
Application Note: AN00231

SPDIF Receive to I2S output using Asynchronous Sample Rate Conversion

The XMOS Asynchronous Sample Rate Conversion (ASRC) library allows audio to be streamed between systems running with asynchronous sample rates while maintaining “Hi-Res” audio quality. Practical examples where ASRC can provide significant benefits include:

- Allow systems to receive and mix multiple digital audio streams
- Remove the cost and hardware complexity of audio clock recovery using a PLL
- Allow systems to use the highest possible sample rate rather than the lowest common denominator
- Addition of interfaces and processing to existing audio architectures

In addition, the ASRC can perform major rate changes between the standard audio sample rates of 44.1, 48, 88.2, 96, 176.4 and 192KHz.

This application note demonstrates the use of the XMOS Sample Rate Conversion library in a system that receives samples from an SPDIF input and outputs them to a DAC over I²S. The I²S subsystem uses a fixed frequency local master clock oscillator and the SPDIF clock is encoded within the received stream. This means that, even if operating at the same nominal sample rate, the clocks will not be synchronized and consequently ASRC is required to properly stream audio between these two audio interfaces.

The sample rates supported for this demonstration are 44.1, 48, 88.2 and 96KHz. This restriction is applied by the SPDIF receiver which is only rated up to 96KHz for optical sources and the ASRC library which can currently only handle rates of 176.4/192KHz on one side. The restricted sample rates also allow significant chip resource usage optimization which is discussed later within this application note.

Using the XMOS I²S and SPDIF libraries this application note demonstrates practical usage of the ASRC library. The software includes additional functions to handle the serial to block conversion of the audio samples and a precise rate monitoring servo task which is required to calculate the fractional frequency ratios for the ASRC processing function.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.2.0 or higher
- XMOS I²S/TDM library - Version 2.1.0
- XMOS GPIO library - Version 1.0.0
- XMOS I²C library - Version 3.1.0
- XMOS SPDIF library - Version 2.0.2
- XMOS Logging library - Version 2.0.0
- XMOS Assertion library - Version 2.0.0

Required hardware

The example code provided with the application has been implemented and tested on the xCORE-200 Multichannel Audio Platform.

Prerequisites

- This document assumes familiarity with I²S and SPDIF interfaces, the principles of ASRC and a basic understanding of the XMOS xCORE architecture, the XMOS tool chain and the xC language. Docu-

mentation related to these aspects which are not specific to this application note are linked to in the references appendix.

- For a description of XMOS related terms found in this document please see the XMOS Glossary¹.

¹<http://www.xmos.com/published/glossary>

1 Overview

1.1 Introduction

The XMOS Asynchronous Sample Rate Converter (ASRC) provides a way of streaming high quality audio between the two clock domains which are not synchronized. The ASRC library itself is a function call that operates on blocks of samples whereas the SPDIF and I²S peripheral libraries provide streaming audio one sample at a time. To fit the ASRC between two audio streaming components requires a software framework that provides buffering and a precise rate difference calculation.

1.2 Block Diagram

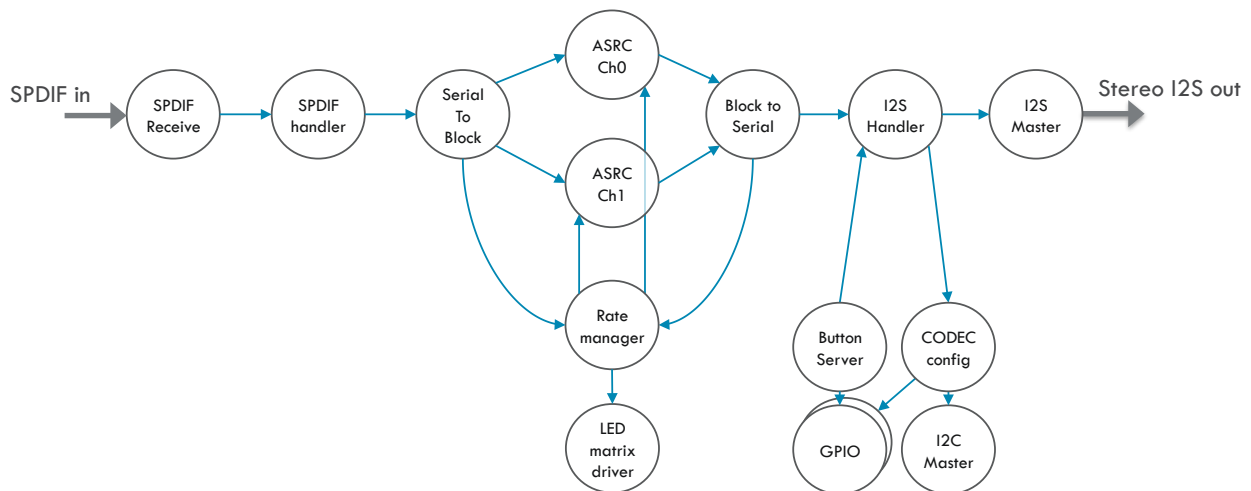


Figure 1: Application Task Diagram

The application uses a number of tasks to send and receive audio samples, convert from serial (streaming) samples to block buffers, time-stamp and monitor the stream rates and of course perform the sample rate conversion. In addition, there are a number of control tasks to complete the application including configuration of the codecs via I²C and GPIO, reading and de-bouncing of the button and displaying information via a matrix of 4 x 4 multiplexed LEDs.

The arrows between tasks show the direction of data-flow between tasks, be it over interfaces or channels. Note that this is not always the same as whether a task communication role is client or server. The roles of the task communications are described later in this application note.

2 SPDIF Receive to I2S using ASRC Demo

2.1 The Makefile

To start using the ASRC, you need to add `lib_src` to your Makefile:

```
USED_MODULES = ... lib_src ...
```

This demo also uses the I²C library (`lib_i2c`), the GPIO library (`lib_gpio`), I²S and SPDIF libraries (`lib_i2s`, `lib_spdif`) and the logging library (`lib_logging`) to provide a lightweight version of `printf()`. The GPIO library abstracts the use of an 8-bit port to drive individual reset and select lines, as well as a 4-bit port for reading the buttons. The SPDIF library provides the source of audio samples, with encoded clock, from the SPDIF receive digital input. The I²S library provides an audio sink using I²S output to drive a DAC.

So the Makefile also includes the following modules:

```
USED_MODULES = .. lib_gpio lib_i2c lib_i2s lib_logging lib_spdif ..
```

2.2 Includes

All xC files which declare the application `main()` function consisting of tasks on multiple tiles need to include `platform.h`. XMOS xCORE specific defines for declaring and initialising hardware are defined in `xs1.h`. `string.h` provides the `memset()` function.

```
#include <xs1.h>
#include <platform.h>
#include <string.h>
```

The ASRC library functions are defined in `src.h`. This header must be included in your code to use the library as it provides the public API to the ASRC function.

```
#include "src.h"
#include "spdif.h"
#include "i2s.h"
#include "i2c.h"
#include "gpio.h"
#include "assert.h"
```

There are four application includes which have the following purposes. `block_serial.h` provides access to the buffer serialization and de-serialization functions, `main.h` provides access to the application helper functions within `main.xc` & `cs4384_5368.h` provides access to the audio CODEC configuration tasks. All contain prototypes of the functions/tasks contained within their respective xC source files as well as defining types of variables and interfaces used to communicate with them.

```
#include "main.h"
#include "block_serial.h"
#include "cs4384_5368.h" //CODEC setup
#include "app_config.h" //General settings
```

The header file `app_config.h` contains defines that are used to configure the application settings including the number of channels, ASRC input block size and maximum input to output ratio (used for defining buffer sizes) and number of logical cores used to carry out the ASRC processing. It also includes information about the local MCLK frequencies used for driving the I²S interface. Note that no MCLK information is required for SPDIF receive because the received samples are timestamped which allows the application

to recover the precise sample rate directly.

```
#define ASRC_N_CHANNELS 2 //Total number of audio channels to be processed by SRC (
↳ minimum 1)
#define ASRC_N_INSTANCES 2 //Number of instances (each usually run a logical core) used to
↳ process audio (minimum 1)
#define ASRC_CHANNELS_PER_INSTANCE (ASRC_N_CHANNELS/ASRC_N_INSTANCES)
//Calculated number of audio channels processed by each core
#define ASRC_N_IN_SAMPLES 8 //Number of samples per channel in each block passed into SRC
↳ each call
//Must be a power of 2 and minimum value is 4 (due to two /2 decimation stages)
#define ASRC_N_OUT_IN_RATIO_MAX 3 //Max ratio between samples out:in per processing step
↳ (44.1->192 is worst case)
#define ASRC_MAX_BLOCK_SIZE (ASRC_N_IN_SAMPLES * ASRC_N_OUT_IN_RATIO_MAX)
#define OUT_FIFO_SIZE (ASRC_MAX_BLOCK_SIZE * 8) //Size per channel of block2serial
↳ output FIFO
#define ASRC_DITHER_SETTING OFF //No output dithering of samples from 32b to 24b

#define DEFAULT_FREQ_HZ_SPDIF 48000
#define DEFAULT_FREQ_HZ_I2S 48000
#define MCLK_FREQUENCY_48 24576000
#define MCLK_FREQUENCY_44 22579200
```

2.3 Allocating hardware resources

The audio interfaces require I/O ports to communicate with the external components, as well as clock blocks to manage the rate of I/O operations. All ports and clock blocks must be declared on the tile they reside since the chip used in the hardware has two tiles.

In the case of SPDIF receive, it needs only a single 1-bit port and clock block.

```
#define SPDIF_TILE 1
port port_spdif_rx = on tile[SPDIF_TILE]: XS1_PORT_10;
clock clk_spdif_rx = on tile[SPDIF_TILE]: XS1_CLKBLK_1;
```

I²S requires multiple 1-bit ports for MCLK input, BCLK output, LRCLK output and DATA, as well as two clock blocks for managing MCLK and BCLK.

```
#define AUDIO_TILE 0
in buffered port:32 ports_i2s_adc[4] = on tile[AUDIO_TILE]: {XS1_PORT_1I,
XS1_PORT_1J,
XS1_PORT_1K,
XS1_PORT_1L};

port port_i2s_mclk = on tile[AUDIO_TILE]: XS1_PORT_1F;
clock clk_mclk = on tile[AUDIO_TILE]: XS1_CLKBLK_2;
out buffered port:32 port_i2s_bclk = on tile[AUDIO_TILE]: XS1_PORT_1H;
out buffered port:32 port_i2s_wclk = on tile[AUDIO_TILE]: XS1_PORT_1G;
clock clk_i2s = on tile[AUDIO_TILE]: XS1_CLKBLK_1;
out buffered port:32 ports_i2s_dac[4] = on tile[AUDIO_TILE]: {XS1_PORT_1M,
XS1_PORT_1N,
XS1_PORT_1O,
XS1_PORT_1P};
```

To support human input and visual output from within the application further I/O ports are declared which map to the buttons and LEDs on the hardware. In addition, there is an 8-bit output port which controls the reset pins configures the clocking modes on the board. Together with configuration via I²C, this allows the DAC and ADC to be setup as required for the application.

```

out port port_leds_col           = on tile[SPDIF_TILE]: XS1_PORT_4C;    //4x4 LED matrix
out port port_leds_row          = on tile[SPDIF_TILE]: XS1_PORT_4D;

port port_i2c                    = on tile[AUDIO_TILE]: XS1_PORT_4A;    //I2C for CODEC configuration

port port_audio_config          = on tile[AUDIO_TILE]: XS1_PORT_8C;
char pin_map_audio_cfg[5]      = {0, 1, 5, 6, 7};
/* Bit map for XS1_PORT_8C
 * 0 DSD_MODE
 * 1 DAC_RST_N
 * 2 USB_SELO
 * 3 USB_SEL1
 * 4 VBUS_OUT_EN
 * 5 PLL_SELECT
 * 6 ADC_RST_N
 * 7 MCLK_FSEL
 */

port port_buttons               = on tile[AUDIO_TILE]: XS1_PORT_4D;    //Buttons and switch

```

2.4 The application main() function

The main() function in the program sets up the tasks in the application.

