Application Note: AN00201

A startKIT audio effects demo

This application note shows a demo that applies audio effects to a stereo audio stream on the XMOS startKIT. It shows the use of driving audio using the I2S library, performing simple DSP in xC and accessing I/O on the startKIT using the startKIT support library.

The application loops audio input back to audio output with a biquad filter and a modulating gain applied to the signal. The effects are controlled via the button and the sliders on the startKIT. The example also shows the xSCOPE tracing functionality of the xTIMEcomposer tools by sending internal signal values to the development PC via program instrumentation.

Required tools and libraries

The code in this application note is known to work on version 14.0.4 of the xTIMEcomposer tools suite, it may work on other versions.

The application depends on the following libraries:

- lib_logging (>=2.0.0)
- lib_i2c (>=3.0.0)
- lib_i2s (>=2.0.0)
- lib_startkit_support (>=2.0.0)
- lib_gpio (>=1.0.0)

Required hardware

The application note is designed to run on the XMOS startKIT with the XMOS audio slice card (XA-SK-AUDIO) connnected to it.

Prerequisites

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

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

1 Overview

1.1 Introduction

startKIT is a low-cost development board for the configurable xCORE multicore microcontroller products from XMOS. It's easy to use and provides lots of advanced features on a small, extremely low cost platform.

xCORE lets you software-configure the interfaces that you need for your system; so with startKIT you can configure the board to match your exact requirements. Its 500MIPS xCORE multicore microcontroller has eight 32bit logical cores that perform deterministically, making startKIT an ideal platform for functions ranging from robotics and motion control to networking and digital audio.

startKIT also has a *slice* connector which allows users to connect to XMOS slice cards to extend the I/O with other capabilities.

1.2 Block diagram

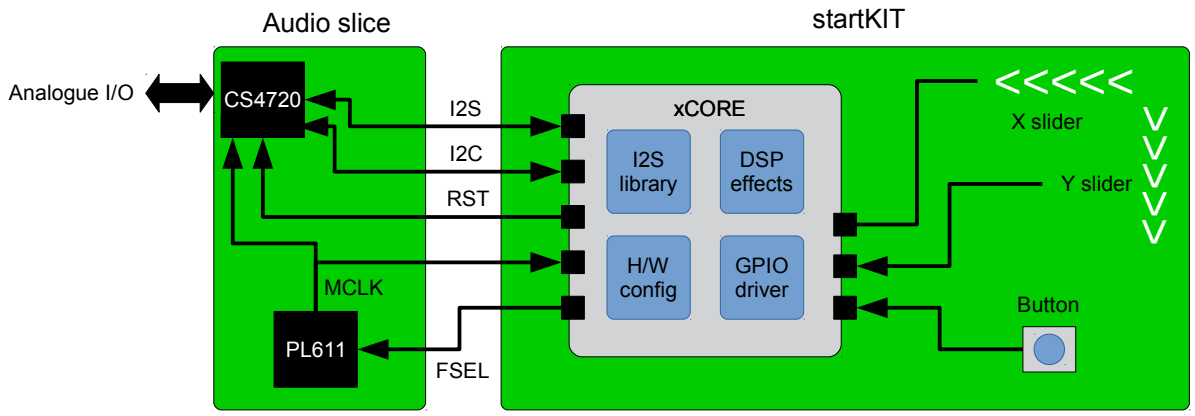

Figure 1: Block diagram of the startKIT audio effects demo

2 A startKIT audio effects demo

The demo loops back an analogue stereo audio input to a analogue stereo output. The xCORE loopbacks digital audio via an I2S bus which is connected to a Cirrus CS4270 audio codec.

The audio has biquad filters (which affect the frequency spectrum of the output) and also have a varying gain applied to it (making the audio signal "vibrate").

The startKIT sliders control modulation depth and frequency of the amplitude modulation wave and the button toggles between having no biquad effect applied or different biquad effects. Each time a new effect is enabled via the button a different filter is used. The startKIT LED grid shows where you are in the cycle. The filter cycles through the following pre-defined types: low pass, high pass, band pass, notch (band stop) and all pass.

