



Extending USB Audio with Digital Signal Processing

Publication Date: 2023/10/3

Document Number: XM006859A

IN THIS DOCUMENT

- ▶ Introduction to USB Audio
 - ▶ API offered by USB Audio
 - ▶ DSP functions available
 - ▶ Timing requirements
 - ▶ Executing the DSP on the other physical core
 - ▶ Parallelising DSP
 - ▶ Data Parallel DSP
 - ▶ Data Pipelining DSP
 - ▶ Controlling
-

In this app note we describe how to extend the XMOS USB Audio stack with DSP capabilities.

USB Audio is a highly configurable piece of software; in its simplest form it may just interface a single ADC to USB Audio; but it can deal with a multitude of I2S, TDM, DSD, S/PDIF, ADAT and other interfaces. Often data is just transported in real-time, but DSP that may be interesting may, for example, include:

- ▶ Equalisation
- ▶ Mixing
- ▶ Dynamic range compression
- ▶ Audio effects

This app note discusses the API that the USB Audio stack offers to enable you to include DSP algorithms inside the stack.

For reference, we refer to the following repositories that you may want to use:

- ▶ http://github.com/xmos/sw_usb_audio.git for the USB Audio reference design
- ▶ https://github.com/xmos/lib_xua.git for the USB Audio library design

1 Introduction to USB Audio

The basic structure of USB Audio is shown below in Figure 1.

On the left is a USB interface to the host - this is dealt with by the XUD and XUA libraries. XUD https://www.github.com/xmos/lib_xud is the low level USB library for XCORE, XUA https://www.github.com/xmos/lib_xua is the USB-Audio protocol implementation on xcore. On the right is a series of interfaces (ADC, DAC, S/PDIF, ADAT). USB Audio provides a path from the left to the right (USB host computer to the interfaces), this is called the output path; and a path from the right to the left (the interfaces to the USB

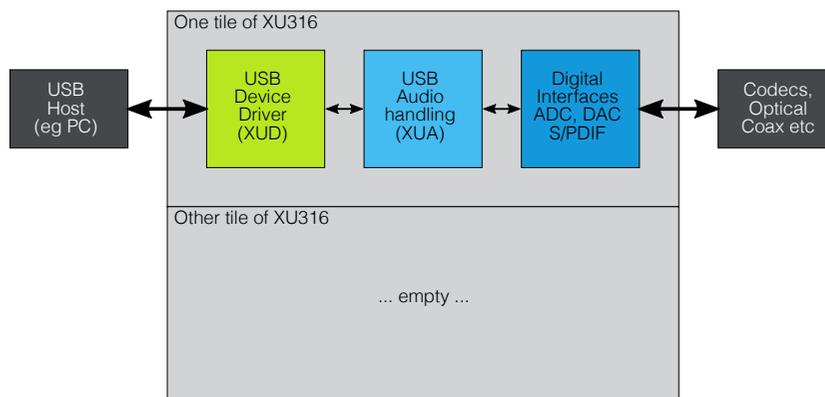


Figure 1:
Structure of
USB Audio

host computer) that is called the input path. The terms input-path and output-path are host-centric names, and we use input and output this was as it is consistent with the USB standard nomenclature.

The XU316 device has two tiles, and for many designs one of the tiles will be empty. This is not always the case, as there may be a situation where the ADC/DAC I/O pins are located on the other tile. This subtlety does not matter for addition of simple DSP. Also, the physical core used for the USB stack may be tile 0 or tile 1 depending on the design.

2 API offered by USB Audio

The USB Audio stack provides one function that you need to override in order to add any DSP capability to your system:

```
extern void UserBufferManagement (
    unsigned output_samples [NUM_OUTPUTS],
    unsigned input_samples [NUM_INPUTS]);
```

For brevity we use `NUM_OUTPUTS` and `NUM_INPUTS` throughout this code to refer to the number of output audio-channels (`NUM_USB_CHAN_OUT`) and the number of input audio-channels (`NUM_USB_CHAN_IN`).

The `UserBufferManagement` function is called at the sample rate of the USB Audio stack (eg, 48 kHz) and between them the two arrays contain a full multi-channel audio-frame. The first array carries all the data that shall be shipped to the interfaces, the second array carries all the data from the interfaces that shall be shipped to the USB host. You can chose to intercept and overwrite the samples stored in these arrays. The interfaces are ordered first all I2S channels, then optional S/PDIF, finally optional ADAT.

A second function that you can overwrite is:

```
extern void UserBufferManagementInit(void);
```

This function is called once before the first call to `UserBufferManagement`. The code in this document does not require this function, but other code may require it.

Note that the values of the type are *unsigned*; a 32-bit number. The use of these 32 bits depends on the data-types used for the audio, typical values are 16-bit PCM (the top 16 bits are a signed PCM value), 24-bit PCM (the top 24 bits are a signed PCM value), 32-bit PCM (the top 32 bits are a signed PCM value), or DSD (the 32 bits are PDM values, with the least significant bit representing the oldest 1-bit value).

