lib_adat: ADAT lightpipe

Publication Date: 2024/10/18
Document Number: XM-015136-UG v2.0.1

XMOS

IN THIS DOCUMENT

1 Introduction

The ADAT Lightpipe, officially the ADAT Optical Interface, is a standard for the transfer of digital audio between equipment via optical cable. The data transmission rate is determined by the transmitter, and the receiver has to recover the sample rate. ADAT can carry eight channels of uncompressed digital audio at sample-rates of 44.1 or 48 kHz.

Important characteristics of `lib_adat` are the following:

- The sample rate(s) supported. Typical values are 44.1 or 48. Higher rates are supported with a reduced number of samples via S/MUX ('sample multiplexing')
- Transmit and Receive support. Some systems require only ADAT output, or only ADAT input. Others require both.

Note that ADAT of eight channels at 48 Khz is identical to four channels at 96 KHz - a single bit in the data stream differentiates it (but the bit rates, transmit, and receive code are identical).

1.1 Transmit

This module can transmit ADAT signals at the following rates (assuming eight threads on a 500 MHz part)

Functionality provided		Resources required		
Channels	Sample Rate	1-bit port	Threads	Memory
8	up to 48 KHz	1-2	1+	3.6K
8	up to 48 KHz	1-2	1	3.5K

It requires a single thread to run the transmit code. The number of 1-bit ports depends on whether the master clock is already available on a one-bit port. If available, then only a single 1-bit port is required to output ADAT. If not, then two ports are required, one for the signal output, and one for the master-clock input.

The precise transmission frequencies supported depend on the availability of an external clock (eg, a PLL or a crystal oscillator) that runs at a frequency of:

```
512 * sampleRate
```

or a power-of-2 multiple. For example, for 48 Khz the external clock has to run at a frequency of 24.576 MHz. If both 44,1 and 48 Khz frequencies are to be supported, both a 24.587 MHz and a 22.579 MHz master clock are required. This is normally not an issue since the same clocks can be used to drive the audio codecs.

When using an *xcore.ai* based device these frequencies can be generated by the on-chip application/secondary PLL.

1.2 Receive

This module can receive ADAT signals at the following rates (assuming 8 threads on a 500 MHz part)

Functionality provided		Resources required	
Channels	Sample Rate	1-bit port	Memory
8	up to 48 KHz	1	3.5 KB

A single 62.5-MIPS core is required. The receiver does not require any external clock, but can only recover 44.1 and 48 KHz sample rates.

2 ADAT receive

The ADAT receiver comprises a single thread that parses data as it arrives on a one-bit port and that outputs words of data onto a streaming channel end. Each word of data carries 24 bits of sample data and 4 bits of channel information.

The receiver requires the *xcore* reference clock being exactly 100 Mhz (default value).

The receiver API comprises two functions, one that receives adat at 48 KHz, and one that receives ADAT at 44.1 KHz. If the frequency of the input signal is known a priori, the call that function in a non terminating `while(1)` loop. If the frequency could be either, then call the two functions in succession from a `while(1)` loop (recommended).

2.1 Receive API

Compile time defines

ADAT_REF

Define this to 100 to state that the reference clock is exactly 100 MHz (for example when using a 20 or 25 MHz crystal), or 999375 to state that the reference clock is 99.9375 MHz (the result of using a 13 MHz crystal on an XS1-L series devices). Other values are at present not supported.

Functions

void **`adatReceiver48000`**(in_buffered_port_32_t p, streaming_chanend_t oChan)

ADAT Receive Thread (48kHz sample rate).

When a data rame is received, samples will be output onto the streaming channel At first a word 0x000000Z1 will be output, where `Z` are the user data; after that eight words 0x0ZZZZZZ0 will be output where `ZZZZZZ` is a 24-bit sample value. The eight words may refer to sample values on eight channels, or on fewer channels if muxing is used.

The function will return if it cannot lock onto a 48,000 Hz signal. Normally the 48000 function is called in a while(1) loop. If both 44,100 and 48,000 need to be

supported, they should be called in sequence in a while(1) loop. Note that the functions are large, and that 44,100 should not be called if 44.1 KHz does not need to be supported.

Parameters

- **p** – ADAT port - should be 1-bit, 32-bit buffered, and clocked at 100MHz
- **oChan** – channel on which decoded samples are output

void **adatReceiver44100**(in_buffered_port_32_t p, streaming_chanend_t oChan)

ADAT Receive Thread (44.1kHz sample rate).

