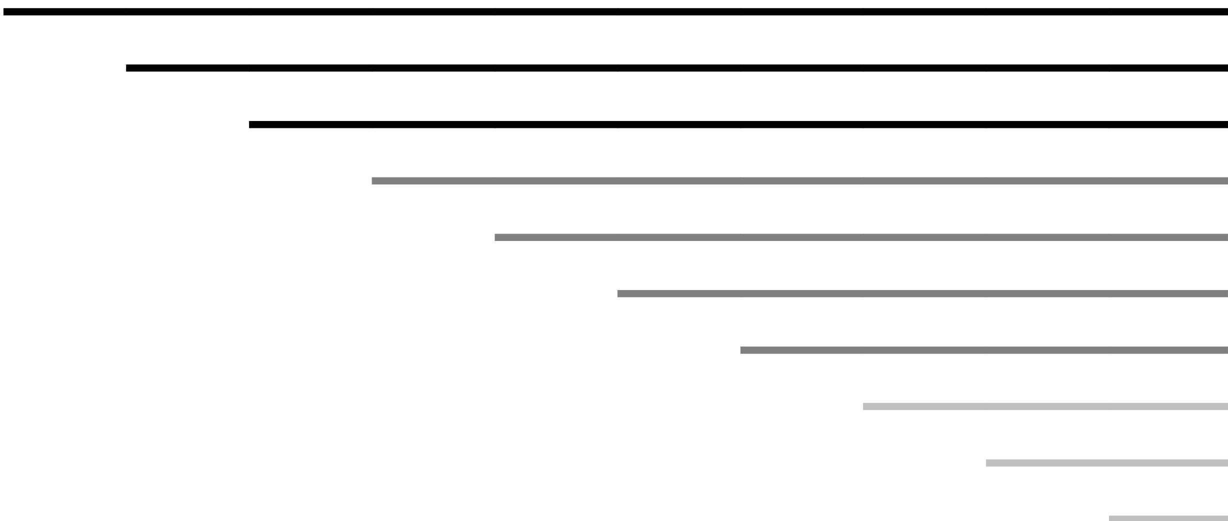
Embedding it better...

μTasker Document

μTasker – USB Audio



Copyright © 2016 M.J.Butcher Consulting

Table of Contents

1.Introduction....3
2.USB OUT Buffer Processing....4
3.USB IN Buffer Processing....8
4.Synchronisation....8
  4.1 Controlling Start Delta Value....8
  4.2 Measuring Synchronisation Deviation and Drift....9
  4.3 USB Watchdog....10
  4.4 Correcting Deviation using Software PLL....11
5.SAI/I2S....12
6.Configuration....12
7.Conclusion....13

1. Introduction

The USB Device Class Definition for Audio Devices species how devices with audio functions can be connected to a USB host.

This document describes the ***µTasker*** USB audio device implementation, its use and various practical technical details that are relevant to the subject.

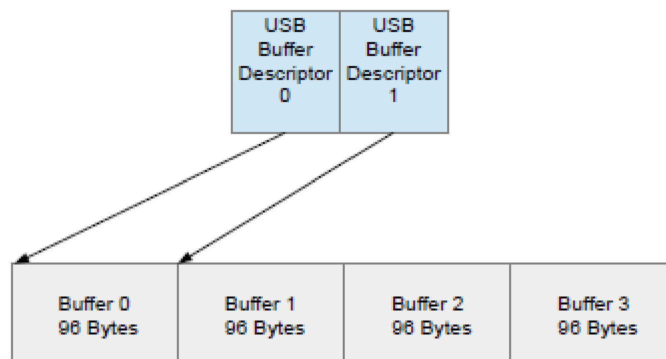
It is recommended to refer also to the ***µTasker*** ADC/DAC User's guide - http://www.utasker.com/docs/uTasker/uTaskerADC.pdf - for additional information about ADC/DAC interface functions that can be used for efficient DMA based audio data transfer operations.