The demo consists of several tasks running in parallel. Specifically:

- The I2S master task from the XMOS I2S library handles the I2S digital audio interface.
- The I2S master task will make callbacks to the I2S handler task which will initialize the audio codec on the audio slice by communicating with the I2C master task (from the XMOS I2C library) and a GPIO task which connects via a multi-bit xCORE port to the reset and clock frequency select lines on the audio slice.
- The other job of the I2S handler tasks is to pass samples from the I2S bus to the audio effects tasks over an xC communication channel.
- The audio effects task applies the audio effects to the samples before passing them back to the I2S handler task. It also connects to the a task driving the startKIT GPIO (i.e. the buttons and sliders).
- Finally, the wave generation task generates a low frequency triangle wave and passes these values to the audio effects task to use.

The following diagram shows the task and communication structure for the application.

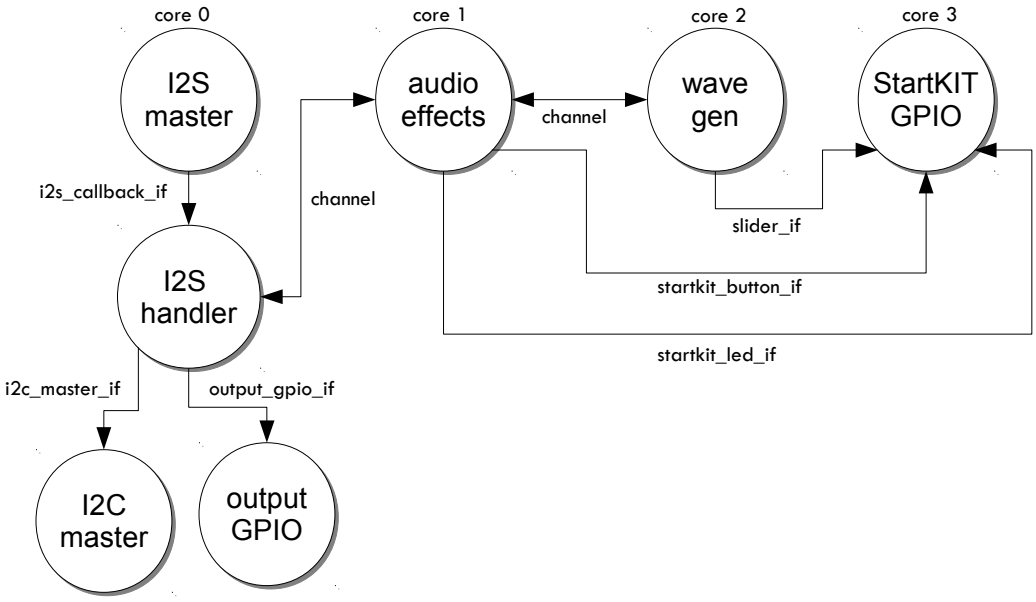

Figure 2: Task diagram of the startKIT audio effects demo

Note that the application consists of 7 tasks over 4 logical cores. The I2S master, audio effects, wavegen and startKIT GPIO tasks consume a logical core each. The I2S handler, I2C master and GPIO tasks do not - these tasks will be scheduled as required when there are communicated with by the other tasks.

2.1 Makefile additions for this example

This example uses many XMOS libraries. These can all be included in the project via the USED_MODULES variable in the Makefile:

```
USED_MODULES = lib_gpio lib_i2c lib_i2s lib_logging lib_startkit_support
```

The compiler flags include -g (to enable debug information), -report (to give a resource usage report after compilation) and -DDEBUG_PRINT_ENABLE=1 which is a define to enable the debug printing provided by the debug_printf function from lib_logging:

```
XCC_FLAGS = -O2 -g -report -DDEBUG_PRINT_ENABLE=1 -save-temps
```