When a data rame is received, samples will be output onto the streaming channel At first a word 0x000000Z1 will be output, where Z are the user data; after that eight words 0x0ZZZZZZ0 will be output where ZZZZZZ is a 24-bit sample value. The eight words may refer to sample values on eight channels, or on fewer channels if muxing is used.

The function will return if it cannot lock onto a 44,100 Hz signal. Normally the 44,100 function is called in a while(1) loop. If both 44,100 and 48,000 need to be supported, they should be called in sequence in a while(1) loop. Note that the functions are large, and that 48,000 should not be called if 48 Khz does not need to be supported.

Parameters

- **p** – ADAT port - should be 1-bit 32-bit buffered, and clocked at 100MHz
- **oChan** – channel on which decoded samples are output

2.2 Receive example

A simple receive program is provided in `examples/app_adat_rx_example`. This application simply receives samples and periodically emits the number of frames received to the terminal. This section examines this simple example.

The input port needs to be declared as a buffered port:

```
buffered in port:32 p_adat_rx = PORT_ADAT_IN;
```

The receive functions should be called from a `while(1)` loop.

```
void receive_adat(streaming chanend c)
{
    while(1)
    {
        adatReceiver48000(p_adat_rx, c);
        adatReceiver44100(p_adat_rx, c);
    }
}
```

A data handler task inspects the received data samples and synchronises with the beginning of each frame. In this case, we expect every 9th value to be marked with a '1' nibble to indicate end-of-frame.

```
void collect_samples(streaming chanend c)
{
    unsigned head, channels[9];
    int count = 0;

    while(1)
    {
        for(int i = 0; i < 9; i++)
        {
            c :> head;
```

(continues on next page)

(continued from previous page)

```
            if ((head & 0xF) == 1)
            {
                break;
            }
            channels[i] = head;
        }
        ++count;

        if ((count % 100000) == 0)
        {
            printstr("Frames received: ");
            printintln(count);
        }
        // One whole frame in channels [0..7]
    }
}
```

`main()` simply forks the data handling task and the receiver in parallel in two threads:

```
int main(void)
{
    streaming chan c;
    par
    {
        on tile[0]:
        {
            board_setup();
            receive_adat(c);
        }
        on tile[0]: collect_samples(c);
    }
    return 0;
}
```

3 ADAT transmit

There are two functions in the API that can produce an ADAT signal. The simplest is a single thread that inputs samples over a channel and that outputs data on a 1-bit port.

A more complex version has a thread that inputs samples over a channel and that produces an ADAT signal onto a second channel. Another thread is required to copy this data from the channel onto a port. This second version is useful if the ADAT output port resides on a different tile.

This document provides example usage for both variants.

An identical protocol is used by both variants for inputting sample values to be transmitted over ADAT. The first word transmitted over the chanend should be the multiplier of the master clock (either 1024 or 512), the second word should be the S/MUX setting (either 0 or 2), then there should be N x 8 words of sample values, terminated by an `XS1_CT_END` control token. If no control token is sent, the transmission process will not terminate, and an infinite stream of ADAT data can be sent.

The multiplier is the ratio between the master clock and the bit-rate; 1024 refers to a 49.152 MHz master-clock, 512 assumes a 24.576 MHz master clock.

The output of the ADAT transmit thread has to be synchronised with an external flip-flop. In order to make sure that the flip-flop captures the signal on the right edge, the output port should be set up as follows:

```
set_clock_src(mck_blk, mck);        // Connect Master Clock Block to mclk pin
set_port_clock(adat_port, mck_blk); // Set ADAT_tx to be clocked from mck_blk
set_clock_fall_delay(mck_blk, 7);   // Delay falling edge of mck_blk
start_clock(mck_blk);               // Start mck_blk
```

3.1 Transmit API

Functions

void **`adat_tx`**(chanend c_data, chanend c_port)

Function that takes data over a channel end, and that outputs this in ADAT format onto a 1-bit port. The 1-bit port should be clocked by the master-clock, and an external flop should be used to precisely align the edge of the signal to the master-clock.

Data should be send onto c_data using outuint only, the first two values should be The multiplier and the smux values, after that output any number of eight samples (24-bit, right aligned), and if the process is to be terminated send it an control token 1.

The data is output onto a channel, which a separate process should output to a port. This process should byte-reverse every word read over the channel, and then output the reversed word to a buffered 1-bit port.