2. USB OUT Buffer Processing

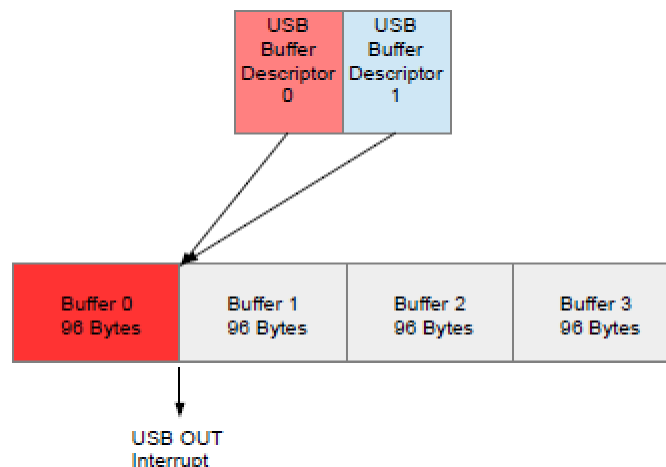
This section looks at the way that USB OUT frames (that is, the audio data that the USB host sends to the device) are handled so that no data loss occurs due to input buffer overruns and the data can be subsequently efficiently transferred to the sink (eg. A loud-speaker after conversion to analogue signal using a DAC (Digital to Analogue Converter)).

Typically USB reception will be controlled by a mechanism allowing at least 2 receptions to be defined in preparation of receiving data. This allows the USB controller to continue receiving data when the first of the packet receptions is still being processed by the processor. This is illustrated in the following sequence where a reception data buffer consists of enough space for 4 standard length data packets to be stored to before it becomes full and further reception starts at the beginning of the buffer again. The length of the buffer could however be 2, 6, 8 packet lengths etc. depending on the most suitable choice for the application and available memory.

Initially the two reception buffers are shows controlled by two buffer descriptors, which indicate that the buffer is ready to receive data and where the data is to be saved to. The first of the buffer descriptors defines that the first received packet is to be saved to the first block in memory and the second is to follow that.

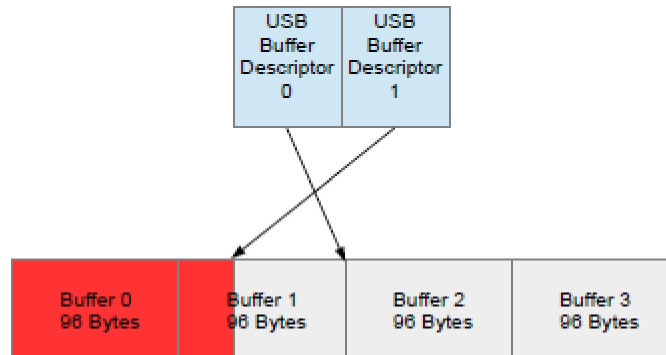


Once the first data packet has been received by the USB controller and saved to the first block in the reception buffer, the buffer descriptor is set by the USB controller to be owned by the processor and so cannot be used again by the USB controller until the processor has freed it.