In this example we just modify the output path - and we use `NUM_OUTPUTS=2` and `NUM_INPUTS=4`. We can run the output_samples through a cascaded_biquad in order equalise the output signal. One can go further and apply independent biquads to the two channels to independently equalise stereo speakers:

```
#define FILTERS 4
//      b2/a0      b1/a0      b0/a0      -a1/a0      -a2/a0
int32_t filter_coeffs[FILTERS*5] = {
    261565110, -521424736, 260038367, 521424736, -253168021,
    255074543, -506484921, 252105451, 506484921, -238744538,
    280274501, -523039333, 245645878, 523039333, -257484924,
    291645146, -504140302, 223757950, 504140302, -246967640,
};

int32_t filter_states[NUM_INPUTS+NUM_OUTPUTS][FILTERS*4];

void UserBufferManagement(
    unsigned output_samples[NUM_OUTPUTS],
    unsigned input_samples[NUM_INPUTS])
{
    for(int i = 0; i < NUM_OUTPUTS; i++) {
        output_samples[i] = dsp_filters_biquads((int32_t) output_samples[i]
        ↪ ),
                                filter_coeffs,
                                filter_states[i],
                                FILTERS,
                                28);
    }
}

void UserBufferManagementInit() {}
```

If one wants, one can combine input_samples and output_samples in order to mix data from interfaces or USB into USB or the interfaces.

The sample rate depends on the environment. The USB application typically has a list of supported sample rates (this may just be one sample-rate), and the user can on the host select which sample rate they want to use. For simplicity, we do not discuss sample-rate changes; we assume that there is just one sample-rate.



3 DSP functions available

There are a few repositories with DSP and general maths functions available, with different trade-offs between speed, accuracy, and ease-of-use.

- ▶ https://github.com/xmos/lib_xcore_math is the xcore.ai library for high performance maths functions. Many of them are optimised to make use of the vector unit and use 40-bit accumulators.
- ▶ https://github.com/xmos/lib_dsp for high-resolution maths functions that execute on the CPU often using 64-bit accumulators. These functions are not as fast as `lib_xcore_math`
- ▶ https://github.com/xmos/lib_audio_effects for audio effects functions. (this is based on `lib_dsp` above)

In this application note we use as a running example a cascaded biquad filter that is set to a fixed operation:

- ▶ First stage Peaking Filter 200 Hz, 1 octave -20 dB,
- ▶ Second stage Peaking Filter 400 Hz, 1 octave +10 dB,
- ▶ Third stage Peaking Filter 800 Hz, 1 octave -20 dB,
- ▶ Fourth stage Peaking Filter 1600 Hz, 1 octave +10 dB,

This is not a necessarily a realistic set of filters, but it is something that can easily be heard.

4 Timing requirements

The XMOS USB Audio stack is designed to operate on single samples in order to minimise latency introduced by the audio stacks. The `UserBufferManagement()` function is called from the core of the USB stack; it is called at the native frame rate of the system (for example 44.1 kHz), and it should therefore take no longer than one sample period to finish its operation. In fact, it has a bit less time than that in order to guarantee that the samples reach the next stage of the pipeline.

Given the speed of a single thread in a system (for example $600 / 8 = 75$ MHz) and the sample rate (say, 44.1 KHz sample rate) we can calculate the number of issue slots available between two samples: $75,000,000 / 44,100 = 1,700$ issue slots. This includes the time taken by the USB stack to shuffle data around. Taking that into account there is no more than 1,300 issue-slots available for DSP using this method, which allows for only a limited number of FIR taps or biquads to be used. The timeline is shown in Figure 2.

What is more, with higher sample rates the overhead of the USB stack is the same, but the time between samples is squeezed, further limiting the number of cycles available for DSP.

As XCORE is a concurrent multi-threaded multi-core processor, there are other threads and cores available for DSP. It depends on the precise configuration of the USB stack (whether you use special interfaces such as S/PDIF, ADAT, MIDI) but in a simple case

