DESIGNING MULTITHREADED AND MULTICORE AUDIO SYSTEMS

HENK L MULLER

XMOS Ltd, Bristol, UK
henk@xmos.com
http://www.xmos.com/

Digital audio represents data as bit patterns. As such they are easily transmitted and stored either for archiving purposes (on hard disk), or for temporary purposes ("buffering" audio data). Buffering data has the undesired effect of delaying audio, and should hence be avoided whereever possible. In this article we present a design tactic for digital audio systems that avoids buffering and that relies on the predictable nature of underlying hardware to deliver data just-in-time. We show how to use this tactic, how to compute minimum buffer sizes required, and how to scale the design to larger systems.

INTRODUCTION

In this article we discuss how to use a multithreaded and multicore design methods to design digital audio systems. Digital audio systems operate on streams of data; this enables the system designer to split the design in parts, and reason about the delay and bandwidth of each part.

Multicore and multithreading are efficient methods to design real-time systems in general, but audio systems in particular. Multithreaded and multicore design views a system as a collection of many tasks that operate independently, and that communicate with each other when required. Breaking the system design down from large monolithic blocks of code into much more manageable tasks greatly simplifies system design and speeds product development. As a result, real-time properties of the system as a whole are more easily understood, and the designer only has to worry about the fidelity of the implementation of each task. For example, “Is the feedback algorithm implemented correctly?”.

We first briefly summarise the notions of digital audio, multicore, and multithreading, before showing how to effectively use multicore and multithreading to design the buffering schemes for digital audio systems. We use several digital audio systems to illustrate the design method, including Asynchronous USB-Audio 2, and AVB over Ethernet.

DIGITAL AUDIO

Digital audio has taken over from analog audio in many consumer markets for two reasons. First, most audio sources are digital. Whether delivered in lossy compressed form (MP3) or in uncompressed formats (CD), digital standards have taken over from the traditional analog standards such as cassettes and tapes. Second, digital audio is easier to deal with than analog audio. Data can be transferred lossless over existing standards, such as TCP/IP or USB, and the hardware design of that part does not need any “black magic” to keep the noise floor down. As far as the digital path is concerned, the noise floor is constant and immune from, for example, TDMA noise that mobile phones may cause. In particular, there is little need for expensive cables with specific analogue characteristics.

A digital audio system operates on *streams of samples*. Each sample represents the amplitude of one or more audio channels at a point in time, with the time between samples being governed by the *sample rate*. CD standards have two channels (left and right) and use a sample rate of 44,100 Hz. Common audio standards use

2, 6 (5.1) and 8 (7.1) channels, and samples rates of 44.1 kHz, 48 kHz, or a multiple. We use 48 kHz as a running example, but this is by no means the only standard that can be supported.

MULTICORE AND MULTITHREADING

In a multithreaded design approach, a system is expressed as a collection of concurrent tasks. Using concurrent tasks rather than using a single monolithic description has several advantages:

- Multiple tasks are a good way to support separation of concerns, one of the most important aspects of software engineering.

  Separation of concerns means that different tasks of the design can be individually designed, implemented, tested and verified. Once the interaction between the tasks has been specified, teams or individuals can each get on with their own task.

- Concurrent tasks provide an easy framework to specify what a system should be doing. For example, a digital audio system will play audio samples that are received over a network interface. In other words, the system should *concurrently* perform two tasks: receive data from the network interface and play samples on its audio interface. Expressing these two tasks as a single sequential task is confusing.

A system that is expressed as a collection of concurrent tasks can be implemented by a collection of threads in one or more multithreaded cores. We assume that threads are scheduled at instruction level, such as is the case on an XMOS XCore processor, because that enables concurrent tasks to operate in real-time. Note that this is different from multithreading on, for example, Linux where threads are scheduled on a uni-processor with context switching. This may make those threads appear concurrent to a human being, but not to a collection of real-time devices.