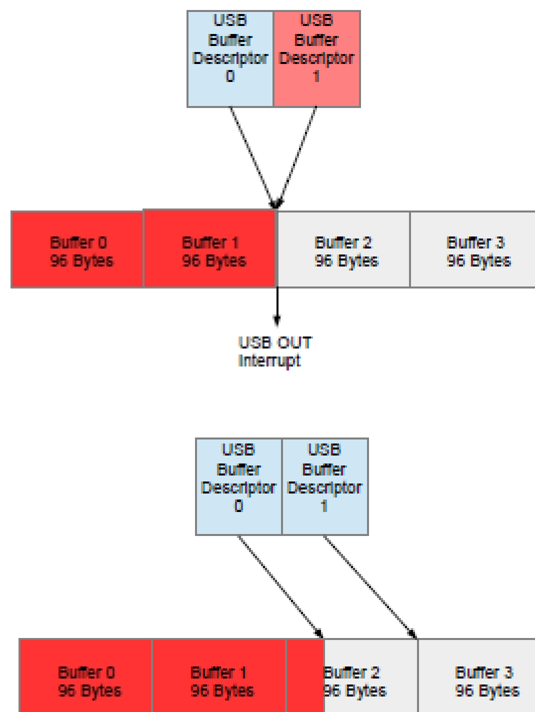


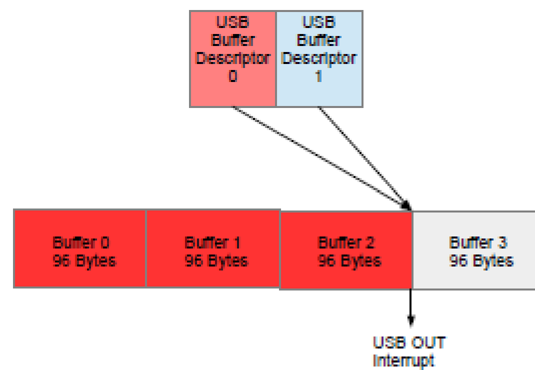
Note that a USB OUT interrupt results when the data packet is ready to be read, which will cause the processor to do whatever is necessary to process the received data and free up the presently blocked buffer descriptor.

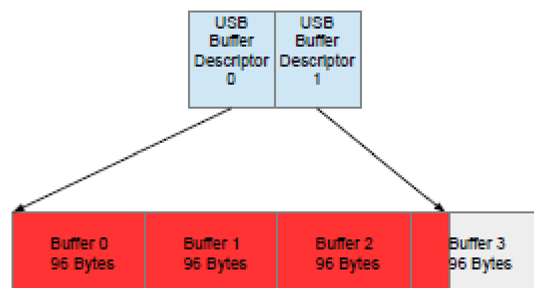
In this example it is assumed that the processor does not need to process the data content and can leave it where it has been stored. The processor thus only needs to free p the first buffer descriptor and it does this by reprogramming it to define that the next data packet that is received by it is put to the thirst bock in the reception buffer. This is illustrated next, where it is to be noted that the second buffer descriptor may still be handling the reception of subsequent data, shown by the fact that there may already be data being collected in the second block in the buffer.

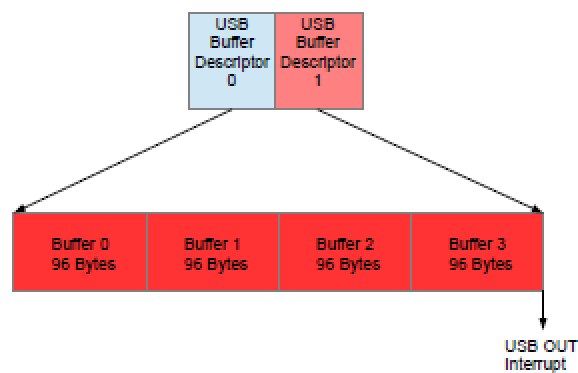


This process is fundamental to the way that the USB receiver operates, generally giving the processor enough time to handle the data in one packet while the UB controller continues receiving subsequent data. This implies that the processor must ensure that there is always one buffer descriptor available when it is needed, otherwise data will be lost, but this can usually be guaranteed during normal operation to achieve the data throughput of the isochronous USB endpoint.
The following illustrations show the process continuing until all four blocks in the input buffer have been filled.

