on tile[0]: clock clk_i2s_mclk = XS1_CLKBLK_4;

on tile[0]: out port p_audio_shared = XS1_PORT_8C;
```

These ports are mapped to external pins on the XE216 device. Some ports can be moved if required to support a different hardware portmap on another board.

2.5 The application main() function

The source code below is taken from the main function of the example application file `main.xc`.

The declarations before the `par` create the connections between the tasks (see Figure 2). Tasks are connected by passing one of these declared variables to both tasks. Some of the connections are arrays which can connect one task to many others.

```
int main(void)
{
  // Ethernet interfaces and channels
  ethernet_cfg_if i_eth_cfg[NUM_ETH_CFG_CLIENTS];
  ethernet_rx_if i_eth_rx_lp[NUM_ETH_RX_LP_CLIENTS];
  ethernet_tx_if i_eth_tx_lp[NUM_ETH_TX_LP_CLIENTS];
  streaming chan c_eth_rx_hp;
  streaming chan c_eth_tx_hp;
  smi_if i_smi;
  streaming chan c_rgmii_cfg;

  // PTP channels
  chan c_ptp[NUM_PTP_CHANS];

  // AVB unit control
  chan c_talker_ctl[AVB_NUM_TALKER_UNITS];
  chan c_listener_ctl[AVB_NUM_LISTENER_UNITS];
  chan c_buf_ctl[AVB_NUM_LISTENER_UNITS];

  // Media control
  chan c_media_ctl[AVB_NUM_MEDIA_UNITS];
  interface media_clock_if i_media_clock_ctl;

  // Core AVB interface and callbacks
  interface avb_interface i_avb[NUM_AVB_MANAGER_CHANS];
  interface avb_1722_1_control_callbacks i_1722_1_entity;

  // I2C and GPIO interfaces
  i2c_master_if i_i2c[NUM_I2C_IFS];
  interface output_gpio_if i_gpio[4];

  // I2S and audio buffering interfaces
  i2s_callback_if i_i2s;
  streaming chan c_audio;
  interface push_if i_audio_in_push;
  interface pull_if i_audio_in_pull;
  interface push_if i_audio_out_push;
  interface pull_if i_audio_out_pull;
```

The par functionality describes several tasks running in parallel across multiple logical cores on 2 tiles.

2.6 The Ethernet MAC and PHY configuration

The first four tasks are related to the Ethernet MAC and its configuration. These tasks must run on tile 1 of the xCORE due to the fixed nature of the RGMII ports. The `rgmii_ethernet_mac` task consumes seven xCORE logical cores internally and the `rgmii_ethernet_mac_config`, `ar8035_phy_driver` and `smi` tasks are combined onto the single spare core on tile 1.

```
on tile[1]: rgmii_ethernet_mac(i_eth_rx_lp, NUM_ETH_RX_LP_CLIENTS,
                               i_eth_tx_lp, NUM_ETH_TX_LP_CLIENTS,
                               c_eth_rx_hp, c_eth_tx_hp,
                               c_rgmii_cfg,
                               rgmii_ports,
                               ETHERNET_DISABLE_SHAPER);

on tile[1].core[0]: rgmii_ethernet_mac_config(i_eth_cfg, NUM_ETH_CFG_CLIENTS, c_rgmii_cfg);
on tile[1].core[0]: ar8035_phy_driver(i_smi, i_eth_cfg[MAC_CFG_TO_PHY_DRIVER]);

on tile[1]: [[distribute]] smi(i_smi, p_smi_mdio, p_smi_mdc);
```

The ar8035_phy_driver task is defined within the application and is intended to be changed if a different Ethernet PHY is used. The function first applies a reset pulse to the Ethernet PHY of 1000 ms:

```
void ar8035_phy_driver(client interface smi_if smi,
                       client interface ethernet_cfg_if eth) {
  ethernet_link_state_t link_state = ETHERNET_LINK_DOWN;
  ethernet_speed_t link_speed = LINK_1000_MBPS_FULL_DUPLEX;
  const int phy_reset_delay_ms = 1;
  const int link_poll_period_ms = 1000;
  const int phy_address = 0x4;
  timer tmr;
  int t;
  tmr :> t;
  p_eth_reset <: 0;
  delay_milliseconds(phy_reset_delay_ms);
  p_eth_reset <: 0xf;
```

Next, the Ethernet MAC is configured with the ingress and egress latencies in nanoseconds through the PHY. These latencies are required to correct for the offset between the 802.1AS timestamp measurement plane relative to the reference plane, as described in IEEE 802.1AS section 8.4.3, and are required for compliant behavior.

The latencies vary with the speed of the link and therefor a speed parameter is also provided. These values are PHY specific and must be updated if a different PHY is used. They can be obtained experimentally or from the PHY vendor.

```
eth.set_ingress_timestamp_latency(0, LINK_1000_MBPS_FULL_DUPLEX, 200);
eth.set_egress_timestamp_latency(0, LINK_1000_MBPS_FULL_DUPLEX, 200);

eth.set_ingress_timestamp_latency(0, LINK_100_MBPS_FULL_DUPLEX, 350);
eth.set_egress_timestamp_latency(0, LINK_100_MBPS_FULL_DUPLEX, 350);
```

A library function smi_phy_is_powered_down is used to wait until the PHY is powered on before reading or writing MDIO registers. The PHY is then configured via the SMI register interface. Energy Efficient Ethernet features of the PHY must be disabled for AVB operation. The smi_configure library function configures the PHY speed and auto-negotiation parameters.

```
while (smi_phy_is_powered_down(smi, phy_address));

// Disable smartspeed
smi.write_reg(phy_address, 0x14, 0x80C);
// Disable hibernation
smi.write_reg(phy_address, 0x1D, 0xB);
smi.write_reg(phy_address, 0x1E, 0x3C40);
// Disable smart EEE
smi.write_reg(phy_address, 0x0D, 3);
smi.write_reg(phy_address, 0x0E, 0x805D);
smi.write_reg(phy_address, 0x0D, 0x4003);
smi.write_reg(phy_address, 0x0E, 0x1000);
// Disable EEE auto-neg advertisement
smi.write_reg(phy_address, 0x0D, 7);
smi.write_reg(phy_address, 0x0E, 0x3C);
smi.write_reg(phy_address, 0x0D, 0x4003);
smi.write_reg(phy_address, 0x0E, 0);

smi_configure(smi, phy_address, LINK_1000_MBPS_FULL_DUPLEX, SMI_ENABLE_AUTONEG);
```

Finally, the task periodically polls the Ethernet link state (up/down) using the SMI library function

smi_get_link_state. Since link speed is not provided by a standard PHY register, a AR8035 specific register must be read to obtain it. If the state has changed, the task communicates with the MAC to inform it of the change. This information is then proxied through the Ethernet MAC to the AVB stack.

```
// Periodically check the link status
while (1) {
  select {
  case tmr when timerafter(t) :> t:
    ethernet_link_state_t new_state = smi_get_link_state(smi, phy_address);
    // Read AR8035 status register bits 15:14 to get the current link speed
    if (new_state == ETHERNET_LINK_UP) {
      link_speed = (ethernet_speed_t)(smi.read_reg(phy_address, 0x11) >> 14) & 3;
    }
    if (new_state != link_state) {
      link_state = new_state;
      eth.set_link_state(0, new_state, link_speed);
    }
    t += link_poll_period_ms * XS1_TIMER_KHZ;
```

2.7 Configuring GPIO

Tile 0 implements the AVB endpoint functionality and thus the non-Ethernet I/O must reside on the ports on this tile.

An I2C bus is used to configure the codecs and PLL on the multichannel audio board. The I2C clock and data pins have been placed on a 4-bit port and therefore the multi-bit port I2C implementation is required. This is instantiated by the i2c_master_single_port task, which takes the port, bus speed (100Kbps) and bit positions of the clock and data within the multibit port as parameters.