Concurrent tasks are logically designed to communicate by message passing, and when two tasks are implemented by two threads, they communicate by sending data and control over *channels*. Inside a core, channel communication is performed by the core itself, and when threads are located on separate cores, channel communication is performed through switches. This is visualised in Figures 1 and 2, which shows a system comprising three cores in two packages.

Multithreaded design has been used by embedded system designers for decades. To implement an embedded system, a system designer used to employ a multitude of micro-controllers. For example, inside a music player one may have found three microcontrollers controlling flash, the DAC, and an MP3 decoder chip.

We argue that modern day multithreaded environments offer a replacement for this design strategy. A single multithreaded chip can replace a number of MCUs and provide an integrated communication model between tasks. Instead of having to implement bespoke communication between tasks on separate MCUs, the system is implemented as a set of threads that communicate over channels.

Using a multithreaded design approach enables the designer to reuse parts of their design in a straightforward manner. In traditional software engineering, functions and modules are combined to perform complex tasks, but this method does not necessarily work in a real-time environment because executing two functions in sequence may break the real-time requirement of either.

In an ideal multithreaded environment composition of real-time tasks is trivial, as it is just a case of adding a thread (maybe a core) for every new real-time task. In reality, the designer will have constraints on the number of cores (for example because of a limitation on the BOM cost) and will hence have to make a decision which tasks to compose as concurrent threads, and which tasks to integrate in a single thread as a collection of functions.

MULTITHREADED DIGITAL AUDIO

In the remaining sections we look at how to design a digital audio system using a multithreaded design approach. In particular we will look at how to avoid buffering, and how to design a low latency system.

A digital audio system is easily split into multiple threads, for example, a network protocol stack thread, a clock recovery thread, an audio delivery thread, and optionally, threads for DSP, device upgrade, driver authentication etc. The network protocol stack may be as complex as an Ethernet TCP/IP stack and comprise multiple concurrent tasks, or as simple as an S/PDIF receiver.

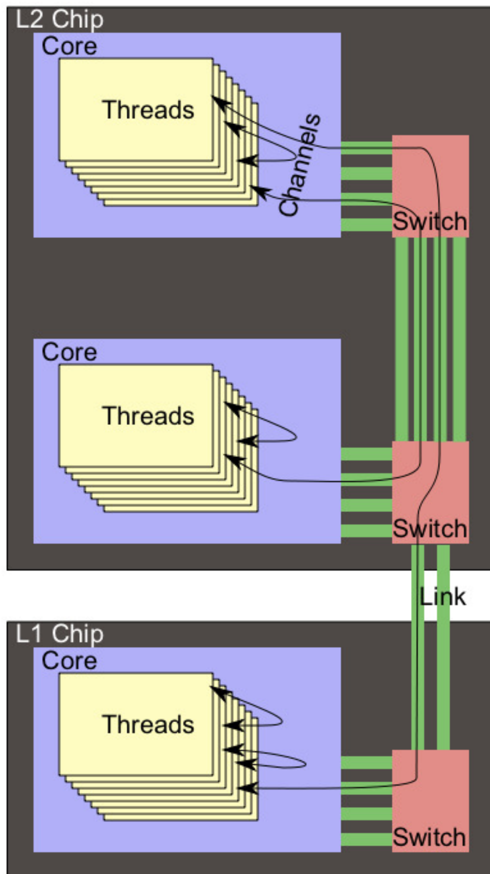

Figure 1: Diagram showing cores, threads, channels and switches

Figure 2: Physical incarnation of Figure 1

We assume that the threads in the system communicate by sending *data samples* over communication-channels. Whether the threads execute on a single core or on a multicore system is not important in this design method, multicore just adds scalability to the design. We assume that the computational requirements for each thread can be established statically and are not data-dependent, which is normally true for uncompressed audio.

We will focus our attention on two key parts of the design: buffering between threads (and their impact on performance) and embedding the design in a complete system, requiring us to perform clock recovery. Once the tasks have been split and the buffering scheme has been designed, implementing the inside of each thread follows normal software engineering principles, and is as hard or easy as one would expect.