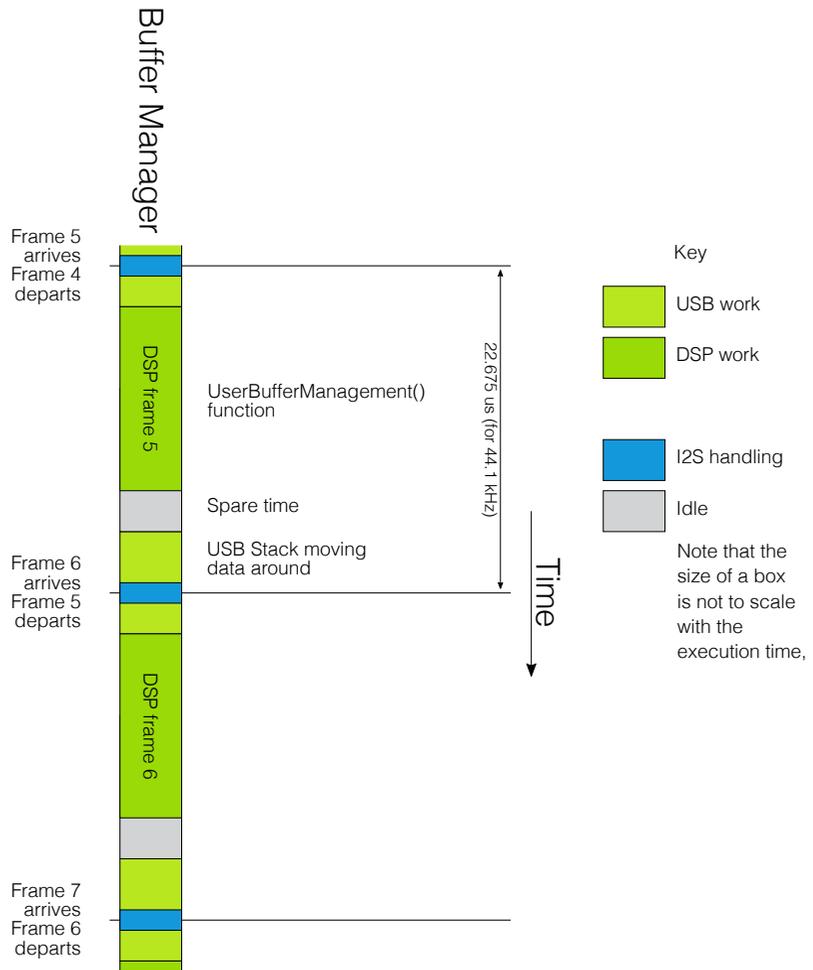


Figure 2:
Timeline of
executing DSP
inside a thread

with just I2S, USB Audio uses around 30% of the compute, with one tile being completely empty.

We will first look at how to use a single thread on the other tile for DSP, then we will look in how to generally parallelise DSP, and then we will look into using multiple threads for DSP.

5 Executing the DSP on the other physical core

The XCORE architecture offers a communication fabric to efficiently transport data between threads and between cores. Communication works on *channels*. A *channel* has two ends, *A* and *B*, and data that is *output* into *A* has to be *input* on *B*, and data that



is output into B has to be input from A . A and B can be inside the same physical core on different threads, or on different cores on the same chip, or on different chips in the same system; communication always works, but performance is lower when the physical distance increases.

A channel is like a two way communication pipe. It has very little buffering capacity, so both ends of the channel have to agree to communicate otherwise one side will wait for the other.

The data types and functions for communicating data provided by `lib_xcore` are:

- ▶ `chanend_t c` ; a type holding the reference to one end of a *channel*
- ▶ `chan ch` ; a type holding a complete channel with both ends
- ▶ `chan_out_word(c, x)` ; a function that outputs a word x over channel-end c .
- ▶ `x = chan_in_word(c)` ; a function that inputs a word x over channel-end c .
- ▶ `chan_out_buf_word(c, x, n)` ; a function that outputs n words from array x over channel-end c .
- ▶ `chan_in_buf_word(c, x, n)` ; a function that inputs n words over channel-end c into array x

We could also use XC instead of C and lib-xcore; the resulting behaviour is identical. There is equivalent functions `chanend_*` that create streaming channels rather than synchronised channels. We do not use them in this app-note, but they can be useful where extra performance and predictability are required.

Typical code to off-load the DSP to the other tile involves a `UserBufferManagement` function that outputs and inputs samples to the DSP task, a `user_main.h` function that declares the extra code needed to create the channels and start the DSP task, and a DSP task that receives and transmits the data.

The `UserBufferManagement` code is:



```

#include "xcore/chanend.h"
#include "xcore/channel.h"

static chanend_t g_c;

void UserBufferManagement(
    unsigned output_samples[NUM_OUTPUTS],
    unsigned input_samples[NUM_INPUTS]
) {
    chan_out_buf_word(g_c, output_samples, NUM_OUTPUTS);
    chan_out_buf_word(g_c, input_samples, NUM_INPUTS);
    chan_in_buf_word(g_c, output_samples, NUM_OUTPUTS);
    chan_in_buf_word(g_c, input_samples, NUM_INPUTS);
}

void UserBufferManagementSetChan(chanend_t c) {
    g_c = c;
}

void UserBufferManagementInit() {}
    
```

The code to be included in the main program is as follows:

```

#define USER_MAIN_DECLARATIONS \
chan c_data_transport; \
interface i2c_master_if i2c[1];

#define USER_MAIN_CORES \
on tile[1]: { \
    dsp_main(c_data_transport); \
} \
on tile[0]: { \
    ctrlPort(); \
    i2c_master(i2c, 1, p_scl, p_sda, 100); \
} \
on tile[1]: { \
    UserBufferManagementSetChan(c_data_transport); \
    unsafe \
    { \
        i_i2c_client = i2c[0]; \
    } \
}
    
```

And finally the code to perform the DSP is the opposite of the buffer-management function:



```

#define FILTERS 4
//      b2/a0      b1/a0      b0/a0      -a1/a0      -a2/a0
int32_t filter_coeffs[FILTERS*5] = {
    261565110, -521424736, 260038367, 521424736, -253168021,
    255074543, -506484921, 252105451, 506484921, -238744538,
    280274501, -523039333, 245645878, 523039333, -257484924,
    291645146, -504140302, 223757950, 504140302, -246967640,
};

int32_t filter_states[NUM_INPUTS+NUM_OUTPUTS][FILTERS*4];

void dsp_main(chanend_t c_data) {
    int for_usb [NUM_INPUTS + NUM_OUTPUTS];
    int from_usb[NUM_INPUTS + NUM_OUTPUTS];
    while(1) {
        chan_in_buf_word( c_data, &from_usb[0],          NUM_OUTPUTS);
        chan_in_buf_word( c_data, &from_usb[NUM_OUTPUTS], NUM_INPUTS);
        chan_out_buf_word(c_data, &for_usb[0],          NUM_OUTPUTS);
        chan_out_buf_word(c_data, &for_usb[NUM_OUTPUTS], NUM_INPUTS);
        for(int i = 0; i < 2; i++) {
            for_usb[i] = dsp_filters_biquads((int32_t) from_usb[i],
                                           filter_coeffs,
                                           filter_states[i],
                                           FILTERS,
                                           28);
        }
    }
}
    
```

The execution of two of the tasks (the USB Task calling `UserBufferManagement`) and the DSP task (`dsp_main`) is shown below in Figure 3.