```
on tile[0]: [[distribute]] i2c_master_single_port(i_i2c, NUM_I2C_IFS, p_i2c, 100, 0, 1, 0);
on tile[0]: [[distribute]] output_gpio(i_gpio, 4, p_audio_shared, gpio_pin_map);
```

An output GPIO task allows a multibit port to be shared in a safe manner via xC interfaces. It takes a 4-bit port and mapping of pins within that port to be used for ADC, DAC reset lines and other strapping functions.

Both tasks are marked with a [[distributable]] attribute which means that they do not consume a logical core and are scheduled on the core that the client interface is used.

2.8 The I2S master task

Before the I2S master interface from lib_i2s is instantiated in its own logical core in the par, the core is set into high priority scheduling which guarantees 100 MIPS and enables up to 192 kHz I2S operation. The master clock input from the PLL is configured to clock a clock block used by the I2S task.

```
set_core_high_priority_on();
configure_clock_src(clk_i2s_mclk, p_i2s_mclk);
start_clock(clk_i2s_mclk);
```

The I2S ports and clocks are passed as parameters to the i2s_master function.

```
i2s_master(i_i2s,
           p_aud_dout, AVB_NUM_MEDIA_OUTPUTS/2,
           p_aud_din, AVB_NUM_MEDIA_INPUTS/2,
           p_i2s_bclk,
           p_i2s_lrclk,
           clk_i2s_bclk,
           clk_i2s_mclk);
```

2.9 The buffer manager to I2S handler task

The i2s_master task is connected to the buffer_manager_to_i2s task which is defined in the application. The connection between the tasks will make 'callbacks' from the i2s_master task to the application. The prototype of the I2S handling task is as below:

```
void buffer_manager_to_i2s(server i2s_callback_if i2s,
                           streaming chanend c_audio,
                           client interface i2c_master_if i2c,
                           client output_gpio_if dac_reset,
                           client output_gpio_if adc_reset,
                           client output_gpio_if pll_select,
                           client output_gpio_if mclk_select)
```

Note that:

- The task takes the server side of the i2s_callback_if interface. This means that the I2S master task will make calls into this task.
- It also takes the client side of connections to the I2C bus and GPIO tasks. These allow this task to make calls to configure the hardware.
- Finally, the task takes a streaming chanend argument which is an un-typed channel connection to the buffer manager task for sending/receiving samples.
- The task is marked as [[distributable]] - this means that the task will only be implementing callbacks and can be run on the same logical core as the task making the calls.
- It is also marked as [[always_inline]] to guarantee performance.

The task implements the callbacks via a 'while(1)-select' construct. This represents an infinite loop that repeatedly responds to calls from other tasks:

The calls it will respond to are defined in the i2s_callback_if in i2s.h. There are four callbacks: initialization, sending a sample, receiving a sample and checking for restart.

2.9.1 Configuring the audio hardware

The init callback occurs when the I2S bus initializes. At this point the task will configure the audio hardware and buffering. The I2S mode and clock ratio fields of the i2s_config structure are first set (see the I2S library documentation for more details).

The section of code following this makes calls on the GPIO and I2C interfaces to configure the clock selection and codecs on the board.

```
case i2s.init(i2s_config_t &?i2s_config, tdm_config_t &?tdm_config):
  // Receive the first free buffer and initial sample rate
  unsafe {
    c_audio :> double_buffer;
    p_in_frame = &double_buffer->buffer[double_buffer->active_buffer];
    c_audio :> cur_sample_rate;
  }
  i2s_config.mode = I2S_MODE_I2S;
  // I2S has 32 bits per sample. *2 as 2 channels
  const unsigned num_bits = 64;
  const unsigned mclk = 512 * 48000;
  // Calculate the MCLK to BCLK ratio using the current sample rate and bits per sample
  i2s_config.mclk_bclk_ratio = mclk / ( cur_sample_rate * num_bits);
```

2.9.2 Communicating audio samples to/from the audio buffer manager

The send and receive callbacks from I2S will pass samples to and from the audio buffer manager task. An array in the audio_frame_t structure is used to store the incoming samples. When the last sample in the frame is requested the task does a channel exchange with the buffer manager task to swap to a unused

buffer.

