XMOS

AN02011: USB Audio with concurrent multi-threaded DSP

Publication Date: 2025/10/10
Document Number: XM-006859-AN v1.1.0

IN THIS DOCUMENT

This application note describes how to implement a multi-threaded DSP system on the *XCORE*. As an example, it is integrated into the *XMOS USB Audio Reference Design*. Integration into other software stacks follows a very similar process.

The *USB Audio Reference Design* 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. As such, it provides a useful framework to support the DSP functionality that may include:

- Equalisation
- Mixing
- Dynamic range compression
- Audio effects

This application note discusses the recommended method of partitioning the DSP functionality between multiple *XCORE* threads, integrating them into a system with the audio interfaces using the USB audio stack API and tuning the DSP system in-situ using *xscope* to monitor signals in the DSP pipeline.

For reference, the developer may wish to use the following repositories:

- <https://www.xmos.com/file/sw_usb_audio>: the USB Audio reference design software
- <https://www.xmos.com/libraries/lib_xua>: the USB Audio library

1 Introduction

In its simplest case, DSP can be added to a program by simply adding a number of function calls. Each function call typically processes a sample and produces a sample. This application note starts with that case, and then extends to the case where so much signal processing needs to be done that multiple concurrent DSP tasks need to be fired off. The latter takes advantage of the multi-threaded nature of the *XCORE.AI* processor where each thread is allocated part of the processing bandwidth.

This process is called developing a DSP pipeline, and it consists of a number of distinct phases:

1. Capture - Assemble chains of DSP component functions using DSP library components optimised for the target micro-architecture with, where necessary, customised DSP elements.
2. Map - Partition the DSP pipeline between the *XCORE* threads.
3. Configure - Determine the filter coefficients to satisfy the frequency domain requirements.
4. Tune - Tune the DSP pipeline using representative signal samples

Typically there would a be a final phase to port the pipeline, however, the *XCORE* devices support the *XSCOPE* high-performance debugging interface with minimal impact on the real-time performance of the application under development. This enables the use of the *XCORE* itself for tuning by providing the ability to drive test samples and monitor any point in the DSP pipeline. For audio applications the high-performance of the *XSCOPE* interface enables continuous monitoring in real-time.

This on-device development environment offers the significant advantages of tuning the DSP pipeline in-situ with real data and eliminating risks associated with the porting of the DSP pipeline to the target device in addition to the use of custom DSP components without the development of an equivalent model for an external development environment.

1.1 Capture

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

- <https://www.xmos.com/libraries/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://www.xmos.com/libraries/lib_src> for synchronous and asynchronous sample rate conversion functions.
- <https://www.xmos.com/libraries/lib_audio_dsp> for audio effects functions (this is based on `lib_xcore_math` above).

These functions and components, along with any custom DSP components are assembled to form the DSP graph. At this point, the DSP graph can reside in a single thread; the performance will be limited to 20% of the performance from a single *XCORE* tile but can be tested with test data.

1.2 Map

The mapping process is a stage that is not necessary with a single-threaded, dedicated DSP device. However, DSP systems often contain multiple interfaces and the dedicated DSP device needs to implement a scheduler to process each individual data stream which is greatly simplified with a multi-threaded platform. More importantly, when the DSP pipeline interfaces between interfaces operating at different sample rates, this scheduling task is very complex but a multi-threaded *XCORE* platform can operate different threads in each clock domain.

Mapping the DSP pipeline to *XCORE* threads is a two stage process:

1. Split the DSP graph into sub-graphs for each sample clock domain if applicable. Interfaces between the sub-graphs are supported through the sample rate conversion component available in the `lib_src` library.

2. For each sub-graph, partition the elementary DSP components into further sub-graphs where the combined instruction count of the elements can be supported by the thread within the sample period.

DSP requires guaranteed, hard real-time execution and the unique multi-threaded micro-architecture of the *XCORE* pipeline along with the single-cycle access to on-chip primary memory, ensures that each instruction completes its path through the pipeline before executing the following instruction, eliminating data hazard and memory latency uncertainty. This means that it is straight forward to calculate the latency of the sub-graph.

Allowing a handful of instruction slots for each exchange of samples between threads, the number of instructions available for the DSP components is readily calculated.

Inside the thread, the DSP sub-graph is statically scheduled through the ordering of the individual DSP component functions. In order to link to the rest of the graph in the other threads, two additional functions are required: one to initiate communication by sending the requests through the channels, followed by another to wait for the response events and updating the state variables appropriately.

