XMOS

AN02050: Extend I2S loopback application with DSP and AI

Publication Date: 2025/10/13
Document Number: XM-015408-AN v1.0.0

IN THIS DOCUMENT

xCORE devices make it easy to add functionality to an existing design. This application note provides a simple example of how to do so by adding two extensions to an existing design:

- Audio DSP functionality, and
- AI-based keyword spotting.

Building up functionality in this way allows the developer to implement and test a design in an incremental manner.

AN00162 describes how to implement a basic I^2S loopback application on an XCORE device. This I^2S loopback application acts as the starting point for this application note.

1 Application overview

The software included in this application note has both the DSP extension and the AI extension integrated into a single application. The use of pre-processor macros allows for easy inclusion or exclusion of each extension.

Each extension has been implemented in C. As in AN00162, the application and I^2S component have been implemented in XC.

Without either extension, the system consists of three threads communicating over two sets of ports as shown in Fig. 1.

2 Common aspects of both extensions

2.1 Common processing pattern

Each extension adds threads of processing to the code running on the xCORE processor. The common pattern for each thread consists of some code that performs initialisation followed by an event-driven processing loop.

The processing loop waits on one or more resources to detect an event. When an event occurs, the loop runs a corresponding event handler. Once the event handler completes, the processing loop waits for the next event. The loop continues in this manner until all

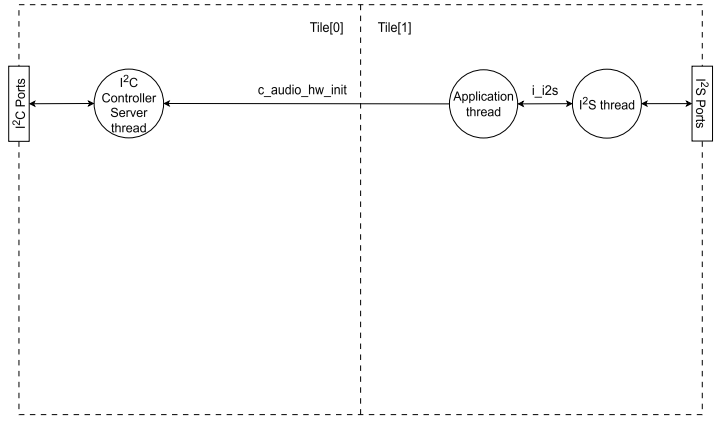

Fig. 1: I^{2}S loopback thread diagram

processing stops, for instance upon removal of power. No mechanism exists to exit a processing loop based on detection of an event.

2.2 Header file declarations

To allow correct linkage regardless of the language of the source file that includes them, each header file includes the `xccompat.h` header file and some conditionally included lines to establish external C-linkage when needed. The `xccompat.h` file defines typedefs for xCORE resources when writing source code in C or C++.

3 Adding DSP

The DSP extension adds a simple volume control capability to the application. A single button controls the volume level and mute/unmute state. Pressing the button cycles through a series of volume levels, from the maximum volume down to mute and then back up to maximum volume.

With the DSP extension, the system consists of five threads communicating over three sets of ports as shown in Fig. 2.

The DSP functionality appears in the `some_dsp.c` source file. Button handling appears in the `some_gpi.c` source file. The corresponding header files `some_dsp.h` and `some_gpi.h` declare the public interface to each module.

3.1 The DSP header file

The `some_dsp()` function provides an entry point to start the DSP functionality. It declares two channel end parameters, one for handling audio data exchanged with the application and one for receiving button-press indications from the button handler.

```
void some_dsp(chanend c_dsp, chanend c_gpi);
```

The `main()` function has the responsibility to provide the correct channel ends when starting the DSP thread.

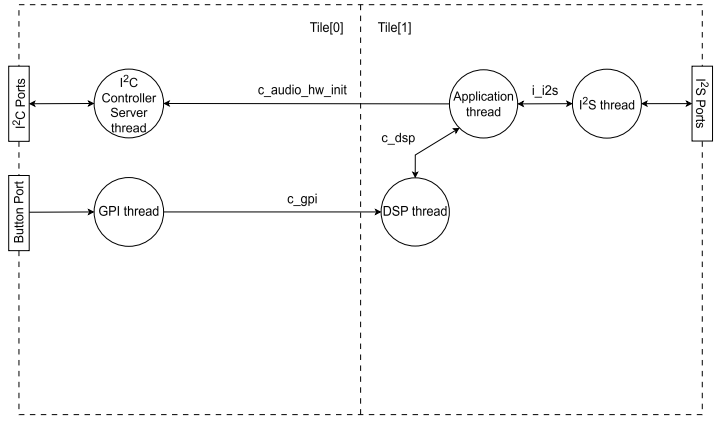

Fig. 2: DSP extension thread diagram

3.2 The DSP source file

The `some_dsp.c` source file implements the DSP functionality. It defines several constants, a few data structures, and a single function, `some_dsp()`, organised as a separate thread of processing.