Firstly, the interfaces and channels are declared. In xC interfaces and channels provide a means of concurrent tasks communicating with each other. Each interface has a concept of client and server where the client initiates the communication and the server responds. Typically, a server is waiting for commands or data to be sent from the client whereupon it performs some processing. The processing may result in a response on the same interface, an I/O event, initiation of a transaction on another interface or a change in some state which may be queried later by another client. All of these cases are demonstrated within this application. A total of 21 interfaces are used to provide communication between the tasks.

```

serial_transfer_push_if i_serial_in;
block_transfer_if i_serial2block[ASRC_N_INSTANCES];
block_transfer_if i_block2serial[ASRC_N_INSTANCES];
serial_transfer_pull_if i_serial_out;
sample_rate_enquiry_if i_sr_input, i_sr_output;
fs_ratio_enquiry_if i_fs_ratio[ASRC_N_INSTANCES];
interface audio_codec_config_if i_codec;
interface i2c_master_if i_i2c[1];
interface output_gpio_if i_gpio[5]; //See mapping of bits 0..7 above in port_audio_config
interface i2s_callback_if i_i2s;
led_matrix_if i_leds;
buttons_if i_buttons;
input_gpio_if i_button_gpio[1];
streaming chan c_spdif_rx;

```

In addition to the interfaces, a single channel is used in which is required by the lib_spdif component. This component uses a streaming channel for performance reasons as it provides a short FIFO allowing outputs within that component to be decoupled.

A version of the task diagram, annotated with the communication roles (Client or Server) is shown below.

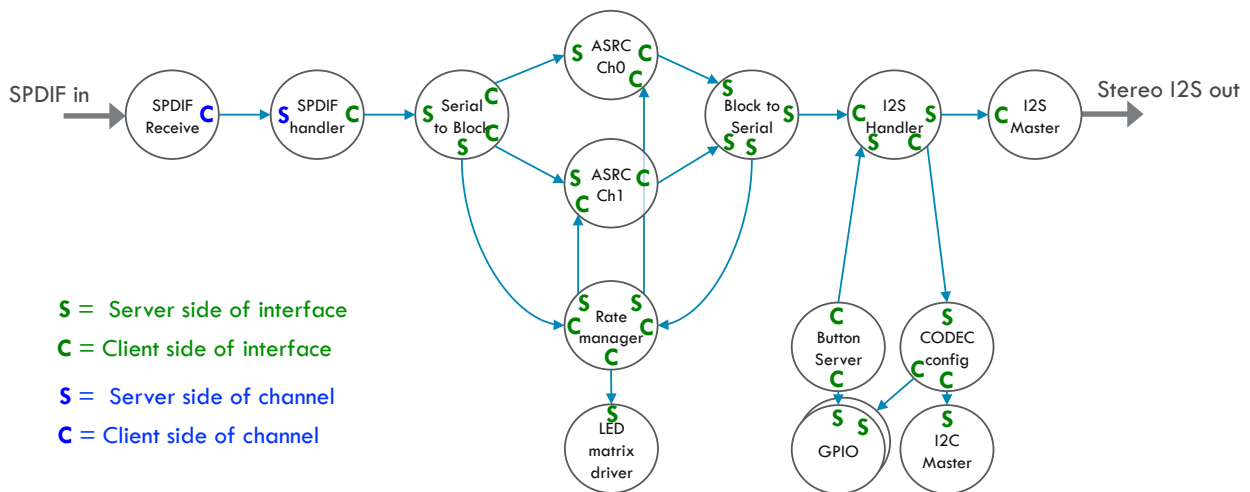


Figure 2: Communication Roles Between Tasks

The rest of the main function starts all the tasks in parallel using the xC par construct:

```

par{
  on tile[SPDIF_TILE]: spdif_rx(c_spdif_rx, port_spdif_rx, clk_spdif_rx, DEFAULT_FREQ_HZ_SPDIF);
  on tile[AUDIO_TILE]: [[combine, ordered]] par{
    rate_server(i_sr_input, i_sr_output, i_fs_ratio, i_leds);
    spdif_handler(c_spdif_rx, i_serial_in);
    button_listener(i_buttons, i_button_gpio[0]);
    input_gpio_with_events(i_button_gpio, 1, port_buttons, pin_map_buttons);
  }

  on tile[AUDIO_TILE]: serial2block(i_serial_in, i_serial2block, i_sr_input);
  on tile[AUDIO_TILE]: unsafe{ par (int i=0; i<ASRC_N_INSTANCES; i++) asrc(i_serial2block[i], i_block2serial
  ← [i], i_fs_ratio[i]);}
  on tile[AUDIO_TILE]: unsafe { par{[[distribute]] block2serial(i_block2serial, i_serial_out, i_sr_output)
  ← ;}}



  on tile[AUDIO_TILE]: audio_codec_cs4384_cs5368(i_codec, i_i2c[0], CODEC_IS_I2S_SLAVE, i_gpio[0], i_gpio
  ← [1], i_gpio[3], i_gpio[4]);
  on tile[AUDIO_TILE]: i2c_master_single_port(i_i2c, 1, port_i2c, 10, 0 /*SCL*/, 1 /*SDA*/, 0);
  on tile[AUDIO_TILE]: output_gpio(i_gpio, sizeof(pin_map_audio_cfg), port_audio_config, pin_map_audio_cfg);
  on tile[AUDIO_TILE]: {
    i_gpio[2].output(0); //Select fixed local clock on MCLK mux
    configure_clock_src(clk_mclk, port_i2s_mclk); //Connect MCLK clock block to input pin
    start_clock(clk_mclk);
    debug_printf("Starting I2S\n");
    i2s_master(i_i2s, ports_i2s_dac, 1, ports_i2s_adc, 1, port_i2s_bclk, port_i2s_wclk, clk_i2s, clk_mclk)
    ← ;
  }
  on tile[AUDIO_TILE]: [[distribute]] i2s_handler(i_i2s, i_serial_out, i_codec, i_buttons);
  on tile[SPDIF_TILE]: led_driver(i_leds, port_leds_row, port_leds_col);
}
return 0;
    
```

This code starts all of the tasks in parallel. Note that each task has an on tile[] placement directive which maps that particular task to a tile. For example, the I²S task must run on the tile to which the DAC is connected on the hardware.



The rate_server(), spdif_handler(), button_listener() and input_gpio_with_events() tasks are instantiated from their own par{} statement. This allows them to be started with the common attributes combine and ordered. Combine instructs the compiler to flatten the tasks into a single large task and ordered allows the select cases in the combined task to fire in a given order. In this case, we need to prioritize spdif_handler() to preserve the client -> server communication flow within the signal

chain and avoid deadlock.

-  The `asrc()` task is preceded by a for-loop like construct `par(int i=0; i<ASRC_N_INSTANCES; i++)`. This construct is called a parallel replicator and instantiates `ASRC_N_INSTANCES` instances of the `asrc()` task. The index within the replicator allows the interface array passed to the task to be indexed appropriately.
-  Tasks marked as `distributable` have their `select{}` cases automatically distributed across logical cores by the compiler. The tasks must reside of the same tile for this to be possible. For further information regarding distributable and combinable tasks, please see section 2.3 of the Xmos Programming guide “Creating tasks for flexible placement”.

2.5 Audio Source and Sink

The ASRC allows two audio clock domains to be bridged. In order to effectively demonstrate this, SPDIF receive was chosen as the audio source because it is synchronized to the clock of the SPDIF transmitter, which is external to the hardware used. The I²S audio output has its own local clock driven by a crystal source on the xCORE-200 Multichannel Audio Platform.

The usage and operation of the SPDIF and I²S functions is described in the associated documentation for those libraries.

2.6 Configuring audio codecs

Two audio codecs are used on the xCORE-200 Multichannel Audio Platform. A Cirrus CS5368 for audio input and a Cirrus CS4384 for audio output, although only the output path is used for this demonstration. These codecs have to be configured before they can be used. The `audio_codec_cs4384_cs5368()` task handles this via an interface call over `i_codec` which in turn makes a call to methods in the `i_i2c` client interface and `i_gpio` client interfaces to make the necessary hardware settings.

```
void audio_codec_cs4384_cs5368(server audio_codec_config_if i_codec,
                               client i2c_master_if i2c,
                               enum codec_mode_t codec_mode,
                               client output_gpio_if i_dsd_mode,
                               client output_gpio_if i_dac_rst_n,
                               client output_gpio_if i_adc_rst_n,
                               client output_gpio_if i_clk_fsel)
```



If you are porting this application to a different xCORE development board then this is the function that will have to be modified to configure the relevant codecs for that board.

The usage and operation of I²C and GPIO functions is described in the associated documentation for those libraries.

2.7 SPDIF Handler

The SPDIF library uses a channel interface to receive samples from the SPDIF receiver component. The rest of the application uses xC interfaces to communicate and so a thin task is included which simply receives samples and forwards them to the next stage over an interface. Because this task initiates the communication, it is connected to the client side of the downstream interface.

```
void spdif_handler(streaming_chanend c_spdif_rx, client serial_transfer_push_if i_serial_in)
```

2.8 I²S Handler

The purpose of the I²S handler is to provide samples to I²S immediately when required and pull samples from the generic `block2serial` interface as needed. The interface to the upstream `block2serial()` task is a client interface meaning that samples are actively requested. It is a thin task that abstracts the I²S interface away from `block2serial` and can easily be modified for different audio sinks.

In addition, the I²S handler calls a method in the `i_codec` client interface to request reconfiguration of the hardware on a sample rate change event. It also listens on the `i_buttons` server side interface to see when a change in sample rate is requested via user input, and restarts the I²S component at the new sample rate.

2.9 Serial to Block

The ASRC processing receives blocks of samples whereas the SPDIF receive component provides left/right samples one at a time. This means we need a task that fills a block buffer up with samples until it is full, whereupon it passes the entire block to the ASRC tasks. The `serial2block()` task receives streaming samples on the server side of the `i_serial_in` interface and pushes out block of samples on the `i_block_transfer` client side interface. Because there are multiple ASRC tasks, `i_block_transfer` is an array of interfaces, where the left and right samples are separated out into individual paths.

```
void serial2block(server serial_transfer_push_if i_serial_in, client block_transfer_if i_block_transfer[
    ↪ ASRC_N_INSTANCES], server sample_rate_enquiry_if i_input_rate)
```

The input block size to the ASRC is fixed on initialization and the rate is matched on both sides so the buffer scheme used for the block can be a simple double buffer (ping-pong). This allows one side of the double buffer to be filled while the other is emptied. See the `lib_src` documentation for details on the buffer format expected by ASRC.

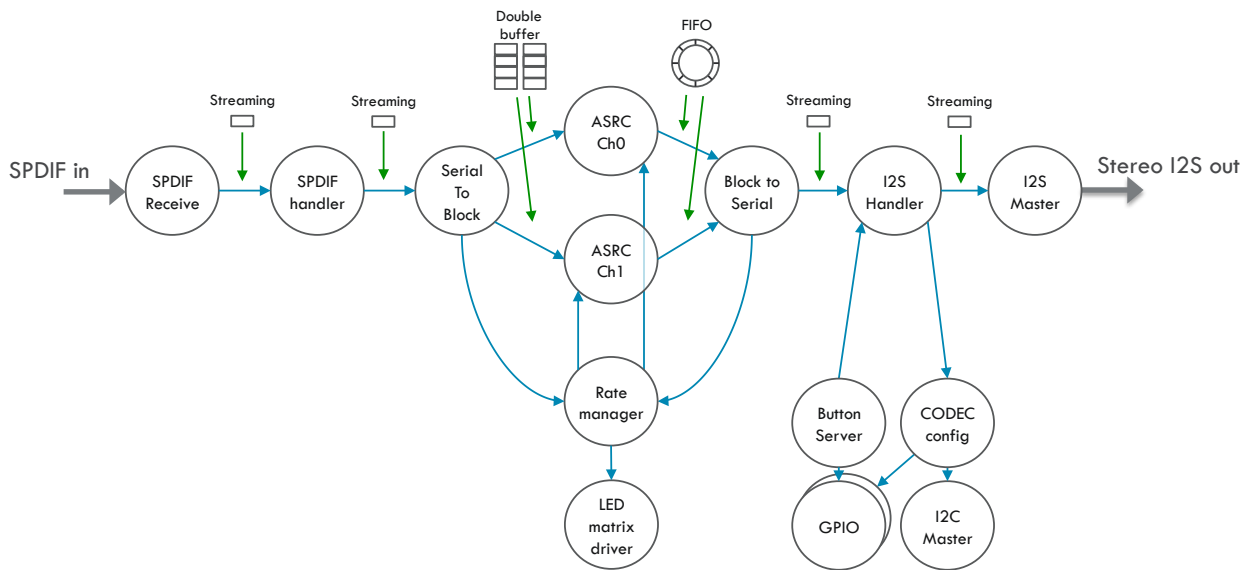


Figure 3: Use of Buffers within the Application

The Serial to Block task has one additional function which is to timestamp the samples it receives on the serial input interface. When queried over the `i_sr_input` server side interface, it returns the number of samples and elapsed time. Using a 10ns timer and a query rate of 20ms, for example, it allows a theoretical rate measurement accuracy of 0.5PPM.