Each DSP component needs to start from a known state. The recommended state is where all sample history is set to zero. The code could start to execute at any point but computing new samples from the initialised state is redundant so it is recommended that each thread starts at the point of exchanging data with other threads. This ensures that the threads are primed for the samples from input interfaces as soon as they become available.

When designing a multi-threaded application with communication between the threads, care is required to avoid creating a deadlock which can arise from a dependency loop from data, code or limited resources.

Data dependencies are avoided by computing the output state from a defined input state, similar to hardware. Given an input state, a series of DSP functions are performed until the final function yields the output state. There are no data dependencies.

The channels that support communication between threads in an *XCORE* are lossless and therefore block. When a communication takes place over a channel between two threads, the code execution of the threads aligns around the data transfer but the ordering of communication in each thread must follow the correct order to avoid deadlock.

In each sample synchronous sub-system, data is exchanged between threads once per sample and we are free to choose the ordering that maximises performance.

The total time taken to complete the data exchange is less important than the time each thread requires to complete its communication; once a thread has completed its communication, it is free to process the next sample while other threads are still communicating. Consequently, the performance optimisation is obtained when the communication time for each thread is minimised, maximising the instructions available to compute the next sample.

In the USB Audio platform the communication from the interfaces to and from the DSP pipeline are adjacent and should, therefore, be consecutive. In general, for threads that have a single sample input and output, the optimal communication ordering will be to propagate the communication along the signal path but we are free to choose the direction which can be the opposite of the signal sample flow.

Exploiting parallel paths within the DSP pipeline allows the use of individual threads for each path, extending the width of DSP elements that can be accommodated within each thread. This reduces the latency of the DSP pipeline implementation.

1.3 Configure

Filter coefficients for, for example, FIR and IIR filters are determined from the required filter characteristics. *XMOS* provides a suite of Python scripts to map filter characteristics to data structures containing the filter coefficients.

The *XCORE* scalar pipeline supports a wide range of data-types including single precision floating point while the vector processing pipeline offers SIMD capabilities supporting eight 32-bit operations in each instruction but is limited to fixed point data formats. Consequently, to maximise the performance, fixed-point data-types should be used where possible. This is also desirable as the interfaces on either side of the pipeline (I2S, TDM, ADAT, S/PDIF, USB, AVB) typically use fixed point formats. While the dynamic range of the signal can be different at each stage of the DSP pipeline, it is static and can be readily accommodated by gain terms.

1.4 Tune

The XSCOPE debugging interface can write data structures to memory in the XCORE as well as stream signals from points along the DSP pipeline. The bandwidth available through the XSCOPE interface is sufficient to monitor several signals in real time for audio applications.

A monitoring point is selected by adding the code:

```
xscope_int(channel, signal);
```

where required in the DSP code. The debugger generates a standard VCD file for the signals to be viewed through the user's preferred VCD viewer.

1.5 Integration

The flexibility of the threads in the *XCORE* enable single device embedded solutions by integrating a diverse range of compute requirements including DSP, IO, Control and AI inference models.

In most cases, the peripheral interfaces will drive the DSP pipeline and define the sample rate. These interfaces can be considered to behave in the same way as any other DSP component and can be integrated into the DSP pipeline like a DSP element.

In an embedded application there will be a control layer which will take responsibility for controlling the operation of the DSP pipeline by setting its parameters, for example, the control layer may be controlling the display and user input peripherals in order to control the characteristics of the DSP pipeline.

An efficient method of acquiring the necessary interfaces is to extend one of the *XMOS*' existing platforms such as the USB Audio platform.

2 DSP Pipelines

This section summarises the principles of what a DSP pipeline looks like typically. It is assumed that the reader is familiar with DSP. A typical pipeline is shown in Fig. 1. In this pipeline digital samples enter the pipeline (on the left-hand-side in this case), flow through a series of DSP blocks, and eventually samples leave the pipeline (on the right-hand-side in this case).

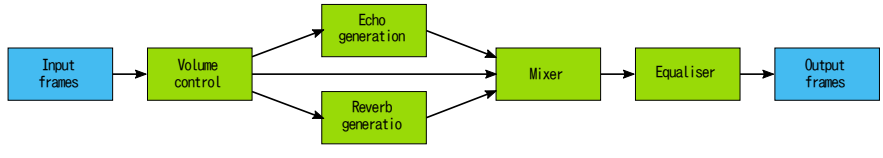

Fig. 1: Typical pipeline of DSP operations

The following terminology is used to describe these systems:

