SharedSound in the future

One of the great things about SharedSound is that it's shared. One of the really difficult things about it is that it's shared.

Introduction

This document is primarily about sound playback with SoundDMA and SharedSound, and is based on the feedback of my earlier document.

How SoundDMA works

SoundDMA is the module that interacts the closest to the hardware. It relies on software to fill up a buffer with the sound data, and hardware emptying that buffer at the right time in order for the sound to be played.

This is done with a *LinearHandler*, which is set up with the *Sound_LinearHandler 1* SWI call.

In order to reduce CPU loading, the LinearHandler code will fill a number of samples in one go, and wait for a signal from the hardware that more is ready. Figure 1 shows the LinearHandler pushing data to the sound buffer, and the audio hardware is pulling it.

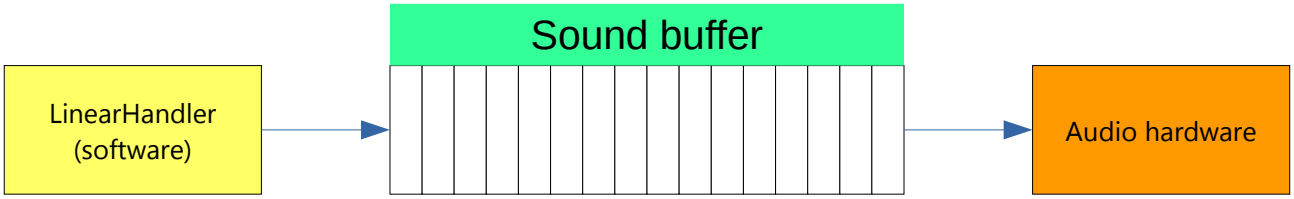

Figure 1: Simplified audio playback

In reality, two buffers are used – the application fills one with its data whilst the hardware, as shown in figure 2:

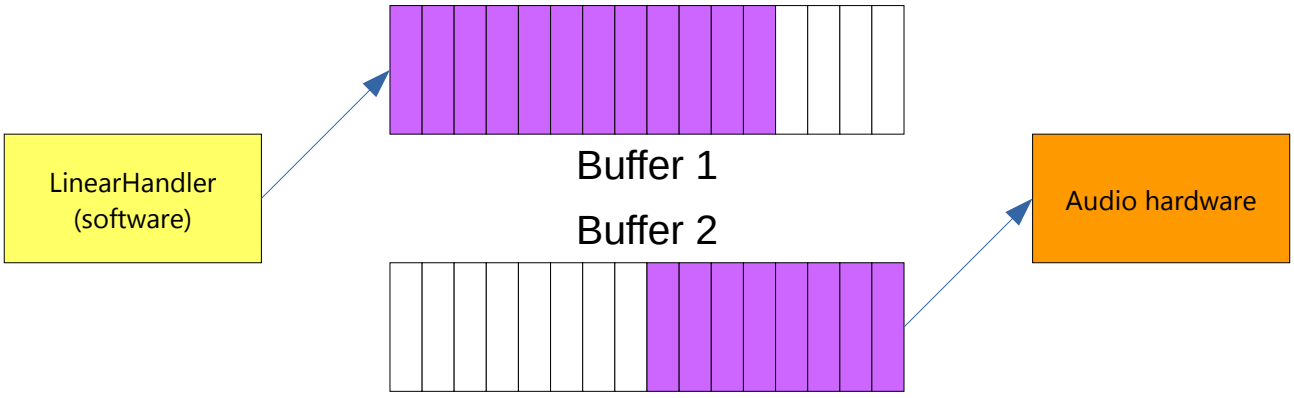

Figure 2: Double buffered

When the hardware has emptied the buffer, it with switch to the other buffer, and the LinearHandler will fill the buffer the hardware had been using moments before, as shown in figure 3:

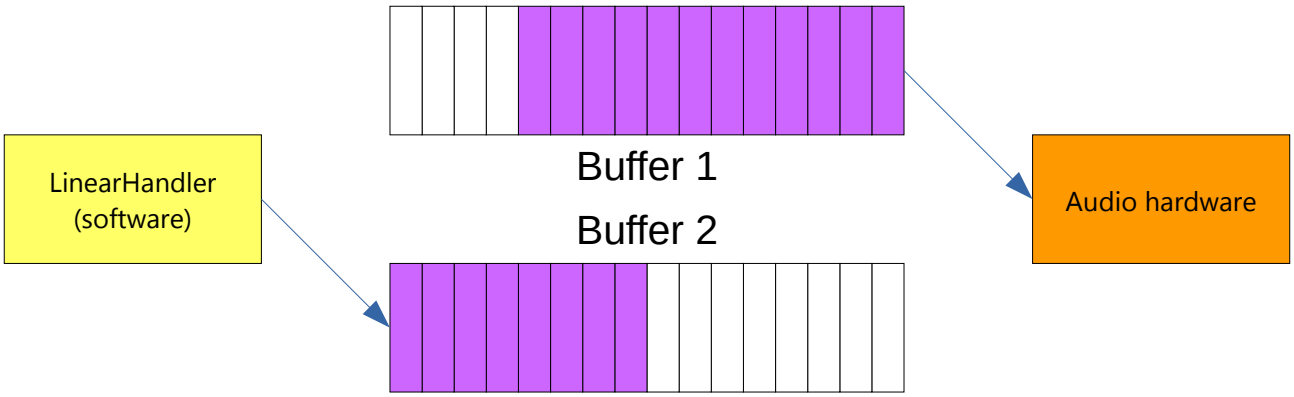

Figure 3: Double buffer after swapping

Note that the old µ-Law adds a little bit of complexity into this system. When the audio hardware has just started on a new buffer, the old buffer is empty, as shown in figure :