```
case i2s.send(size_t index) -> int32_t sample:
  unsafe {
    if (index == 0) {
      c_audio :> sample_out_buf;
    }
    sample = sample_out_buf[index];
    if (index == (AVB_NUM_MEDIA_INPUTS-1)) {
      tmr :> p_in_frame->timestamp;
      audio_frame_t *unsafe new_frame = audio_buffers_swap_active_buffer(*double_buffer);
      c_audio <: p_in_frame;
      p_in_frame = new_frame;
    }
  }
  break; // End of send
```

2.9.3 Changing sample rate

The audio buffer manager task indicates to the I2S task that a sample rate change has been requested via a positive integer in element 8 of the output sample array. The I2S task must consume any unused audio buffers in the c_audio channel before indicating that it wishes to restart I2S. The new MCLK to BCLK ratio is calculated when the `init` callback occurs after restart.

```
case i2s.restart_check() -> i2s_restart_t restart:
  unsafe {
    if (sample_out_buf[8]) {
      restart = I2S_RESTART;
      while (!stestct(c_audio)) {
        c_audio :> int;
      }
      sinct(c_audio);
    }
    else {
      restart = I2S_NO_RESTART;
    }
  }
  break; // End of restart check
```

2.10 Setting the Ethernet MAC address

A unique Ethernet MAC address is required for every device on an Ethernet network. XMOS has pre-programmed the OTP memory on the xCORE-200 multichannel audio boards to contain a unique MAC address in the XMOS OUI-24 range.

This MAC address can be read using the XMOS `lib_otpinfo` library and setup in the MAC using the `set_macaddr` interface function on the MAC configuration interface as follows.

```
char mac_address[6];
if (otp_board_info_get_mac(otp_ports0, 0, mac_address) == 0) {
  fail("No MAC address programmed in OTP");
}
i_eth_cfg[MAC_CFG_TO_AVB_MANAGER].set_macaddr(0, mac_address);
```

Customers designing their own hardware must use their own MAC address range assigned by the IEEE. XMOS provides the xburn utility to program unique MAC addresses into OTP via JTAG. See the tools user guide for more information.

MAC addresses may be read via a different method, such as from flash memory, but this is outside the scope of this application note.

2.11 Configuring the AVB endpoint

The main application control task, as prototyped below, is responsible for initializing the AVB stack via the core AVB API and receiving control callbacks via 1722.1.

```
// The main application control task
[[combinable]]
void application_task(client interface avb_interface avb,
                      server interface avb_1722_1_control_callbacks i_1722_1_entity)
```

2.11.1 Setting up the media clock

Firstly, the audio clock is configured to a default of 48 kHz and set to type `INPUT_STREAM_DERIVED`. This means that the audio clock will be a slaved to an audio clock master and recovered from a Listener stream (stream #0). The clock is then enabled.

```
const unsigned default_sample_rate = 48000;
unsigned char aem_identify_control_value = 0;

// Initialize the media clock
avb.set_device_media_clock_type(0, DEVICE_MEDIA_CLOCK_INPUT_STREAM_DERIVED);
avb.set_device_media_clock_rate(0, default_sample_rate);
avb.set_device_media_clock_state(0, DEVICE_MEDIA_CLOCK_STATE_ENABLED);
```

2.11.2 Configuring the Talker and Listener streams

A Talker (source) and Listener (sink) stream is setup to use 8 channel, 24-bit MBLA formats and a one-to-one channel mapping is configured to map stream channel 0 to I2S channel 0 and so on.

The streams are also configured to use the media clock defined above.

```
for (int j=0; j < AVB_NUM_SOURCES; j++)
{
  const int channels_per_stream = AVB_NUM_MEDIA_INPUTS/AVB_NUM_SOURCES;
  int map[AVB_NUM_MEDIA_INPUTS/AVB_NUM_SOURCES];
  for (int i = 0; i < channels_per_stream; i++) map[i] = j ? j*channels_per_stream+i : j+i;
  avb.set_source_map(j, map, channels_per_stream);
  avb.set_source_format(j, AVB_FORMAT_MBLA_24BIT, default_sample_rate);
  avb.set_source_sync(j, 0);
  avb.set_source_channels(j, channels_per_stream);
}