2.2 Declaring resources used by the application

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

```
/* Ports and clocks used by the application */
startkit_gpio_ports gpio_ports =
  on tile[0] : {XS1_PORT_32A, XS1_PORT_4A, XS1_PORT_4B, XS1_CLKBLK_3};

out buffered port:32 p_dout[1] = on tile[0]: {XS1_PORT_1G};

in buffered port:32 p_din[1]   = on tile[0]: {XS1_PORT_1K};

in port p_mclk                 = on tile[0]: XS1_PORT_1E;
out buffered port:32 p_bclk    = on tile[0]: XS1_PORT_1F;
```

These ports are mapped to external pins on the device which then are connected to the I/O on the startKIT and attached slice card. For other boards these ports could be changed.

2.3 Application defines

The main.xc file also contains some defines that are used within that file to configure the application for the sample frequency to run at, the master clock frequency to use and the I2C address of the codec.

```
#define SAMPLE_FREQUENCY 48000
#define MASTER_CLOCK_FREQUENCY 24576000
#define CODEC_I2C_DEVICE_ADDR 0x48
```

These are only used within main.xc to configure the hardware and libraries.

2.4 The application main() function

Below is the source code for the main function of this application, which is taken from the source file `main.xc`. This function sets up all the tasks running in parallel that make up the application:

```
int main(void)
{
  streaming chan c_aud_dsp;
  chan c_gain;
  startkit_led_if i_led;
  startkit_button_if i_button;
  slider_if i_slider_x, i_slider_y;
  i2s_callback_if i_i2s;
  i2c_master_if i_i2c[1];
  output_gpio_if i_gpio[2];
  par {
    /* These four tasks are library functions that drive the hardware
       peripherals */
    on tile[0]: i2c_master_single_port(i_i2c, 1, p_i2c, 10, 0, 1, 0);
    on tile[0]: output_gpio(i_gpio, 2, p_gpio, gpio_pin_map);
    on tile[0]: startkit_gpio_driver(i_led, i_button,
                                     i_slider_x,
                                     i_slider_y,
                                     gpio_ports);
    on tile[0]: {  configure_clock_src(mclk, p_mclk);
                   start_clock(mclk);
                   i2s_master(i_i2s, p_dout, 1, p_din, 1,
                              p_bclk, p_lrclk, bclk, mclk);
                }

    /* These three tasks are defined in this application */
    on tile[0]: i2s_handler(i_i2s, i_i2c[0], i_gpio[0], i_gpio[1],
                            c_aud_dsp);
    on tile[0]: audio_effects(c_aud_dsp, i_led, i_button, c_gain, 2);
    on tile[0]: wavegen(c_gain, i_slider_x, i_slider_y);


  }
  return 0;
}
```

Looking at this in a more detail you can see the following:

- The par functionality describes running seven separate tasks in parallel.
- The declarations before the par create the connections between the tasks (see Figure 2). Tasks are connected by passing one of these declared variables to both tasks. Some of the connections are arrays which can connect one task to many others.
- The first four tasks are library functions that drive the hardware peripherals: the I2C bus, some general GPIO, the startKIT GPIO (LEDs, buttons and sliders) and the I2S bus. Before calling the `i2s_master` function a small section of code is executed to configure the I2S clock. The tasks take the ports declared at the top of the file as arguments. This is how the software drivers link to the hardware.
- The final three tasks are the ones defined in the application to make up the demo. The `i2s_handler` task handles the I2S bus and configures the audio hardware, it passes samples to the `audio_effects` task to implement the DSP transforms. The wavegen task calculates a triangle amplitude modulation wave and passes the values from this wave to the `audio_effects` task.

2.5 The I2S handler task

The i2s_master task is connected to the i2s_handler task (which is defined in the application). The connection between the tasks will make 'callbacks' from the i2s_master task to the application. The prototype of the I2S handling task is as below:

```
[[distributable]]
void i2s_handler(server i2s_callback_if i2s,
                 client i2c_master_if i2c,
                 client output_gpio_if clock_select,
                 client output_gpio_if codec_reset,
                 streaming chanend c_dsp)
```

Note that:

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

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

```
int32_t in_samps[2] = {0};
int32_t out_samps[2] = {0};
while (1) {
  select {
```

The calls it will respond to are defined in the i2s_master_callback_if in i2s.h. There are four callbacks: initialization, sending a sample, receiving a sample and checking for restart.