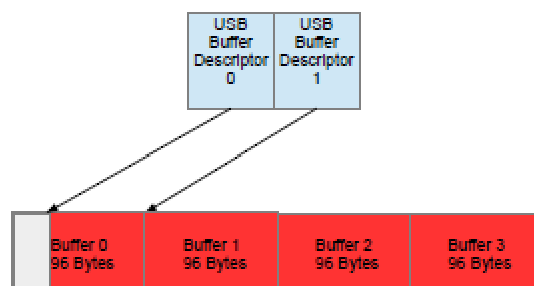






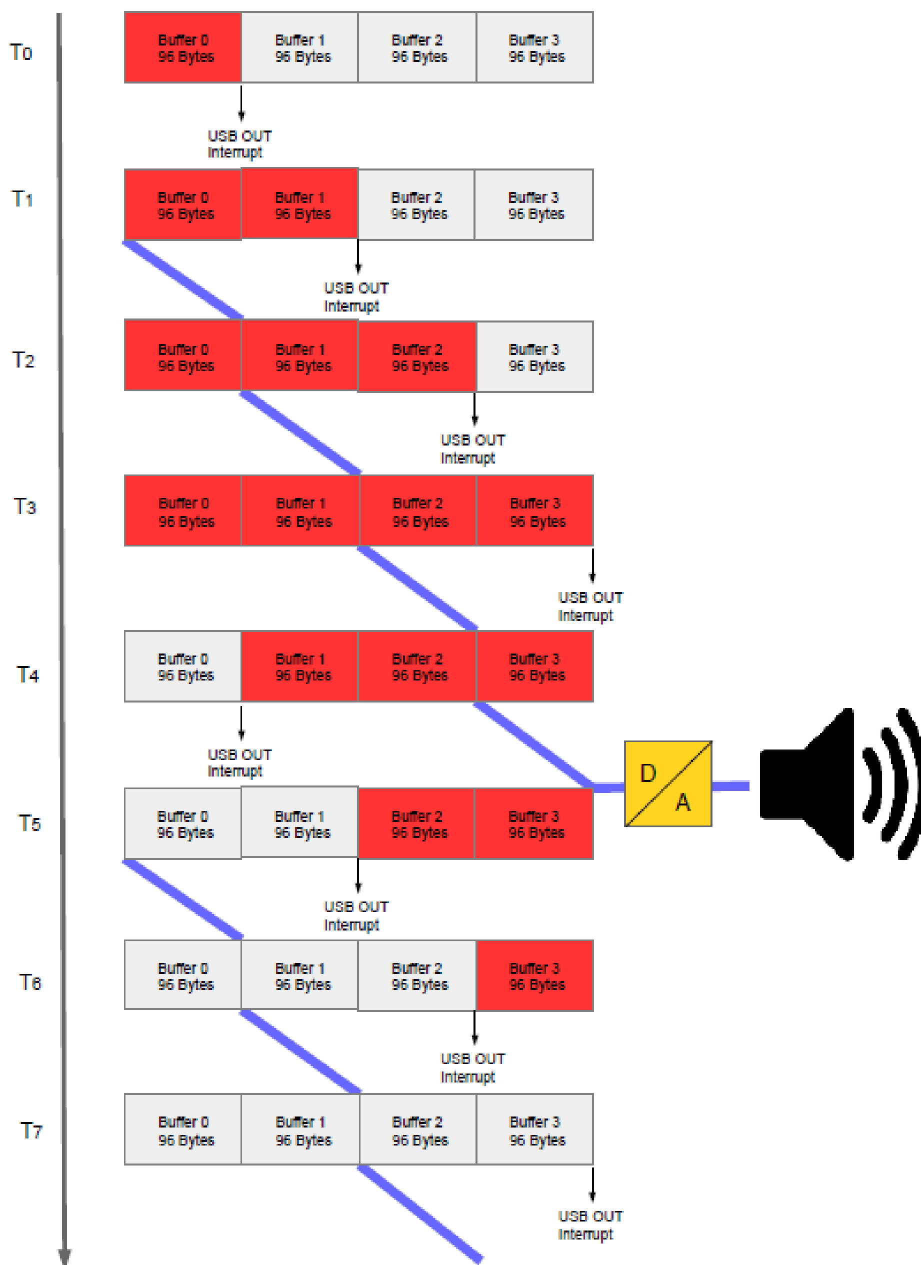


Finally, once the linear input buffer has been filled, further reuse starts at the beginning of the buffer again and the data in the buffer gradually becomes overwritten as the process continues.



The result is that data values exist in the input buffer for a period of time defined by the quantity of blocks in the linear buffer's total memory. If more memory is used by the linear buffer, more reception packets can be stored before buffer re-use and this the longer each data value exists before being overwritten.

The next illustration indicates how received audio data content could subsequently be output to a sink, such as a loud-speaker. This assumes that there is some suitable process that takes the data from the linear buffer and allows the content to be converted to an analogue signal suitable for driving the load (this may also involve decoding the format). It also assumes that the process of copying and conversion takes place in time with the data arriving, for example, if the average data rate of the signal being received is 48kBytes/s the DAC processing is also exactly 48kBytes/s so that the overall process remains synchronised.



Notice that the data output starts once two buffers are available, resulting in the output being always delayed by two buffer periods. It can be seen that it is also possible for the output timing to fluctuate by +/- 2 buffer blocks without data corruption resulting, as long as the synchronisation remains within this accuracy over all time.

3. USB IN Buffer Processing

IN buffer processing is the inverse of OUT buffer processing. Instead of there being a sink that uses the received data, there is a source – such as a microphone – whose sampled data (eg. using an ADC (Analogue to Digital Converter)) is saved to a linear buffer at the sample rate. Instead of the USB host sending data packets, it reads data packets from the buffer using IN frames.