The Serial to Block function is designed to be generic so that it can be used with a range of audio sources. Only the SPDIF Handler task needs to be customized when adapting this demonstration for use with other audio sources.

2.10 Block to Serial

The Block to Serial function provides the opposite functionality to Serial to Block; that is it takes blocks of samples from the ASRC output and serves up a serial stream to the the downstream audio sink. You may have noticed from the commincation roles diagram that this is the task which consists purely of server interfaces. This is because it is the task which provides elasticity within the data path.

```
unsafe void block2serial(server block_transfer_if i_block2serial[ASRC_N_INSTANCES], server
↳ serial_transfer_pull_if i_serial_out, server sample_rate_enquiry_if i_output_rate)
```

However, it differs from Serial to Block in one main aspect. The output from ASRC is a *variable* size block of samples, the size of which will vary on each call to the ASRC. This means a double-buffer scheme cannot be used because it doesn't provide any support for varying block sizes. This would especially be an issue where the emptying block is a different size to the filling block, which often occurs.

To solve this problem, a FIFO is used which introduces some elasticity permitting the varying rates. The FIFO has been enhanced to support filling using contiguous blocks, which is required for maximum efficiency at the output of the ASRC which expects contiguous memory. Use of the FIFO has the added benefit that the fill level can be monitored. The fill level provides important information about the rate ratio between input and output and effectively provides an I term for a PI loop, which is necessary for the rate matching servo algorithm to work. See the Rate Server section for further details.

The FIFO also helps buffer any small errors in rate difference when either the source or sink rate wanders. The size of the FIFO is set by OUT_FIFO_SIZE and is initialized to eight times the maximum number of samples that will be produced by a single call to the ASRC function.

```
#define ASRC_N_INSTANCES 2 //Number of instances (each usually run a logical core) used to
↳ process audio (minimum 1)
#define ASRC_CHANNELS_PER_INSTANCE (ASRC_N_CHANNELS/ASRC_N_INSTANCES)
//Calculated number of audio channels processed by each core
#define ASRC_N_IN_SAMPLES 8 //Number of samples per channel in each block passed into SRC
↳ each call
//Must be a power of 2 and minimum value is 4 (due to two /2 decimation stages)
#define ASRC_N_OUT_IN_RATIO_MAX 3 //Max ratio between samples out:in per processing step
↳ (44.1->192 is worst case)
#define ASRC_MAX_BLOCK_SIZE (ASRC_N_IN_SAMPLES * ASRC_N_OUT_IN_RATIO_MAX)
#define OUT_FIFO_SIZE (ASRC_MAX_BLOCK_SIZE * 8) //Size per channel of block2serial
↳ output FIFO
```

To enable the use of C-style pointers (without bounds checking to improve performance and to enable compatibility with C-code), unsafe pointers have been used. This requires that the entire scope of this block is declared unsafe to indicate to the reader that memory safety is maintained by the code's functionality rather than the compiler.



The serial2block(), asrc() and block2serial() tasks must reside on the same tile because they use a shared memory interface to provide the highest performance data passing between tasks.

2.11 Rate Server

The Rate Server function performs an essential function with the ASRC demonstration. It monitors the rate of the input and output streams and calculates the fs_ratio (sample rate ratio) to be provided to the ASRC algorithm each time it is called. The Rate Server acquires information regarding stream rate and FIFO fill level from the serial/block tasks via client side interfaces i_spdif_rate and i_2s_rate on a periodic basis. It then implements a PI controller and applies some simple filtering using a first order low pass filter to reduce any jitter introduced from rate measurements. The calculated fs_ratio is served

up to the ASRC tasks via the `i_fs_ratio` server side interface using the `get_ratio` method.

```
void rate_server(client sample_rate_enquiry_if i_spdif_rate, client sample_rate_enquiry_if i_i2s_rate,
                server fs_ratio_enquiry_if i_fs_ratio[ASRC_N_INSTANCES], client led_matrix_if i_leds)
```

It uses an unsigned Q20.12 fixed point variable to calculate the current sample rate and passes the calculated and filtered `fs_ratio` to the ASRC task using an unsigned Q4.28 fixed point representation.

The Rate Server also calculates the nominal sample rate using a table which specifies the range of values considered acceptable. The table is initialized to accept a deviation of up to 1000PPM, as supported by the ASRC library.

```
#define SR_TOLERANCE_PPM    1000    //How far the detect_frequency function will allow before declaring
    ↪ invalid in p.p.m.
#define LOWER_LIMIT(freq) (freq - (((long long) freq * SR_TOLERANCE_PPM) / 1000000))
#define UPPER_LIMIT(freq) (freq + (((long long) freq * SR_TOLERANCE_PPM) / 1000000))
```

The nominal sample rates are monitored for change and written to a VALID/INVALID flag. For example, a change in rate will always transition through an INVALID state. When this occurs, a notification is sent to the ASRC task using the server side interface `fs_ratio_enquiry_if` which then queries back to the Rate Server for the nominal sample rate. This information is used to re-initialize the ASRC core filters to support the new major sample rate change.

The Rate Server also acts as a simple application and outputs information about buffer level and sample rates to the LED matrix using the client side interface method `"i_leds.set()"` as well as printing internal state to the console on a periodic basis.

The rates at which the `fs_ratio` is calculated and the internal state is printed to the console is defined by the following values:

```
#define SR_CALC_PERIOD    2000000    //20ms The period over which we count samples to find the rate
//Because we timestamp at 10ns resolution, we get 20000000/10 = 21bits of precision
#define REPORT_PERIOD    500100000  //5.001s. How often we print the rates to the screen for debug. Chosen to
    ↪ not clash with above
```

2.12 Asynchronous Sample Rate Conversion

These tasks handle calling the core ASRC algorithm functions. There is one task per audio channel (2) and each is given a dedicated task running on a logical core to ensure that a guaranteed amount of performance is available for the ASRC.

```
unsafe void asrc(server block_transfer_if i_serial2block, client block_transfer_if i_block2serial, client
    ↪ fs_ratio_enquiry_if i_fs_ratio)
```

There are three interface connections to each of the tasks. Indication of availability of a new sample block is provided by a push method call into the ASRC task over the `i_serial2block` interface, which also returns a pointer to the producer for the next buffer to write to. This triggers calling of the `asrc_process()` function which, on completion, writes the output sample block to the FIFO and notifies downstream using the client side `i_block2serial` interface. Included within each of these interface calls is a parameter indicating the number of samples within the block, although this is only used on the output side of the ASRC where this is a variable.

The ASRC processing call is not actually made within the input sample push select case. The reason for this is that the task on the client side of the interface would block until completion of the case (i.e. `break;` is reached). To allow the `serial2block` client to continue unblocked after the sample block push, a flag is set which acts as a guard indicating that the ASRC processing event can be fired on the default case.

An additional `select` case is included within the main `select` loop which is used to notify the ASRC task of a change in either the input or output nominal sample rates. When this occurs a call to `asrc_init()`

is made which initializes the ASRC and sets up the cascaded FIR filters to support the new required rate change.

To ensure that 100MHz (Assuming 500MHz core clock) is allocated to each of the ASRC tasks, a built-in function is called to force the logical core to be issued every fifth core clock.

```
set_core_high_priority_on(); //Give me guaranteed 1/5 of the processor clock i.e. 100MHz
```

2.13 LED Driver

The 16 LEDs on the xCORE-200 Multichannel Audio Platform are arranged in a 4x4 matrix and are multiplexed as rows and columns. Consequently, to be able to address each LED individually, a simple task to multiplex the LEDs is included which displays a 16-bit bitmap. The LEDs are set and cleared when the `i_leds.set()` server side interface method is called. A timer keeps track of the column multiplexing and outputs the appropriate nibbles in turn to display the 16-bit bitmap on the LEDs.

```
[[combinable]]void led_driver(server led_matrix_if i_leds, out port port_leds_col, out port port_leds_row){
```

The LED matrix display task is a low performance task and can be combined with other tasks onto a single logical core.

2.14 Button Listener

A simple task that periodically reads and de-bounces the button inputs is included. It detects when the button is pressed and sends the index of the pressed button over the client side interface method `i_buttons.pressed()`.

```
[[combinable]]void button_listener(client buttons_if i_buttons, client input_gpio_if i_button_port){
```

The button listener task is a low performance task that is ideal to be combined with other tasks onto a single logical core.

2.15 Optimization

The task diagram shows the logical tasks defined within the application and the communications between them. However, many of these tasks do not require a whole logical core to function and so can be either combined with or distributed across other tasks, performance requirement permitting. This optimization is made by the compiler, with help from the programmer, by writing tasks with a certain form and by adding attributes as compiler hints. The result of this optimization is that 15 tasks are able to be placed on just 6 logical cores. Further, the LED matrix task has low performance requirements and may be combined with additional user tasks if needed.

The below diagram shows the mapping to physical processing elements (logical cores). Note that the ASRC tasks have been configured to be scheduled once every 5 core clocks to ensure they receive a guaranteed 100MHz, although they will automatically receive this performance level within this example application due to five or less tasks being placed on the tile.

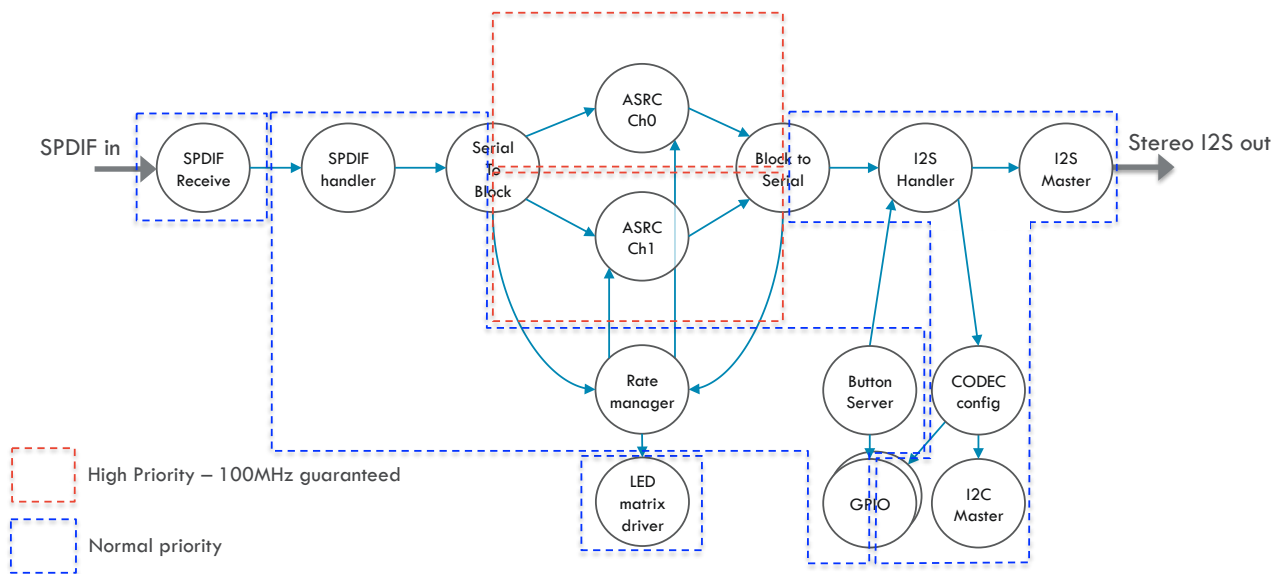


Figure 4: Mapping of Tasks to Logical Cores in xCORE Processor

APPENDIX A - Demo hardware setup

The demo is designed to run on the xCORE-200 Multichannel Audio Platform. To run the demo:

- Connect the XTAG-2 or XTAG-3 USB debug adapter to the xCORE-200 Multichannel Audio Platform.
- Connect the XTAG-2 or XTAG-3 to the host PC using USB extension cable.
- Connect an optical SPDIF source to the Rx TOSLINK connector.
- Connect headphones or speakers to the channels 1/2 of the line out 3.5mm jack.
- Connect the 12V power adapter to the xCORE-200 Multichannel Audio Platform.