2.5.1 Configuring the audio hardware

The `init` callback occurs when the I2S bus initializes. At this point the task will configure the audio hardware.

```
case i2s.init(i2s_config_t &?i2s_config, tdm_config_t &?tdm_config):
  /* Set CODEC in reset */
  codec_reset.output(0);

  /* Set master clock select appropriately */
  if ((SAMPLE_FREQUENCY % 22050) == 0) {
    clock_select.output(0);
  }else {
    clock_select.output(1);
  }

  /* Hold in reset for 2ms while waiting for MCLK to stabilise */
  delay_milliseconds(2);

  /* CODEC out of reset */
  codec_reset.output(1);

  cs4270_configure(i2c, CODEC_I2C_DEVICE_ADDR,
                   SAMPLE_FREQUENCY, MASTER_CLOCK_FREQUENCY,
                   CODEC_IS_I2S_SLAVE);

  /* Configure the I2S bus */
  i2s_config.mode = I2S_MODE_I2S;
  i2s_config.mclk_bclk_ratio = (MASTER_CLOCK_FREQUENCY/SAMPLE_FREQUENCY)/64;
  break;
```

This section of code makes calls on the GPIO and I2C interfaces to configure the clock selection and codecs on the board. The supplementary `cs4270.h` and `cs4270.xc` files contain the `cs4270_configure` helper function to configure the CS4270 audio codec.

At the end it sets the mode and clock ratio fields of the `i2s_config` structure provided by the I2S task (see the I2S library documentation for more details).

2.5.2 Communicating audio samples to/from the audio effects task

The send and receive callbacks from I2S will pass samples to and from the audio effects task. Arrays are used to store the incoming and outgoing samples. When the sample number 0 is requested the task does a channel exchange with the audio effects task to fill/send the arrays:

```
case i2s.receive(size_t index, int32_t sample):
  if (index == 0) {
    for (size_t i = 0; i < 2; i++) {
      c_dsp <: in_samps[i];
      c_dsp :> out_samps[i];
    }
  }
  in_samps[index] = sample;
  break;

case i2s.send(size_t index) -> int32_t sample:
  sample = out_samps[index];
  break; // end of select
```

2.6 The audio effects task

The audio effects task follows the flow diagram show in Figure 3.

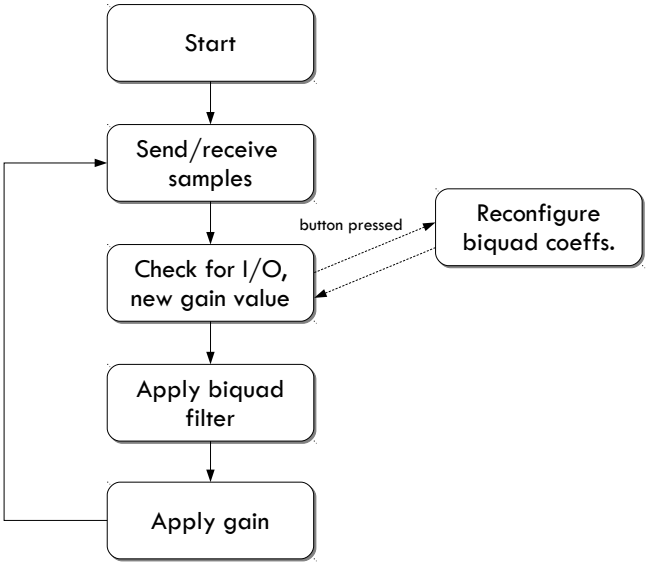

Figure 3: Flow diagram for the audio effects task.