- Samples enter the pipeline at a sample-rate. This sample rate may be 48,000 Hz for an audio pipeline. Samples also exit the pipeline at a sample rate, and each of the blocks pass data along at a sample rate. Where the sample rate on input and output is not the same, typically a sample-rate-conversion is had. This document assumes that all sample-rates are synchronous to a single clock, and therefore all sample rate conversions are synchronous
- There may be multiple channels. For example, a system may be mono (single channel), stereo (two channels) or 7.1 (eight channels). We call one sample on each channel a *frame*, and hence the word frame-rate may be used interchangeably with the word sample-rate.
- Frames may be blocked for efficiency or for algorithm considerations. For example, it may be chosen that the DSP pipeline operates on blocks of 48 frames. If the frame-rate is 48,000 Hz that means that the DSP pipeline operates at a rate of 1,000 Hz (1 ms). Blocking may improve efficiency but will also increase latency through the pipeline. Blocking may be unavoidable for certain algorithms such as an FFT.
- Samples are stored in a data-format, for example, 16-bit int, 24-bit int, or float. This document is agnostic to the data-format used, but it is generally advised that for all integer formats data is stored in the higher-order-bits of a 32-bit word (that makes all modules agnostic to the particular input format). Floating point is also possible but usually not as efficient.

Given a pipeline of DSP operations, these can be trivially mapped onto a single C-function where each statement implements one of the blocks of the DSP pipeline (by calling an appropriate DSP function), and data is passed from one function to the next through the use of an array of data where the array is large enough to hold a block of frames. Each function may require some local state (e.g., biquads, FIRs) that is passed along subsequent iterations.

This document uses those functions as a starting point, and discusses how to integrate those functions on an *XCORE*, and how to split those functions up over multiple threads.

3 Introduction to USB Audio

The basic structure or USB Audio is shown in Fig. 2.

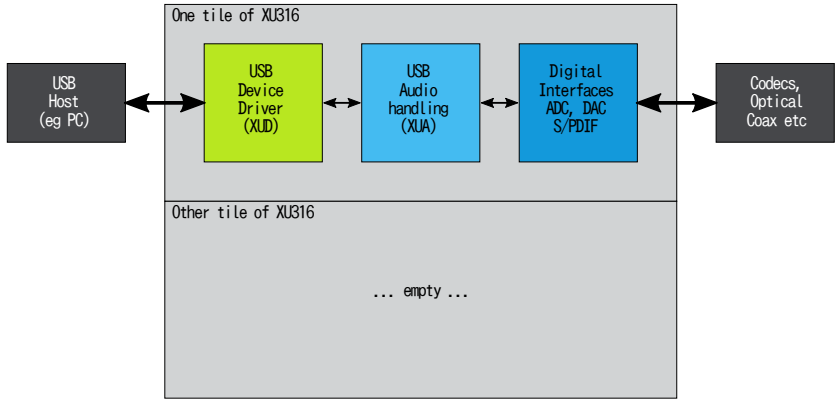

Fig. 2: Structure of USB Audio

On the left is a USB interface to the host - this is dealt with by the **lib_xud** (*XMOS* USB Device) and **lib_xua** (*XMOS* USB Audio) libraries. XUD is the low level USB library for *XCORE*, 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 host computer) that is called the input path. The terms input-path and output-path are host-centric names, consistent with the USB specification(s) nomenclature.

The *XMOS xcore.ai* 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.

3.1 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 **NUM_OUTPUTS** and **NUM_INPUTS** are used throughout this code to refer to the number of output audio-channels (**XUA_NUM_USB_CHAN_OUT**) and the number of input audio-channels (**XUA_NUM_USB_CHAN_IN**).

The **UserBufferManagement()** function is called at the sample rate of the USB Audio stack (e.g., 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 transferred to the audio interfaces, the second array carries all the data from the audio interfaces that shall be transferred to the USB host. A developer may choose 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 can be overridden is:

```
extern void UserBufferManagementInit(unsigned int curSampFreq);
```

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).

This example only modifies the output path - **NUM_OUTPUTS=2** and **NUM_INPUTS=4**. The output_samples can be run 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(
```

(continues on next page)

(continued from previous page)