The USB controller's buffer descriptors also operate in the opposite sense and control the data packets that are transmitted in each isochronous data frame.

In the case of OUT data packets the data is processed at the device after a short delay (a quantity of buffer periods to ensure that all data is available) but for IN data packets the device has to ensure that there are complete data packets available when each IN token is received; this means that there is an equivalent delay controlled between the source data being saved until the IN transfer takes place.

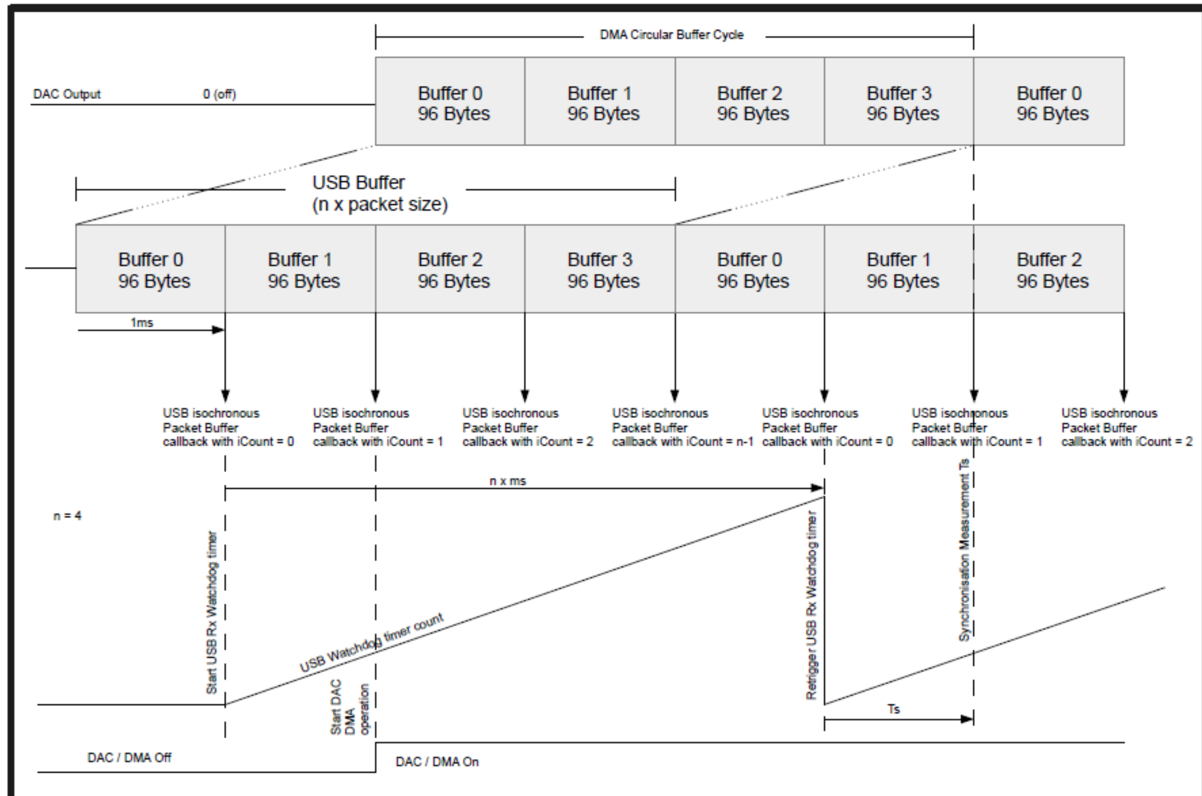
4. Synchronisation

The perfect situation exists when the clocks used for audio processing are synchronised to the clock used for data transmission via USB to and from the host. In this case there is essentially no drift between clocks (although if sample periods are missed for some reason, such as when controlled by interrupts which can be delayed, there is still potential for sample drift to take place even if the underlying clocks are always locked).

As shown in the diagrams in the previous sections a small amount of sample deviation is acceptable but drifts exceeding the buffer margins would cause errors (under-run or overrun) to occur which would lead to noise, clicks or distortion.