Time progresses from top to bottom, and we show a snapshot of what happens around the time that Frame numbers 5..7 arrive over I2S. The small dark blue box is when Frame 5 arrives over I2S whilst a processes Frame 3 is sent out over I2S. The light blue boxes below are the communication between the two tasks; `UserBufferManagement()` on the left, and the first four lines of the while-loop in `dsp_main()` on the right. After that, the USB task has a bit of idle time (to cope with higher sample rates and more channels), and the DSP task starts the DSP. Whilst the DSP is operating on Frame 5; Frame 6 arrives in the USB task, and the DSP task must finish before the next communication phase. Please note that the boxes are not drawn to scale otherwise some of them would be too small to see.

It is important to note that the grey area where the Buffer Manager is idle is time that can be used by other threads. This means that up to five DSP threads can be active at this time, taking all the bandwidth of the processor. During the period where the Buffer Manager is working, the DSP threads will run slightly slower; probably hardly noticeable as they will also be having some down time over this period.

In this example, we assume a 44,100 Hz sample rate. If the DSP thread is too late, then all the timings will fail; it has to be on time, but it is allowed to be *just in time*. Note that the DSP processing is synchronous with the frame transmissions, but the phase is off. Every sample is processed a bit later than arriving, leading to a whole sample delay



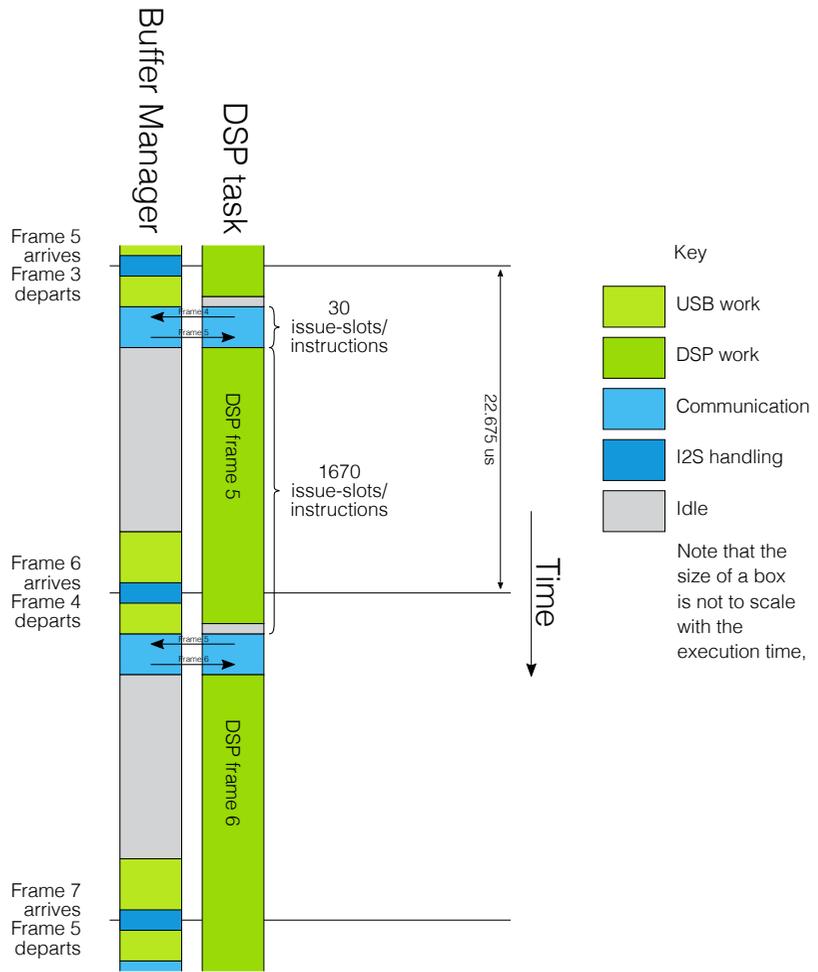


Figure 3:
Timeline of
executing the
two
concurrent
threads

6 Parallelising DSP

Parallelisation involves splitting work into a multitude of *tasks*. *Tasks* can then be mapped onto threads. The reason to separate these two words is that a *task* is a software concept: a set of instructions that does something meaningful, for example a shelf-filter. If we have 10 of those tasks then we can execute five of them in *Thread 1* and five of them in *Thread 2* and we have achieved 2x parallelism.

Typically tasks are dependent on each other, and when the design is drawn out that is reflected by arrows from one task into the other, representing data being transported from

one task to the next. When the tasks are mapped onto threads these data dependencies have to be adhered to.