```
    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(unsigned int curSampFreq) {}
```

If desired, `input_samples` and `output_samples` can be combined in order to mix data from interfaces or USB into USB or the audio 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 select, on the host, the sample rate that is to be used. For simplicity, this application note does not discuss sample-rate changes; it is assumes that there is just one, fixed, sample-rate.

This application note uses, as an 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.

3.2 Timing requirements

The *XMOS* USB Audio Reference Design is designed to operate on single samples in order to minimise latency introduced by the audio stacks. The `UserBufferManagement()` function is called from 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 it's 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 i.e. system frequency / thread count (for example 600 / 8 = 75 MHz) and the sample rate (say, 44.1 KHz) one can calculate the number of instruction issues 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 Fig. 3.

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

As *XCORE* is a concurrent multi-threaded multi-core processor, there are other threads and cores/tiles 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 with just I2S, USB Audio uses around 30% of the compute, with one core/tile being completely empty.

Initially using a single thread on the other tile for DSP is examined, then how to generally parallelise DSP, and then finally using multiple threads for DSP is examined.

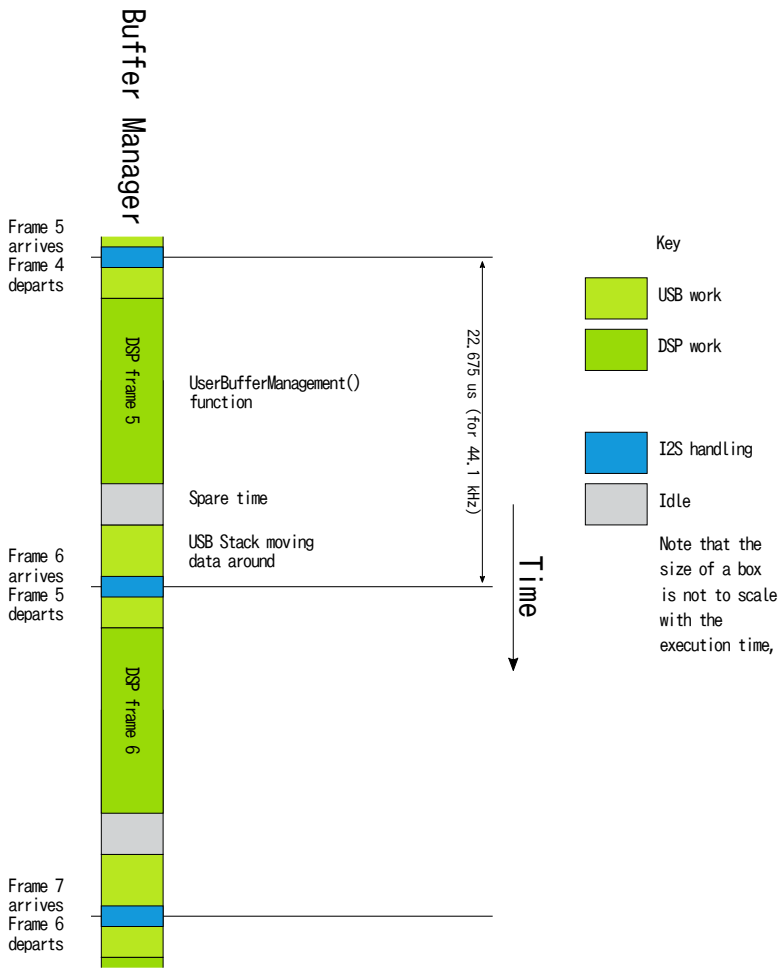

Fig. 3: Timeline of executing DSP inside a thread

4 Executing the DSP on the other physical core

The *XCORE* architecture offers a communication fabric to efficiently transport data between threads and between tiles/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 device, or on different devices in the same system; communication always works, but performance is lower when the physical distance increases.

A channel is analogous to 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`

XC language could be used instead of *C* and `lib_xcore`; the resulting behaviour is identical. There is equivalent functions `chanend_*` that create streaming channels rather than synchronised channels. These are not used in this this application note, but they can be useful where extra performance and predictability are required.

Typical code to off-load the DSP to the other core 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:

```
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]: {                                   \
        board_setup();                              \
        xk_audio_316_mc_ab_i2c_master(i2c);         \
    }                                               \
    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) {
```

(continues on next page)

(continued from previous page)

```
    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 in Fig. 4.