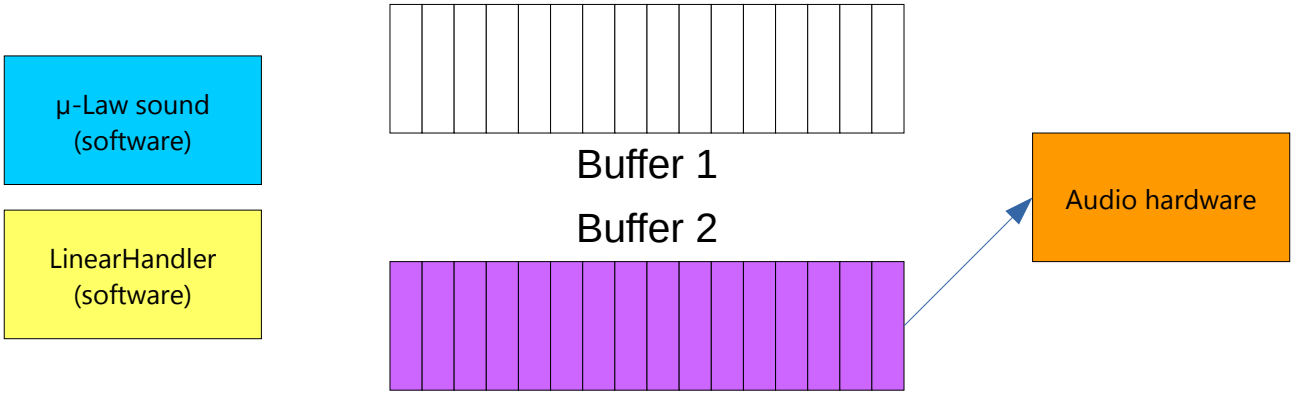

Figure 4: New buffer being started

Some code converts the current µ-Law sound into 16-bit sound, filling the next hardware buffer (which is being emptied), as shown in figure 5:

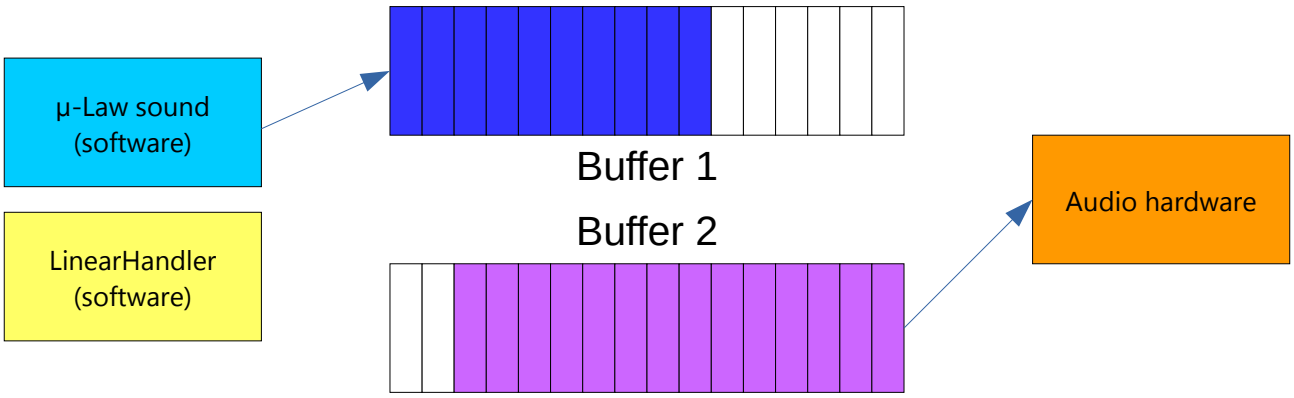

Figure 5: µ-Law buffer filling up

When that is complete, the LinearHandler is given the buffer for it to fill (as the hardware as pulled more of its buffer), as shown in figure 6:

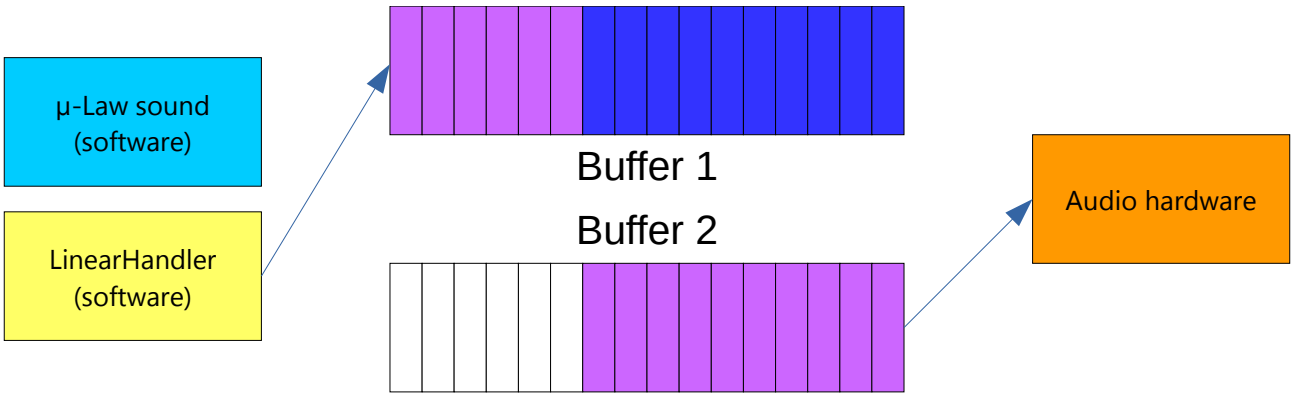

Figure 6: LinearHandler filling buffer

At this point in time, the LinearHandler has the option to merge its sound with that of the µ-Law output, or simply replace what is there (when it is called, the LinearHandler is told whether the buffer contains µ-Law data, is empty, or simply needs to be overwritten).

Merging data could potentially be computationally expensive – but as long as the audio hardware is pulling data from its buffer at a slower rate than the two buffer fillers are filling it, then there will be no break in audio.