DSP lends itself to parallelism as there are typically large clusters of compute on identified sets of data. Each DSP problem will be parallelised individually, and in this document we distinguish two models on which the rest can be built:

- ▶ Data parallelism, for example, output-conditioning on stereo speakers. In this case, one could put the DSP for the left speaker in task 1, and the DSP for the right speaker in task 2.
- ▶ Data Pipelining. A series of DSP tasks are executed one after the other on an audio stream.

In general this gives rise to two sorts of designs. The first design is one where each sample is being fed into a task, and the tasks independently of each other all produce the output samples. The second design is one where the samples run through a sequence of tasks before finally producing the output samples. The latter architecture has an inherent higher latency than the former design and a slightly more complex design. The former is a very simple design that we shall discuss first.

7 Data Parallel DSP

Data parallelism is a simple extension of the previous example. Instead of using a single channel we use multiple channels to communicate the data onto the DSP task. This gives rise to the timeline shown below in Figure 4.

Like before, we use channels to communicate between the DSP tasks, what is new is that we have to create those DSP tasks, and create the channels between them. The only difference is in the `dsp_main` function.



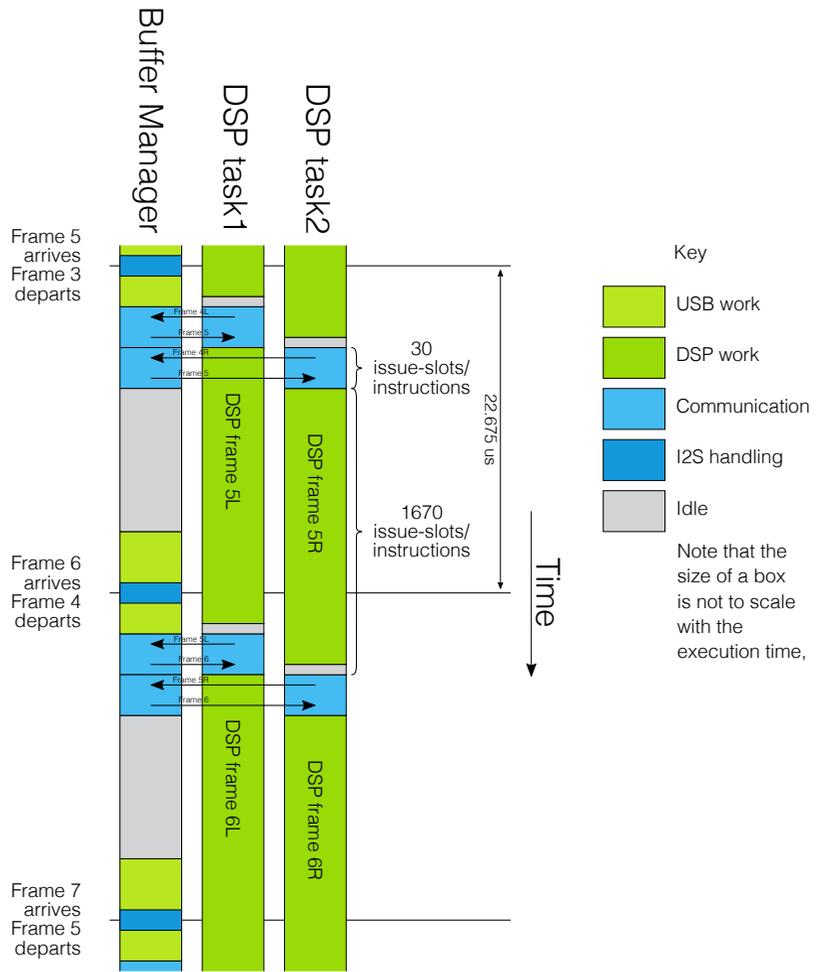


Figure 4:
Timeline of executing the two concurrent threads

The UserBufferManagement code is:

```

static chanend_t g_c, g_c2;

void UserBufferManagement(
    unsigned output_samples [NUM_OUTPUTS],
    unsigned input_samples [NUM_INPUTS]
) {
    chan_out_buf_word(g_c, output_samples, NUM_OUTPUTS);
    chan_out_buf_word(g_c, input_samples, NUM_INPUTS);
    chan_in_buf_word(g_c, output_samples, NUM_OUTPUTS/2);
    chan_in_buf_word(g_c, input_samples, NUM_INPUTS/2);
    chan_out_buf_word(g_c2, output_samples, NUM_OUTPUTS);
    chan_out_buf_word(g_c2, input_samples, NUM_INPUTS);
    chan_in_buf_word(g_c2, output_samples+NUM_OUTPUTS/2, NUM_OUTPUTS/2);
    chan_in_buf_word(g_c2, input_samples +NUM_INPUTS/2, NUM_INPUTS/2);
}

void UserBufferManagementSetChan(chanend_t c, chanend_t c2) {
    g_c = c;
    g_c2 = c2;
}

void UserBufferManagementInit() {}
    
```

The code to be included in the main program is as follows:

```

#define USER_MAIN_DECLARATIONS \
chan c1, c2; \
interface i2c_master_if i2c[1]; \

#define USER_MAIN_CORES \
on tile[1]: { \
    dsp_main1(c1); \
} \
on tile[1]: { \
    dsp_main2(c2); \
} \
on tile[0]: { \
    ctrlPort(); \
    i2c_master(i2c, 1, p_scl, p_sda, 100); \
} \
on tile[1]: { \
    UserBufferManagementSetChan(c1, c2); \
    unsafe \
    { \
        i_i2c_client = i2c[0]; \
    } \
} \
    
```