We pick out buffering and clock recovery because they have a qualitative impact on the user experience (facilitating stable low latency audio) and are easy to express in a multithreaded programming environment.

BUFFERING: THE CORE OF THE SYSTEM

Within a digital solution data samples are not necessarily transported at the time that they are to be delivered. This requires digital audio to be *buffered*.

The design challenge is to establish the right amount of buffering. In an analogue system, buffering is not an issue: the signal is delivered on time. In a digital system designed on top of a non-real-time O/S, programmers usually stick in a reasonably large buffer (say 250 or 1000 samples) in order to cope with uncertainties in scheduling policies. However, large buffers are costly in terms of memory, in terms of adding latency and in

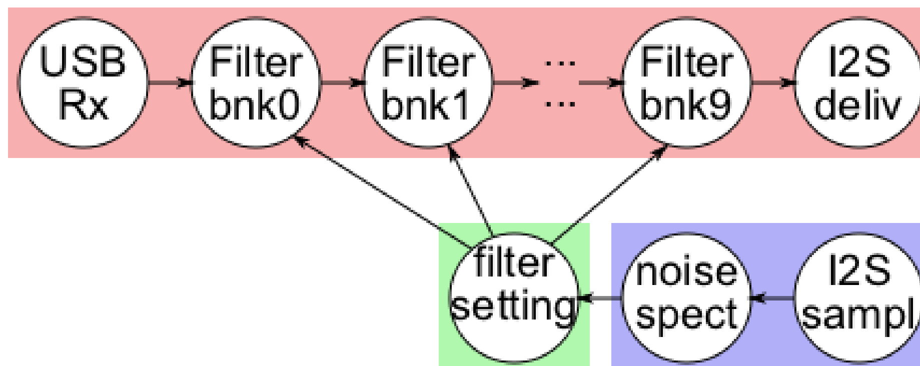

Figure 1: The threads grouped together based on their frequency

terms of proving that they are large enough to guarantee click-free delivery.

As an example on how to design with little buffering, consider a USB-2 speaker with a 48 kHz sample rate. The USB layer will transport a burst of six samples in every 125 us window. There is no guarantee at which time in the 125 us window the six samples will be delivered, hence a buffer of at least 12 samples is required in order to guarantee that samples can be streamed out to the speaker in real-time; assuming that all samples are dealt with without delay.

Multithreaded design provides a good framework to both informally and formally reason about buffering and avoids unnecessarily large buffers. In order to illustrate the reasoning we augment the above USB speaker with an ambient noise correction system. This system will comprise the following threads:

- A thread that receives USB samples over the network.
- A series of, say, 10 threads that filter the stream of samples; each with a different set of coefficients.
- A thread that delivers a filtered output sample to the stereo codec using I2S.
- A thread that reads samples from a codec connected to a microphone sampling ambient noise.
- A thread that subsamples the ambient noise to an 8 kHz sample rate.
- A thread that establishes the spectral characteristics of the ambient noise.
- A thread that changes the filter coefficients based on the computed spectral characteristics.

All threads will operate on some multiple of the 48 kHz base period. For example, each of the filtering threads will filter exactly one sample every 48 kHz period; the delivery thread will deliver a sample every period. Each of the threads also has a defined window over which it operates, and a defined method by which this window is advanced. For example, if our filter thread is implemented using a Biquad, then it will operate on a window of three samples that is advanced by one sample every period. The spectral thread may operate on a 256 sample window (to perform an FFT) that is advanced by 64 samples every 64 samples.

One can now establish all parts of the system that operate on the same period and compose these together in synchronous parts. No buffers are required inside those synchronous parts, although if threads are to operate in a pipeline, single buffers are required. Between the various synchronous parts buffers are required. In our example, we end up with three parts:

1. The part that receives samples from USB. This part filters and delivers samples at a rate of 48 kHz.
2. The part that samples ambient noise. This part samples at 48 kHz and delivers samples with a rate of 8 kHz.
3. The part that establishes the spectral characteristics and changes the filter settings.