If the system is running too slowly, then one of three things could happen:

1. The audio hardware stops playing
2. The audio hardware repeats its last buffered
3. The audio hardware starts playing the other buffer regardless of the fact that it has not been filled yet

That's pretty much it for what SoundDMA needs to do (in the context of this discussion) – it's just there to marshal the data from the LinearHandler to the audio hardware.

How SharedSound works

SharedSound is actually a LinearHandler. On a modern system that has just booted, calling *Sound_LinearHandler 0* to get the address of the current LinearHandler actually provides an address within the SharedSound module (on my system, SharedSound was located at &FC31A8B0, and ended at &FC31C278 – the current linear handler was &FC31BBD4).

How SharedSound differs from LinearHandlers is that it is actually a chain of "enhanced LinearHandlers" – each one passes data into the next one, and as they go along, they merge the sounds.

Here is the same examples as above, except from a SharedSound perspective. Note that the µ-Law is partially represented as a SharedSound client when in reality it is not.

Figure 7 shows the buffer about to be filled:

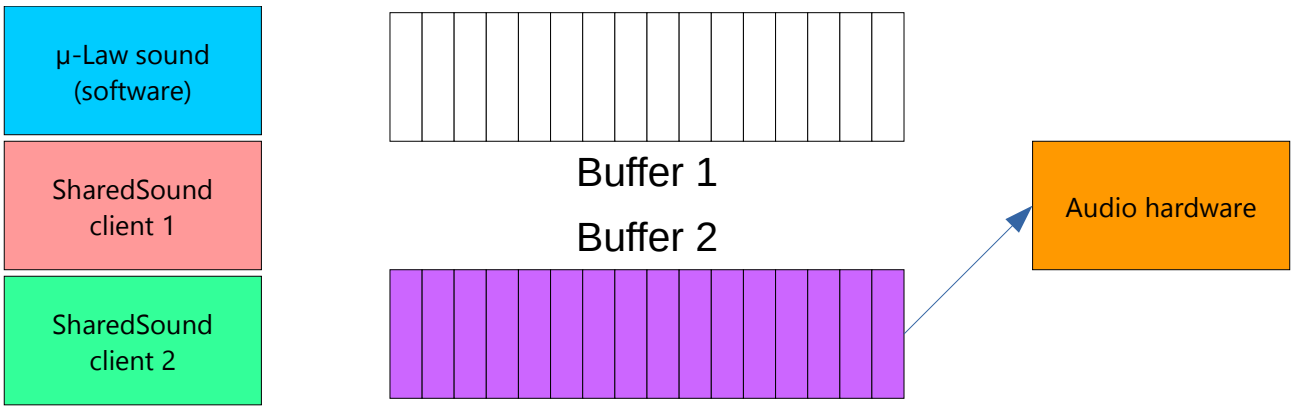

Figure 7: SharedSound with empty buffer

The µ-Law buffer fills up first, as shown in figure 8:

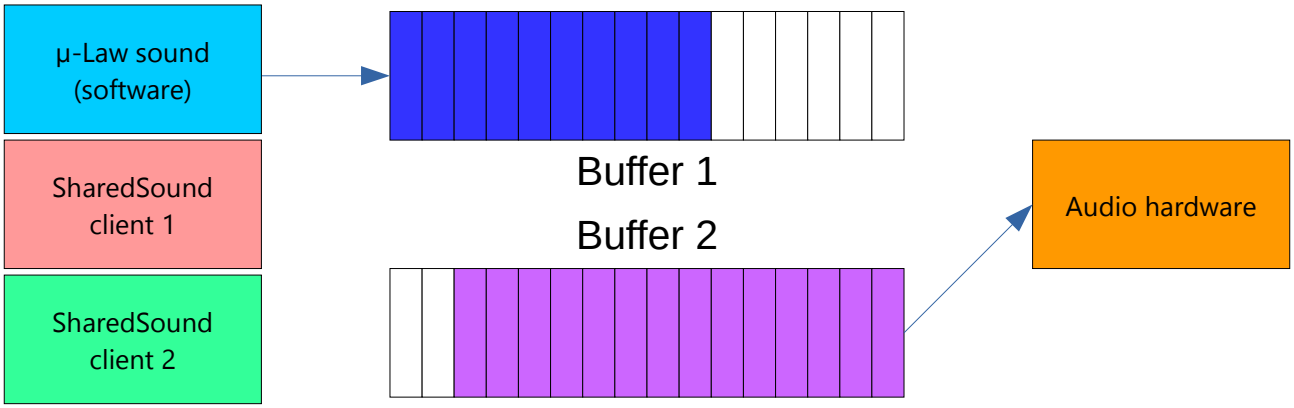

Figure 8: SharedSound, buffer filled up by µ-Law

And then SharedSound client 1 starts merging its sounds with that of the µ-Law sounds, as shown in figure 9:

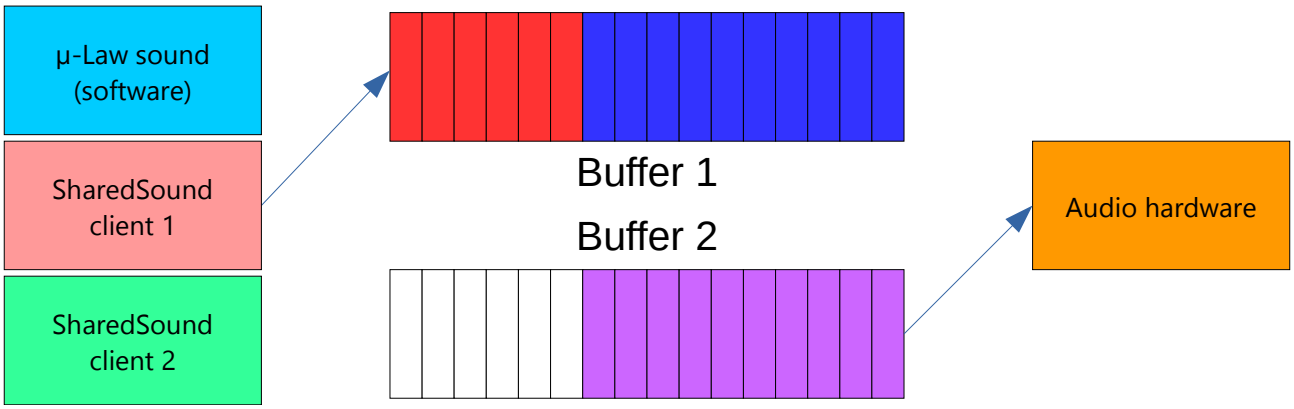

Figure 9: SharedSound client 1 filling buffer

When that is complete, SharedSound client 2 starts merging its sounds with both the µ-Law and ShareSound client 1, as shown in figure 10:

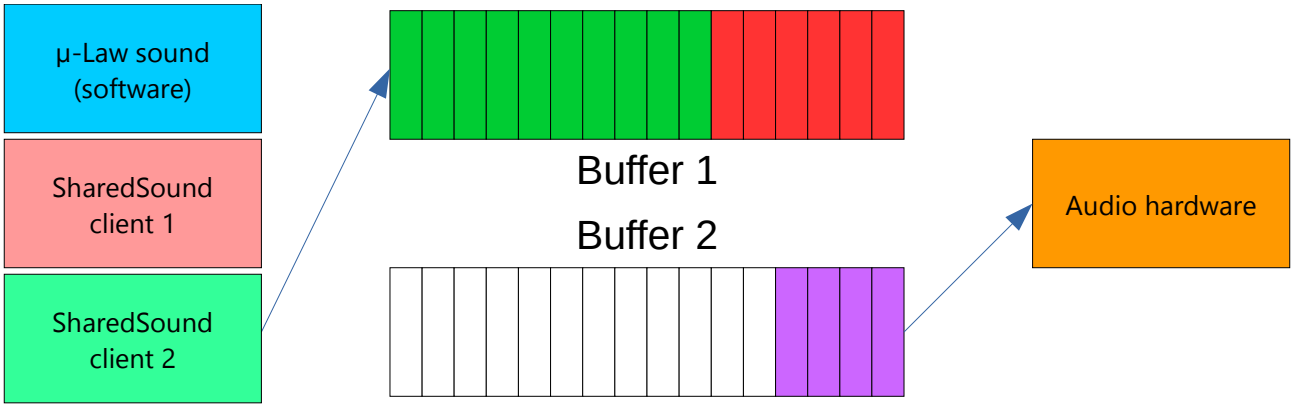

Figure 10: SharedSound client 2 filling buffer

During all this time, the hardware is emptying its buffer. If too many SharedSound clients are attached, then the buffer may not be filled in time.

16-bit, 24-bit and 32-bit sound formats

Currently, RISC OS supports 16-bit sound data only. This is good enough for most people, but modern chipsets support higher quality, and applications on other operating systems can take advantage of it.

A 16-bit format is shown in figure 11:

	31..24	23..16	15..8	7..0
Sample 1	Right (15..0)		Left (15..0)	
Sample 2	Right (15..0)		Left (15..0)	
Sample 3	Right (15..0)		Left (15..0)	
Sample 4	Right (15..0)		Left (15..0)	
Sample 5	Right (15..0)		Left (15..0)	
Sample 6	Right (15..0)		Left (15..0)	
Sample 7	Right (15..0)		Left (15..0)	
Sample 8	Right (15..0)		Left (15..0)	

Figure 11: 16-bit data format

A 32-bit word is split into two 16-bit values containing the left and right hand samples for that particular time interval. So if you have 32 bytes of data to fill in a buffer, you have got 8 samples.

With a 32-bit data format (as shown in figure 12), it’s different:

	31..24	23..16	15..8	7..0
Sample 1	Left (31..0)			
Sample 1	Right (31..0)			
Sample 2	Left (31..0)			
Sample 2	Right (31..0)			
Sample 3	Left (31..0)			
Sample 3	Right (31..0)			
Sample 4	Left (31..0)			
Sample 4	Right (31..0)			

Figure 12: 32-bit data format

Now, in the same 32 bytes you've got just 4 samples of data.

Note that the 24-bit version would be the same, except bits 7..0 of every word will be ignored (so could be zero).

SharedSound and higher quality formats

At the moment, SharedSound currently only supports 16-bit formatted data.

Adding support for 24-bit or 32-bit formats to SharedSound is only part of the problem – the only functionality that needs to know about the format is the 'helper' functions it provides for lazy programmers to do filling of data (or silence).

However, if a SharedSound client does its own mixing (and a quick straw-poll suggests this is the case), then the clients tend to be 16-bit only as well.

What this means in reality is that as it currently stands, SharedSound works really well if you are dealing with 16-bit data, but when you get to 24-bit or 32-bit data, every client **must** be able to deal with that format of data – otherwise it'll try to interpret the data incorrectly and everything will go wrong.

Actually, what'll happen is that it'll play its sample at twice the sample rate, with weird things going on in stereo. I think.

How to solve this

There are a number of ways in which this can be solved.

Firstly, you could force the sound system to always be in 16-bit when you got at least one 16-bit capable SharedSound clients active. This would be fairly easy – but probably less understandable by users ("Why can't I use 24-bit audio?"). Users could be presented with a greyed-out list of formats that can't be supported, but they may not know why this is the case.

Secondly, you could disable any SharedSound clients that can't cope with the higher resolutions while an application wants to use higher resolutions. This, again, would be fairly easy – but even less understandable by users ("Why did my CD player stop working?"). Users could be forewarned that this is going to happen by the application that is going to change the audio quality.

Thirdly, you could allow 16-bit only SharedSound clients to work by providing a mechanism for 16-bit clients to buffer to a temporary area, and when all of these have finished their mixing, then SharedSound takes this area of memory, and expands it into 32-bit samples while merging with the 24-bit or 32-bit stream.

