S/PDIF Software Component

Document Number: X9128A

Publication Date: 2012/3/21

XMOS

CONTENTS

1 SPDIF software

IN THIS CHAPTER

S/PDIF, or Sony/Philips Digital Interface is a protocol to transmit audio data over either coaxial or optical cables. The data transmission rate is determined by the transmitter, and the receiver has to recover the sample rate.

Important characteristics of S/PDIF software are the following:

- The number of audio channels: 2 (Stereo)
- The sample rate(s) supported. Typical values are 44.1, 48, 96, and 192 Khz. Some systems require only a single frequency to be supported, others need to support all frequencies and need to auto-detect the frequency.
- Transmit and Receive support. Some systems require only S/PDIF output, or only S/PDIF input. Others require both.

1.1 module_spdif_tx

This module can transmit S/PDIF signals at the following rates (assuming a 50 MIPS thread):

Functionality provided		Resources required		Status
Channels	Sample Rate	1-bit port	Memory	
2	up to 192 KHz	1-2	3.5 KB	Implemented and tested

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 S/PDIF. If not, then two ports are required, one for the signal output, and one for the master-clock input.

The jitter on the output-pin is within tolerances allowed by the spec provided a 500 MHz part is used. It is recommended to use an external flip-flop to resynchronise the data signal to the master-clock, which will eliminate the remaining jitter on the S/PDIF edges.

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:

```
channels * sampleRate * 64
```

or a power-of-2 multiple. For example, for 2 channels at 192 Khz the external clock has to run at a frequency of 24.576 MHz. This same frequency also supports 2 channels at 48 KHz (which requires a minimum frequency of 6.144 MHz). If both 44,1 and 48 Khz frequencies are to be supported, both a 24.576 MHz and a 22.579 MHz master clock is required. This is normally not an issue since the same clocks can be used to drive the audio codecs.

Typical applications for this module include iPod docks, digital microphones, digital mixing desks, USB audio, and AVB.

1.2 module_spdif_rx

This module can receive S/PDIF signals at three different rates. It automatically adjusts to the incoming rate, but for high rates a fast thread is required. The thread will fail silently if it does not have enough MIPS to parse the input stream.

The S/PDIF receiver is generated from a state machine description. The generated code requires a one bit buffered input port (transfer width of 4), and a clock block to work.

Functionality provided		Resources required			Status
Channels	Sample Rate	1-bit port	Memory	Thread rate	
2	up to 192 KHz	1	3 KB	80 MIPS	Implemented and teste

The receiver does not require any external clock, but can only recover 44.1, 48, 88.2, 96, and 192 KHz sample rates.

Typical applications for this module include digital speakers, digital mixing desks, USB audio, and AVB.

2 S/PDIF Receive

IN THIS CHAPTER

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

This module requires the reference clock to be exactly 100 MHz.

2.1 Symbolic constants

`FRAME_X` This constant defines the four least-significant bits of the first sample of a frame (typically a sample from the left channel).

`FRAME_Y` This constant defines the four least-significant bits of the second or later sample of a frame (typically a sample from the right channel, unless there are more than two channels).

`FRAME_Z` This constant defines the four least-significant bits of the first sample of the first frame of a block (typically a sample from the left channel).

2.2 API

```
void SpdifReceive(in buffered port:4 p,
                  streaming chanend c,
                  int initial_divider,
                  clock clk)
```

S/PDIF receive function.

This function needs 1 thread and no memory other than ~2800 bytes of program code. It can do 11025, 12000, 22050, 24000, 44100, 48000, 88200, 96000, and 192000 Hz. When the decoder encounters a long series of zeros it will lower the divider; when it encounters a short series of 0-1 transitions it will increase the divider.

Output: the received 24-bit sample values are output as a word on the streaming channel end. Each value is shifted up by 4-bits with the bottom four bits being one of FRAME_X, FRAME_Y, or FRAME_Z. The bottom four bits should be removed whereupon the sample value should be sign extended.

The function does not return unless compiled with TEST defined in which case it returns any time that it loses synchronisation.

This function has the following parameters:

`p` S/PDIF input port. This port must be 4-bit buffered, declared as `in buffered port:4`

`c` channel to output samples to

`initial_divider` initial divide for initial estimate of sample rate For a 100Mhz reference clock, use an initial divider of 1 for 192000, 2 for 96000/88200, and 4 for 48000/44100.

`clk` clock block sourced from the 100 MHz reference clock.