Parameters

- **c_data** – Channel over which to send sample values to the transmitter
- **c_port** – Channel on which to generate the ADAT stream

void **adat_tx_port**(chanend c_data, out_buffered_port_32_t p_data)

Function that takes data over a channel end, and that outputs this in ADAT format onto a 1-bit port. The 1-bit port should be clocked by the master-clock, and an external flop should be used to precisely align the edge of the signal to the master-clock.

Data should be send onto c_data using outuint only, the first two values should be The multiplier and the smux values, after that output any number of eight samples (24-bit, right aligned), and if the process is to be terminated send it an control token 1.

Parameters

- **c_data** – Channel over which to send sample values to the transmitter
- **p_data** – 1-bit, 32-bit buffered, port on which to generate the ADAT stream

3.2 Transmit example

Example applications are provided for both the 'direct port' and 'remote port' API variants. These are `app_adat_tx_direct_example` and `app_adat_tx_example` respectively.

Both examples transmit sine waves on all channels and are described in this section.

Direct port example

The output port is declared as a 32-bit buffered port, and the master clock input must be declared as an unbuffered input port. A clock block is also required:

```
buffered out port:32 p_adat_tx = PORT_ADAT_OUT;
in port p_mclk_in = PORT_MCLK_IN;
out port p_ctrl = PORT_CTRL;
on tile[1]: clock clk_audio = XS1_CLKBLK_2;
```

The ports are setup such that the output port is clocked from the master clock with a suitable delay (to enable the external flop to latch the signal). Starting the clock block is a critical step, otherwise outputs to the transmit port will pause:

```
    set_clock_src(clk_audio, p_mclk_in);
    configure_out_port_no_ready(p_adat_tx, clk_audio, 0);
    set_clock_fall_delay(clk_audio, 7);
    start_clock(clk_audio);
```

The data generator task initially communicates the clock multiplier and the S/MUX flags, prior to transmitting data.

The task uses a table to generate sine-waves of various frequencies and phase-shifts (allowing for channel identification):

```
void generate_samples(chanend c) {
    int count1 = 0;
    int count2 = 0;
    int count4 = 0;
    outuint(c, MCLK_FREQUENCY_48 / 48000);  // clock multiplier value
    outuint(c, 1);                          // S/MUX value
    unsafe {
        volatile unsigned * unsafe sample_ptr = (unsigned * unsafe) &samples[0];
        outuint(c, (unsigned) sample_ptr);
    }

    while(1) {
        inuint(c);

        // Update sample values
        samples[0] = sine_table[count1];                            // 500Hz sine
        samples[1] = sine_table[SINE_TABLE_SIZE - 1 - count1];      // 500Hz sine, phase-shifted from channel 0
        samples[2] = sine_table[count2];                            // 1000Hz sine
        samples[3] = sine_table[SINE_TABLE_SIZE - 1 - count2];      // 1000Hz sine, phase-shifted from channel 2
        samples[4] = sine_table[count4];                            // 2000Hz sine
        samples[5] = sine_table[SINE_TABLE_SIZE - 1 - count4];      // 2000Hz sine, phase-shifted from channel 4
        samples[6] = sine_table[count1];                            // same as channel 0
        samples[7] = sine_table[SINE_TABLE_SIZE - 1 - count1];      // same as channel 1

        unsafe {
            volatile unsigned * unsafe sample_ptr = (unsigned * unsafe) &samples[0];
            outuint(c, (unsigned) sample_ptr);
        }

        // Handle rollover of the sine_table array indices
        count1 += 1;
        count2 += 2;
        count4 += 4;
        if (count1 == SINE_TABLE_SIZE) {
            count1 = 0;
            count2 = 0;
            count4 = 0;
        } else if (count2 == SINE_TABLE_SIZE) {
            count2 = 0;
            count4 = 0;
        } else if (count4 == SINE_TABLE_SIZE) {
            count4 = 0;
        }
    }
}
```

The main program simply forks the data generating and the transmitter tasks in parallel in two threads. A channel is declared and passed to both tasks to allow communication. A `board_setup` tasks is also spawned that configures the external hardware and configures the *xcore.ai* application PLL to generate a suitable master-clock.

```
int main(void)
{
    chan c;
    par
    {
        on tile[0]: board_setup();
        on tile[1]: transmit_adat(c);
        on tile[1]: generate_samples(c);
    }
    return 0;
}
```