The function that implements the task is found in `audio_effects.xc` and its prototype is:

```
void audio_effects(streaming chanend c_dsp,
                   client startkit_led_if i_led,
                   client startkit_button_if i_button,
                   chanend c_gain,
                   static const size_t num_chans)
```

It takes connections to the I2S handler task, the wavegen task and the startKIT GPIO task to send/receive samples, obtain gain level updates and drive I/O respectively. It also takes the number of audio channels to process as an argument. The num_chans argument is marked as `static` which is an xC extension that allows declaring local arrays of this size.

The task has state to hold the current incoming and outgoing samples, the current gain level, the state of the biquad filters and a value that represents the current biquad effect type:

```
// Unprocessed input audio sample buffer
int32_t in_samps[num_chans] = {0};

// Sample buffer to hold samples after processing
int32_t out_samps[num_chans] = {0};

int32_t gain = 0; // Start with volume at 0

biquad_state_t biquad_state[num_chans];

int cur_effect = 0; // This value is either 0 (meaning no effect should
                    // be applied) or a value representing one more than
                    // an effect type value as decribed by the enum in
                    // biquad.h
```

The implementation of the flow diagram in Figure 3 is a `while (1)` loop. It starts by sending and receiving samples over a channel connected to the I2S handler task:

```
while(1) {
  // Send/Receive samples
  for (size_t i = 0; i < num_chans; i++) {
    c_dsp :> in_samps[i];
    c_dsp <: out_samps[i];
  }
```

The next step in the loop is to check for I/O events or a new incoming value from the wavegen task. This is done via a `select` statement. It waits for a button change event from the startKIT GPIO task or a new incoming gain value over the channel connected to the wavegen task:

```
select{
case i_button.changed():
  ...
  break;
case c_gain :> gain:
  break;
default:
  break;
}
```

The `default` case means that the select will not wait if no events are ready i.e. control flow will continue.

The code within the case for a button press reconfigures the biquad filters to a new effect:

```
case i_button.changed():
  if (i_button.get_value() == BUTTON_DOWN) {
    cur_effect++;
    if (cur_effect == NUM_BIQUAD_TYPES + 1)
      cur_effect = 0;

    i_led.set_multiple(0b111111111, LED_OFF);
    if (cur_effect == 0) {
      debug_printf("Biquad effect off\n");
      break;
    }
    enum biquad_type_t btype = cur_effect - 1;

    debug_printf("Effect: %s: Freq 1000Hz\n", filt_names[btype]);
    for (size_t i = 0; i < num_chans; i++) {
      init_biquad_state(btype, 1000, 48000, BIQUAD_Q_BUTTERWORTH,
                        biquad_state[i]);
    }
    i_led.set_multiple(1 << btype, LED_ON);
  }
  break; // end of button handling case
```

Here we can see that if the button has been pressed then the code updates the LEDs and updates the `cur_effect` state variable. If there is an effect to be applied it will call the `init_biquad_state` function described in the next section.

The next step applies biquad filters to all channels (providing the current effect is not set to 'no effect'):

```
// Do DSP processing ...
for (size_t i = 0; i < num_chans; i++) {
  if (cur_effect != 0)
    out_samps[i] = apply_biquad(in_samps[i], biquad_state[i]);
  else
    out_samps[i] = in_samps[i];
 }
```

The apply_biquad function is defined in biquad.xc and described in the next section.

The final part of the loop applies the variable gain to the samples:

```
// Apply gain
for (size_t i = 0; i < num_chans; i++)  {
  out_samps[i] = apply_gain(out_samps[i], gain);
}
```

Where the apply_gain function performs a fixed point multiplication:

```
int apply_gain(int sample, int gain) {
  return (((int64_t) sample) * (int64_t) gain) >> 31;
}
```

Note that the calculation casts to 64-bit integers to prevent overflow.