Allowing 16-bit and 24/32-bit SharedSound clients

The following sequence demonstrates this in action.

Figure 13 shows three sound producers (µ-Law, SharedSound client 1 and SharedSound client 2). SharedSound client 2 is 16-bit only.

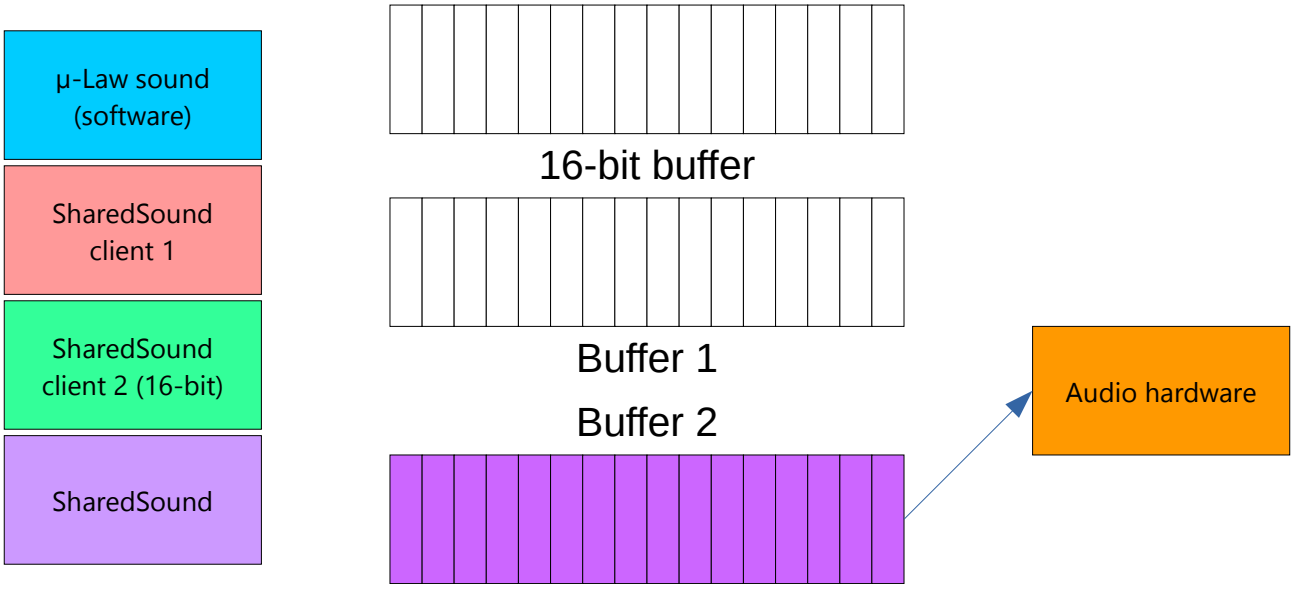

Figure 13: 32-bit SharedSound with a 16-bit client initialising

Initially, the µ-Law converter inserts into the main audio buffer, as shown in figure 14:

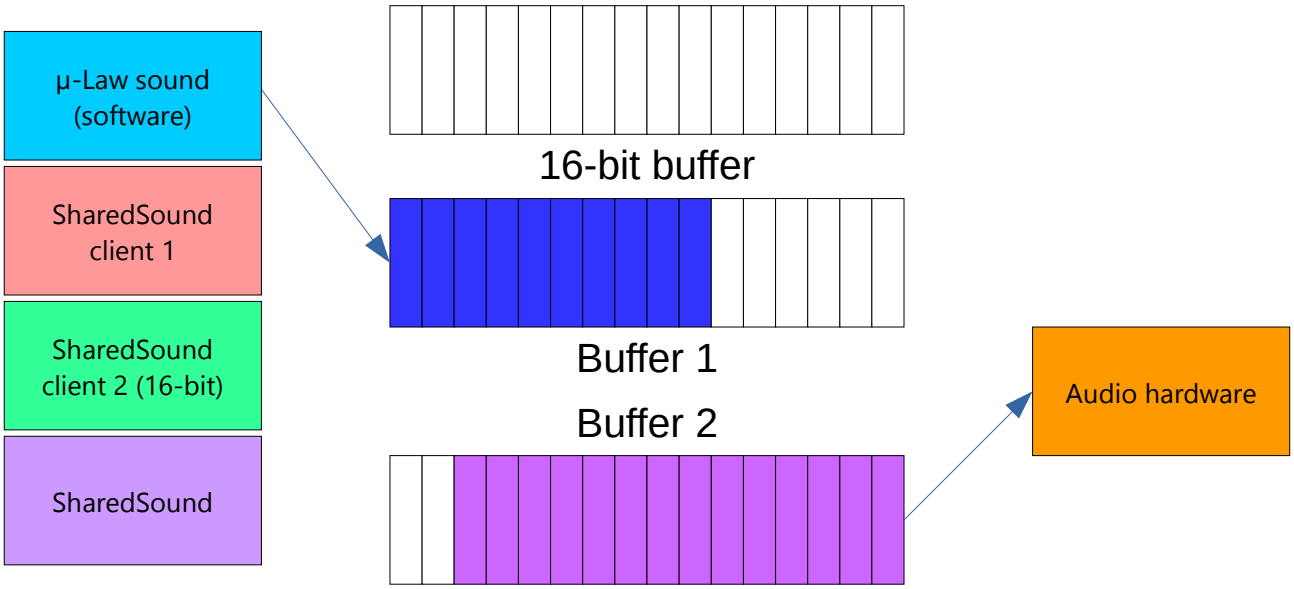

Figure 14: µ-Law filling 32-bit buffer

Next, SharedSound client 1 merges its sound with that generated by the µ-Law generator (as shown in figure 15).