Initialisation includes setting the initial volume level, un-muting the audio, and setting the value of the variable that controls the direction of volume changes.

```
    // Set initial volume control gain and make sure it's un-muted
    int volume_idx = 0; // Start at -6dB
    adsp_volume_control_set_gain(&volume_control, VOLUME_LEVEL[volume_idx]);
    adsp_volume_control_unmute(&volume_control);
    bool direction = DECREASE;
```

It also initialises a buffer and index for storing audio samples. Finally, it defines a variable to hold the button-press data it will receive from the button handler.

```
    // Initialize the audio sample buffer
    int idx = 0;
    int32_t sample[BUFFER_SIZE];

    // Create a place to sink the button-pressed data
    bool button_pressed;
```

The thread's event-driven processing loop uses the `SELECT_RES` macro. This macro expands into an infinite loop which allows individual handlers to respond to events on different resources.

```
    SELECT_RES(
        CASE_THEN(c_dsp, event_dsp_chanend),
        CASE_THEN(c_gpi, event_button_pressed)
    ) // SELECT_RES
```

Each `CASE` statement associates a resource with an event handler. In the first `CASE`, the resource is the channel end `c_dsp`, and any event on that channel end invokes the `event_dsp_chanend` handler. In the second `CASE`, the resource is the channel end `c_gpi`, and any event on that channel end invokes the `event_button_pressed` handler.

The loop responds to two events: audio data arriving on the `c_dsp` channel end or button-press data arriving on the `c_gpi` channel end. The first event handler reads audio samples from the `c_dsp` channel end and stores them in the `sample` buffer. When the buffer is full, it applies the current volume level to each sample and returns it through the same channel end.

```
        event_dsp_chanend:
            sample[idx] = chan_in_word(c_dsp);

            if(BUFFER_SIZE <= ++idx) {
                for(size_t i = 0; i < BUFFER_SIZE; ++i) {
                    int32_t result = adsp_volume_control(&volume_control, sample[i]);
                    chan_out_word(c_dsp, result);
                }
                idx = 0;
            }

            continue; // end of event_dsp_chanend
```

The second event handler reads button-press data arriving on the `c_gpi` channel end and stores it in the `button_pressed` variable. The button handler has been designed to only send data when a button has been pressed. Consequently, storing the data in the `button_pressed` variable only serves to remove it from the channel so that subsequent button-presses can be received. Having cleared the channel, the event handler changes the volume level and mute/unmute state based on values in the `DECREASE_VOLUME`, `INCREASE_VOLUME`, and `VOLUME_LEVEL` arrays.

```
#define VOLUME_LEVEL_COUNT (4)
// Arrange violume levels in decreasing order of gain to allow an index equal to VOLUME_LEVEL_COUNT to signify a
↪muted state.
static const int32_t VOLUME_LEVEL[VOLUME_LEVEL_COUNT] = {
    INT_MAX>>2,  // -6dB
    INT_MAX>>3,  // -9dB
    INT_MAX>>4,  // -12dB
    INT_MAX>>5,  // -15dB
};

#define DECREASE (false)
#define INCREASE (true)

typedef struct {
    size_t next_index;
    bool next_direction; // DECREASE or INCREASE
} volume_change_t;

static const volume_change_t DECREASE_VOLUME[VOLUME_LEVEL_COUNT] = {
    {1, DECREASE},  // Move to -9dB, continue decreasing
    {2, DECREASE},  // Move to -12dB, continue decreasing
    {3, DECREASE},  // Move to -15dB, continue decreasing
    {4, INCREASE}   // Mute, start increasing
}; // DECREASE_VOLUME

static const volume_change_t INCREASE_VOLUME[VOLUME_LEVEL_COUNT+1] = {
    {0, DECREASE},   // Should not happen, start decreasing
    {0, DECREASE},   // Move to -6dB, start decreasing
    {1, INCREASE},   // Move to -9dB, continue increasing
    {2, INCREASE},   // Move to -12dB, continue increasing
    {3, INCREASE}    // Unmute, move to -15dB, continue increasing
}; // INCREASE_VOLUME
```

```
        event_button_pressed:
            button_pressed = chan_in_word(c_gpi);

            if(direction == DECREASE) {
                direction  = DECREASE_VOLUME[volume_idx].next_direction;
                volume_idx = DECREASE_VOLUME[volume_idx].next_index;
                if(VOLUME_LEVEL_COUNT <= volume_idx) {
                    adsp_volume_control_mute(&volume_control);
                } else {
                    adsp_volume_control_set_gain(&volume_control, VOLUME_LEVEL[volume_idx]);
                }
            } else {
                direction  = INCREASE_VOLUME[volume_idx].next_direction;
                volume_idx = INCREASE_VOLUME[volume_idx].next_index;
                adsp_volume_control_set_gain(&volume_control, VOLUME_LEVEL[volume_idx]);
                if(volume_idx == VOLUME_LEVEL_COUNT-1) {
                    adsp_volume_control_unmute(&volume_control);
                }
            }

            continue; // end of event_button_pressed
```

3.3 The button handler header file

The `some_gpi()` function provides an entry point to start the button handler. It accepts a channel end which the button logic will use to send button-press data to the DSP module. It also accepts a port argument.

```
#if defined(__XC__)
void some_gpi(chanend c_gpi, in port p_buttons);
#else
void some_gpi(chanend c_gpi, port p_buttons);
#endif // defined(__XC__)
```

> **Note**
>
> The XC language requires inclusion of a port's direction when declaring or defining it. Neither C nor C++ recognise the direction keyword when declaring or defining a port. Hence `some_gpi.h` declares `some_gpi()` twice with slightly different signatures, once for use within XC files and again for use within C or C++ files.