for (int j=0; j < AVB_NUM_SINKS; j++)
{
  const int channels_per_stream = AVB_NUM_MEDIA_OUTPUTS/AVB_NUM_SINKS;
  int map[AVB_NUM_MEDIA_OUTPUTS/AVB_NUM_SINKS];
  for (int i = 0; i < channels_per_stream; i++) map[i] = j ? j*channels_per_stream+i : j+i;
  avb.set_sink_map(j, map, channels_per_stream);
  avb.set_sink_format(j, AVB_FORMAT_MBLA_24BIT, default_sample_rate);
  avb.set_sink_sync(j, 0);
  avb.set_sink_channels(j, channels_per_stream);
```

2.11.3 Reacting to 1722.1 control commands

A callback interface avb_1722_1_control_callbacks is used to implement custom functionality on receipt of a 1722.1 control command. Currently supported commands via this interface are GET_CONTROL, SET_CONTROL, GET_SIGNAL_SELECTOR and SET_SIGNAL_SELECTOR.

When the 1722.1 stack receives one these commands from a Controller, it will cause an event to be fired on the relevant case statement. It is the responsibility of the application to process the index of the control and return the correct status code to the 1722.1 stack, which will in turn respond to the Controller.

For example, when a SET_CONTROL command is received with index 0, the application looks up this value and understands that this is the Identify control. It then reads the current identify control value, sets it in the values array and returns a status code of SUCCESS to indicate that the operation was successful.

```
case i_1722_1_entity.set_control_value(unsigned short control_index,
                                       unsigned short values_length,
                                       unsigned char values[]) -> unsigned char return_status:
{
  return_status = AECP_AEM_STATUS_NO_SUCH_DESCRIPTOR;