2.3 Example

An example program is shown below. An input port and a clock block must be declared. Neither should be configured:

```
#include <xs1.h>
#include "SpdifReceive.h"

buffered in port:4 oneBitPort = XS1_PORT_1F;
clock clockblock = XS1_CLKBLK_1;
```

All data samples are being received on a streaming channel, after being parsed by the receive process. After reading a sample value from the channel, it must be converted to a signed sample value whilst removing the tag identifying the channel information. In this example, we perform this operation by masking off the bottom four bits and shifting the sample-data into the most significant 24-bits, ready to be used on, for example, I2S:

```
void handleSamples(streaming chanend c) {
    int v, left, right;
    while(1) {
        c :> v;
        if((v & 0xF) == FRAME_Y) {
            right = (v & ~0xf) << 4;
            // operate on left and right
        } else {
            left = (v & ~0xf) << 4;
        }
    }
}
```

The main program in this example simply starts the S/PDIF receive thread, and the data handling thread in parallel:

```
int main(void) {
```

```
    streaming chan c;
    par {
        SpdifReceive(oneBitPort, c, 1, clockblock);
        handleSamples(c);
    }
    return 0;
}
```

3 S/PDIF Transmit

IN THIS CHAPTER

This module is a single thread that receives samples over a channel and that outputs data on the port.

The S/PDIF transmit module require a one-bit buffered output port (with transfer width of 32), a clock block, and a master clock coming in on an unbuffered one-bit port.

3.1 API

Call SpdifTransmitPortConfig to set up the clock then SpdifTransmit to output data.

```
void SpdifTransmitPortConfig(out buffered port:32 p,
                             clock cl,
                             in port p_mclk)
```

Configure out port to be clocked by clock block, driven from master clock input.

Must be called before SpdifTransmit()

This function has the following parameters:

`p`	S/PDIF tx port
`cl`	Clock block to be used
`p_mclk`	Master-clock input port

```
void SpdifTransmit(buffered out port:32 p, chanend c)
```

Function expects a buffered single bit port clock from the master clock.

All channel communication is done via builtins (e.g. outuint, outct etc.)

On startup expects two words over the channel:

1. Desired sample frequency (in Hz)
2. Master clock frequency (in Hz)

Then sample pairs:

1. Left sample
2. Right sample

The data format is 24-bit signed left aligned in a 32-bit word.

If a XS1_CT_END token is received, the thread stops and waits for new sample/-master freq pair

This function has the following parameters:

p	S/PDIF tx port
c	Channel-end for sample freq and samples

3.2 Example

This example generates a triangle sound wave on the SPDIF interface from a USB Audio 2.0 multichannel interface board. On this board the master clock input is from a PLL. The program is shown below (excluding code to set up the PLL on the board).

An output port, a master-clock input port and a clock block must be declared:

```
#include <xs1.h>
#include <platform.h>
#include "SpdifTransmit.h"

#define SAMPLE_FREQUENCY_HZ 96000
#define MASTER_CLOCK_FREQUENCY_HZ 12288000

on stdcore[1] : buffered out port:32 oneBitPort = XS1_PORT_1K;
on stdcore[1] : in port masterClockPort = XS1_PORT_1L;
on stdcore[1] : clock clockblock = XS1_CLKBLK_1;
```

In this example transmitSpdif sets up the clock and starts the transmit function to receive on a chanend.

```
void transmitSpdif(chanend c) {
    SpdifTransmitPortConfig(oneBitPort, clockblock, masterClockPort);
    SpdifTransmit(oneBitPort, c);
}
```

The generate function sends configuration settings over a channel then a triangle wave.

```
#define WAVE_LEN 512
void generate(chanend c) {
    int i = 0;
    outuint(c, SAMPLE_FREQUENCY_HZ);
    outuint(c, MASTER_CLOCK_FREQUENCY_HZ);
    while(1) {
       // Generate a triangle wave
       int sample = i;
```

```
        if (i > (WAVE_LEN / 4)) {
           // After the first quarter of the cycle
           sample = (WAVE_LEN / 2) - i;
        }
        if (i > (3 * WAVE_LEN / 4)) {
           // In the last quarter of the cycle
           sample = i - WAVE_LEN;
        }
        sample <<= 23; // Shift to highest but 1 bits
        outuint(c, sample); // Left channel
        outuint(c, sample); // Right channel

        i++;
        i %= WAVE_LEN;
    }
    //outct(c, XS1_CT_END); // to stop SpdifTransmit thread
}
```

- S/PDIF transmit
- the data generator
- clock generator for the PLL

An XC channel connects the generator and the transmit thread.

```
void example(void) {
   chan c;
   setupPll();
   par {
      transmitSpdif(c);
      generate(c);
      clockGen();
   }
}

int main(void) {
   par {
      on stdcore[1]: example();
   }
   return 0;
}
```

Table of Contents