2.7 Calculating biquad filters

The biquad filter code is in `biquad.xc` and `biquad.h`. It provides simple yet effective code for perfoming a digital biquad filter. It is just one example of the DSP that the xCORE is capable of. More examples can be found in the open source repository:

http://github.com/xcore/sc_dsp_filters

The key functions in `biquad.xc` are `apply_biquad` which performs the filter:

```
int32_t apply_biquad(int32_t val, biquad_state_t &st);
```

and `init_biquad_state` which initializes the biquad for use:

```
void init_biquad_state(enum biquad_type_t type,
                       unsigned significant_freq,
                       unsigned sample_freq,
                       uint32_t Q,
                       biquad_state_t &st);
```

Both these functions take a pass-by-reference `biquad_state_t` structure which holds the state of the filter for a channel. This contains the sample history required for the filter and its coefficients:

```
struct biquad_state_t {
  int32_t b[3];
  int32_t a[2];
  int32_t prev_inputs[2];
  int32_t prev_outputs[2];
  int64_t error;
};
```

All the coefficients and parameters of the filter are represented as fixed point integers with 28 bits after the point.

The init function takes a type of filter, this is declared in the following enum:

```
enum biquad_type_t {
  BIQUAD_LOPASS = 0,
  BIQUAD_HIPASS,
  BIQUAD_BANDPASS,
  BIQUAD_NOTCH,
  BIQUAD_ALLPASS,

  NUM_BIQUAD_TYPES
};
```

We can see that the biquad can be a lo-pass filter, hi-pass filter, band-pass filter, notch (band-stop) filter or an all-pass (phase change) filter.

The apply_biquad function performs the filter calculation, it calculates the IIR filter using the 'Direct Form 1' calculation.

```
int32_t apply_biquad(int32_t val, biquad_state_t &st)
{
  int64_t res;

  res = ((int64_t) st.b[0]) * val +
        ((int64_t) st.b[1]) * st.prev_inputs[0] +
        ((int64_t) st.b[2]) * st.prev_inputs[1] -
        ((int64_t) st.a[0]) * st.prev_outputs[0] -
        ((int64_t) st.a[1]) * st.prev_outputs[1] +
        st.error;

  int64_t scaled_res = res >> FBITS;

  if (scaled_res > INT32_MAX)
    scaled_res = INT32_MAX;
  if (scaled_res < INT32_MIN)
    scaled_res = INT32_MIN;

  st.error = res - (scaled_res << FBITS);
  st.prev_inputs[1] = st.prev_inputs[0];
  st.prev_inputs[0] = val;
  st.prev_outputs[1] = st.prev_outputs[0];
  st.prev_outputs[0] = (int32_t) scaled_res;

  return (int32_t) scaled_res;
}
```

Here FBITS is a define for the number of fractional bits used in the coefficients (i.e. 28). The arithmetic is done using 64-bits to avoid overflow. This compiles down to the 64-bit arithmetic instructions on the xCORE.

The `init_biquad_state` will setup the coefficients of the biquad according to the filter type required. This is also done in fixed point and requires trigonometric functions. Fixed point versions of sine and cos are provided in `biquad.xc`. A good resource on calculating biquad coefficents can be found here:

http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt

2.8 Generating an triangle amplitude modulation wave

The wavegen task creates a traingle wave for amplitude modulation and every millsecond it passes the values from this wave to the `audio_effects` task.

The function implementing the tasks takes arguments consisting of a chanend connected to the audio effects task and interface connections to the startKIT GPIO task to access the sliders.

```
void wavegen(chanend c,
             client slider_if i_slider_x, client slider_if i_slider_y)
```

The state of the tasks consists of the current gain, the direction the gain is moving in, the current depth (amplitude) of the wave and the current step (i.e. the amount the current gain will move each time step):

```
int gain = MAX_DEPTH, dir = 1, time;
int hertz = 30;
int depth = 0; //0 = no effect, MAX_DEPTH = max depth
int step = (MAX_DEPTH/((TICKS_PER_SECOND/TICKS_PER_MILLISECOND) / (2 * hertz)));
```

The main loop of the task is a 'while(1)-select' construct where the select will react to a timer event (the 1 millisecond time step) or a slider event from the startKIT GPIO task:

```
while(1) {
  select {
     case tmr when timerafter(time) :> time:
        ...
        break;
     case i_slider_y.changed_state():
        ...
        break;
     case i_slider_x.changed_state():
        ...
        break;
   }
}
```