  switch (control_index) {
    case DESCRIPTOR_INDEX_CONTROL_IDENTIFY: {
      if (values_length == 1) {
        aem_identify_control_value = values[0];
        if (aem_identify_control_value) {
          debug_printf("IDENTIFY Ping\n");
        }
        return_status = AECP_AEM_STATUS_SUCCESS;
      }
      else
      {
        return_status = AECP_AEM_STATUS_BAD_ARGUMENTS;
```

The example does a debug print on receipt of this control, but can be modified to any custom behavior as required.

APPENDIX A - Demo hardware setup and requirements

The application note is designed to run on the XMOS xCORE-200 Multichannel Audio platform version 2. Version 2 is denoted by a silkscreen label '2V0' beside the XMOS logo.

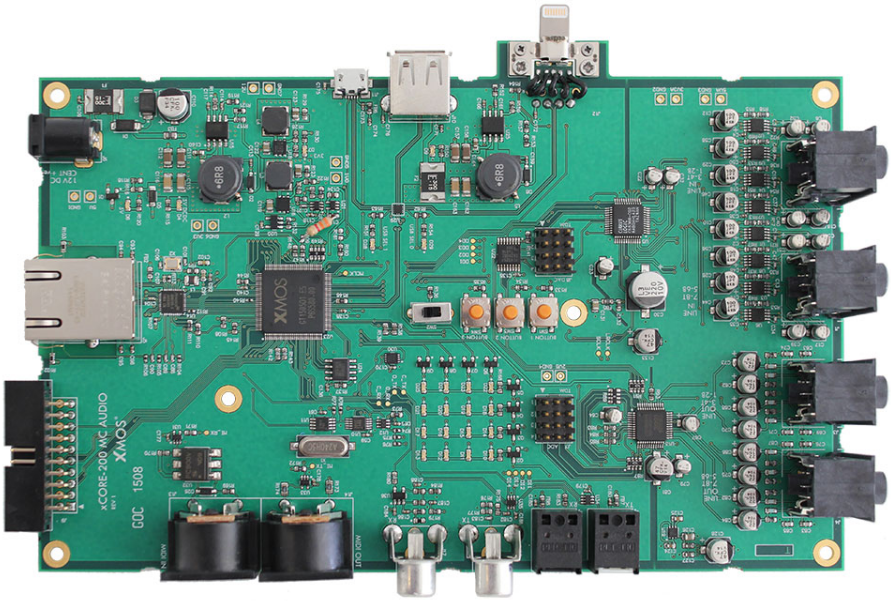


Figure 3: XMOS xCORE-200 Multichannel Audio platform

A full overview of the hardware and its features is provide in the xCORE-200 Multichannel Audio Platform Hardware Manual.

This example firmware is a fully compliant AVB endpoint that will interoperate with other compliant, third-party endpoints. A Late 2013 MacBook Pro running OS X version 10.10.3 is used to demonstrate this capability. An official Apple Thunderbolt to Gigabit Ethernet adapter is required if the Mac does not have a built-in Ethernet port.

All Apple Macs with a Thunderbolt port are AVB enabled.

APPENDIX B - Importing, building and running the example

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

1. Choose **File ▶ Import**.
2. Choose **General ▶ Existing Projects into Workspace** and click **Next**.
3. Click **Browse** next to **'Select archive file'** and select the firmware .zip file associated with this application note.
4. Make sure that all projects are ticked in the *Projects* list.
5. Click **Finish**.
6. Open the **Edit** perspective, select the `AN00202_gige_avb_i2s_demo` project in the Project Explorer and click the **Build** icon in the main toolbar.
7. One or more **Import Wizard** windows may appear. Click **Finish** to automatically download and import the library dependencies for this example.
8. The example will now build. Build information can be seen in the **Console** tab and will print *Build Complete* when finished.