The `main()` function supplies the correct channel end and the actual port connected to the buttons when starting the button handler thread.

3.4 The button handler source file

The `some_gpi.c` source file implements the button handler functionality. It defines some constants useful for isolating data from button 0 and a single function, `some_gpi()`, organised as a separate thread of processing.

Initialisation prepares the button port for input and configures a timer to handle bouncing of the button signal. The `port_set_trigger_in_equal()` function configures the port to generate an event when someone presses button 0 and button 1 is not pressed.

```
    // Prepare the button port for input
    port_enable(p_buttons);
    int32_t button_state = port_in(p_buttons);
    port_set_trigger_in_equal(p_buttons, BUTTON_0_PRESSED);

    // Prepare a timer for handling bouncing by the button input
    hwtimer_t t_settle = hwtimer_alloc();
    const uint32_t SETTLE_TIME = 5000000; // 50ms in 10ns ticks

    const bool BUTTON_PRESSED = true;
    bool stable = true;
```

The SELECT_RES macro establishes the thread's event-driven processing loop.

```
    SELECT_RES(
        CASE_GUARD_THEN(p_buttons, stable, event_button_port),
        CASE_NGUARD_THEN(t_settle, stable, event_button_settle_timer)
    ) // SELECT_RES
```

Each CASE statement associates a resource with an event handler. In the first CASE, the resource is the port `p_buttons`, and events on that port invoke the `event_button_port` handler only if the `stable` variable equals `true`. This conditional guard prevents invocation of the handler while the button signal is bouncing. In the second CASE, the resource is the timer `t_settle`, and events on that timer invoke the `event_button_settle_timer` handler only if the `stable` variable equals `false`. This conditional guard enables invocation of the handler while the button signal settles down after someone presses it.

The loop responds to two events: a button press and a timer event. The first event handler responds to an event on the `p_buttons` port. Due to the port's trigger configuration, this event indicates that button 0 has been pressed. The handler takes several actions. First, it reads the button's state from the port and clears the event trigger so that subsequent events due to the button signal bouncing do not invoke the handler again. Second, it sends a button-pressed indication to the DSP module over the `c_gpi` channel. Third, it sets up a trigger time for the `t_settle` timer and enables the `event_button_settle_timer` handler by setting the `stable` variable to `false`. The timer will generate an event after an interval of 50 ms.

```
        event_button_port:
            button_state = port_in(p_buttons);
            port_clear_trigger_in(p_buttons);
            chan_out_word(c_gpi, BUTTON_PRESSED);
            hwtimer_set_trigger_time(t_settle, hwtimer_get_time(t_settle) + SETTLE_TIME);
            stable = false;
            continue; // end of event_button_port
```

The second event handler responds to an event on the **t_settle** timer. This event indicates that the timer's trigger interval has elapsed. The handler reads the current value of the buttons from the **p_buttons** port. Since the **event_button_port** handler cleared the trigger condition for the **p_buttons** port, no constraint exists on the value of each button. Both button 0 and button 1 may be either pressed or not pressed. Consequently, this handler masks the button state to check only the value of button 0. If button 0 reports an asserted (i.e., pressed) state, the handler establishes a new trigger time for the **t_settle** timer. If button 0 reports an unasserted (i.e., released) state, the handler clears the **t_settle** timer trigger, sets the **stable** variable to **true** and re-establishes the trigger condition on the **p_buttons** port. These actions re-enable the **event_button_port** handler to respond to the next button press and disable the **event_button_settle_timer** handler.

```
        event_button_settle_timer:
            button_state = port_in(p_buttons);

            // Active low; ignore values for buttons other than button 0
            if(0 == (button_state & BUTTON_0_MASK)) {
                hwtimer_change_trigger_time(t_settle, hwtimer_get_time(t_settle) + SETTLE_TIME);
            } else {
                hwtimer_clear_trigger_time(t_settle);
                stable = true;
                port_set_trigger_in_equal(p_buttons, BUTTON_0_PRESSED);
            }

            continue; // end of event_button_settle_timer
```

4 Adding AI

The AI extension adds a keyword spotting capability to the application. It includes a Neural Network that has been trained to spot ten different words. The extension has been designed so that detection of two of those words lights up an LED. When the AI extension detects the word "right", it lights up LED 0. When it detects the word "left", it lights up LED 3.

With the AI extension, the system consists of six threads communicating over three sets of ports as shown in Fig. 3.