When the timer event fires, the new gain is calculated and sent over the channel to the audio effects task:

```
case tmr when timerafter(time) :> time: //Time for a new sample to be generated
  if(dir==1) gain += step;            //Rising phase of triangle
  else gain -= step;                  //Falling phase of triangle
  if (gain >= (MAX_DEPTH-step)) dir = -1;
  if (gain <= step) dir = 1;

  c <: MAX_DEPTH - (int) (((long long)depth * (long long)gain) >> 31);
  time += TICKS_PER_MILLISECOND;
  break;
```

The slider events let the user change the depth and frequency of the wave. The following case handles the depth adjustment:

```
case i_slider_x.changed_state():
  sliderstate state = i_slider_x.get_slider_state();
  if (state == RIGHTING) {
    depth -= MAX_DEPTH/5;
    if (depth < 0)
      depth = 0;
    debug_printf("Amplitude modulation depth decreased to 0x%x\n", depth);
  }
  if (state == LEFTING) {
    if (depth > MAX_DEPTH - MAX_DEPTH/5)
      depth = MAX_DEPTH;
    else
      depth += MAX_DEPTH/5;
    debug_printf("Amplitude modulation depth increased to 0x%x\n", depth);
  }
  break;
```

Details on the API to query the slider state can be found in the startKIT support library user guide.

2.9 xSCOPE instrumentation

The application also provides xSCOPE instrumentation for various values. The `config.xscope` file in the application declares the values that should be traced.

```
<xSCOPEconfig ioMode="basic" enabled="true">

   <Probe name="Left in" type="CONTINUOUS" datatype="INT" units="Value" enabled="true"/>
   <Probe name="Left out" type="CONTINUOUS" datatype="INT" units="Value" enabled="true"/>
   <Probe name="Gain" type="CONTINUOUS" datatype="INT" units="Value" enabled="true"/>

</xSCOPEconfig>
```

This declares three probes:

- The 'Left In' probe - the sample values on the left channel before effects are applied.
- The 'Left Out' probe - the sample values on the left channel after effects are applied.
- The 'Gain' probe - the current gain value of the amplitude modulation wave.

In `audio_effects.xc` the `xscope.h` header is included:

```
#include <xscope.h>
```

This lets the code access functions to output trace values. In the main loop, the three integers values are output each iteration:

```
xscope_int(LEFT_IN, in_samps[0]);
xscope_int(LEFT_OUT, out_samps[0]);
xscope_int(GAIN, gain);
```

These values can then be seen in the Real-Time xSCOPE perspective in the xTIMEcomposer (see §B for details).

APPENDIX A - Demo Hardware Setup

To run the demo, connect the startKIT to the audio slice card and then connected the startKIT micro-USB connector to your development PC via a micro-USB cable.