And finally the code to perform the DSP is the opposite of the buffer-management function:

```

#define FILTERS 4
//      b2/a0      b1/a0      b0/a0      -a1/a0      -a2/a0
int32_t filter_coeffs[FILTERS*5] = {
    261565110, -521424736, 260038367, 521424736, -253168021,
    255074543, -506484921, 252105451, 506484921, -238744538,
    280274501, -523039333, 245645878, 523039333, -257484924,
    291645146, -504140302, 223757950, 504140302, -246967640,
};

int32_t filter_states [NUM_OUTPUTS/2][FILTERS*4];
int32_t filter_states2[NUM_OUTPUTS/2][FILTERS*4];

void dsp_main1(chanend_t c_data) {
    int for_usb [NUM_INPUTS/2 + NUM_OUTPUTS/2];
    int from_usb [NUM_INPUTS + NUM_OUTPUTS];
    while(1) {
        chan_in_buf_word( c_data, &from_usb[0],          NUM_OUTPUTS);
        chan_in_buf_word( c_data, &from_usb[NUM_OUTPUTS], NUM_INPUTS);
        chan_out_buf_word(c_data, &for_usb[0],          NUM_OUTPUTS/2);
        chan_out_buf_word(c_data, &for_usb[NUM_OUTPUTS/2], NUM_INPUTS/2);
        for(int i = 0; i < NUM_OUTPUTS/2; i++) {
            for_usb[i] = dsp_filters_biquads((int32_t) from_usb[i],
                                           filter_coeffs,
                                           filter_states[i],
                                           4,
                                           28);
        }
    }
}

```

`dsp_main2` is identical, and the code may be shared provided they have separate state to operate on.

This method expands to five threads, after which the XCORE.AI pipeline is fully used. More threads can be used, but no performance will be gained. This is because the full number of issue cycles will be divided between more threads.

8 Data Pipelining DSP

We can make an arbitrary pipeline of DSP processes by creating an extra thread that acts as the source of the data and as the sync of the data. This thread's purpose is to perform just those tasks. The reason that this task is special is that it loops the data path around, because what came out of the pipe has to go back into the USB Audio stack at a determined point in time. The pipeline that we're building is shown in Figure 5.

The pipeline that we are building requires a bit of plumbing to make it all work but the code is reasonably straightforward otherwise.

DSP task 1B is implemented by `dsp_thread1b` and picks up data from the distributor, and outputs data to dsp tasks 1A and 1B:

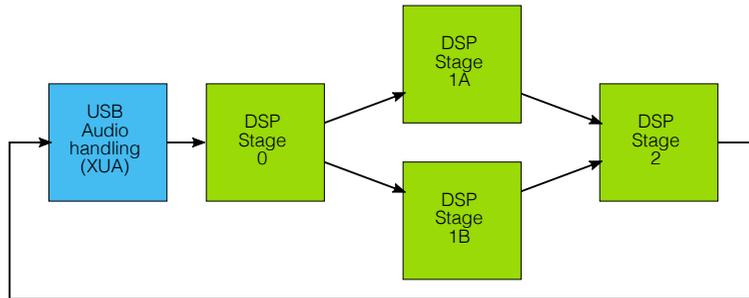


Figure 5:
Example pipeline

```

#define FILTERS0 1

static __attribute__((aligned(8))) int32_t filter_coeffs0[FILTERS0*5] = {
    261565110, -521424736, 260038367, 521424736, -253168021,
};

static __attribute__((aligned(8))) int32_t filter_states0[NUM_OUTPUTS][
    ↪ FILTERS0*4];

void dsp_thread0(chanend_t c_fromusb,
                 chanend_t c_to1a, chanend_t c_to1b) {
    int from_usb[NUM_OUTPUTS];
    int for_1[NUM_OUTPUTS];
    while(1) {
        // Pick up my chunk of data to work on
        chan_in_buf_word(c_fromusb, &from_usb[0], NUM_OUTPUTS);

        for(int i = 0; i < NUM_OUTPUTS; i++) {
            for_1[i] = dsp_filters_biquads((int32_t) from_usb[i],
                                          filter_coeffs0,
                                          filter_states0[i],
                                          FILTERS0,
                                          28);
        }

        // And forward answer to next stage
        chan_out_buf_word(c_to1a, &for_1[0], NUM_OUTPUTS);
        chan_out_buf_word(c_to1b, &for_1[0], NUM_OUTPUTS);
    }
}
    
```

DSP task 1A is implemented by `dsp_thread1a` and picks up data from the DSP task 0, and outputs data to dsp task 2:

```

#define FILTERS1a 2
//      b2/a0      b1/a0      b0/a0      -a1/a0      -a2/a0
static __attribute__((aligned(8))) int32_t filter_coeffs1a[FILTERS1a*5] = {
    261565110, -521424736, 260038367, 521424736, -253168021,
    255074543, -506484921, 252105451, 506484921, -238744538,
};

static __attribute__((aligned(8))) int32_t filter_states1a[NUM_OUTPUTS/2][
    ↪ FILTERS1a*4];

void dsp_thread1a(chanend_t c_from0,
                 chanend_t c_to2) {
    int from_0[NUM_OUTPUTS];
    int for_2[NUM_OUTPUTS/2];
    while(1) {
        // Pick up my chunk of data to work on
        chan_in_buf_word(c_from0, &from_0[0], NUM_OUTPUTS);

        for(int i = 0; i < NUM_OUTPUTS/2; i++) {
            for_2[i] = dsp_filters_biquads((int32_t) from_0[i],
                                         filter_coeffs1a,
                                         filter_states1a[i],
                                         FILTERS1a,
                                         28);
        }

        // And forward answer to next stage
        chan_out_buf_word(c_to2, &for_2[0], NUM_OUTPUTS/2);
    }
}