Remote port example

Much of the remote port example matches the direct port example. The output port is declared as a buffered port, and the master clock input must be declared as an unbuffered input port. A clock block is also required:

```
buffered out port:32 p_adat_tx = PORT_ADAT_OUT;
in port p_mclk_in = PORT_MCLK_IN;
out port p_ctrl = PORT_CTRL;
on tile[1]: clock clk_audio = XS1_CLKBLK_2;
//*

#define MCLK_FREQUENCY_48  24576000

void board_setup(void)
{
    set_port_drive_high(p_ctrl);

    // Drive control port to turn on 3V3.
    // Bits set to low will be high-z, pulled down.
    p_ctrl <: 0xA0;

    // Wait for power supplies to be up and stable.
    delay_milliseconds(10);

    sw_pll_fixed_clock(MCLK_FREQUENCY_48);

    while (1) {}
}

/* Port driver */
void drive_port(chanend c_port)
{
    while (1)
    {
        p_adat_tx <: byterev(inuint(c_port));
    }
}
```

Again, the ports are setup so that the output port is clocked from the master clock with a suitable delay (to enable the external flop to latch the signal). Starting the clock block is a critical step, otherwise outputs to the transmit port will pause:

```
    set_clock_src(clk_audio, p_mclk_in);
    configure_out_port_no_ready(p_adat_tx, clk_audio, 0);
    set_clock_fall_delay(clk_audio, 7);
    start_clock(clk_audio);
```

The thread that drives the port should input words from the channel, and output them with *reversed byte order*. Note that this activity of input, byte-reverse and output takes only three instructions and can often be merged with other task; for example if there is an I²S thread that delivers data synchronised to the same master clock, then that thread can simultaneously drive the ADAT and I²S ports:

```
void drive_port(chanend c_port)
{
    while (1)
    {
        p_adat_tx <: byterev(inuint(c_port));
    }
}
```

For simplicity, this example spawns the **`adat_tx`** and port-driving tasks on the same tile:

```
    par
    {
        adat_tx(c, c_port);
        drive_port(c_port);
    }
```

The data generation task closely matches the previous example, first communicating the clock multiplier and the S/MUX flags, prior to transmitting data:

```
void generate_samples(chanend c) {
    int count1 = 0;
    int count2 = 0;
    int count4 = 0;
    outuint(c, MCLK_FREQUENCY_48 / 48000);  // clock multiplier value
    outuint(c, 0);                          // S/MUX value

    for (int idx = 0; idx < 8; ++idx) {
        outuint(c, 0);
    }

    while(1) {
```

(continues on next page)

(continued from previous page)

```
        // Send the next samples
        outuint(c, sine_table[count1]);                         // 500Hz sine
        outuint(c, sine_table[SINE_TABLE_SIZE - 1 - count1]);   // 500Hz sine, phase-shifted from channel 0
        outuint(c, sine_table[count2]);                         // 1000Hz sine
        outuint(c, sine_table[SINE_TABLE_SIZE - 1 - count2]);   // 1000Hz sine, phase-shifted from channel 2
        outuint(c, sine_table[count4]);                         // 2000Hz sine
        outuint(c, sine_table[SINE_TABLE_SIZE - 1 - count4]);   // 2000Hz sine, phase-shifted from channel 4
        outuint(c, sine_table[count1]);                         // same as channel 0
        outuint(c, sine_table[SINE_TABLE_SIZE - 1 - count1]);   // same as channel 1

        // Handle rollover of the sine_table array indices
        count1 += 1;
        count2 += 2;
        count4 += 4;
        if (count1 == SINE_TABLE_SIZE) {
            count1 = 0;
            count2 = 0;
            count4 = 0;
        } else if (count2 == SINE_TABLE_SIZE) {
            count2 = 0;
            count4 = 0;
        } else if (count4 == SINE_TABLE_SIZE) {
            count4 = 0;
        }
    }
}
```

The main program simply forks the data generating thread and the transmitter in parallel in two threads. Prior to starting the transmitter, the clocks should be set up:

```
int main(void) {

    chan c;
    par
    {
        on tile[0]: board_setup();
        on tile[1]: transmit_adat(c);
        on tile[1]: generate_samples(c);
    }
    return 0;
}
```

4 Additional examples

An additional example, `app_adat_loopback`, is also provided. This expects a cable to be connected between transmit and receive. The application transmits counters on all channels and checks for the correct reception of these counters. This application can be useful for initial debugging and validation of external transmit/receive circuitry.