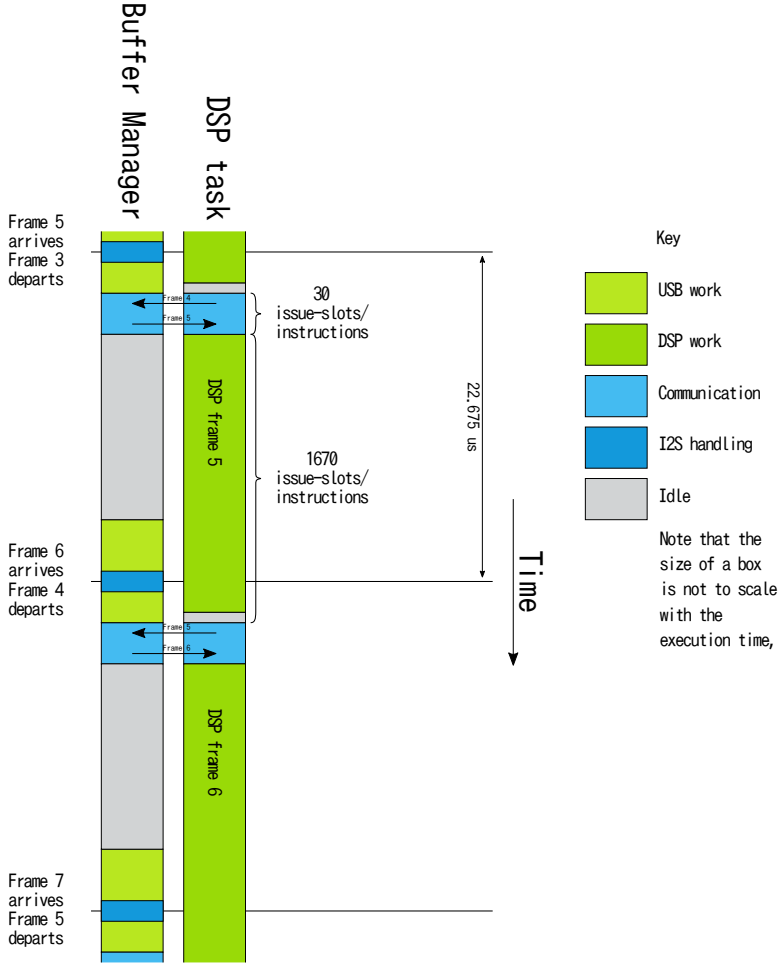

Fig. 4: Timeline of executing the two concurrent threads

Time progresses from top to bottom and what is depicted is 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. Note that the boxes are not drawn to scale otherwise some of them would be too small to be visible.

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, a 44,100 Hz sample rate is assumed. 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.

5 Parallelising DSP

Parallelisation involves splitting work into a multitude of *tasks*. *Tasks* can then be mapped onto threads. The reason to distinguish between these two words is that a *task* is a software concept: a set of instructions that performs some meaningful operation, for example a shelf-filter. If, say, ten tasks exist then five of them can be executed in *Thread 1* and five of them in *Thread 2* then 2x parallelism has been achieved.

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.

6 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 in Fig. 5.

As previously, channels are used to communicate between the DSP tasks, what is new is that to these DSP tasks need to be created as well as the channels between them. The only difference is in the `dsp_main()` function.

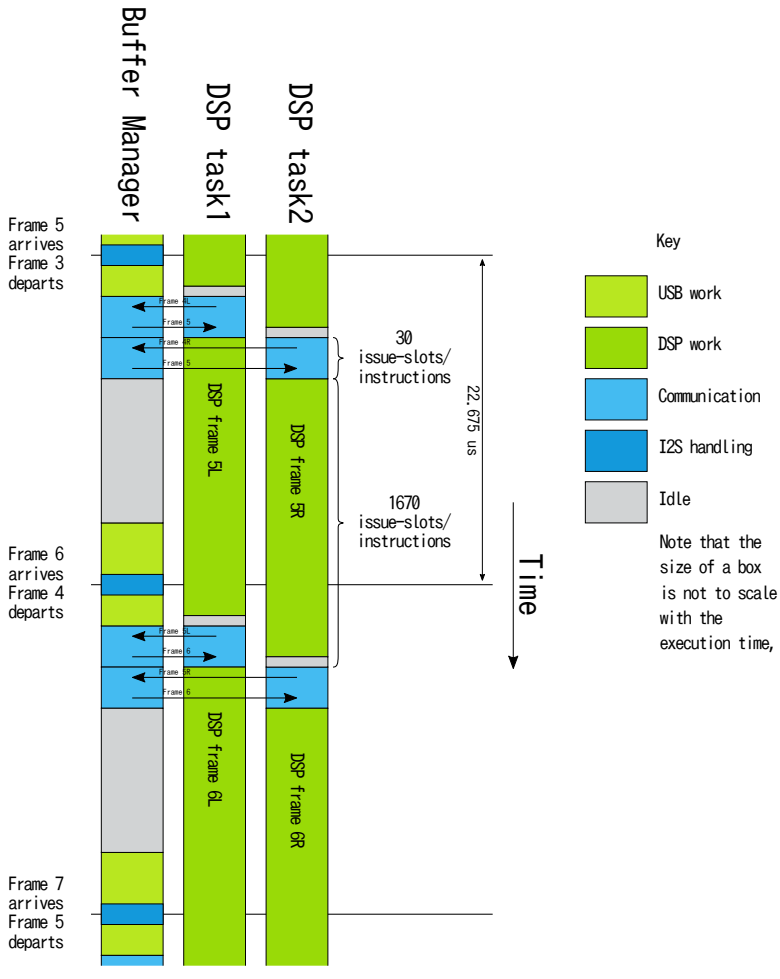

Fig. 5: Timeline of executing the two concurrent threads

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;
}
```

(continues on next page)

(continued from previous page)