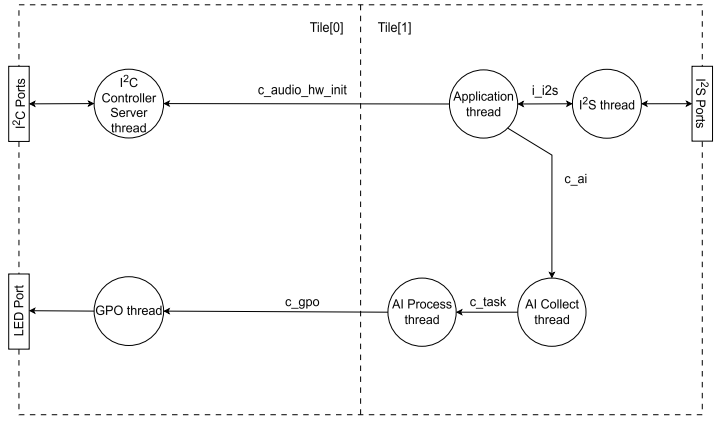

Fig. 3: AI extension thread diagram

The entry point for the AI functionality appears in the `some_ai.cpp` source file. The LED handling logic appears in the `some_gpo.c` source file. The corresponding header files `some_ai.hpp` and `some_gpo.h` declare the public interface to each module.

4.1 The AI header file

The `some_ai()` function provides an entry point to start the AI functionality. The function declares two channel end parameters, one for handling audio data received from the application and one for sending LED commands to the LED handler.

```
void some_ai(chanend c_ai, chanend c_gpo);
```

The `main()` function has the responsibility to provide the correct channel ends when starting the AI thread.

4.2 The AI source file

The `some_ai.cpp` source file provides the entry point and coordinating logic for the keyword spotting functionality. It defines two functions, `some_ai_collect()` and `some_ai_process()`. It also defines a single entry function, `some_ai()`, which operates as a separate thread of processing.

The `some_ai()` function

The `some_ai()` thread of processing starts two further threads of processsing, `some_ai_collect()` and `some_ai_process()`. It also creates a channel, `c_task`, which is used to send data from `some_ai_collect()` to `some_ai_process()`.

```
void some_ai(chanend_t c_ai,  chanend_t c_gpo) {
    assert((SAMPLE_FREQUENCY / AI_SAMPLE_FREQUENCY == 2) && (SAMPLE_FREQUENCY % AI_SAMPLE_FREQUENCY == 0));
    channel_t c_task = chan_alloc();
    PAR_JOBS(
        PJOB(some_ai_collect, (c_ai, c_task.end_a)),
        PJOB(some_ai_process, (c_task.end_b, c_gpo))
    );
    chan_free(c_task);
} // End of some_ai
```

> **Note**
>
> Although the `PAR_JOBS` macro in `some_ai()` includes two `PJOB` macros, only one new thread of processing comes into existence. The scheduler uses the same thread for `some_ai_process()` as for `some_ai()` since nothing prohibits running them sequentially. Execution of the `chan_free()` function is held off until both `some_ai_collect()` and `some_ai_process()` return.

The `some_ai_collect()` function

The keyword spotter operates at 16 kHz on a monaural audio signal collected into units of 320 samples within a 512-sample frame. The rest of the application operates at 32 kHz on a stereo audio signal, and it works on a sample-by-sample basis. The `some_ai_collect()` thread of processing provides the conversion between these two sets of characteristics.

Initialisation code defines a sample buffer and a frame buffer plus some variables that index these buffers. The `samples` buffer has enough space to hold two samples of stereo data. By holding two samples, it allows 2-to-1 decimation through averaging sample values. The `frame` buffer has enough space to hold 512 samples of monaural data.

```
    // Define buffers and index variables
    int32_t samples[SAMPLE_COUNT][CHANS_PER_FRAME] = {{0}};
    unsigned samples_idx = 0;
    unsigned channel_idx = 0;

    int32_t frame[KWD_FRAME_SIZE] = {0};
    unsigned frame_idx = 0;
```

A `SELECT_RES` macro establishes the thread's event-driven processing loop.

```
    SELECT_RES(CASE_THEN(c_ai, event_ai_chanend))
    {
        event_ai_chanend:{
            samples[samples_idx][channel_idx] = chan_in_word(c_ai);
            channel_idx = (channel_idx + 1) % CHANS_PER_FRAME;
            if (channel_idx == 0) {
                samples_idx = (samples_idx + 1) % SAMPLE_COUNT;
                if (samples_idx == 0) {
                    // get decimated mono sample
                    int32_t sample = (samples[0][0] >> 1) + (samples[1][0] >> 1);
                    frame[frame_idx++] = sample;

                    if (N_SAMPLES <= frame_idx) { // frame collected
                        frame_idx = 0;
                        chan_out_buf_word(c_collect, (uint32_t *)frame, N_SAMPLES);
                    } // frame collected
                }
            }
            continue;
        } // event_ai_chanend
    } // select
```

The single `CASE` statement associates the `c_ai` channel end with the `event_ai_chanend` event handler. The handler runs upon receiving data through the `c_ai` channel end. It collects the data received through the `c_ai` channel end into the `samples` buffer with `if` statements and use of the modulo operator advancing the `channel_idx` and `sample_idx` values correctly. Each time the `samples` buffer holds two new samples of stereo data, the handler copies the average value of the samples for channel 0 into the frame buffer.

```
                    int32_t sample = (samples[0][0] >> 1) + (samples[1][0] >> 1);
                    frame[frame_idx++] = sample;
```