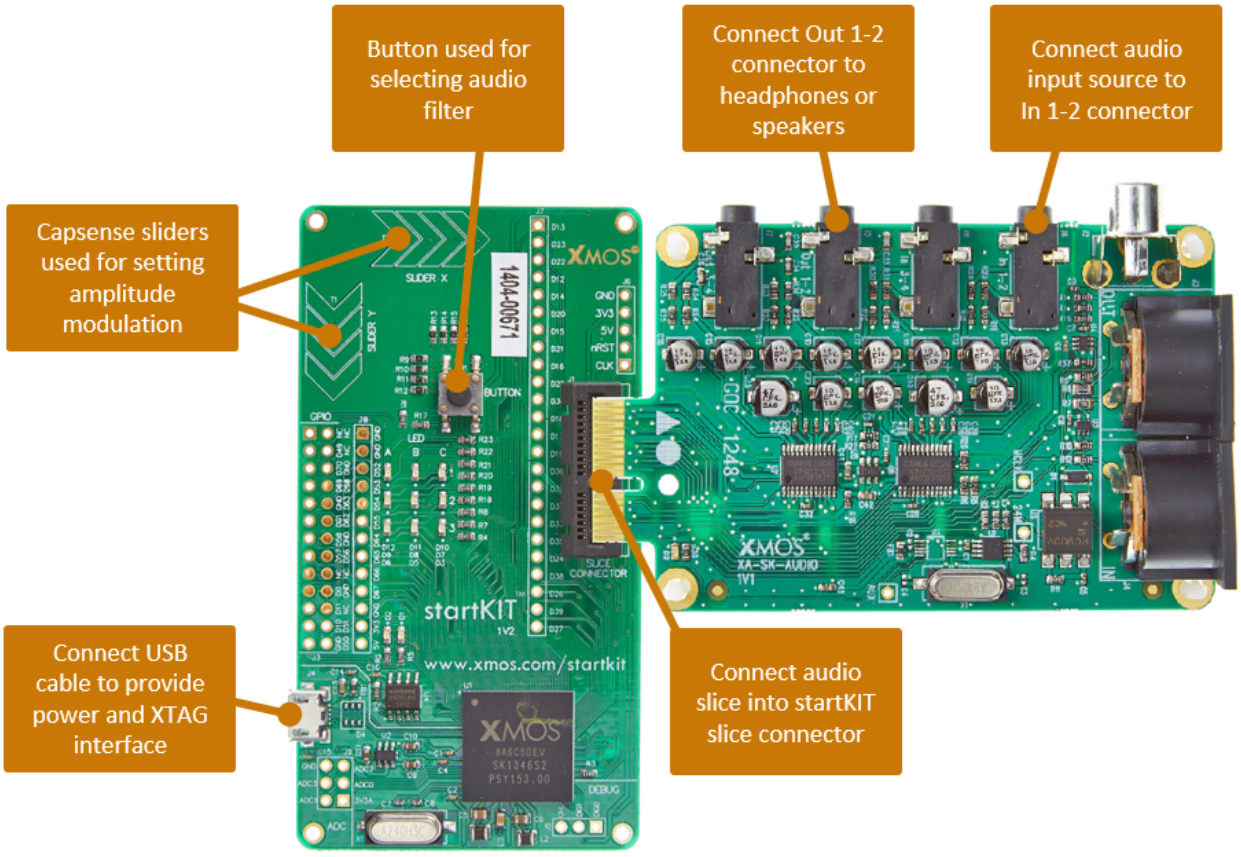

Figure 4: XMOS startKIT with audio slice

The demo will loopback the first analogue input of the slice CARD to the first analogue output.

APPENDIX B - Launching the demo application

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio we can execute the application on the startKIT.

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

B.1 Launching from the command line

From the command line we use the `xrun` tool to download code to both the xCORE devices. If we change into the bin directory of the project we can execute the code on the xCORE microcontroller as follows:

```
> xrun --xscope AN00201_startKIT_audio_effects_demo.xe      <-- Download and execute the xCORE code
```

Once this command has executed the demo should run and audio will loopback on the audio slice. Debug output will be displayed on the console and you can use the buttons and slider to control the effects.

B.2 Launching from xTIMEcomposer Studio

To run the application binary, first the application must be built by pressing the **Build** button in the xTIMEcomposer. This will create the `AN00201_startKIT_audio_effects_demo.xe` binary in the `bin` folder of the project. The xTIMEcomposer may have to import the some libraries if you do not already have it in your workspace; this will occur automatically on build.

Then a *Run Configuration* 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**.

Within the new run configuration, you should selec the **Run on Hardware** option and select the startKIT target from the target list. Within the **XScope** tab the **Real-Time Mode** should be selected.

By clicking on the **Run** icon (a green arrow) in the Edit Perspective of the xTIMEcomposer or by clicking the **Run** button in the run configuration dialog, the program will run. It will automatically take you to the **Trace** perspective in Real-Time xSCOPE mode where you will see the traced values in a scope-like view. The console window will show debug output and the audio slice will loopback its first input to its first output.

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 Software Libraries

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