```
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]: {                                         \
        board_setup();                                    \
        xk_audio_316_mc_ab_i2c_master(i2c);               \
    }                                                     \
    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 `UserBufferManagement()` 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.

7 Data Pipelining DSP

An arbitrary pipeline of DSP processes can be made 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 Fig. 6.

The pipeline being built requires some additional setup to make everything work, but the code itself is otherwise straightforward.

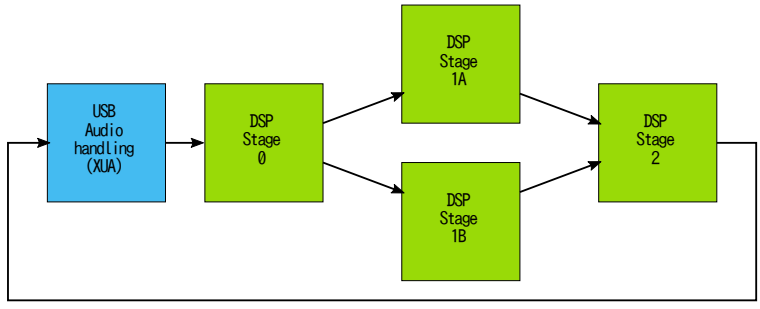

Fig. 6: Example pipeline

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

```
#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 deliver my answer back
        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);
```

(continues on next page)

(continued from previous page)

```
        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 deliver my answer back
        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 deliver my answer back
        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 deliver my answer back
        chan_out_buf_word(c_todist, &for_usb[0], NUM_OUTPUTS);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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, &from_usb[0], NUM_OUTPUTS);
        // Now pick up data from DSP task 2
        chan_in_buf_word( c_from2, &for_usb[0], NUM_OUTPUTS);
    }
}
```

Finally, code to start all the parallel threads is required. 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))
        );
}
```

A diagram shows complete operation is shows in Fig. 7. Note that the distribution task is mostly idle; it consumes very little processing in the beginning and the end of the sample-cycle. This means that five other threads can be used to use up the available DSP.

8 Optimised Data Pipelining DSP

Since there is flexibility to choose the order of the communication processes in a sample synchronous system, the DSP distribution thread can be eliminated.

The communication timing is based on the communication from the Buffer Manager thread. DSP task0 receives a sample from the Buffer Manager thread. It then sends data to DSP task 1A and DSP taskB in that order which is a sequence of three sequential communication processes. DSP task2 sends data to the Buffer Manager thread then receives data from DSP task1A and DSP task1B in that order which is, again, a sequence of three sequential communication processes. The communication to and from each of DSP task1A and DSP task1B are adjacent. This pipeline completes the communication to and from each thread in the minimum possible time, maximising the instructions available for the computation of the next samples.

The pipeline that we're building is shown in Fig. 8.

The pipeline being built requires a bit of setup to make it function 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:

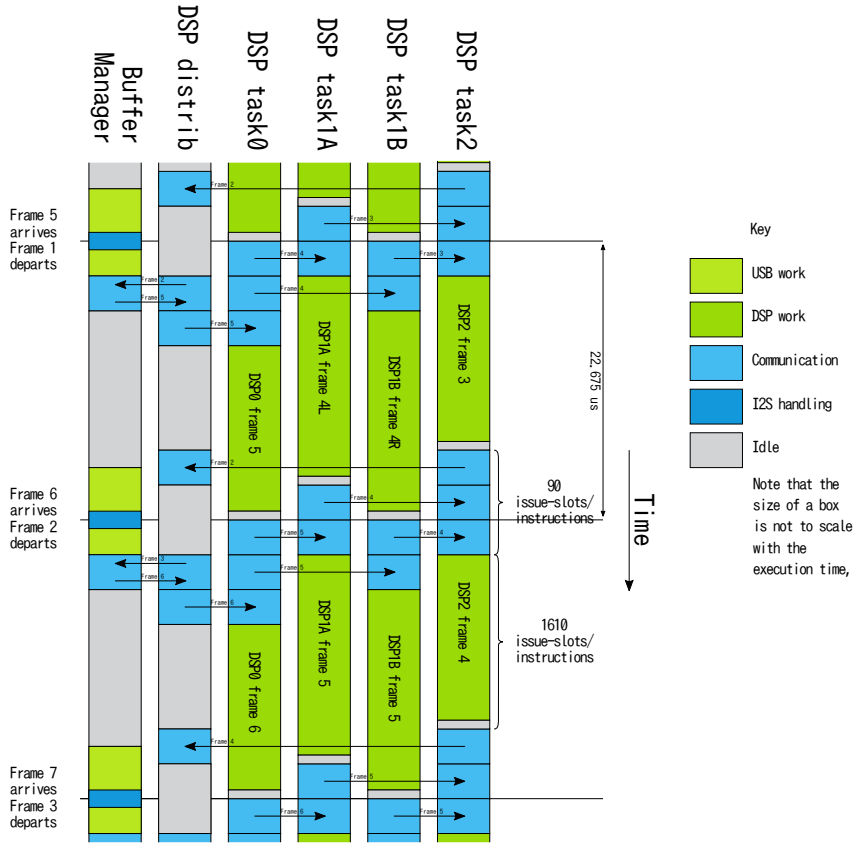

Fig. 7: Timeline of the pipelined example

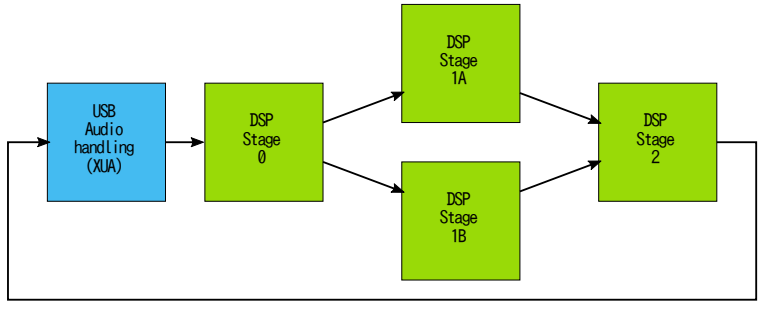

Fig. 8: Example pipeline