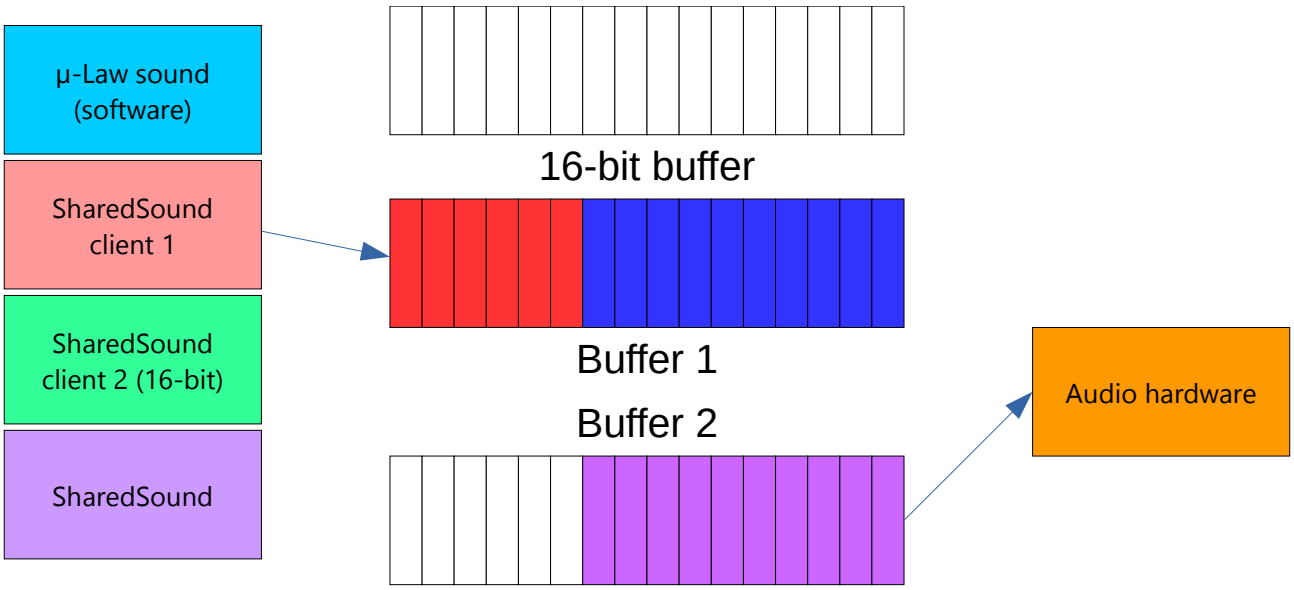

Figure 15: SharedSound client 1 filling 32-bit buffer

When that is complete, SharedSound client 2 starts writing its data to a separate area of memory – the 16-bit buffer, as demonstrated in figure 16:

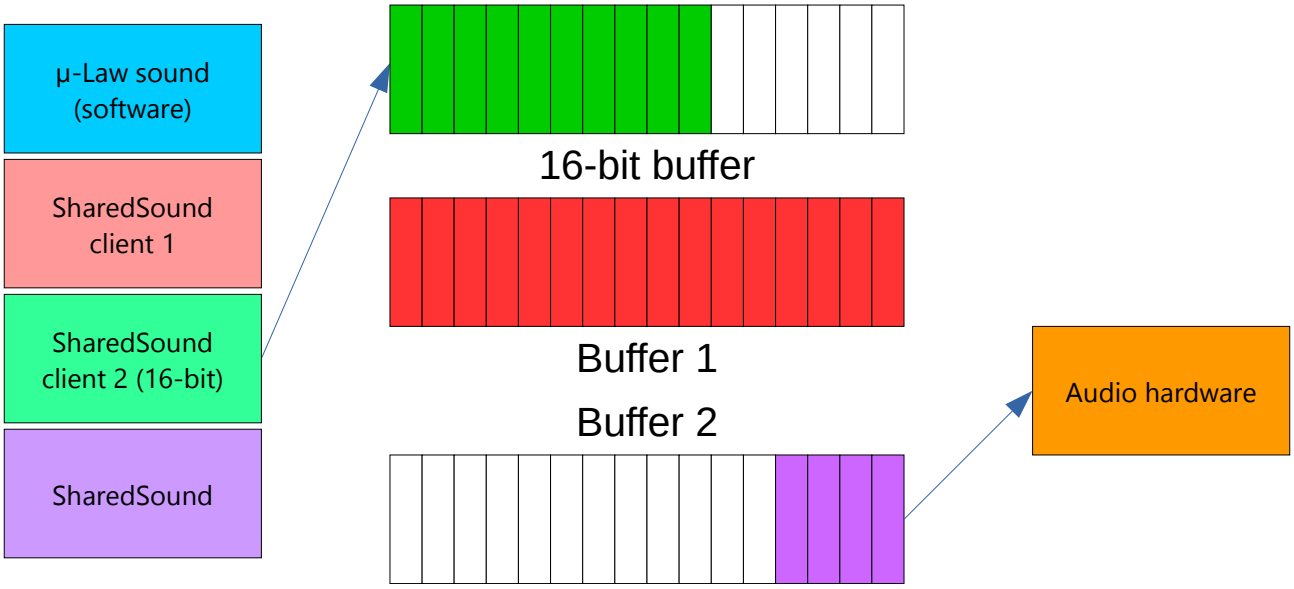

Figure 16: SharedSound client 2 filling 16-bit buffer

When that is complete, SharedSound now takes the 16-bit buffer, and merges it into the sound buffer, as shown in figure 17:

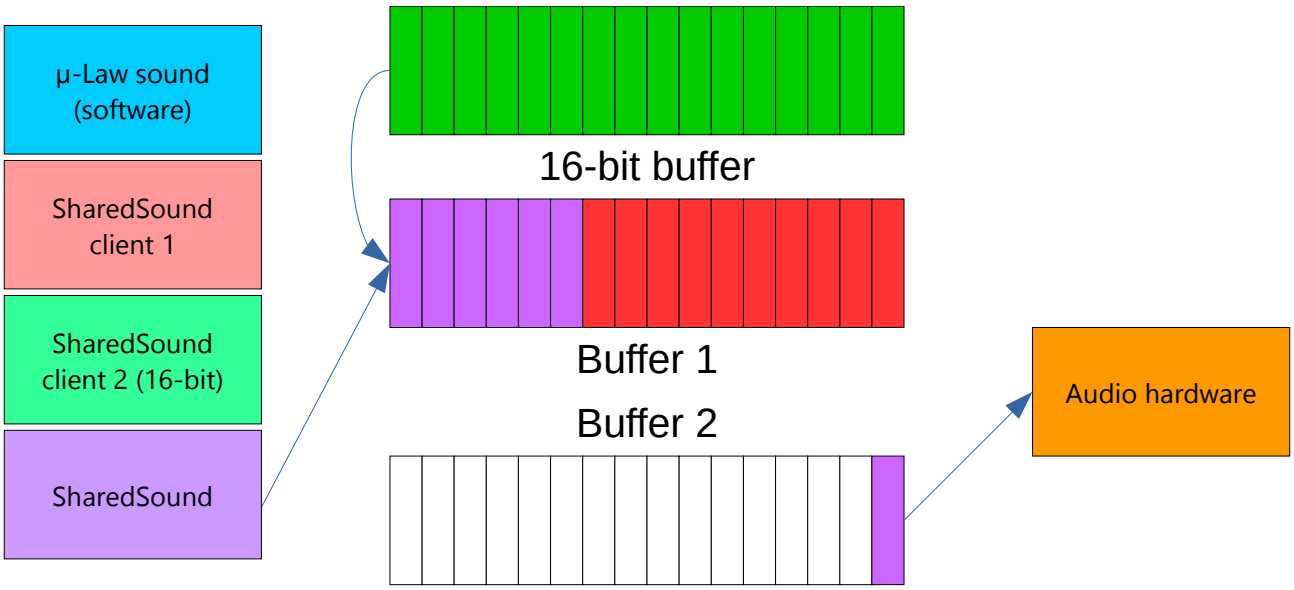

Figure 17: SharedSound merging 16-bit buffer into 32-bit buffer

But what this also shows is that this could be quite CPU intensive – and we may run out of CPU cycles to do the sound.

If multiple 16-bit SharedSound clients are in the system, then they should use the same 16-bit buffer, and SharedSound should only merge this buffer with the 32-bit buffer after all the 16-bit clients have done what they need to.

One possible encapsulation is to perform all the 16-bit clients first, then create the 32-bit buffer from this, and then call the 32-bit clients, as shown in figure 18:

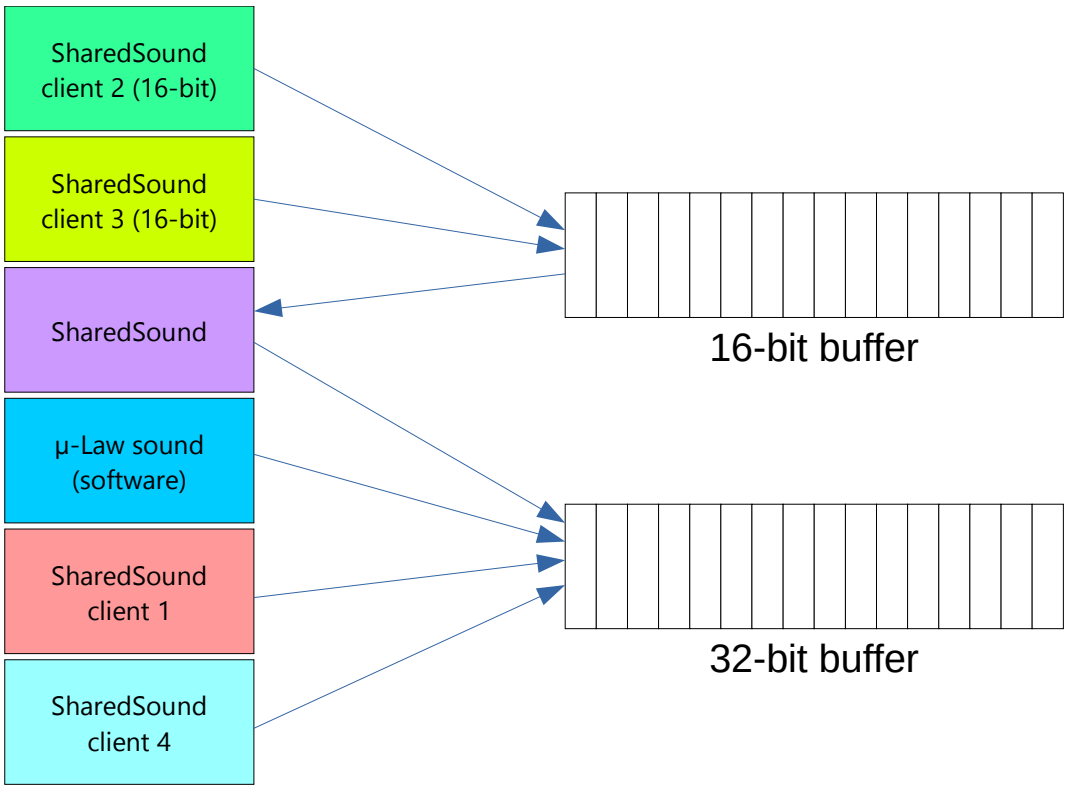

Figure 18: Possible sequence to fill buffers

LinearHandlers and SharedSound

One question has been asked, which I have paraphrased as “Can LinearHandlers and SharedSound work together?”

The answer is yes – but not by one method which I’ve described as “Can SharedSound replace the SoundDMA LinearHandler code?” (it’s no, by the way – because SharedSound *is* a LinearHandler, and is arguably simpler than SoundDMA. Functionally, they could be merged into a single module, but supporting multiple SWI numbers in the same module isn’t trivial).