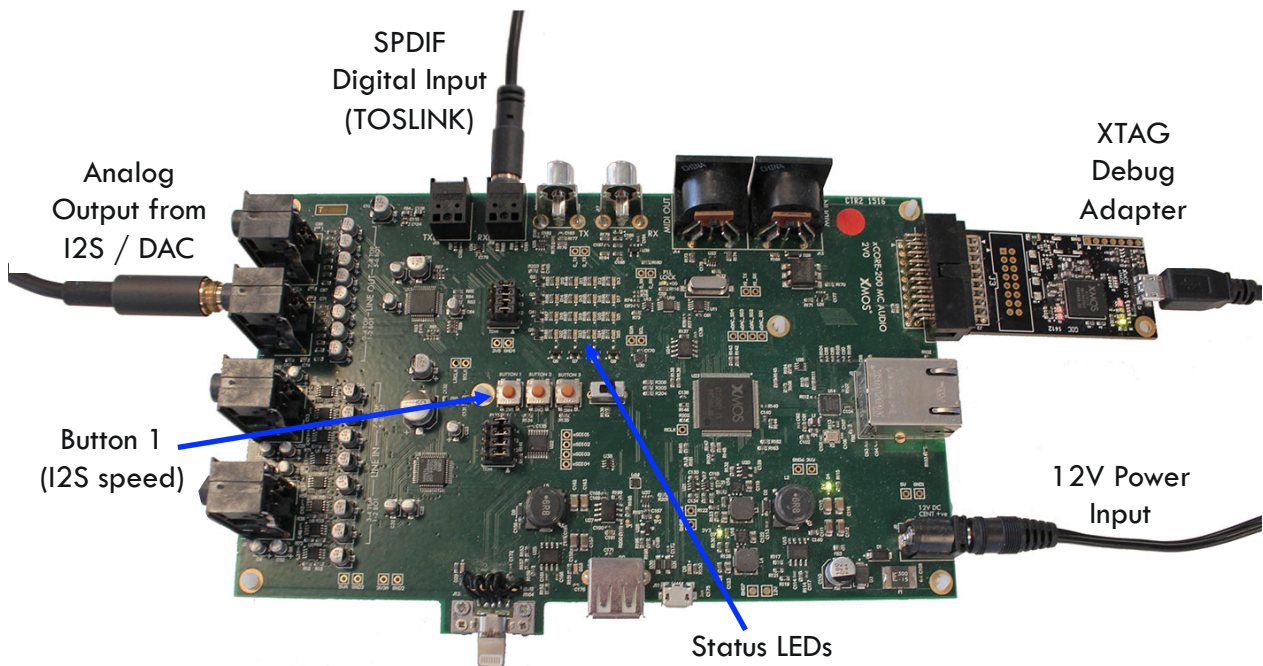


Figure 5: Hardware setup for XMOS SPDIF to I2S using ASRC

APPENDIX B - Launching the demo device

Once the application source code is imported into the tools it can be built by pressing the **Build** button in xTIMEcomposer. This will create the AN00231_ASRC_SPDIF_TO_DAC.xe binary in the bin folder of the project. xTIMEcomposer may have to import the dependent libraries if you do not already have them in your workspace; this will occur automatically on build.

A *Run Configuration* then needs to be set up. This can be done by selecting the **Run ► Run Configurations...** menu. You can create a new run configuration by right clicking on the **xCORE application** group in the left hand pane and **New**.

Select your XMOS XTAG in the Target box and click **Apply**. You can now click the **Run** button to launch the application.

If you wish to view the reported values from the Rate Server task in the console, ensure that the **Run Configurations ► Target I/O ► xSCOPE via xCONNECT...** option is set. xSCOPE printing is already enabled within the application.

Alternatively, the command line tools can be used to build and run the application.

Building is done by changing to the folder containing the src sub-folder and doing a call to xmake. Running the application is then done using the command: `xrun --xscope bin/AN00231_ASRC_SPDIF_TO_DAC.xe`.

The audio from the SPDIF input should now be playing on channels 1/2 of the DAC output.

B.1 LED Indicators

The column of four LEDs closest to the analog outputs indicates the sample rate of I²S. The column closest the crystal indicates the sample rate of the received SPDIF stream. The two columns in between show the current buffer fill level, where zero indicates exactly half full (the target level). The table below provides further detail of the LED meanings.

	Row			
Column	1	2	3	4
4	I ² S >= 96KHz	Buffer > 12 samples	Buffer > 6 samples	SPDIF >= 96KHz
3	I ² S >= 88.2KHz	N/A	Buffer > 0 samples	SPDIF >= 88.2KHz
2	I ² S >= 48KHz	N/A	Buffer < 0 samples	SPDIF >= 44.1KHz
1	I ² S >= 44.1KHz	Buffer < -12 samples	Buffer < -6 samples	SPDIF >= 44.1KHz

Table 1: LED Indicator meaning

If the buffer level is exactly half, then no LEDs are illuminated in either columns 2 or 3.

B.2 Change the I2S rate

The rate of the I²S interface is initialized to 48000 but can be changed at run-time by pressing button 1. Each press of the button cycles through 44.1KHz -> 48KHz -> 88.2KHz -> 96KHz and wraps back to the beginning.

A mute function is included to output zero samples from the point of an I²S rate change for a fixed number of sample cycles. This prevents old samples being outputted during the period when `rate_server()` detects a change in frequency and re-configures the ASRC.

APPENDIX C - 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>

XMOS I²S/TDM Library

http://www.xmos.com/support/libraries/lib_i2s

XMOS I²C Library

http://www.xmos.com/support/libraries/lib_i2c

XMOS GPIO Library

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

APPENDIX D - Full source code listing

D.1 Source code for app_config.h

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#ifndef APP_CONFIG_H_
#define APP_CONFIG_H_

#define ASRC_N_CHANNELS 2 //Total number of audio channels to be processed by SRC (
    ↳ minimum 1)
#define ASRC_N_INSTANCES 2 //Number of instances (each usually run a logical core) used to
    ↳ process audio (minimum 1)
#define ASRC_CHANNELS_PER_INSTANCE (ASRC_N_CHANNELS/ASRC_N_INSTANCES)
    ↳ //Calculated number of audio channels processed by each core
#define ASRC_N_IN_SAMPLES 8 //Number of samples per channel in each block passed into SRC
    ↳ each call
    ↳ //Must be a power of 2 and minimum value is 4 (due to two /2
    ↳ ↳ decimation stages)
#define ASRC_N_OUT_IN_RATIO_MAX 3 //Max ratio between samples out:in per processing step
    ↳ (44.1->192 is worst case)
#define ASRC_MAX_BLOCK_SIZE (ASRC_N_IN_SAMPLES * ASRC_N_OUT_IN_RATIO_MAX)
#define OUT_FIFO_SIZE (ASRC_MAX_BLOCK_SIZE * 8) //Size per channel of block2serial
    ↳ output FIFO
#define ASRC_DITHER_SETTING OFF //No output dithering of samples from 32b to 24b

#define DEFAULT_FREQ_HZ_SPDIF 48000
#define DEFAULT_FREQ_HZ_I2S 48000
#define MCLK_FREQUENCY_48 24576000
#define MCLK_FREQUENCY_44 22579200

#endif /* APP_CONFIG_H_ */
```

D.2 Source code for main.h

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#ifndef MAIN_H_
#define MAIN_H_

//Interface for obtaining timing information about the serial stream
typedef interface sample_rate_enquiry_if {
    unsigned get_sample_count(int &elapsed_time_in_ticks); //Returns sample count and elapsed time in 10ns
    ↳ ticks since last query
    {unsigned, unsigned} get_buffer_level(void); //Returns the buffer level. Note this will return
    ↳ zero for the serial2block function which uses a double-buffer
} sample_rate_enquiry_if;

//Interface for serving up the last calculated fs_ratio between input and output
typedef interface fs_ratio_enquiry_if {
    unsigned get_ratio(unsigned nominal_fs_ratio); //Get the current ratio
    [[notification]] slave void new_sr_notify(void);
    [[clears_notification]] unsigned get_in_fs(void); //Get the nominal sample rate
    [[clears_notification]] unsigned get_out_fs(void);
} fs_ratio_enquiry_if;

//Sets or clears pixel. Origin is bottom left scanning right
typedef interface led_matrix_if {
    void set(unsigned col, unsigned row, unsigned val);
} led_matrix_if;

//Notifies when the button is pressed
typedef interface buttons_if {
    void pressed(void);
} buttons_if;

//Type which tells us the current status of the detected sample rate
//Is it with range of a supported rate?
typedef enum sample_rate_status_t{
    INVALID,
    VALID }

```

```

sample_rate_status_t;

//Q4.28 fixed point frequency ratio type
typedef unsigned fs_ratio_t;

#endif /* MAIN_H_ */

```

D.3 Source code for main.xc

```

// Copyright (c) 2016, XMOS Ltd, All rights reserved
//Standard includes
#include <xs1.h>
#include <platform.h>
#include <string.h>

//Supporting libraries
#include "src.h"
#include "spdif.h"
#include "i2s.h"
#include "i2c.h"
#include "gpio.h"
#include "assert.h"

//Application specific includes
#include "main.h"
#include "block_serial.h"
#include "cs4384_5368.h" //CODEC setup
#include "app_config.h" //General settings

//Debug includes
#include <debug_print.h> //Enabled by -DDEBUG_PRINT_ENABLE=1 in Makefile
#include <xscope.h>

//These port assignments all correspond to XU216 multichannel audio board 2V0
//The port assignments can be changed for a different port map.

#define AUDIO_TILE 0
in buffered port:32 ports_i2s_adc[4] = on tile[AUDIO_TILE]: {XS1_PORT_1I,
    XS1_PORT_1J,
    XS1_PORT_1K,
    XS1_PORT_1L};

port port_i2s_mclk = on tile[AUDIO_TILE]: XS1_PORT_1F;
clock clk_mclk = on tile[AUDIO_TILE]: XS1_CLKBLK_2;
out buffered port:32 port_i2s_bclk = on tile[AUDIO_TILE]: XS1_PORT_1H;
out buffered port:32 port_i2s_wclk = on tile[AUDIO_TILE]: XS1_PORT_1G;
clock clk_i2s = on tile[AUDIO_TILE]: XS1_CLKBLK_1;
out buffered port:32 ports_i2s_dac[4] = on tile[AUDIO_TILE]: {XS1_PORT_1M,
    XS1_PORT_1N,
    XS1_PORT_1O,
    XS1_PORT_1P};

#define SPDIF_TILE 1
port port_spdif_rx = on tile[SPDIF_TILE]: XS1_PORT_1O;
clock clk_spdif_rx = on tile[SPDIF_TILE]: XS1_CLKBLK_1;

out port port_leds_col = on tile[SPDIF_TILE]: XS1_PORT_4C; //4x4 LED matrix
out port port_leds_row = on tile[SPDIF_TILE]: XS1_PORT_4D;

port port_i2c = on tile[AUDIO_TILE]: XS1_PORT_4A; //I2C for CODEC configuration

port port_audio_config = on tile[AUDIO_TILE]: XS1_PORT_8C;
char pin_map_audio_cfg[5] = {0, 1, 5, 6, 7};
/* Bit map for XS1_PORT_8C
 * 0 DSD_MODE
 * 1 DAC_RST_N
 * 2 USB_SELO
 * 3 USB_SEL1
 * 4 VBUS_OUT_EN
 * 5 PLL_SELECT
 * 6 ADC_RST_N
 * 7 MCLK_FSEL