Once the frame contains 320 samples of data, the handler sends it to the `some_ai_process()` thread over the `c_collect` channel end.

```
                    if (N_SAMPLES <= frame_idx) { // frame collected
                        frame_idx = 0;
                        chan_out_buf_word(c_collect, (uint32_t *)frame, N_SAMPLES);
                    } // frame collected
```

The `some_ai_process()` function

The `some_ai_process()` thread invokes the keyword spotter and determines whether either of the two desirable keywords have been detected based on the information the keyword spotter returns.

Its initialisation code defines variables and structures needed by the keyword spotter and initialises it. It also defines a variable for indicating which keyword, if any, the spotter detects.

```
void some_ai_process(chanend_t c_process, chanend_t c_gpo) {

    kwd_spotter_state_t state = KWD_SPOTTER_NOT_READY;
    kwd_spotter_ctx_t ctx = {{0}};

    state = kwd_spotter_init(&ctx);
    assert(state == KWD_SPOTTER_INIT_DONE);

    uint32_t *fft_frame_ptr = ctx.kwd_frame_ptr;

    kwd_label_indices_t keyword_detected = UNKNOWN;
```

A `SELECT_RES` macro establishes the thread's event-driven processing loop.

```
    SELECT_RES(CASE_THEN(c_process, event_frame_received))
    {
        event_frame_received:{
            chan_in_buf_word(c_process, fft_frame_ptr, N_SAMPLES);
            state = kwd_spotter_compute(&ctx);
            if (state == KWD_SPOTTER_DONE){
                if (ctx.kwd_output[LEFT] > THRESHOLD){
                    keyword_detected = LEFT;
                }
                else if (ctx.kwd_output[RIGHT] > THRESHOLD){
                    keyword_detected = RIGHT;
                }
                else{
                    keyword_detected = UNKNOWN;
                }
                chan_out_word(c_gpo, keyword_detected); // Notify GPO task
            }
            continue;
        } // event_frame_received
    } // select
```

The single `CASE` statement associates the `c_process` channel end with the `event_frame_received` event handler. The handler runs upon receiving data through the `c_process` channel end. It collects a block of data received through the `c_process` channel end into a buffer within the keyword spotter's context structure and then invokes the keyword spotter. The handler then determines whether or not the spotter has detected either of the two chosen keywords. It sends an indication of the keyword detected through the `c_gpo` channel end for every result returned by the keyword spotter. Any detection of the word "left" results in the handler sending the `LEFT` value. Likewise, any detection of the word "right" results in the handler sending the `RIGHT` value. If the keyword spotter detects anything else, including silence, it sends the `UNKNOWN` value.

Note

The keyword spotter implementation resides in the `kwd_spotter` directory and its sub-directories. This application note does not discuss the keyword spotter design or algorithm. However, AN02013 includes some information about these areas of keyword spotting.

Because the keyword spotter operates on a 320-sample segment of a 512-sample frame of data and because a human utterance spans multiple frames, the keyword spotter usually detects the same word multiple times in a row. However, the spotter's output will occasionally drop below the threshold used for detection only to rise above it again on a subsequent invocation. The design of `event_frame_received` does not attempt to process these multiple positive detections or the occasional false negative ones. It simply sends all of the data through the `c_gpo` channel end to the LED thread allowing that thread to manage the operation of the LEDs.

4.3 The LED header file

The `some_gpo()` function accepts a channel end which the GPO logic will use to receive LED data from the AI module. It also accepts a port argument.

```
#if defined(__XC__)
void some_gpo(chanend c_gpo, out port p_leds);
#else
void some_gpo(chanend c_gpo, port p_leds);
#endif // defined(__XC__)
```

Note

The XC language requires inclusion of a port's direction when declaring or defining it. Neither C nor C++ recognise the direction keyword when declaring or defining a port.

Hence `some_gpo.h` declares `some_gpo()` twice with slightly different signatures, once for use within XC files and again for use within C or C++ files.

The `main()` function supplies the correct channel end and the actual port connected to the LEDs when starting the LED handler thread.

4.4 The LED source file

The `some_gpo.c` source file implements the LED handler functionality. It defines the constants to turn on and off two of the LEDs and a single function, `some_gpo()`, organised as a separate thread of processing.