```

DSP task 1B is implemented by `dsp_thread1b` and picks up data from the DSP task 0, and outputs data to dsp task 2:



```

#define FILTERS1b 2
//      b2/a0      b1/a0      b0/a0      -a1/a0      -a2/a0
static __attribute__((aligned(8))) int32_t filter_coeffs1b[FILTERS1b*5] = {
    280274501, -523039333, 245645878, 523039333, -257484924,
    291645146, -504140302, 223757950, 504140302, -246967640,
};

static __attribute__((aligned(8))) int32_t filter_states1b[NUM_OUTPUTS/2][
    ↪ FILTERS1b*4];

void dsp_thread1b(chanend_t c_from0,
                 chanend_t c_to2) {
    int from_0[NUM_OUTPUTS];
    int for_2[NUM_OUTPUTS/2];
    while(1) {
        // Pick up my chunk of data to work on
        chan_in_buf_word(c_from0, &from_0[0], NUM_OUTPUTS);

        for(int i = 0; i < NUM_OUTPUTS/2; i++) {
            for_2[i] = dsp_filters_biquads((int32_t) from_0[i],
                                         filter_coeffs1b,
                                         filter_states1b[i],
                                         FILTERS1b,
                                         28);
        }

        // And forward answer to next stage
        chan_out_buf_word(c_to2, &for_2[0], NUM_OUTPUTS/2);
    }
}

```

Similarly, DSP task 2 is implemented by `dsp_thread2` and picks up data from the DSP tasks 1A and 1B, and outputs data to the distribution task. The weird part of the code is that we need to push some data into the output channel end prior to starting the loop - otherwise the `data_distribution` task would hang:



```

#define FILTERS2 1

static __attribute__((aligned(8))) int32_t filter_coeffs2[FILTERS2*5] = {
    291645146, -504140302, 223757950, 504140302, -246967641,
};

static __attribute__((aligned(8))) int32_t filter_states2[NUM_OUTPUTS][
    ↪ FILTERS2*4];

void dsp_thread2(chanend_t c_from1a, chanend_t c_from1b,
    chanend_t c_todist) {
    int from_1a[NUM_OUTPUTS];
    int from_1b[NUM_OUTPUTS];
    int for_usb[NUM_OUTPUTS];
    chan_out_buf_word(c_todist, &for_usb[0], NUM_OUTPUTS); // Sample -2
    chan_out_buf_word(c_todist, &for_usb[0], NUM_OUTPUTS); // Sample -1
    while(1) {
        // Pick up my chunk of data to work on
        chan_in_buf_word(c_from1a, &from_1a[0], NUM_OUTPUTS/2);
        chan_in_buf_word(c_from1b, &from_1b[0], NUM_OUTPUTS/2);

        for_usb[0] = dsp_filters_biquads((int32_t) from_1a[0],
            filter_coeffs2,
            filter_states2[0],
            FILTERS2,
            28);

        for_usb[1] = dsp_filters_biquads((int32_t) from_1b[0],
            filter_coeffs2,
            filter_states2[1],
            FILTERS2,
            28);

        // And forward answer to the distributor for completion
        chan_out_buf_word(c_todist, &for_usb[0], NUM_OUTPUTS);
    }
}

```

The distributor picks up data from the USB stack, posts it to DSP task 0, and picks up an answer from DSP task 2:

```

void dsp_data_distributor(chanend_t c_usb, chanend_t c_to0, chanend_t
    ↪ c_from2) {
    int for_usb [NUM_OUTPUTS + NUM_INPUTS];
    int from_usb [NUM_OUTPUTS + NUM_INPUTS];
    while(1) {
        // First deal with the USB side
        chan_in_buf_word( c_usb, &from_usb[0],
            NUM_OUTPUTS);
        chan_in_buf_word( c_usb, &from_usb [NUM_OUTPUTS], NUM_INPUTS);
        chan_out_buf_word(c_usb, &for_usb[0],
            NUM_OUTPUTS);
        chan_out_buf_word(c_usb, &for_usb [NUM_OUTPUTS], NUM_INPUTS);
        // Now supply output data to DSP task 0
        chan_out_buf_word(c_to0, &for_usb[0], NUM_OUTPUTS);
        // Now pick up data from DSP task 2
        chan_in_buf_word( c_from2, &for_usb [0], NUM_OUTPUTS);
    }
}

