RISC OS Sound – playtime is over

In the beginning was the beep, and the beep was WaveSynth-Beep.

Introduction

Historically, RISC OS has only directly supported audio playback on a single audio system, with the audio playback performed as a side-function of the video controller (VIDC).

On the original Archimedes, the audio playback was with an up to 8 channel 8-bit µ-Law representation that was adequate, and provided a signal to noise ratio approaching 12-bit audio.

The µ-Law representation chosen was quite clever because it was trivial in software to allow the volume to be reduced (just two single cycle instructions were required – a subtract and a set to zero if the value went negative).

The VIDC1 had to be given the data under DMA with help from the IOC using a double-buffered array. The VIDC1/IOC would be using data in one buffer, whilst the software would be filling the other buffer. The buffer sizes were controllable, although few applications did so.

This format was used in the early RiscPCs with the VIDC20, although was replaced later on with a 16-bit stereo sound system (this was also available as an upgrade for the 8-bit sound system RiscPCs).

Older applications that used the 8-bit µ-Law system were supported by a module that presented a similar interface to the applications, took the output and converted it into the 16-bit sound format required in the new hardware, whilst merging the output from newer applications that were capable of supporting the 16-bit sound. This is the *SoundChannels* module.

Applications that wanted to use the 16-bit sound system had to interact with the *SoundDMA* module – this provided applications with an API to fill in a memory buffer with the audio that is to be played, and it pushed the buffers to the VIDC20 (under DMA control).

Acorn’s demise didn’t mean the end of development.

Applications could use an interface that allowed sound to be played by multiple applications at the same time. This is the *SharedSound* module.

More importantly, several new manufacturers came to the scene, producing hardware that was capable of running RISC OS. The existing *SoundDMA* module was able to interact with the hardware on these new machines via a hardware abstraction layer. This meant that existing applications could use the sound output on the new hardware without having to know anything about it.

Sound input

In all of this, sound input was never a core part of the RISC OS sound system. This meant that companies such as Armadillo, Clares, ESR and VTi produced their own applications that (largely) only worked with their hardware (some software had the capability of using another manufacturer’s sampler, but this depended on the ease of access to the sampler ICs or API).

As sound output was largely catered for by the computer itself, many of the audio sampler interfaces only provided audio input capabilities – although a few (such as Clares Armadeus, Computer Concepts Lark, and Audio Dynamics DMI50) had audio output capabilities. Audio playback on these hardware units would’ve only been possible using the applications provided with the hardware.

In the mid-1990s, Acorn realised that there was no unified sound input API that would allow applications to use any manufacturer’s hardware without having to write special drivers. A study

was started, and several manufacturers went to Acorn's headquarters in Cambridge to discuss the requirements of this API. Sadly, it did not appear to conclude before the demise of Acorn.

Today, most home computers and operating systems have the capability of of supporting multiple audio streams at the same time – both input and output, at many different sample rates and formats. Hardware support is generally through on-board devices, USB or PCI/PCIe (FireWire did enjoy a brief renaissance, but USB has largely replaced it).

Unfortunately, RISC OS is not on that list; it can support a single stereo output stream, and while it has the capability of multiple sample rates, it does not support anything other than 16-bit.

Principal of sounds

This section can be ignored by anyone who is familiar with how sound is captured and played back on a computer system. It is provided to give a background for the proposed changes.

Sound output basics

In the most part, sound output is done with LPCM samples (Linear Pulse Code Modulated). In a 16-bit signed LPCM, a sound is captured as a value from -32,768 (minimum) to 32,767 (maximum). Silence is the value 0.

When playing back audio, a digital to analogue converter (DAC) takes this value and creates an analogue level between two voltages, say -3V to 3V, with the lowest number producing the lowest voltage, and the highest number producing the highest voltage. A value of 0 would produce 0V.

Sample rates

In order to keep things simple, the DAC will update its value at a certain rate – this is the sample rate. For audio CDs, this is 44.1 kHz, which means that every 1/44100th of a second, the output voltage will change.

Professional audio systems will have sample rates much higher than this, with 192 kHz being generally the highest used, which means 1/192,000th of a second.