Initialisation prepares the LED port for output, ensures that all LEDs are turned off, and configures a timer to handle transient variablility in the keyword detection data received from the AI processing thread.

```
void some_gpo(chanend c_gpo, port p_leds)
{
    // Prepare the LED port for output
    // For simplicity, this example turns off all four LEDs initially,
    //  and only changes the state of LED 0
    port_enable(p_leds);
    uint32_t current_led_state = LEDS_OFF;
    uint32_t next_led_state    = LEDS_OFF;
    port_out(p_leds, current_led_state);

    // Prepare a timer to sustain the LED output
    hwtimer_t t_sustain = hwtimer_alloc();
    const uint32_t SUSTAIN_DURATION = 85000000; // 850ms in 10ns ticks
    bool sustaining = false;

    kwd_label_indices_t led_signal = UNKNOWN;
```

The `SELECT_RES` macro establishes the thread's event-driven processing loop.

```
    SELECT_RES(
        CASE_THEN(c_gpo, event_gpo_chanend),
```

Each `CASE` statement associates a resource with an event handler. In the first `CASE`, the resource is the channel end `c_gpo`, and data received over that channel end invoke the `event_gpo_chanend` handler. In the second `CASE`, the resource is the timer `t_sustain`, and events on that timer invoke the `event_sustain_timer_trigger` handler when the `sustaining` flag equals `true`.

The loop responds to two events: data received over a channel end and a timer event. The first event handler responds to data received over the `c_gpo` channel end. The handler takes two actions. First, it reads the data from the channel end so that subsequent attempts to use the channel by the keyword spotter thread do not block. Second, it either processes that data or ignores it depending on the value of the `sustaining` flag.

```
        event_gpo_chanend:
            led_signal = chan_in_word(c_gpo);
            if(!sustaining) {
                switch (led_signal) {
                    case LEFT:
                        next_led_state = LED_3_ON;
                        break;
                    case RIGHT:
                        next_led_state = LED_0_ON;
                        break;
                    case UNKNOWN:
                        next_led_state = LEDS_OFF;
                        break;
                    default:
                        next_led_state = LEDS_OFF;
                        break;
                }

                if(next_led_state != current_led_state) {
                    port_out(p_leds, next_led_state);
                    current_led_state = next_led_state;

                    if(next_led_state != LEDS_OFF) {
                        sustaining = true;
```

(continues on next page)

(continued from previous page)

```
                hwtimer_set_trigger_time(t_sustain, hwtimer_get_time(t_sustain) + SUSTAIN_DURATION);
            }
        }
    }
    continue; // end of event_gpo_chanend
```

As noted earlier, the keyword spotter thread sends a stream of keyword detection data through the channel that connects it to this thread. If the `event_gpo_chanend` handler did not read the data from the `c_gpo` channel end every time the keyword spotter thread sent some, the `chan_out_word()` function call in `some_ai_process()` would block further processing within that loop, which in turn would prohibit the processing of new events there. Consequently, the `some_gpo()` thread must respond to events on the `c_gpo` channel promptly regardless of any other responsibilities it has.

The `sustaining` flag ensures that the `event_gpo_chanend` handler pays attention to the received data when the keyword spotter thread has detected a new keyword. The `sustaining` flag equals `true` during the period that the `some_gpo()` thread keeps an LED turned on in a sustained manner, that is, after the keyword spotter thread has detected a new keyword. While `sustain` equals `true`, the `event_gpo_chanend` handler ignores data received over the `c_gpo` channel end. Once the `some_gpo` thread turns the LED off, the `sustaining` flag equals `false`, and the `event_gpo_chanend` handler processes new keyword detection data.

To process the keyword detection data, the `event_gpo_chanend` handler maps the data to specific LED states through a `switch` statement. If the new LED state differs from the current one, the handler turns on or off one of the LEDs through the `port_out()` function. When the handler turns on an LED, it changes the value of `sustaining` to true to ignore subsequent data received over the `c_gpo` channel end. It also sets a trigger on the `t_sustain` timer at 850 ms from the current time using the `hwtimer_set_trigger_time` function.

When the `t_sustain` timer reaches the trigger time and the `sustaining` flag equals `true`, the `event_sustain_timer_trigger` handler runs. This handler sets the `sustaining` flag to `false` and clears the trigger condition on the `t_sustain` timer. Setting the `sustaining` flag to `false` results in the `event_gpo_chanend` handler processing the next keyword detection data it receives over the `c_gpo` channel. Clearing the trigger condition for the `t_sustain` timer and setting the `sustaining` flag to `false` ensures that the `event_sustain_timer_trigger` handler will not run again until a new keyword has been detected.

```
event_sustain_timer_trigger:
    hwtimer_clear_trigger_time(t_sustain);
    sustaining = false;
    continue; // end of event_sustain_timer_trigger
```

5 Modifications to the `main()` function

Compared to AN00162, the `main()` function contains only a few changes. For specific build configurations, it:

- Defines the channels `c_ai`, `c_dsp`, `c_gpi`, and `c_gpo`,
- Starts the `some_gpi()` and `some_gpo()` threads on `tile[0]`, and
- Starts the `some_ai()` and `some_dsp()` threads on `tile[1]`.