4.1 Controlling Start Delta Value

When USB Audio operation starts it is important to start with the optimal delay between USB and the audio input/output. This is achieved at the sink by allowing the USB reception buffer to first fill up to half of its capacity and then starting the output transfer (from USB buffer to sink output). The timing involved is illustrated in the following illustration whereby this is based on the USB reception's interrupt each time an isochronous packet has been received form the USB host. The optional µTasker USB interrupt callback contains a parameter `iCount` that indicates which buffer the interrupt corresponds to. With the reference case of 4 packet buffers in the complete reception buffer space, the parameter counts from 0 to 3 and then repeats.



Initially the audio interface may be set to its active configuration but the USB host does not yet send any audio data. The first packet reception, with count value 0, is used to start a USB watchdog timer that counts in order to monitor the period between each 0 count interrupt. The timer's instantaneous counter value will also be used to monitor the synchronisation as explained in the following section.

The DAC output (or other type of audio output) is initially held at its 0 output value and the USB buffer to sink output DMA operation is disabled so that the output doesn't yet change.

When the USB interrupt's parameter signals that half of the USB buffer has been filled with audio data (when the packet count n is even this is at the value `(n/2 - 1)`) [1 in the example case] the optimal delay (delta) exists and the ADC DMA operation is started, which causes the DAC output to start following the initial USB packet buffer's content and the optimal `n/2 ms` Delta value exists.

4.2 Measuring Synchronisation Deviation and Drift

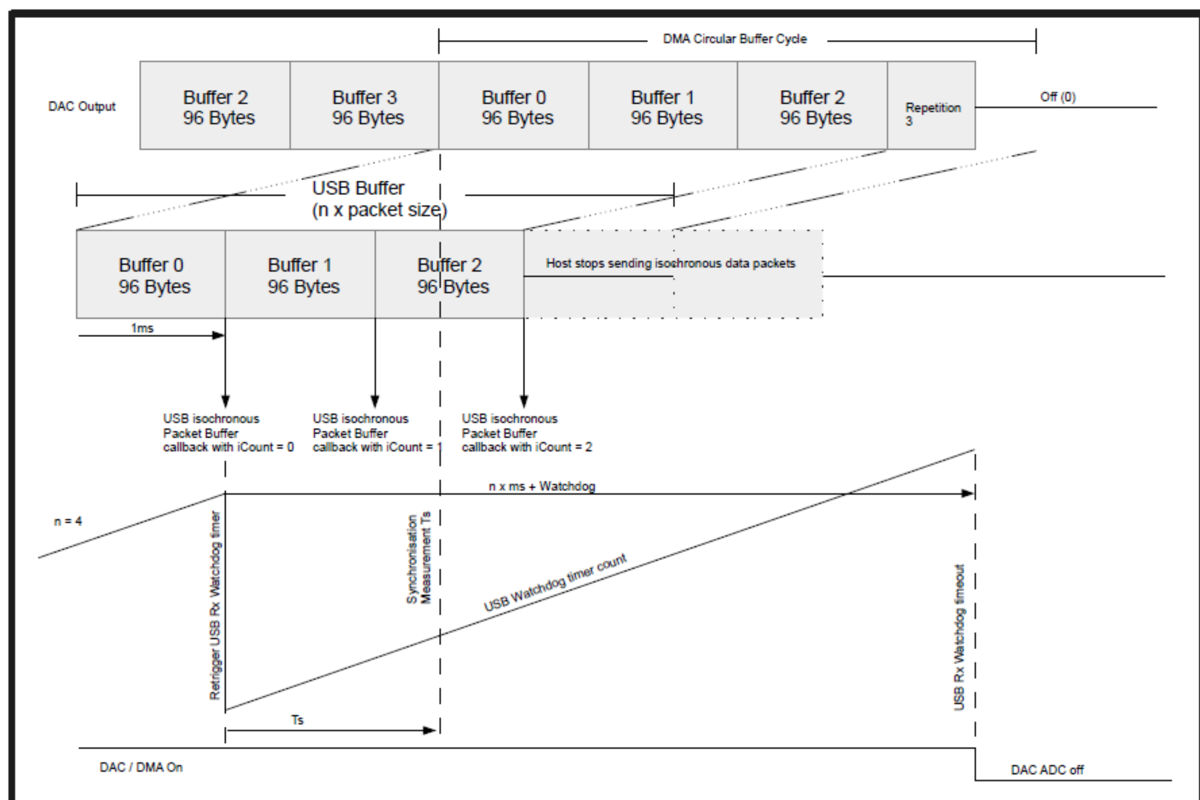
The perfect situation with respect to maximum drift margin exists when the USB reception/transmission is shifted by half of the complete USB buffer size (`n/2 ms`) to the sink/source. This means that this initial half-buffer deviation needs to be monitored so that any drift can be compensated for.