The way I would implement this is to have SharedSound called after the buffer has been filled by a LinearHandler **all the time**. This means that when SharedSound fills the 32-bit buffer, it’s already got data present within it. Taking figure 18 as an example, we can produce figure 19:

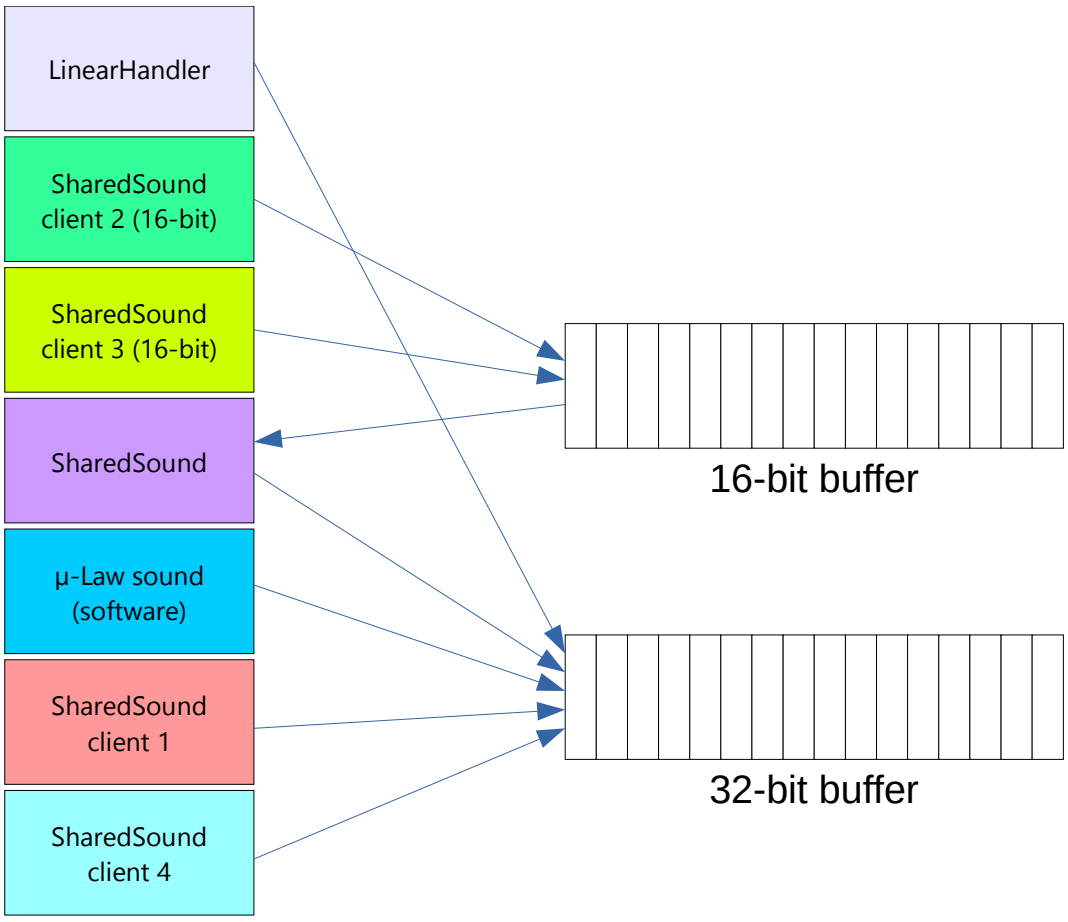

Figure 19: LinearHandler with SharedSound

Very little code would need to be changed to permit this method of operation – SoundDMA simply calls SharedSound at the appropriate time.

The only question is should the user have the option to disable SharedSound if they want to?

One of my hobbies is composing music, and I would like the facility to concentrate on what the music application is generating – and not have the sound cluttered. However, it could be argued that I could stop all the clients from making any sound…

Non-LPCM streams

This is something that will need a lot of ironing out, and there's two areas to think about.

Sound players with non-LPCM streams

There are benefits in streaming non-LPCM data to something that can decode it natively – for example, an audio file player that a file encoded as MP3 can pass data directly to a CODEC that supports MP3. This improves both CPU usage (since the MP3 data will not need to be encoded as LPCM), and I/O usage (since the MP3 data will be a lot smaller than the LPCM data).

The problem is that SharedSound will not like this one iota – it is expecting the data to be in LPCM format in order to merge it with the other LPCM streams. It would be compounded if two MP3 streams were attempted to be merged, since both of them will need to be re-encoded as LPCM.

Transport mechanisms with non-LPCM streams

As indicated in my earlier document, some transport mechanisms do not cope with LPCM streams, and the data has to be encoded as something else (most notably, Bluetooth).

For this, I would propose a mechanism where encoders and decoders of popular streaming formats are available by calling a SWI to get the encoder/decoder of choice, which gives various entry points for initialisation, giving a chunk of data, and getting data out again.

This should be done with a new module that encoder and decoder modules can register themselves with.

This could also be used by sound players (and recorders) to get data into an LPCM format.

However, it should be used judiciously – if a transport mechanism needs it in format A, and the sound player is fetching it in format A, then it makes sense not to convert it from format A into LPCM, and then from LPCM back into format A unless necessary (for example, other active SharedSounds modules).

SharedSoundBuffer and StreamManager

I was not aware of these modules until about 1 hour ago. However, I did have the idea of pseudo-hardware drivers, but mainly for capturing audio output (if anyone has used OBS on Linux or Windows, then it has this capability).

These pseudo-drivers would simply pretend that an interrupt has been generated, and ask the modules to fill in the sound into its own buffer. It can then do what it wants with the data – for example, writing to a file, or sending over a TCP/IP socket.

In the case of RDPClient, it would probably change the default sound output device to be its own pseudo-driver upon a successful connection, so the applications will continue playing as before.