On `tile[1]`, the `i2s_loopback()` function has been renamed `application()`. Channel ends have been conditionally included as arguments to the `application()` function for communication with the AI and DSP modules in specific build configurations. Due to the use of these channel ends within the `application()` function, use of the `[[distribute]]` attribute is no longer valid. The `application()` function must occupy its own thread.

```
int main(void)
{
    chan c_audio_hw_init; // Channel for cross-tile communication

    #if (ADD_AI_PROCESSING == 1)
        chan c_ai;  // Channel for application <-> AI communication
        chan c_gpo; // Channel for LED events
    #endif

    #if (ADD_DSP_PROCESSING == 1)
        chan c_dsp; // Channel for application <-> DSP communication
        chan c_gpi; // Channel for button press events
    #endif

    par {
        on tile[0]: {
            par {
                xk_evk_xu316_AudioHwRemote(c_audio_hw_init);

                #if (ADD_AI_PROCESSING == 1)
                    some_gpo(c_gpo, p_leds);
                #endif

                #if (ADD_DSP_PROCESSING == 1)
                    some_gpi(c_gpi, p_buttons);
                #endif
            } // Inner par
        } // tile[0]

        on tile[1]: {
            interface i2s_frame_callback_if i_i2s;

            xk_evk_xu316_AudioHwChanInit(c_audio_hw_init);

            par {
                application(
                    #if (ADD_AI_PROCESSING == 1)
                        c_ai,
                    #endif
                    #if (ADD_DSP_PROCESSING == 1)
                        c_dsp,
                    #endif
                    i_i2s);

                i2s_frame_master(
                    i_i2s,
                    p_dac,
                    NUM_I2S_LINES,
                    p_adc,
                    NUM_I2S_LINES,
                    DATA_BITS,
                    p_bclk,
                    p_lrclk,
                    p_mclk,
                    bclk);

                #if (ADD_AI_PROCESSING == 1)
                    some_ai(c_ai, c_gpo);
                #endif

                #if (ADD_DSP_PROCESSING == 1)
                    some_dsp(c_dsp, c_gpi);
                #endif
            } // Inner par
        } // tile[1]
    } // Outer par
    return 0;
```

6 Modifications to the `application()` function

Likewise, the `application()` function has only a few changes compared with the `i2s_loopback()` function in AN00162. A minor modifications has taken place in defining the size of the sample buffer. The expression `NUM_I2S_LINES * CHANS_PER_FRAME` has been replaced with `BUFFER_SIZE` with that constant defined in `buffer.h` to allow the static definition of corresponding buffers in multiple modules.

For specific build configurations, the `application()` function:

- Accepts up to two channel ends as parameters, one each for the AI and DSP extensions,
- Initialises two more variables for use with the DSP extension,
- Sends audio data to these extensions as part of the `i_i2s.receive()` method, and

- Receives and stores DSP-processed audio data from the DSP module over the c_dsp channel.

```
void application(
    #if (ADD_AI_PROCESSING == 1)
        chanend c_ai,
    #endif
    #if (ADD_DSP_PROCESSING == 1)
        chanend c_dsp,
    #endif
    server i2s_frame_callback_if i_i2s
)
{
    xk_evk_xu316_AudioHwInit(hw_config);

    // Array used for looping back samples
    int32_t samples[BUFFER_SIZE] = {0};

    #if (ADD_DSP_PROCESSING == 1)
        size_t idx = 0;
        int32_t sample;
    #endif

    while (1) {
    select {
        case i_i2s.init(i2s_config_t &?i2s_config, tdm_config_t &?tdm_config):
            i2s_config.mode = I2S_MODE_I2S;
            i2s_config.mclk_bclk_ratio = (MASTER_CLOCK_FREQUENCY / (SAMPLE_FREQUENCY * CHANS_PER_FRAME * DATA_
↪BITS));
            xk_evk_xu316_AudioHwConfig(SAMPLE_FREQUENCY, MASTER_CLOCK_FREQUENCY, 0, DATA_BITS, DATA_BITS);
            break;

        case i_i2s.receive(size_t n_chans, int32_t in_samps[n_chans]):
            for (size_t i = 0; i < n_chans; ++i){
                #if (ADD_AI_PROCESSING == 1)
                    // Send samples for AI processing
                    c_ai <: in_samps[i];
                #endif

                #if (ADD_DSP_PROCESSING == 1)
                    // Send samples to DSP processing
                    c_dsp <: in_samps[i];
                #else
                    samples[i] = in_samps[i]; // copy samples for loopback
                #endif
            }
            break;

        #if (ADD_DSP_PROCESSING == 1)
        // Receive processed samples from DSP
        case c_dsp :> sample:
            samples[idx] = sample;
            if (BUFFER_SIZE <= ++idx){
                idx = 0;
            }
            break;
        #endif

        case i_i2s.send(size_t n_chans, int32_t out_samps[n_chans]):
            for (size_t i = 0; i < n_chans; ++i){
                out_samps[i] = samples[i]; // loopback samples
            }
            break;

        case i_i2s.restart_check() -> i2s_restart_t restart:
            restart = I2S_NO_RESTART; // Keep on looping
            break;
        } // End select
    } // End while (1)
} // End application
```

7 Combined extensions

When both extensions are included, the system consists of eight threads communicating over four sets of ports as shown in Fig. 4.

8 Tutorial application

8.1 Prerequisites for building

This application note assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found in the XTC Install Guide.