This part receives data at 8 Khz and changes the settings at 125 Hz

These three parts are shown graphically in Figure 3.

The first part that receives samples from the USB buffer needs to buffer 250 us worth of samples, plus one sample: 250 us to cover two USB microframes, and one sample overlap. Given a 48 Khz sample rate this translates to a 13 sample buffer. A few extra samples buffer can be added to cope with medium term jitter in the clock recovery algorithm. The part that delivers needs to buffer one stereo sample. Operating the 10 filter threads as a pipeline requires 11 buffers. All threads in this part operate at 48 kHz. The second part that samples ambient noise needs to store one sample on the input side and six samples for subsampling, hence there is a seven sample delay at 48 kHz, or 145 us. The third part that establishes the spectral characteristics needs to store 256 samples, at an 8 kHz sample rate.

No other buffers are required hence the delay between ambient noise and filter correction is 256 samples at 8 kHz and 145 us for the sub sampling, or just over 32 ms. The delay in the audio delivery is 25 sample times, or just over 500 us. Note that these are minimum buffer sizes for the algorithm and thread partitioning that we have chosen to use; if this latency is unacceptable, a different algorithm or different thread partition has to be chosen. For example, we can choose to glue together threads that implement the DSP, and reduce the buffer count between the filter elements.

There is often a temptation to design the threads to operate on blocks of data instead of single samples, but that will increase the overall latency experienced, increase the memory requirements, and increase complexity. This should be considered only if there is a clear benefit, such as an increased throughput.

Note that this number of buffers is independent of the number of cores used to implement the system. Splitting the design over cores will add some marginal extra latency for inter-core communication; hundreds of microseconds.

CLOCKING DIGITAL AUDIO

A big difference between digital and analog audio is that digital audio is based on this underlying sample rate and digital audio requires a clock signal to be distributed to all parts of the system. Although components can all use different sample rates (for example, some parts of the system may use 48 kHz and some other parts may use 96 kHz with a sample rate conversion in between), all components should agree on the length of a second, and therefore agree on a basis to measure frequencies. The measurements above simply counted integral numbers of samples and assumes that all threads run at an identical rate. In fact, all threads can execute *data-driven* as long as the three threads that interface with the outside world operate synchronously.

All threads inside the system are agnostic to clock frequency, and it does not matter if multiple cores in the system use different crystals, as long as they operate on whole samples. However, at the edges of the system, the true clock frequency is important and we must ensure that the edges operate synchronously.

In a multithreaded environment, we will set a thread aside to explicitly measure the true clock sample rate that the signal wishes to use, to implement the clock recovery algorithm, and to measure the local clock versus the global clock and agree with a master clock on the clock offset. The latter enables the master to deliver samples synchronously to multiple devices (eg, multiple AVB speakers) and synchronously with other media such as a TV screen.

The clock may be measured implicitly using the underlying bit-rate of interconnects such as S/PDIF or ADAT. Measuring the number of bits per second on either of those networks will give a measure for the master clock. The clock may be measured explicitly by using protocols designed for this purpose such as PTP over Ethernet. The latter also have facilities to measure latency and mitigate against latency.

The clock recovery thread itself is a control loop that estimates the clock frequency and that adjusts the clock based on the error observed. In its simplest form, the error is used as a metric to adjust the frequency, but filters can be used to reduce jitter. This software thread implements what would have been traditionally been performed by a PLL, but in software and hence it can be adjusted to the environment cheaply. If a hardware master clock is required, an analogue PLL is required to generate a local low-jitter audio master audio clock.

CONCLUSIONS

A multithreaded development method enables digital audio systems to be developed using a divide-and-

conquer approach, where a problem is split into a set of concurrent tasks that are each executed in a separate thread on a multithreaded core.

Like many real time systems, digital audio lends itself to a multithreaded design method because digital audio systems obviously consist of a group of tasks that work on data and also require those tasks to execute concurrently.