```

```

*/
port port_buttons = on tile[AUDIO_TILE]: XS1_PORT_4D; //Buttons and switch
char pin_map_buttons[1] = {0}; //Port map for buttons GPIO
↳ task. We are just interested in bit 0

out port port_debug_tile_1 = on tile[SPDIF_TILE]: XS1_PORT_1N; //MIDI OUT. A good test point
↳ to probe..
out port port_debug_tile_0 = on tile[AUDIO_TILE]: XS1_PORT_1D; //SPDIF COAX TX. A good test
↳ point to probe..

//Application task prototypes. For functionality of these tasks, see comments in implementations below
[[combinable]] void spdif_handler(streaming chanend c_spdif_rx, client serial_transfer_push_if i_serial_in);
unsafe void asrc(server block_transfer_if i_serial2block, client block_transfer_if i_block2serial, client
↳ fs_ratio_enquiry_if i_fs_ratio);
[[distributable]] void i2s_handler(server i2s_callback_if i2s, client serial_transfer_pull_if i_serial_out,
↳ client audio_codec_config_if i_codec, server buttons_if i_buttons);
[[combinable]] void rate_server(client sample_rate_enquiry_if i_spdif_rate, client sample_rate_enquiry_if
↳ i_output_rate, server fs_ratio_enquiry_if i_fs_ratio[ASRC_N_INSTANCES], client led_matrix_if i_leds);
[[combinable]] void led_driver(server led_matrix_if i_leds, out port port_leds_row, out port port_leds_col);
[[combinable]] void button_listener(client buttons_if i_buttons, client input_gpio_if i_button_port);

int main(void){
  serial_transfer_push_if i_serial_in;
  block_transfer_if i_serial2block[ASRC_N_INSTANCES];
  block_transfer_if i_block2serial[ASRC_N_INSTANCES];
  serial_transfer_pull_if i_serial_out;
  sample_rate_enquiry_if i_sr_input, i_sr_output;
  fs_ratio_enquiry_if i_fs_ratio[ASRC_N_INSTANCES];
  interface audio_codec_config_if i_codec;
  interface i2c_master_if i_i2c[1];
  interface output_gpio_if i_gpio[5]; //See mapping of bits 0..7 above in port_audio_config
  interface i2s_callback_if i_i2s;
  led_matrix_if i_leds;
  buttons_if i_buttons;
  input_gpio_if i_button_gpio[1];
  streaming chan c_spdif_rx;

  par{
    on tile[SPDIF_TILE]: spdif_rx(c_spdif_rx, port_spdif_rx, clk_spdif_rx, DEFAULT_FREQ_HZ_SPDIF);
    on tile[AUDIO_TILE]: [[combine, ordered]] par{
      rate_server(i_sr_input, i_sr_output, i_fs_ratio, i_leds);
      spdif_handler(c_spdif_rx, i_serial_in);
      button_listener(i_buttons, i_button_gpio[0]);
      input_gpio_with_events(i_button_gpio, 1, port_buttons, pin_map_buttons);
    }

    on tile[AUDIO_TILE]: serial2block(i_serial_in, i_serial2block, i_sr_input);
    on tile[AUDIO_TILE]: unsafe{ par (int i=0; i<ASRC_N_INSTANCES; i++) asrc(i_serial2block[i],
      ↳ i_block2serial[i], i_fs_ratio[i]);}
    on tile[AUDIO_TILE]: unsafe { par{[[distribute]] block2serial(i_block2serial, i_serial_out,
      ↳ i_sr_output);}}

    on tile[AUDIO_TILE]: audio_codec_cs4384_cs5368(i_codec, i_i2c[0], CODEC_IS_I2S_SLAVE, i_gpio[0],
      ↳ i_gpio[1], i_gpio[3], i_gpio[4]);
    on tile[AUDIO_TILE]: i2c_master_single_port(i_i2c, 1, port_i2c, 10, 0 /*SCL*/, 1 /*SDA*/, 0);
    on tile[AUDIO_TILE]: output_gpio(i_gpio, sizeof(pin_map_audio_cfg), port_audio_config,
      ↳ pin_map_audio_cfg);
    on tile[AUDIO_TILE]: {
      i_gpio[2].output(0); //Select fixed local clock on MCLK mux
      configure_clock_src(clk_mclk, port_i2s_mclk); //Connect MCLK clock block to input pin
      start_clock(clk_mclk);
      debug_printf("Starting I2S\n");
      i2s_master(i_i2s, ports_i2s_dac, 1, ports_i2s_adc, 1, port_i2s_bclk, port_i2s_wclk, clk_i2s,
        ↳ clk_mclk);
    }
    on tile[AUDIO_TILE]: [[distribute]] i2s_handler(i_i2s, i_serial_out, i_codec, i_buttons);
    on tile[SPDIF_TILE]: led_driver(i_leds, port_leds_row, port_leds_col);
  }
  return 0;
}

```

```

//Shim task to handle setup and streaming of SPDIF samples from the streaming channel to the interface of
↳ serial2block
[[combinable]] //Run on same core as other tasks if possible
void spdif_handler(streaming chanend c_spdif_rx, client serial_transfer_push_if i_serial_in)
{
    unsigned index;                //Channel index
    signed long sample;            //Sample received from SPDIF

    delay_microseconds(10000);     //Bug 17263 workaround (race condition in distributable task
    ↳ init)
    while (1) {
        select {
            case spdif_receive_sample(c_spdif_rx, sample, index):
                i_serial_in.push(sample, index); //Push them into serial to block

                break;
        }
    }
}

//Helper function for converting sample to fs index value
static fs_code_t samp_rate_to_code(unsigned samp_rate){
    unsigned samp_code = 0xdead;
    switch (samp_rate){
        case 44100:
            samp_code = FS_CODE_44;
            break;
        case 48000:
            samp_code = FS_CODE_48;
            break;
        case 88200:
            samp_code = FS_CODE_88;
            break;
        case 96000:
            samp_code = FS_CODE_96;
            break;
        case 176400:
            samp_code = FS_CODE_176;
            break;
        case 192000:
            samp_code = FS_CODE_192;
            break;
    }
    return samp_code;
}

//The ASRC processing task - has it's own logical core to reserve processing MHz
unsafe void asrc(server block_transfer_if i_serial2block, client block_transfer_if i_block2serial, client
↳ fs_ratio_enquiry_if i_fs_ratio)
{
    int input_db1_buf[2][ASRC_CHANNELS_PER_INSTANCE * ASRC_N_IN_SAMPLES]; //Double buffers for to block/
    ↳ serial tasks
    unsigned buff_idx = 0;
    int * unsafe asrc_input = input_db1_buf[0]; //pointer for ASRC input buffer
    int * unsafe p_out_fifo; //C-style pointer for output FIFO

    p_out_fifo = i_block2serial.push(0); //Get pointer to initial write buffer

    set_core_high_priority_on(); //Give me guaranteed 1/5 of the processor clock i.e. 100MHz

    fs_code_t in_fs_code = samp_rate_to_code(DEFAULT_FREQ_HZ_SPDIF); //Sample rate code 0..5
    fs_code_t out_fs_code = samp_rate_to_code(DEFAULT_FREQ_HZ_I2S);

    asrc_state_t asrc_state[ASRC_CHANNELS_PER_INSTANCE]; //ASRC state machine state
    int asrc_stack[ASRC_CHANNELS_PER_INSTANCE][ASRC_STACK_LENGTH_MULT * ASRC_N_IN_SAMPLES]; //
    ↳ Buffer between filter stages
    asrc_ctrl_t asrc_ctrl[ASRC_CHANNELS_PER_INSTANCE]; //Control structure
    asrc_adfir_coefs_t asrc_adfir_coefs; //Adaptive filter coefficients

    for(int ui = 0; ui < ASRC_CHANNELS_PER_INSTANCE; ui++)
    unsafe {
        //Set state, stack and coefs into ctrl structure
        asrc_ctrl[ui].psState = &asrc_state[ui];
    }
}

```

```

    asrc_ctrl[ui].piStack          = asrc_stack[ui];
    asrc_ctrl[ui].piADCoefs       = asrc_adfir_coefs.iASRCADFIRCoefs;
  }

  //Initialise ASRC
  unsigned nominal_fs_ratio = asrc_init(in_fs_code, out_fs_code, asrc_ctrl, ASRC_CHANNELS_PER_INSTANCE,
    ↪ ASRC_N_IN_SAMPLES, ASRC_DITHER_SETTING);

  int do_dsp_flag = 0;           //Flag to indicate we are ready to process. Minimises blocking on
    ↪ push case below

  while(1){
    select{
      case i_serial2block.push(const unsigned n_samps) -> int * unsafe new_buff_ptr:
        asrc_input = input_dbl_buf[buff_idx]; //Grab address of freshly filled buffer
        do_dsp_flag = 1;                       //We have a fresh buffer to process
        buff_idx ^= 1;                         //Flip double buffer for filling
        new_buff_ptr = input_dbl_buf[buff_idx]; //Return pointer for serial2block to fill
        break;

      case i_fs_ratio.new_sr_notify():         //Notification from SR manager that we need to
        ↪ initialise ASRC
          in_fs_code = samp_rate_to_code(i_fs_ratio.get_in_fs()); //Get the new SRs
          out_fs_code = samp_rate_to_code(i_fs_ratio.get_out_fs());
          debug_printf("New rate in SRC in=%d, out=%d\n", in_fs_code, out_fs_code);
          nominal_fs_ratio = asrc_init(in_fs_code, out_fs_code, asrc_ctrl, ASRC_CHANNELS_PER_INSTANCE,
            ↪ ASRC_N_IN_SAMPLES, ASRC_DITHER_SETTING);
          break;

      do_dsp_flag => default:                 //Do the sample rate conversion
        //port_debug <: 1;                    //debug
        unsigned n_samps_out;
        fs_ratio_t fs_ratio = i_fs_ratio.get_ratio(nominal_fs_ratio); //Find out how many samples to
          ↪ produce

        //Run the ASRC and pass pointer of output to block2serial
        n_samps_out = asrc_process((int *)asrc_input, (int *)p_out_fifo, fs_ratio, asrc_ctrl);
        p_out_fifo = i_block2serial.push(n_samps_out); //Get pointer to next write buffer

        do_dsp_flag = 0;                     //Clear flag and wait for next input block
        //port_debug <: 0;                    //debug
        break;
    }
  } //While 1
} //asrc

#define MUTE_MS_AFTER_SR_CHANGE 350 //350ms. Avoids incorrect rate playing momentarily while new rate is
  ↪ detected

//Shim task to handle setup and streaming of I2S samples from block2serial to the I2S module
[[distributable]]
#pragma unsafe arrays //Performance optimisation of i2s_handler task
void i2s_handler(server i2s_callback_if i2s, client serial_transfer_pull_if i_serial_out, client
  ↪ audio_codec_config_if i_codec, server buttons_if i_buttons)
{
  unsigned sample_rate = DEFAULT_FREQ_HZ_I2S;
  unsigned mclk_rate;
  unsigned restart_status = I2S_NO_RESTART;
  unsigned mute_counter; //Non zero indicates mute. Initialised on I2S init SR change

  while (1) {
    select {
      case i2s.init(i2s_config_t &i2s_config, tdm_config_t &tdm_config):
        if (!(sample_rate % 48000)) mclk_rate = MCLK_FREQUENCY_48; //Initialise MCLK to appropriate
          ↪ multiple of sample_rate
        else mclk_rate = MCLK_FREQUENCY_44;
        i2s_config.mclk_bclk_ratio = mclk_rate / (sample_rate << 6);
        i2s_config.mode = I2S_MODE_I2S;
        i_codec.reset(sample_rate, mclk_rate);
        debug_printf("Initializing I2S to %dHz and MCLK to %dHz\n", sample_rate, mclk_rate);
        restart_status = I2S_NO_RESTART;
        mute_counter = (sample_rate * MUTE_MS_AFTER_SR_CHANGE) / 1000; //Initialise to a number of
          ↪ milliseconds
        break;
    }
  }
}