```
#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 deliver my answer back
        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 deliver my answer back
        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,
```

(continues on next page)

(continued from previous page)

```
                                        28);
        }

        // And deliver my answer back
        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 t 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 deliver my answer back
        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, &from_usb[0], NUM_OUTPUTS);
        // Now pick up data from DSP task 2
        chan_in_buf_word( c_from2, &for_usb[0], NUM_OUTPUTS);
    }
}
```

Finally, code is required to start the parallel threads. This code starts five tasks, and connects them 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();
```

(continues on next page)

(continued from previous page)

```
    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))
        );
}
```

The diagram in Fig. 9 depicts complete operation.

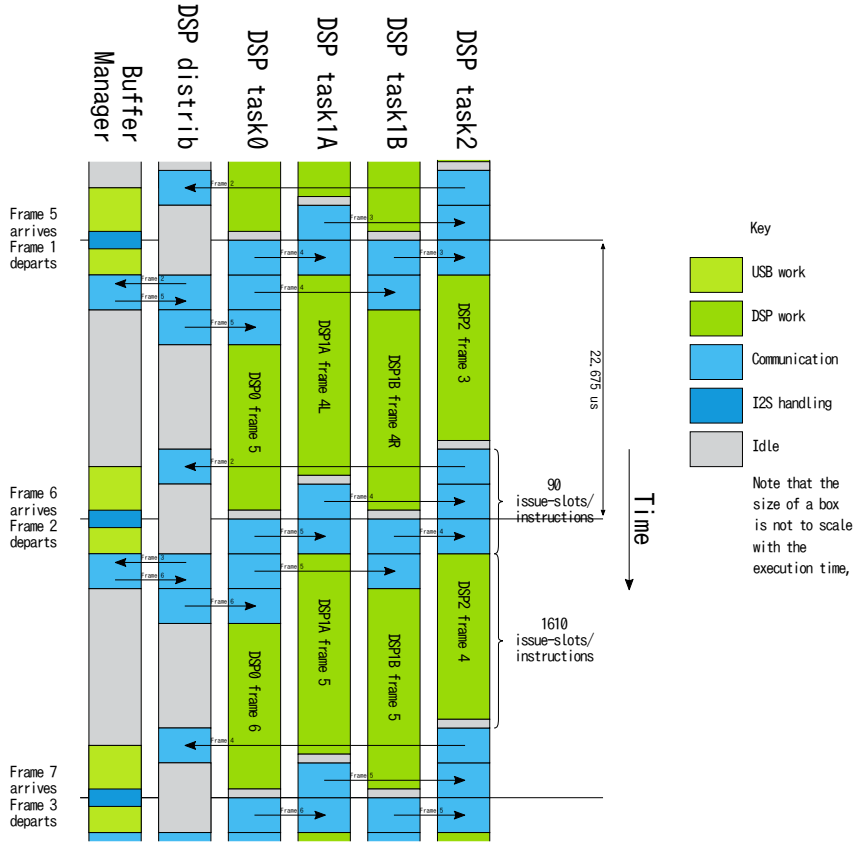

Fig. 9: Timeline of the pipelined example

9 Control

In order to control the DSP that has been inserted into the code (e.g., volume control, equaliser settings), it is required to be able to control values in the various DSP components. Often this control happens asynchronously to the data pipeline, for example, a developer may use a touch-screen to change the settings of an equaliser pipeline, as this change happens outside the audio domain it is intrinsically asynchronous to it. Deciding on how to synchronise the control with the audio stream affects how this is encoded in the solution.

Factors affecting the decision on synchronisation may include:

- The stability of the algorithms used. In particular, algorithms that use a feedback loop such as an IIR may exhibit undesirable behaviour
- Whether all elements of the pipeline are updated simultaneously or not
- Whether all settings of a single algorithm are updated simultaneously or not.
- The output of the DSP pipeline as a whole.
- The desired speed at which the controls take effect.

A number of scenarios on how to update control-values is discussed, concluding with a comparison and trade-offs to be made.

9.1 Control values directly in unguarded shared memory

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 variables in memory effectively become control registers. As long as only one thread writes and another one only reads this is thread-safe.

For many applications this is an adequate solution. For example, when changing a mixer setting, it is not typically a concern that the setting is changed just before a sample is processed or just after a sample is processed. However, in the case of changing the values controlling an IIR, this method may not be adequate. The b0, b1, b2, a0, a1, and a2 values ought to be all changed simultaneously, as changing one value first may cause the IIR to behave in unpredictable ways.

9.2 Control values in guarded shared memory

A further step is to place the values in shared memory, but to explicitly guard their use. Using a lock for guarding it is not appropriate due to the real-time nature of the data-pipeline. However, one can use one or more memory cells to state which set of parameters is now valid, for example using a pointer or an index in an array. It is essential that both the old values and the new values are available for some period of time, enabling the pipeline to make the choice whether to apply the old or the new values.