```

Finally, we need the code to start all the parallel threads. This code starts five tasks, and connects them up using six channels:

```

DECLARE_JOB(dsp_data_distributor, (chanend_t, chanend_t, chanend_t));
DECLARE_JOB(dsp_thread0, (chanend_t, chanend_t, chanend_t));
DECLARE_JOB(dsp_thread1a, (chanend_t, chanend_t));
DECLARE_JOB(dsp_thread1b, (chanend_t, chanend_t));
DECLARE_JOB(dsp_thread2, (chanend_t, chanend_t, chanend_t));

void dsp_main(chanend_t c_data) {
    channel_t c_dist_to_0 = chan_alloc();
    channel_t c_0_to_1a = chan_alloc();
    channel_t c_0_to_1b = chan_alloc();
    channel_t c_1a_to_2 = chan_alloc();
    channel_t c_1b_to_2 = chan_alloc();
    channel_t c_2_to_dist = chan_alloc();
    PAR_JOBS(
        PJOB(dsp_data_distributor, (c_data, c_dist_to_0.end_a, c_2_to_dist.
            ↪ end_b)),
        PJOB(dsp_thread0, (c_dist_to_0.end_b, c_0_to_1a.end_a, c_0_to_1b.
            ↪ end_a)),
        PJOB(dsp_thread1a, (c_0_to_1a.end_b, c_1a_to_2.end_a)),
        PJOB(dsp_thread1b, (c_0_to_1b.end_b, c_1b_to_2.end_a)),
        PJOB(dsp_thread2, (c_1a_to_2.end_b, c_1b_to_2.end_b, c_2_to_dist.
            ↪ end_a))
    );
}

```

In order to show how this code works, we show a diagram in Figure 6. Note that the distribution task is mostly idle; it only consumes very little processing in the beginning and the end of the sample-cycle. This means that five other threads can be used to soak up the available DSP.

9 Controlling

In order to control the DSP that you have inserted into the code (eg, volume control, equaliser settings), the easiest method is to store the settings in memory, and run an asynchronous thread that has access to those variables. This asynchronous thread could be controlled from an A/P (over, say, I2C or SPI), or it can interface directly with, for example, rotary encoders, push buttons, sliders, or a touch screen. The memory registers effectively become control registers. As long as one side writes and the other side reads this is thread safe.

This method of updating needs to be done with some care as the memory is updated asynchronously to the DSP. Updating a volume control is completely safe; it either takes effect this sample or the next. Updating the taps of an FIR filter is also safe, the worst that can happen is that for a sample it will use some of the old and some of the new parameters. When updating an IIR filter (a biquad), it can be damaging to update the biquad coefficients in the middle of the execution. In particular, it may make the filter unstable. This can be avoided by updating the biquad in small steps (which is generally a good idea because the internal state needs to settle too), or one can use a synchronous thread instead.

If using a synchronised thread, the idea is that the thread does not just update the variables, but it requests the variables to be updated by the DSP thread itself, at a time that is safe for

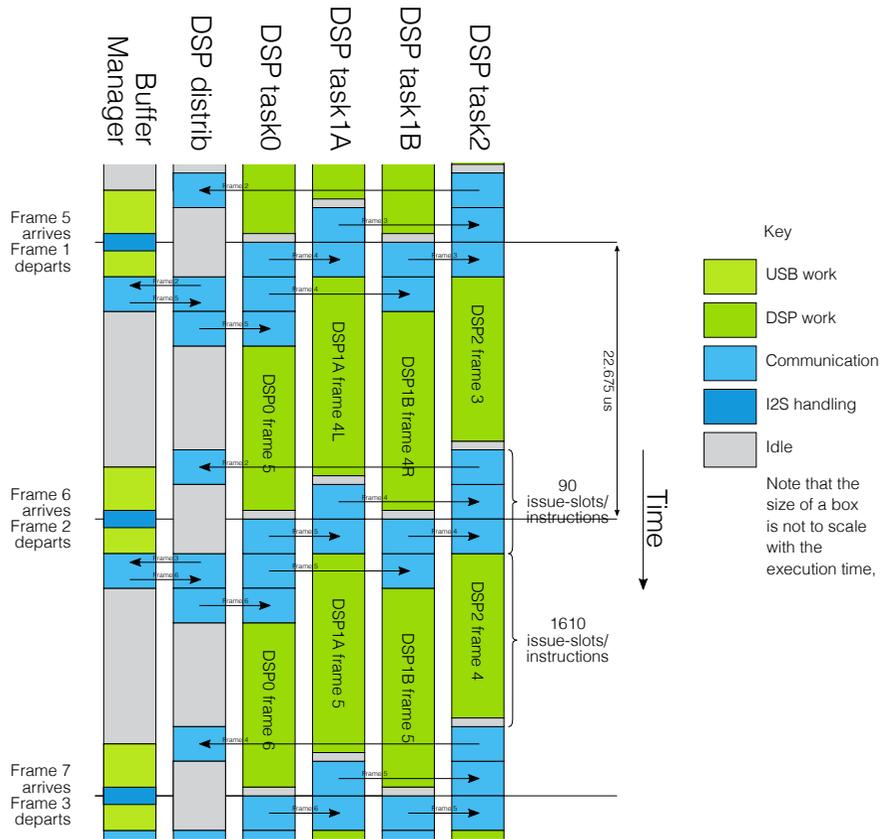


Figure 6:
Timeline of the pipelined example

the DSP. This will require a channel between the two threads and a protocol that causes the control-thread to request an update, and an answer from the DSP task when it is ready, whereupon the control-task posts the new filter coefficients that can be used by the DSP-thread.





Copyright © 2023, 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, xcore.ai, 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.