Once built there will be a `bin` directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

To run the example, a *Run Configuration* needs to be set up:

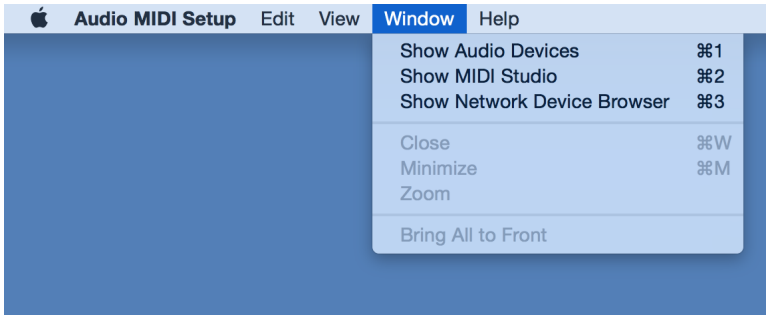
1. Select the **Run ▶ Run Configurations..** menu.
2. Right click on the **xCORE application** group in the left hand pane and **New**.
3. Select the **Run on: hardware** option and from the target list select the xTAG connected to the xCORE-200 multichannel audio board.
4. Within the **Target I/O options** section, the **xSCOPE (via xCONNECT/UART)** option should be selected.
5. By clicking on the **Run** icon (a green arrow) in the **Edit** perspective of the xTIMEcomposer, or by clicking the **Run** button in the run configuration dialog, the program will run on hardware.

Once running the the console tab will show debug output messages from the AVB endpoint.

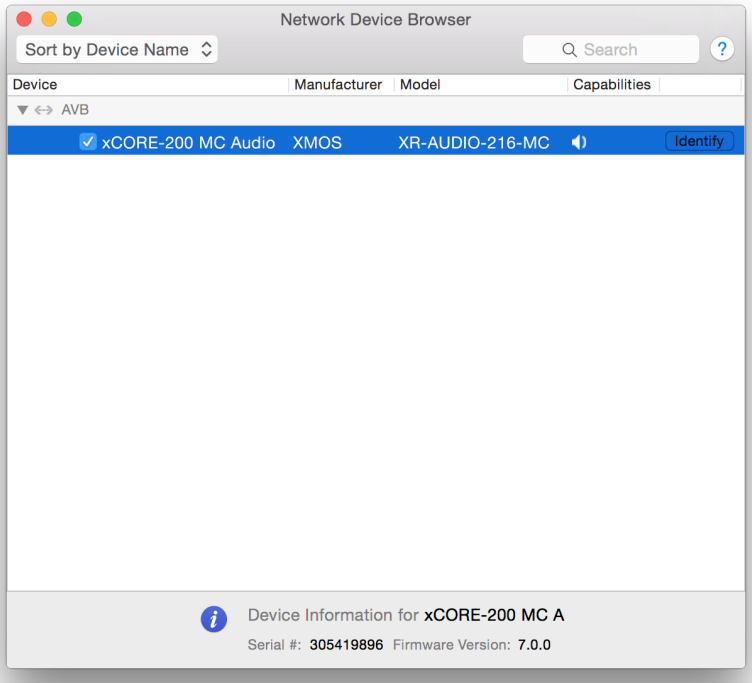
APPENDIX C - Apple Mac OS X AVB setup

To enumerate the XMOS AVB device as an audio device under OS X 10.10:

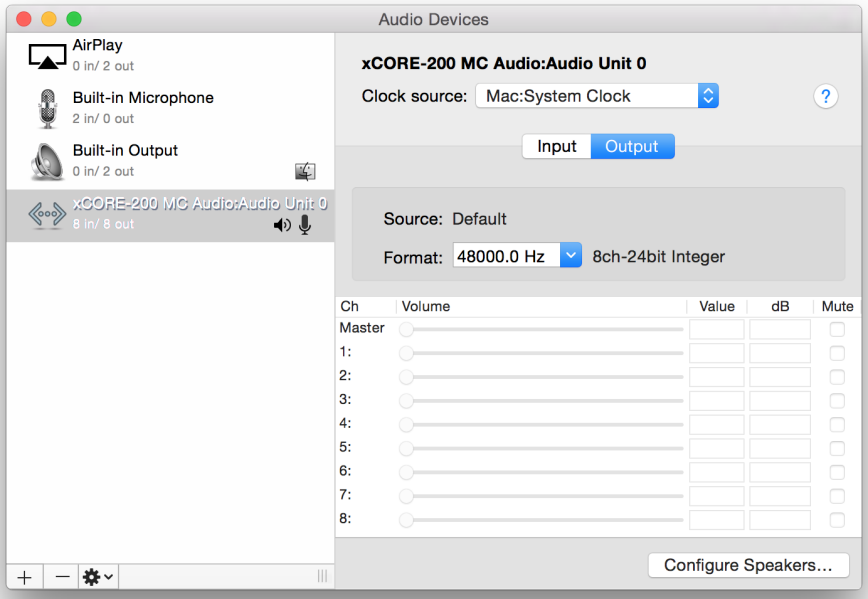
1. Connect the xCORE-200 MC Audio board to the Mac via the Ethernet port or Thunderbolt to Ethernet adapter.
2. Open the *Audio MIDI Setup* utility.
3. In the menu bar, select **Window ▶ Show Network Device Browser**.



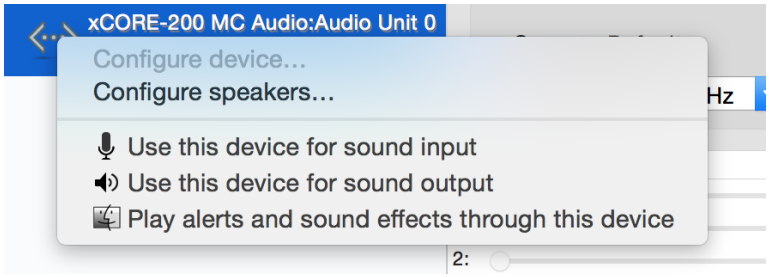
4. The endpoint will enumerate in this list as *xCORE-200 MC Audio*. Select the checkbox to the left of the entry to connect the device.



5. On successful connection, the device will appear as an 8 in/ 8 out Audio Device in the *Audio MIDI Setup* window.



6. To enable audio streaming to/from a device, right click on the device in the left pane and select *Use this device for sound input* and *Use this device for sound output*.



7. Multichannel audio can now be played and recorded via the endpoint.

APPENDIX D - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

xCORE-200 Multichannel Audio Platform 2v0 Hardware Manual

http://www.xmos.com/support/boards?product=18334&component=18687

XMOS Software Libraries

http://www.xmos.com/support/libraries