This method still updates values asynchronously, but it is now in the hands of the DSP pipeline whether to use the old or the new values. For example, a component that implements a bank of IIR filters may on receiving the new frame of data also lookup which set of control values to use. It is either using or old values or all new values. It will be up to the control thread to not make sudden changes that would destabilise the state of the filter-bank, but there is a guarantee that all filter values are applied synchronously.

9.3 Passing control values along the DSP pipeline

A preferred method to solve this problem is to pass the control parameters along the DSP pipeline together with the DSP samples. They can be passed by value or by reference, i.e., a single pointer or even a single byte would be sufficient to inform each stage of the pipeline as to what control parameters to use.

With this method, each sample is processed using a known set of control parameters and the parameters are applied as a wave running through the DSP pipeline. The resynchronisation of the control settings happens only once on entry to the pipeline. This makes the pipeline itself operate synchronous with the control values.

9.4 Comparison of control methods

In all cases control values have to be distributed over the various DSP components; this can always take place through shared memory. The difference is the method by which the DSP component knows which settings to use. In one extreme the DSP component uses directly the only settings that it can observe; on the other extreme, the DSP is given directions to use a specific set of settings with each frame of audio data that arrives. The other methods offer gradually more control over the synchronisation between the audio pipeline and the control settings.

10 Example application

10.1 Build configurations

The example application provides four build configurations, these are:

- `PIPELINE`
- `MULTI_THREAD`
- `SINGLE_THREAD`
- `USB_THREAD`

Each build configuration produces an executable binary.

10.2 Building the example

This section 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 here.

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 `an02011` 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 an02011
cd app_an02011
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.

The application binaries should then be built using `xmake`:

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

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `app_an02011/bin` directory — one for each supported build configuration.

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.

10.3 Running the example

From an XTC command prompt, the following command should be run from the `an02011/app_an02011` directory:

```
xrun ./bin/<BUILD CONFIG>/app_an02011.xe
```

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

```
xflash ./bin/<BUILD CONFIG>/app_an02011.xe
```

11 Further reading

- *XMOS* XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest

XMOS

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, 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.