From the previous illustration is is seen that the USB watchdog timer value can be used to check the synchronisation accuracy based on the instantaneous timer counter value when the DMA circular buffer completes - this is configured to generate an interrupt callback to perform the monitoring. The first time that this takes place it is expected that there will not yet be any noticeable drift that has taken place and the timer counter value will accurately correspond to the delay between the 0 count USB interrupt and the mid-packet interrupt, which is a USB packet delta of (`n/2 - 1`) or 1 packet period (1ms) in the reference case.

A USB watchdog timer count value corresponding to 1ms thus indicates the perfect delta case, whereas shorter or longer count values indicate that the delta has drifted over time and some form of compensation is required in case the delta value drifts outside of an acceptable range. Larger values of n means that the delay (delta) increases but also allows a arger drift to take place before the situation becomes critical and requires adjustment.

4.3 USB Watchdog

The USB watchdog timer is used to monitor synchronisation accuracy during normal operation and has a second function to disable the ADC output when the USB host stops sending audio data. This is illustrated in the following diagram:



When the USB host stops sending audio data it may send packets with 0 values so that the output is silent. However it may also stop sending any data packets before it disables the audio interface (*practically this state often exists for several seconds with some PC audio applications*). If there is no USB reception watchdog operation the effect is that the DAC DMA continues outputting the previous buffer content which tends to sound like some form of noise of interference. In order to avoid this, or at least keep it as short as possible so that it doesn't disturb the USB watchdog is used to detect this situation and silence the DAC output by disabling it or its DMA operation.

To achieve this the USB watchdog timer is programmed to generate an interrupt when its value reaches a threshold which is a little more than the normal buffer repetition period. Normally the USB watchdog timer is reset before this threshold is reached when the USB host is sending continuous isochronous packets and so only fires when these stop for a period determined by the threshold value set. Depending on the relationship between the 0 count USB interrupt and the exact point at which the USB data stops there may still be a short period of time where the DAC DMA transfers continue with repeated buffer content but

this tends not to be noticeable as long as n is not very large and the watchdog threshold is set to be not much longer than the DMA circular buffer period.

4.4 Correcting Deviation using Software PLL

The perfect situation with respect to maximum drift-margin exists when the USB reception/transmission is shifted by half of the complete USB buffer size (n/2 ms) to the sink/source. This means that this initial half-buffer deviation needs to be monitored so that any drift can be compensated for and the technique used in the µTasker project has been discussed in the previous sections.

Although there exists a possibility to allow synchronisation to be controlled using a feedback USB endpoint the simplest technique is often to control the time base of the DAC/ADC directly. Also, external audio processor circuits usually contain some possibility of adjusting the time base to achieve accurate synchronisation.

Assuming the case where the processor controls the ADC/DAC time base using an internal timer with a high resolution (allowing fine frequency adjustments) it is very simple to implement a method of keeping this synchronised to the USB time base based on the the measured deviation from the optimal synchronisation delta value. The ADC/DAC time base will essentially be controlling the rate at which DMA transfer between the USB buffer and the source/sink take place, whereby this is set up to normally operate at the nominal sampling rate - for example 48000Hz.

In the ideal case the 48000Hz sample rate and also the host's USB time base (based on 48MHz and a full-speed start of frame period of 1.000ms) will be highly accurate. The resulting drift will also be ideally zero. Practically however a drift to result may result requiring some intervention after maybe several minutes or, in less accurate cases, several seconds.

A simple software PLL algorithm is achieved by allowing the ADC/DAC sample frequency to run at its middle frequency as long as the measured deviation is within an acceptable region. If the measurement detects that the drift has increased or reduced the delta value from its ideal value to outside the acceptable region the sample rate is increased or decreased to compensate until the delta value is again within the acceptable range.