```

```

//Start of I2S frame
case i2s.restart_check() -> i2s_restart_t ret:
    ret = restart_status;
    break;

//Get samples from ADC
case i2s.receive(size_t index, int32_t sample):
    break;

//Send samples to DAC
case i2s.send(size_t index) -> int32_t sample:
    sample = i_serial_out.pull(index);
    if (mute_counter){
        sample = 0;
        mute_counter --;
    }
    break;

//Cycle through sample rates of I2S on button press
case i_buttons.pressed():
    switch (sample_rate) {
        case 44100:
            sample_rate = 48000;
            break;
        case 48000:
            sample_rate = 88200;
            break;
        case 88200:
            sample_rate = 96000;
            break;
        case 96000:
            sample_rate = 44100;
            break;
    }
    restart_status = I2S_RESTART;
    break;
}
}
}

#define SR_TOLERANCE_PPM    1000    //How far the detect_frequency function will allow before declaring
    ↪ invalid in p.p.m.
#define LOWER_LIMIT(freq) (freq - (((long long) freq * SR_TOLERANCE_PPM) / 1000000))
#define UPPER_LIMIT(freq) (freq + (((long long) freq * SR_TOLERANCE_PPM) / 1000000))

static const unsigned sr_range[6 * 3] = {
    44100, LOWER_LIMIT(44100), UPPER_LIMIT(44100),
    48000, LOWER_LIMIT(48000), UPPER_LIMIT(48000),
    88200, LOWER_LIMIT(88200), UPPER_LIMIT(88200),
    96000, LOWER_LIMIT(96000), UPPER_LIMIT(96000),
    176400, LOWER_LIMIT(176400), UPPER_LIMIT(176400),
    192000, LOWER_LIMIT(192000), UPPER_LIMIT(192000) };

//Helper function for rate_server to check for validity of detected sample rate. Takes sample rate as integer
static sample_rate_status_t detect_frequency(unsigned sample_rate, unsigned &nominal_sample_rate)
{
    sample_rate_status_t result = INVALID;
    nominal_sample_rate = 0;
    for (int i = 0; i < 6 * 3; i+=3) {
        if ((sr_range[i + 1] < sample_rate) && (sample_rate < sr_range[i + 2])){
            nominal_sample_rate = sr_range[i];
            result = VALID;
        }
    }
    return result;
}

#define SR_CALC_PERIOD    2000000    //20ms The period over which we count samples to find the rate
    //Because we timestamp at 10ns resolution, we get 20000000/10 = 21bits of
    ↪ precision
#define REPORT_PERIOD    500100000    //5.001s. How often we print the rates to the screen for debug. Chosen to
    ↪ not clash with above
#define SR_FRAC_BITS    12    //Number of fractional bits used to store sample rate

```

```

//Using 12 gives us 20 bits of integer - up to 1.048MHz SR before overflow
//Below is the multiplier is used to work out SR in 20.12 representation. There is enough headroom in a long
↳ long calc
//to support a measurement period of 1s at 192KHz with over 2 order of magnitude margin against overflow
#define SR_MULTIPLIER ((1<<SR_FRAC_BITS) * (unsigned long long) XS1_TIMER_HZ)
#define SETTLE_CYCLES 3 //Number of measurement periods to skip after SR change (SR change blocks
↳ spdif momentarily so corrupts SR calc)

typedef struct rate_info_t{
    unsigned samp_count; //Sample count over last period
    unsigned time_ticks; //Time in ticks for last count
    unsigned current_rate; //Current average rate in 20.12 fixed point format
    sample_rate_status_t status; //Lock status
    unsigned nominal_rate; //Snapped-to nominal rate as unsigned integer
} rate_info_t;

//Task that queires the de/serialisers periodically and calculates the number of samples for the SRC
//to produce to keep the output FIFO in block2serial rougly centered. Uses the timestamped sample counts
//requested from serial2block and block2serial and FIFO level as P and I terms
[[combinable]]
#pragma unsafe arrays //Performance optimisation
void rate_server(client sample_rate_enquiry_if i_spdif_rate, client sample_rate_enquiry_if i_i2s_rate,
    server fs_ratio_enquiry_if i_fs_ratio[ASRC_N_INSTANCES], client led_matrix_if i_leds)
{
    rate_info_t spdif_info = { //Initialise to nominal values for default frequency
        ((DEFAULT_FREQ_HZ_SPDIF * 10000000ULL) / XS1_TIMER_HZ),
        SR_CALC_PERIOD,
        DEFAULT_FREQ_HZ_SPDIF << SR_FRAC_BITS,
        INVALID,
        DEFAULT_FREQ_HZ_SPDIF};

    rate_info_t i2s_info = { //Initialise to nominal values for default frequency
        ((DEFAULT_FREQ_HZ_I2S * 10000000ULL) / XS1_TIMER_HZ),
        SR_CALC_PERIOD,
        DEFAULT_FREQ_HZ_I2S << SR_FRAC_BITS,
        INVALID,
        DEFAULT_FREQ_HZ_I2S};

    unsigned i2s_buff_level = 0; //Buffer fill level. Initialise to empty.
    unsigned i2s_buff_size = OUT_FIFO_SIZE;
    unsigned skip_validity = 0; //Do SR validity check - need this to allow SR to settle after
    ↳ SR change

    timer t_print; //Debug print timers
    int t_print_trigger;

    t_print := t_print_trigger;
    t_print_trigger += REPORT_PERIOD;

    fs_ratio_t fs_ratio; //4.28 fixed point value of how many samples we want SRC to
    ↳ produce //input fs/output fs. ie. below 1 means inoput faster than
    ↳ output

    fs_ratio_t fs_ratio_old; //Last time round value for filtering
    fs_ratio_t fs_ratio_nominal; //Nominal fs ratio reported by SRC
    timer t_period_calc; //Timer to govern sample count periods
    int t_calc_trigger; //Trigger comparison for above

    int sample_time_spdif; //Used for passing to get_sample_count method by refrence
    int sample_time_i2s; //Used for passing to get_sample_count method by refrence

    t_period_calc := t_calc_trigger; //Get current time and set trigger for the future
    t_calc_trigger += SR_CALC_PERIOD;

    while(1){
        select{
            //Serve up latest sample count value when required. Note selects over array of interfaces
            case i_fs_ratio[int if_index].get_ratio(unsigned nominal_fs_ratio) -> fs_ratio_t fs_ratio_ret:
                fs_ratio_nominal = nominal_fs_ratio; //Allow use outside of this case
                if ((spdif_info.status == VALID) && (i2s_info.status == VALID)){
                    fs_ratio_ret = fs_ratio; //Pass back calculated value
                }
            else {
                fs_ratio = nominal_fs_ratio; //Pass back nominal until we have valid rate data
            }
        }
    }
}

```



```

break;

//Serve up the input sample rate
case i_fs_ratio[int if_index].get_in_fs(void) -> unsigned fs:
    fs = spdif_info.nominal_rate;
break;

//Serve up the output sample rate
case i_fs_ratio[int if_index].get_out_fs(void) -> unsigned fs:
    fs = i2s_info.nominal_rate;
break;

//Timeout to trigger calculation of new fs_ratio
case t_period_calc when timerafter(t_calc_trigger) := int _:
    t_calc_trigger += SR_CALC_PERIOD;

    unsigned samp_count_spdif = i_spdif_rate.get_sample_count(sample_time_spdif); //get spdif
        ↪ sample count;
    unsigned samp_count_i2s = i_i2s_rate.get_sample_count(sample_time_i2s); //And I2S
    {i2s_buff_size, i2s_buff_level} = i_i2s_rate.get_buffer_level();

    if (sample_time_spdif){ //If time is non-zero - avoids divide by zero if no input
        spdif_info.current_rate = (((unsigned long long)samp_count_spdif * SR_MULTIPLIER) /
            ↪ sample_time_spdif);
    }
    else spdif_info.current_rate = 0;
    if (sample_time_i2s){
        i2s_info.current_rate = (((unsigned long long)samp_count_i2s * SR_MULTIPLIER) /
            ↪ sample_time_i2s);
    }
    else i2s_info.current_rate = 0;

    //Find lock status of input/output sample rates
    sample_rate_status_t spdif_status_new = detect_frequency(spdif_info.current_rate >>
        ↪ SR_FRAC_BITS, spdif_info.nominal_rate);
    sample_rate_status_t i2s_status_new = detect_frequency(i2s_info.current_rate >> SR_FRAC_BITS,
        ↪ i2s_info.nominal_rate);

    //If either has changed from invalid to valid, send message to SRC to initialise
    if ((spdif_status_new == VALID && i2s_status_new == VALID) && ((spdif_info.status == INVALID
        ↪ || i2s_info.status == INVALID)) && !skip_validity){
        for(int i = 0; i < ASRC_N_INSTANCES; i++){
            i_fs_ratio[i].new_sr_notify();
        }
        skip_validity = SETTLE_CYCLES; //Don't check on validity for a few cycles as will be
            ↪ corrupted by SR change and SRC init
        fs_ratio = (unsigned) ((spdif_info.nominal_rate * 0x10000000ULL) / i2s_info.nominal_rate);
            ↪ //Initialise rate to nominal
    }

    if (skip_validity) skip_validity--;

    //Update current sample rate status flags for input and output
    spdif_info.status = spdif_status_new;
    i2s_info.status = i2s_status_new;

#define BUFFER_LEVEL_TERM 20000 //How much to apply the buffer level feedback term (effectively 1/I term)
#define OLD_VAL_WEIGHTING 5 //Simple low pass filter. Set proportion of old value to carry over

    //Calculate fs_ratio to tell asrc how many samples to produce in 4.28 fixed point format
    int i2s_buffer_level_from_half = (signed)i2s_buff_level - (i2s_buff_size / 2); //Level w.r.
        ↪ t. half full
    if (spdif_info.status == VALID && i2s_info.status == VALID) {
        fs_ratio_old = fs_ratio; //Save old value
        fs_ratio = (unsigned) ((spdif_info.current_rate * 0x10000000ULL) / i2s_info.current_rate);

        //If buffer is negative, we need to produce more samples so fs_ratio needs to be < 1
        //If positive, we need to back off a bit so fs_ratio needs to be over unity to get more
            ↪ samples from asrc
        fs_ratio = (unsigned) (((BUFFER_LEVEL_TERM + i2s_buffer_level_from_half) * (unsigned long
            ↪ long)fs_ratio) / BUFFER_LEVEL_TERM);
        //debug_printf("sp=%d\ti2s=%d\tbuff=%d\tfs_raw=0x%x\tfs_av=0x%x\n", spdif_info.
            ↪ current_rate, i2s_info.current_rate, i2s_buffer_level_from_half, fs_ratio,

```

```

    ↪ fs_ratio_old);
    //Apply simple low pass filter
    fs_ratio = (unsigned) (((unsigned long long)(fs_ratio_old) * OLD_VAL_WEIGHTING + (unsigned
    ↪ long long)(fs_ratio) ) /
    (1 + OLD_VAL_WEIGHTING));
  }

  //Set Sample rate LEDs
  unsigned spdif_fs_code = samp_rate_to_code(spdif_info.nominal_rate) + 1;
  unsigned i2s_fs_code = samp_rate_to_code(i2s_info.nominal_rate) + 1;
  if (spdif_info.status == INVALID) spdif_fs_code = 0;
  if (i2s_info.status == INVALID) i2s_fs_code = 0;

  for (int i = 0; i < 4; i++){
    if (spdif_fs_code > i) i_leds.set(3, i, 1);
    else i_leds.set(3, i, 0);
    if (i2s_fs_code > i) i_leds.set(0, i, 1);
    else i_leds.set(0, i, 0);
  }

#define THRESH_0    6 //First led comes on when non-zero. Second when > THRESH_0
#define THRESH_1   12 //Third led comes on when > THRESH_1

    //Show buffer level in column 3
    if (i2s_buffer_level_from_half > 0) i_leds.set(2, 2, 1);
    else i_leds.set(2, 2, 0);
    if (i2s_buffer_level_from_half > THRESH_0) i_leds.set(2, 3, 1);
    else i_leds.set(2, 3, 0);
    if (i2s_buffer_level_from_half > THRESH_1) i_leds.set(1, 3, 1);
    else i_leds.set(1, 3, 0);

    if (i2s_buffer_level_from_half < 0) i_leds.set(2, 1, 1);
    else i_leds.set(2, 1, 0);
    if (i2s_buffer_level_from_half < -THRESH_0) i_leds.set(2, 0, 1);
    else i_leds.set(2, 0, 0);
    if (i2s_buffer_level_from_half < -THRESH_1) i_leds.set(1, 0, 1);
    else i_leds.set(1, 0, 0);

    break;

    case t_print when timerafter(t_print_trigger) := int _:
      t_print_trigger += REPORT_PERIOD;
      //Calculate sample rates in Hz for human readability
#if 1
      debug_printf("spdif rate ave=%d, valid=%d, i2s rate=%d, valid=%d, i2s_buff=%d, fs_ratio=0x%x,
        ↪ nom_fs=0x%x\n",
        spdif_info.current_rate >> SR_FRAC_BITS, spdif_info.status,
        i2s_info.current_rate >> SR_FRAC_BITS, i2s_info.status,
        (signed)i2s_buff_level - (i2s_buff_size / 2), fs_ratio, fs_ratio_nominal);
#endif
    break;
  }
}

//Task that drives the multiplexed 4x4 display on the xCORE-200 MC AUDIO board. Very low performance
↪ requirements so can be combined
#define LED_SCAN_TIME 200100 //2ms - How long each column is displayed. Any more than this and you start
↪ to see flicker
[[combinable]]void led_driver(server led_matrix_if i_leds, out port port_leds_col, out port port_leds_row){
  unsigned col_frame_buffer[4] = {0xf, 0xf, 0xf, 0xf}; //4 x 4 bitmap frame buffer scanning from left to
  ↪ right

  unsigned col_idx = 0; //Active low drive hence initialise to 0b1111
  unsigned col_sel = 0x1; //Index for above
  timer t_scan; //Column select 0x1 -> 0x2 -> 0x4 -> 0x8
  int scan_time_trigger;

  t_scan := scan_time_trigger; //Get current time

  while(1){
    select{
      //Scan through 4 columns and output bitmap for each
      case t_scan when timerafter(scan_time_trigger + LED_SCAN_TIME) := scan_time_trigger:

```

```

        port_leds_col <: col_sel;
        port_leds_row <: col_frame_buffer[col_idx];
        col_idx = (col_idx + 1) & 0x3;
        col_sel = col_sel << 1;
        if(col_sel > 0x8) col_sel = 0x1;
        break;

//Sets a pixel at col, row (origin bottom left) to 0 or on
case i_leds.set(unsigned col, unsigned row, unsigned val):
    row = row & 0x3; //Prevent out of bounds access
    col = col & 0x3;
    if (val) { //Need to clear corresponding bit (active low)
        col_frame_buffer[col] &= ~(0x8 >> row);
    }
    else { //Set bit to turn off (active low)
        col_frame_buffer[col] |= (0x8 >> row);
    }
    break;
}
}
}

#define DEBOUNCE_PERIOD      2000000 //20ms
#define DEBOUNCE_SAMPLES    5 //Sample 5 times in this period to ensure we have a true value
#define BUTTON_PRESSED_VAL  0
#define BUTTON_NOT_PRESSED_VAL 1

//Button listener task. Applies a debounce function by checking several times for the same value
[[combinable]]void button_listener(client buttons_if i_buttons, client input_gpio_if i_button_port){
    timer t_debounce;
    int t_debounce_time;
    unsigned debounce_counter = 0; //Counts debounce sequence has started
    int button_released_flag = 0; //Flag showing we have just had a press and are waiting to start again

    i_button_port.event_when_pins_eq(BUTTON_PRESSED_VAL); //setup button event

    while(1){
        select{
            case i_button_port.event(): //The button has reached the expected value
                if (button_released_flag){ //If being released
                    i_button_port.event_when_pins_eq(BUTTON_PRESSED_VAL); //Setup event for being pressed
                    button_released_flag = 0;
                    break;
                }
                debounce_counter = DEBOUNCE_SAMPLES; //Kick off debounce sequence
                t_debounce := t_debounce_time; //Get current time
                break;

            case debounce_counter => t_debounce when timerafter(t_debounce_time + (DEBOUNCE_PERIOD/
                ↪ DEBOUNCE_SAMPLES)) := t_debounce_time:
                unsigned port_val = i_button_port.input();
                if (port_val != BUTTON_PRESSED_VAL) { //Read port n times. If not what we want, start
                    ↪ over
                    debounce_counter = 0;
                    i_button_port.event_when_pins_eq(BUTTON_PRESSED_VAL);
                    break;
                }
                debounce_counter--;
                if (debounce_counter == 0){ //We have seen DEBOUNCE_SAMPLES samples the same,
                    ↪ so button is pressed
                    button_released_flag = 1;
                    i_button_port.event_when_pins_eq(BUTTON_NOT_PRESSED_VAL); //setup event for button
                    ↪ released
                    i_buttons.pressed(); //Send button pressed message
                }
                break;
            }
        }
    }
}

```

D.4 Source code for block_serial.h

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
```

```

#ifndef BLOCK_SERIAL_H_
#define BLOCK_SERIAL_H_

#include <stddef.h>
#include "app_config.h"
#include "main.h"

typedef interface block_transfer_if {
    int * unsafe push(const unsigned n_samps);
} block_transfer_if;

typedef interface serial_transfer_pull_if {
    int pull(const size_t chan_idx);
} serial_transfer_pull_if;

typedef interface serial_transfer_push_if {
    void push(int sample, const size_t chan_idx);
} serial_transfer_push_if;

//Structure containing information about the FIFO
typedef struct b2s_fifo_t{
    int * unsafe ptr_base;
    int * unsafe ptr_top_max;
    int * unsafe ptr_top_curr;
    int * unsafe ptr_rd;
    int * unsafe ptr_wr;
    int size_curr;
    int fill_level;
} b2s_fifo_t;

#define PASS    1
#define FAIL    0

[[distributable]] void serial2block(server serial_transfer_push_if i_serial_in, client block_transfer_if
↳ i_block_transfer[ASRC_N_INSTANCES], server sample_rate_enquiry_if i_input_rate);
[[distributable]] unsafe void block2serial(server block_transfer_if i_block2serial[ASRC_N_INSTANCES], server
↳ serial_transfer_pull_if i_serial_out, server sample_rate_enquiry_if i_output_rate);

#endif /* BLOCK_SERIAL_H_ */

```

D.5 Source code for block_serial.xc

```

// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <string.h>
#include <debug_print.h>
#include <xscope.h>
#include "block_serial.h"

extern out port port_debug_tile_0;

[[distributable]]
#pragma unsafe arrays //Performance optimisation for serial2block. Removes bounds check
void serial2block(server serial_transfer_push_if i_serial_in, client block_transfer_if i_block_transfer[
↳ ASRC_N_INSTANCES], server sample_rate_enquiry_if i_input_rate)
{
    int * unsafe buff_ptr[ASRC_N_INSTANCES]; //Array of buffer pointers. These point to the input double
↳ buffers of ASRC instances

    unsigned samp_count = 0; //Keeps track of samples processed since last query
    unsigned buff_idx = 0; //Index keeping track of input samples per block

    timer t_tick; //100MHz timer for keeping track of sample time
    int t_last_count, t_this_count; //Keeps track of time when querying sample count
    t_tick := t_last_count; //Get time for zero samples counted

    for (int i=0; i<ASRC_N_INSTANCES; i++) {
        buff_ptr[i] = i_block_transfer[i].push(0); //Get initial buffers to write to
    }
}

```

```

int t0, t1; //debug

while(1){
  select{
    //Request to receive one channels of one sample period into double buffer
    case i_serial_in.push(int sample, const unsigned chan_idx):
      if (chan_idx == 0) {
        t_tick := t_this_count;    //Grab timestamp of this sample group
        samp_count++;              //Keep track of samples received
      }

      unsafe{
        *(buff_ptr[chan_idx % ASRC_N_INSTANCES] + buff_idx) = sample;
      }
      if (chan_idx == ASRC_CHANNELS_PER_INSTANCE - 1) buff_idx++; //Move index when all channels
      ↪ received

      if(buff_idx == ASRC_N_IN_SAMPLES){ //When full..
        buff_idx = 0;
        t_tick := t0;
        for(int i=0; i < ASRC_N_INSTANCES; i++) unsafe{//Get new buffer pointers
          buff_ptr[i] = i_block_transfer[i].push(0);
        }
        t_tick := t1;
      }
    break;

    //Request to report number of samples processed since last time
    case i_input_rate.get_sample_count(int &elapsed_time_in_ticks) -> unsigned count:
      elapsed_time_in_ticks = t_this_count - t_last_count; //Set elapsed time in 10ns ticks
      t_last_count = t_this_count;                          //Store for next time around
      count = samp_count;
      samp_count = 0;
    break;

    //Request to report buffer level. Note this is always zero because we do not have a FIFO here
    //This method is more useful for block2serial, which does have a FIFO. This should never be called
    ↪
    case i_input_rate.get_buffer_level() -> {unsigned curr_size, unsigned fill_level}:
      curr_size = 0;
      fill_level = 0;
    break;
  }
}

unsafe{
  //block2serial helper - sets FIFOs pointers to half and clears contents
  static inline void init_fifo(b2s_fifo_t *fifo, int * fifo_base, int size){

    fifo->ptr_base      = fifo_base;
    fifo->ptr_top_max   = fifo_base + size;
    fifo->ptr_top_curr  = fifo_base + size;
    fifo->ptr_rd        = fifo_base + (size >> 1);
    fifo->ptr_wr        = fifo_base;

    memset(fifo_base, 0, size * sizeof(int));
  }

  //block2serial helper - returns size and fill level
  {int, int} static get_fill_level(b2s_fifo_t *fifo){

    int size, fill_level;
    size = fifo->ptr_top_curr - fifo->ptr_base; //Total current size if FIFO
    if (fifo->ptr_wr >= fifo->ptr_rd){
      fill_level = fifo->ptr_wr - fifo->ptr_rd; //Fill level if write pointer
      ↪ ahead of read pointer
    }
    else { //Fill level if read pointer
      ↪ ahead of write pointer
      fill_level = (fifo->ptr_top_curr - fifo->ptr_rd) + (fifo->ptr_wr - fifo->ptr_base);
    }
  }
}

```

```

    }
    return {size, fill_level};
  }

//block2serial helper - checks supplied wr_ptr and modifies if necessary
static inline unsigned confirm_wr_block_address(b2s_fifo_t *fifo){

    if (fifo->ptr_wr >= fifo->ptr_rd){ //rd_ptr behind, need to check
        ↪ for hitting top
        if ((fifo->ptr_wr + ASRC_MAX_BLOCK_SIZE) >= fifo->ptr_top_max) { //cannot fit next block in top
            ↪ , need to wrap
            if((fifo->ptr_base + ASRC_MAX_BLOCK_SIZE) > fifo->ptr_rd) { //No space at bottom either
                return FAIL; //Overflow. We expect to init
                ↪ FIFOs so leave vals as is
            }
            //Space available at bottom
            //Set new top of FIFO at last
            fifo->ptr_top_curr = fifo->ptr_wr;
            ↪ write pointer location
            fifo->size_curr = fifo->ptr_top_curr - fifo->ptr_base; //Shrink size to current top -
            ↪ base
            fifo->ptr_wr = fifo->ptr_base; //Start writing at base
        }
        else { //write space can fit at top
        }
    }

    else { //rd_ptr ahead, need to check
        ↪ for hitting rd_ptr
        if((fifo->ptr_wr + ASRC_MAX_BLOCK_SIZE) >= fifo->ptr_rd) { //We hit rd_ptr
            return FAIL;
        } //We have room
    }
    return PASS;
}

//block2serial helper - pulls a single sample from the FIFO
static unsigned pull_sample_from_fifo(b2s_fifo_t *fifo, int &samp) {

    samp = *fifo->ptr_rd; //read value from fifo
    (fifo->ptr_rd)++; //increment write pointer
    if (fifo->ptr_rd >= fifo->ptr_top_curr) {
        fifo->ptr_rd = fifo->ptr_base; //wrap pointer
    }

    if (fifo->ptr_rd == fifo->ptr_wr){ //We have hit write pointer
        return FAIL; //Underflow - assume will init
        ↪ after this so leave as is
    }
    return PASS;
}
} //unsafe region