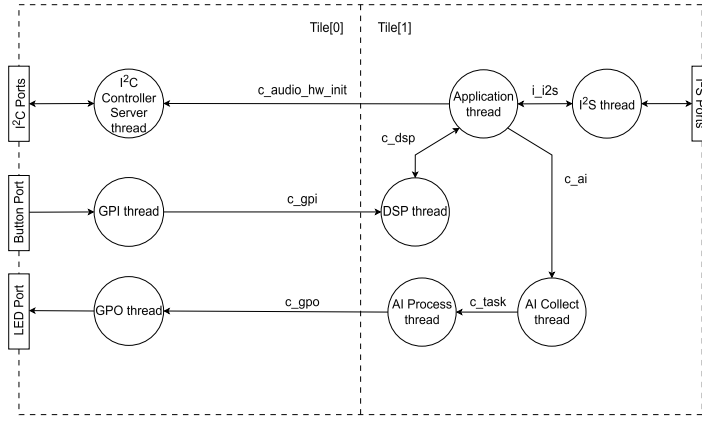

Fig. 4: AI and DSP extensions thread diagram

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The **an02050** software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an02050
cd app_an02050
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

8.2 Building the application

This application note includes four build configurations:

- **loopback**: the basic I2S loopback application from AN00162.
- **dsp**: the application with the DSP extension.
- **ai**: the application with the AI extension.
- **dsp_ai**: the application with both the DSP and AI extensions.

All four application binaries can be built using **xmake**:

```
xmake -j -C build
```

To build a specific configuration, use:

```
xmake -j -C build app_an02050_<config>
```

where **<config>** is one of **loopback**, **dsp**, **ai**, or **dsp_ai**.

Note

Building the **ai** and **dsp_ai** applications can take several minutes each.

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the **app_an02050/bin** directory.

For subsequent builds, the **cmake** step may be omitted. If **CMakeLists.txt** or other build files are modified, **cmake** will be re-run automatically by **xmake** as needed.

8.3 Hardware setup

Please refer to the XCORE.AI Evaluation Kit hardware platform documentation.

The application is designed to run on the XCORE.AI Evaluation Kit. Before running the application:

- Connect a USB cable from a host computer to the DEBUG connector.
- Connect a USB cable from a host computer to the USB connector.
- Connect a sound source to the 3.5mm LINE IN connector.
- Connect headphones or speakers to the corresponding LINE OUT connector.

Note

It may be helpful to connect the host computer to the LINE IN connector using an external USB sound card. Doing so will allow a host-based audio application, such as Audacity or Sound eXchange, to act as the sound source.

8.4 Running the application

From an XTC command prompt, run the following command from the **an02050/app_an02050** directory:

```
xrun ./bin/app_an02050_<config>.xe
```

where **<config>** is one of **loopback**, **dsp**, **ai**, or **dsp_ai**.

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/app_an02050_<config>.xe
```

8.5 Testing the application

The application accepts analogue sound received through the LINE IN connector and produces it through the LINE OUT connector. The behaviour of the application varies depending on the build configuration.

The **loopback** configuration does not alter or interpret the sound received through the LINE IN connector. It simply reproduces the received sound without any change through the LINE OUT connector. In the **loopback** configuration, Button 0 and Button 1 have no effect and none of the general purpose LEDs light up.

The **dsp** configuration adds a volume control to the sound processing. Pressing Button 0 changes the volume of the sound produced through the LINE OUT connector. Subsequent presses of Button 0 lower the volume in steps until the sound is muted, then increase the volume in steps to a maximum level.

The `ai` configuration adds keyword spotting to the sound processing. If the application detects the word "left" in the sound received through the LINE IN connector, it lights up LED 3 briefly. Likewise, if the application detects the word "right", it lights up LED 0 briefly.

The `dsp_ai` configuration combines the features of the `dsp` and `ai` configurations. Button 0 will control the volume of the sound received through the LINE IN connector when reproduced through the LINE OUT connector. Detection by the application of `left` or `right` will briefly light up LED 3 or LED 0 respectively.

9 Resource usage

Table 1: Tile[0] resources used

Build Configuration	Channel Ends	Clock Blocks	HW Timers	Memory	Ports (1b)	Ports (4b)	Threads
Loopback	1	0	1	10748	2	0	1
DSP	2	0	2	11500	2	1	2
AI	2	0	2	11524	2	1	2
DSP plus AI	3	0	3	12036	2	2	3

Table 2: Tile[1] resources used

Build Configuration	Channel Ends	Clock Blocks	HW Timers	Memory	Ports (1b)	Ports (4b)	Threads
Loopback	3	1	2	10904	5	0	3
DSP	6	1	2	12040	5	0	4
AI	6	1	2	133740	5	0	4
DSP plus AI	9	1	2	134868	5	0	5

Note

The `i_i2s` XC interface on Tile[1] in the `main()` function uses one channel end for the `application()` and one channel end for the `i2s_frame_master()`.

10 References

- AN00162 Implementing an I2S loopback using the lib_i2s library
- AN02013 Face ID and Keyword Spotting Example
- XCommon CMake build system for XCORE applications and libraries
- XCORE.AI Evaluation Kit
- XTC Install Guide
- XMOS XTC Tools

Copyright © 2025, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.