The delta value can be easily converted to a +/- deviation from the optimal value and then the simplest (example) algorithm that results is the following, whereby it is assumed that the resolution of the sampling speed is fine but still adequate to effectively compensate drift within a fairly short time.

```
// DMA interrupt after circular buffer completion - the DMA operation has wrapped around to
// continue reusing the buffer but this interrupt is used to monitor the relationship
// between the USB reception (controlled by the USB host time-base) and the audio sink
//
static void buffer_wrap(void)
{
    signed long slDelta = ((PIT_MS_DELAY(USB_WATCHDOG) - PIT_CVAL1) -
((PIT_MS_DELAY(USB_WATCHDOG) - PIT_MS_DELAY(USB_WATCHDOG - AUDIO_BUFFER_COUNT))/2));
                                             // present delta between the USB reference
                                             // interrupt and the DMA buffer interrupt
                                             //(this should ideally remain close to zero)
    // Adjust the PIT time base to compensate for the deviation to ideal audio delay
    // and so hold synchronision to the host time base
    if (slDelta > DRIFT_LIMIT_MAX) {
        PIT_LDVAL0 = (PIT_FREERUN_FREQ(48000) - 1);              // speed up the timer
    }
    else if (slDelta < DRIFT_LIMIT_MIN) {
        PIT_LDVAL0 = (PIT_FREERUN_FREQ(48000) + 1);              // slow down the timer
    }
    else {       // set the exact frequency since the synchronisation is in the optimal region
        PIT_LDVAL0 = PIT_FREERUN_FREQ(48000);
    }
    #endif
}
```

5. SAI/I^{2}S

Audio applications will usually be based on an audio device which will digitally process the audio data stream and convert it to the analogue domain where it may also be amplified and driven by a power amplifier or matched to different possible sources (such as microphones). Processing will usually include some filtering, gain control and muting but could also compress and decompress to reduce the data's bandwidth requirements.

A common interface to the audio processing module is SAI (Synchronus Audio Interface).

I2S is a mode supported by SAI that is often used.

6. Configuration

USB device is enabled in the ***µTasker*** project by setting the define `USB_INTERFACE` and the USB audio class operation by setting also the define `USE_USB_AUDIO`. The USB audio class can be combined with other classes to simply create composite devices.

The isochronous endpoints used by the audio device are defined by the following configuration code:

```
USBTABLE tInterfaceParameters;                 // table for passing information to driver

tInterfaceParameters.usEndpointSize = 96;        // isochronous endpoint size
tInterfaceParameters.Paired_RxEndpoint = 1;      // the endpoint used by the audio device for reception
tInterfaceParameters.Endpoint = 2;               // paired endpoint for transmission (IN)
tInterfaceParameters.usb_callback = fnAudioPacket; // user callback when audio packet has been received
tInterfaceParameters.queue_sizes.TxQueueSize = (96 * AUDIO_BUFFER_COUNT);
tInterfaceParameters.queue_sizes.RxQueueSize = (96 * AUDIO_BUFFER_COUNT);
                                     // reception buffer with space for two complete HID raw messages
tInterfaceParameters.usConfig = (USB_IN_ZERO_COPY | USB_OUT_ZERO_COPY | USB_IN_CALLBACK);
USBPortID_Audio = fnOpen(TYPE_USB, 0, &tInterfaceParameters);
                               // open the endpoints with defined configurations (initially inactive)
```

The isochronous endpoint size is equal to 96 bytes and input and output buffers of size `AUDIO_BUFFER_COUNT`, multiplied by the endpoint packet size, are requested to be created. The configuration parameters `USB_IN_ZERO_COPY` and `USB_OUT_ZERO_COPY` inform the USB driver that the endpoint's buffers are zero-copy types, meaning that they are used as described in the previous chapters which dealt with the buffer processing strategy.

A user OUT and IN interrupt callback (`fnAudioPacket()`) is defined, which will be called at every reception interrupt and is usually used to control and monitor synchronisation.

7. Conclusion

This document is presently in progress.
It is being written in parallel with practical product developments and optimisations based on various USB capable processors.
The code is presently available for Alpha developments and final documentation and fully verified code release is anticipated during April 2016.

Modifications:

- V0.01 23.12.2015: - First draft – in progress
- V0.02 24.03.2016: - Additional synchronisation measurement details – in progress