//Task that takes blocks of samples from SRC, buffers them in a FIFO and serves them up as a stream
//This task is marked as unsafe keep pointers in scope throughout function
[[distributable]]
#pragma unsafe arrays //Performance optimisation for block2serial. Removes bounds check
unsafe void block2serial(server block_transfer_if i_block2serial[ASRC_N_INSTANCES], server
    ↪ serial_transfer_pull_if i_serial_out, server sample_rate_enquiry_if i_output_rate)
{

    int samps_b2s[ASRC_N_CHANNELS][OUT_FIFO_SIZE]; //FIFO buffer storage

    b2s_fifo_t b2s_fifo[ASRC_N_CHANNELS]; //Declare FIFO control structs

    unsigned samp_count = 0; //Keeps track of number of samples passed through

    timer t_tick; //100MHz timer for keeping track of sample ime
    int t_last_count, t_this_count; //Keeps track of time when querying sample count
    t_tick := t_last_count; //Get time for zero samples counted

    for (unsigned i=0; i<ASRC_N_CHANNELS; i++) { //Initialise FIFOs

```

```

    init_fifo(&b2s_fifo[i], samps_b2s[i], OUT_FIFO_SIZE);
  }
  while(1){
    select{
      //Request to pull one channel of a sample over serial
      case i_serial_out.pull(const unsigned chan_idx) -> int samp:
        if(chan_idx == 0){
          t_tick -> t_this_count;          //Grab timestamp of request for channel 0 only
          samp_count ++;                  //Keep track of number of samples served
        }

        unsigned success = pull_sample_from_fifo(&b2s_fifo[chan_idx], samp);
        if (success == FAIL) { //FIFO empty
          //debug_printf("-");
          for (int i=0; i<ASRC_N_CHANNELS; i++){
            init_fifo(&b2s_fifo[i], samps_b2s[i], OUT_FIFO_SIZE);
          }
        }
        break;

      //Request to push block of samples from SRC
      //selects over the entire array of interfaces
      case i_block2serial[int if_index].push(const unsigned n_samps) -> int * unsafe p_buffer_wr:
        b2s_fifo[if_index].ptr_wr += n_samps;          //Move on write pointer.
        ↪ We already know we have space
        unsigned result = confirm_wr_block_address(&b2s_fifo[if_index]); //Get next available block
        ↪ for write pointer
        if (result == FAIL) { //FIFO full
          //debug_printf("+");
          for (int i=0; i<ASRC_N_CHANNELS; i++){
            init_fifo(&b2s_fifo[i], samps_b2s[i], OUT_FIFO_SIZE);
          }
        }
        p_buffer_wr = b2s_fifo[if_index].ptr_wr;
        break;

      //Request to report number of samples processed since last request
      case i_output_rate.get_sample_count(int &elapsed_time_in_ticks) -> unsigned count:
        elapsed_time_in_ticks = t_this_count - t_last_count; //Set elapsed time in 10ns ticks
        t_last_count = t_this_count; //Store for next time around
        count = samp_count;
        samp_count = 0;
        break;

      //Request to report on the current buffer level
      case i_output_rate.get_buffer_level() -> {unsigned curr_size, unsigned fill_level}:
        //Currently just reports the level of first FIFO. Each FIFO should be the same
        {curr_size, fill_level} = get_fill_level(&b2s_fifo[0]);
        break;
    }
  }
}

```

D.6 Source code for cs4384_5368.h

```

// Copyright (c) 2016, XMOS Ltd, All rights reserved
#ifndef AUDIO_CODEC_H_
#define AUDIO_CODEC_H_
#include <xs1.h>
#include <gpio.h>
#include <i2c.h>
#include <stdint.h>

enum codec_mode_t {
  CODEC_IS_I2S_MASTER,
  CODEC_IS_I2S_SLAVE
};

```

```

/** This interface provides a generic set of functions that control audio
    codec functionality */
typedef interface audio_codec_config_if
{
    /** Reset a codec.
     *
     * This function resets a codec and configures it for a particular
     * sample frequency.
     *
     * \param sample_frequency      The required sample frequency to run at
     *                               after reset (in Hz).
     * \param master_clock_frequency The frequency of the master clock
     *                               supplied to the codec (in Hz).
     */
    void reset(unsigned sample_frequency, unsigned master_clock_frequency);
} audio_codec_config_if;

/** A driver for a the adc/dac pair on the xu216/mc hardware.
 *
 * This driver provides the audio codec configuration interface for
 * the XU216 Multichannel Audio board 2.0
 *
 * \param i                The provided codec configuration interface.
 * \param i_i2c            I2C interface. This should connect to an
 *                          I2C component connected to the I2C bus
 *                          connected to the codec.
 * \param device_addr      The I2C bus address of the codec
 * \param mode              Whether the codec should be in master of
 *                          slave mode.
 */
[[distributable]]
void audio_codec_cs4384_cs5368(server audio_codec_config_if i_codec,
                               client i2c_master_if i2c,
                               enum codec_mode_t codec_mode,
                               client output_gpio_if i_dsd_mode,
                               client output_gpio_if i_dac_rst_n,
                               client output_gpio_if i_adc_rst_n,
                               client output_gpio_if i_mclk_fsel);

#endif /* AUDIO_CODEC_H_ */

```

D.7 Source code for cs4384_5368.xc

```

// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <xassert.h>
#include <timer.h>
#include <debug_print.h>
#include "cs4384_5368.h"

//Address on I2C bus
#define CS4384_I2C_ADDR      (0x18)

//Register Addresses
#define CS4384_CHIP_REV      0x01
#define CS4384_MODE_CTRL    0x02
#define CS4384_PCM_CTRL     0x03
#define CS4384_DSD_CTRL     0x04
#define CS4384_FLT_CTRL     0x05
#define CS4384_INV_CTRL     0x06

```



```

#define CS4384_GRP_CTRL      0x07
#define CS4384_RMP_MUTE     0x08
#define CS4384_MUTE_CTRL   0x09
#define CS4384_MIX_PR1     0x0a
#define CS4384_VOL_A1      0x0b
#define CS4384_VOL_B1      0x0c
#define CS4384_MIX_PR2     0x0d
#define CS4384_VOL_A2      0x0e
#define CS4384_VOL_B2      0x0f
#define CS4384_MIX_PR3     0x10
#define CS4384_VOL_A3      0x11
#define CS4384_VOL_B3      0x12
#define CS4384_MIX_PR4     0x13
#define CS4384_VOL_A4      0x14
#define CS4384_VOL_B4      0x15
#define CS4384_CM_MODE     0x16

//Address on I2C bus
#define CS5368_I2C_ADDR     (0x4C)

//Register Addresses
#define CS5368_CHIP_REV     0x00
#define CS5368_GCTL_MDE     0x01
#define CS5368_OVFL_ST     0x02
#define CS5368_OVFL_MSK     0x03
#define CS5368_HPF_CTRL    0x04
#define CS5368_PWR_DN      0x06
#define CS5368_MUTE_CTRL   0x08
#define CS5368_SDO_EN      0x0a

[[distributable]]
void audio_codec_cs4384_cs5368(server audio_codec_config_if i_codec,
                               client i2c_master_if i2c,
                               enum codec_mode_t codec_mode,
                               client output_gpio_if i_dsd_mode,
                               client output_gpio_if i_dac_rst_n,
                               client output_gpio_if i_adc_rst_n,
                               client output_gpio_if i_mclk_fsel)
{
    while (1) {
        select {
            case i_codec.reset(unsigned sample_frequency, unsigned master_clock_frequency):

                i_dac_rst_n.output(0); //Assert DAC reset
                i_adc_rst_n.output(0); //Assert ADC reset

                delay_milliseconds(10); //Allow reset to establish. Also ensures that invalid SR will be detected
                    ↳ briefly

                if ((sample_frequency % 48000) != 0) i_mclk_fsel.output(0);
                else i_mclk_fsel.output(1);
                /* dsdMode == 0 */
                /* Set MUX to PCM mode (muxes ADC I2S data lines) */
                i_dsd_mode.output(0);

                /* Configure DAC with PCM values. Note 2 writes to mode control to enable/disable freeze/power
                    ↳ down */
                i_dac_rst_n.output(1); //De-assert DAC reset

                delay_microseconds(10); //Allow exit from reset

                /* Mode Control 1 (Address: 0x02) */
                /* bit[7] : Control Port Enable (CPEN) : Set to 1 for enable
                * bit[6] : Freeze controls (FREEZE) : Set to 1 for freeze
                * bit[5] : PCM/DSD Selection (DSD/PCM) : Set to 0 for PCM
                * bit[4:1] : DAC Pair Disable (DACx_DIS) : All Dac Pairs enabled
                * bit[0] : Power Down (PDN) : Powered down
                */
                i2c.write_reg(CS4384_I2C_ADDR, CS4384_MODE_CTRL, 0b11000001);

#ifdef I2S_MODE_TDM
                /* PCM Control (Address: 0x03) */
                /* bit[7:4] : Digital Interface Format (DIF) : 0b1100 for TDM
                * bit[3:2] : Reserved
                * bit[1:0] : Functional Mode (FM) : 0x11 for auto-speed detect (32 to 200kHz)
                */

```

```

    */
    i2c.write_reg(CS4384_I2C_ADDR, CS4384_PCM_CTRL, 0b11000111);
#else
    /* PCM Control (Address: 0x03) */
    /* bit[7:4] : Digital Interface Format (DIF) : 0b0001 for I2S up to 24bit
    * bit[3:2] : Reserved
    * bit[1:0] : Functional Mode (FM) : 0x00 - single-speed mode (4-50kHz)
    *                                     : 0x01 - double-speed mode (50-100kHz)
    *                                     : 0x10 - quad-speed mode (100-200kHz)
    *                                     : 0x11 - auto-speed detect (32 to 200kHz)
    *                                     (note, some Mclk/SR ratios not supported in auto)
    */
    unsigned char regVal = 0;
    if(sample_frequency < 50000)
        regVal = 0b00010100;
    else if(sample_frequency < 100000)
        regVal = 0b00010101;
    else //if(sample_frequency < 200000)
        regVal = 0b00010110;

    i2c.write_reg(CS4384_I2C_ADDR, CS4384_PCM_CTRL, regVal);
#endif

    /* Mode Control 1 (Address: 0x02) */
    /* bit[7] : Control Port Enable (CPEN)           : Set to 1 for enable
    * bit[6] : Freeze controls (FREEZE)            : Set to 0 for freeze
    * bit[5] : PCM/DSD Selection (DSD/PCM)         : Set to 0 for PCM
    * bit[4:1] : DAC Pair Disable (DACx_DIS)       : All Dac Pairs enabled
    * bit[0] : Power Down (PDN)                   : Not powered down
    */
    i2c.write_reg(CS4384_I2C_ADDR, CS4384_MODE_CTRL, 0b10000000);

    /* Take ADC out of reset */
    i_adc_rst_n.output(1);

    delay_microseconds(10); //Allow exit from reset

    {
        unsigned dif = 0, mode = 0;
#ifdef I2S_MODE_TDM
        dif = 0x02; /* TDM */
#else
        dif = 0x01; /* I2S */
#endif

        if(codec_mode == CODEC_IS_I2S_MASTER) {
            /* Note, only the ADC device supports being I2S master.
            * Set ADC as master and run DAC as slave */
            if(sample_frequency < 54000)
                mode = 0x00; /* Single-speed Mode Master */
            else if(sample_frequency < 108000)
                mode = 0x01; /* Double-speed Mode Master */
            else if(sample_frequency < 216000)
                mode = 0x02; /* Quad-speed Mode Master */
        }
        else { //CODEC_IS_I2S_SLAVE - i.e. xCore is master
            mode = 0x03; /* Slave mode all speeds */
        }

        /* Reg 0x01: (GCTL) Global Mode Control Register */
        /* Bit[7]: CP-EN: Manages control-port mode
        * Bit[6]: CLKMODE: Setting puts part in 384x mode
        * Bit[5:4]: MDIV[1:0]: Set to 01 for /2
        * Bit[3:2]: DIF[1:0]: Data Format: 0x01 for I2S, 0x02 for TDM
        * Bit[1:0]: MODE[1:0]: Mode: 0x11 for slave mode
        */
        i2c.write_reg(CS5368_I2C_ADDR, CS5368_GCTL_MDE, 0b10010000 | (dif << 2) | mode);
    }

    /* Reg 0x06: (PDN) Power Down Register */
    /* Bit[7:6]: Reserved
    * Bit[5]: PDN-BG: When set, this bit powers-own the bandgap reference
    * Bit[4]: PDM-OSC: Controls power to internal oscillator core

```

```
* Bit[3:0]: PDN: When any bit is set all clocks going to that channel pair are turned off
*/
i2c.write_reg(CS5368_I2C_ADDR, CS5368_PWR_DN, 0b00000000);

debug_printf("SR change in lib_audio_codec - %d\n", sample_frequency);
break;
} //select
} //while (1)
}
```