In order for a computer to output audio waves, something needs to generate the LPCM values that are fed into the DAC. This can be an audio file held on a hard disc, or a synthesizer that is generating the sounds programmatically.

CPUs have the capability to do this with interrupts – these temporarily pause the currently executing application in order to do some work, and then return control back to the application.

Unfortunately, this is a relatively expensive process – everything that the application is doing needs to be stored and everything the interrupt needs to do needs to be retrieved before the interrupt does what it needs to do. When the interrupt has finished, it needs to store its current state before the application can retrieve its state and then continue.

In order to reduce this process, sound output will use buffers – the interrupt will insert the next few hundred samples into an output buffer that is held in memory (or in hardware). The hardware will retrieve the values as and when it needs them. When the buffer is empty, it will signal to the CPU that it needs more data via an interrupt signal. The CPU will then pause the current application and allow the interrupt code to fill the buffer.

This can cause problems if the code required to produce the samples is slow (for example, reading the next block of data off a hard disc). In order to get around this, two buffers are used. The first one is being filled by the interrupt code, and the second one is being emptied by the sound hardware.

When the sound hardware has emptied its buffer, it will indicate an interrupt request to the CPU for it to fill it again – however, the sound hardware can work on the other buffer as the interrupt code has already filled it.

Sound resolution

The number of bits in a sound sample determines how many different voltage levels can be produced. A 16-bit sound sample can have one of 65,536 sample levels; a 24-bit sound sample can have one of 16,777,216 values and a 32-bit sound sample can have one of 4,294,967,296 levels.

In most cases, 16-bit sound samples are used, and professional sound systems tend to use 24-bit samples.

Internally, it is not uncommon for applications to perform calculations to 32-bit levels (or higher), as the CPU offers that level of accuracy by default.

Sound output latency

Latency is important for sound playback since it gives the amount of time an event can occur before the audio will reflect that event.

For example, when a user presses a key on a musical keyboard, the CPU can detect this press, and start generating the sound for this.

If the sound buffers hold five seconds of sound, then this means that the note will not start playing until five seconds after it was pressed. This would be far too late in real-time.

The sound output latency is a combination of the sound sample rate, and the sound buffer size, which tend to be quite small – of the order of a few hundred samples.

At 44.1 kHz, each sample needs to arrive at 22.7µs, so a buffer size of 1000 samples means a latency of 22.7ms.

It may be tempting to lower the buffer size, but this causes increase load in the CPU as it has to spend more time switching between applications and the interrupt code.

On the Archimedes, the default sound buffer was 208 samples, at a sample rate of 20.833 kHz – this meant that the latency was 1ms.

Sound input basics

Sound input is pretty much the same as the sound output – except the hardware (in this case, an analogue to digital converter, or ADC) is filling the buffers, and the interrupt code is emptying them.

Note that on the early sound sampling hardware for RISC OS, buffers were not used – this meant that sampling was performed with no applications able to be run in a co-operative manner.

Some hardware also lacked the capability of producing their own sample clocks, and used the IOC for generating the timing required to sample at a consistent rate.

Real-time processing

Real-time processing of sound requires both input and output capabilities. For example, an echo generation unit needs to record the audio, process it and then play it back. The total latency is the sum of the input and output latency.

Audio transfer encoding

There are a number of ways that hardware can send data to a DAC and receive it from an ADC – however, the main ones all involve converting the audio data to a serial stream in order to send it to the relevant hardware.

I²S

The primary one used for short-range transfer is I²S, I2S, IIS, or Inter-IC Sound. There are three key signals in an I²S bus:

- Serial Clock – this provides the timing for all the signals
- Data – this provides the LPCM data stream. If it is for sound output, then it is the signal into the DAC; for sound input, it is the output from the ADC.
- Word Select – this is also known as “left-right clock”, or “frame sync”, and tells the receiver that the left or right channel is being produced

Some I²S devices can cope with different formats of the data itself.

The most common one is the left-justified I²S format, and is quite neat in that the data stream can have more bits of resolution than the DAC can support. For example, 32-bits of data can be sent to a left-justified 16-bit DAC; the DAC will ignore the remaining 16 bits. The DAC can be easily replaced with a 20-bit or even 24-bit DAC, and the extra quality will be available for use.

Conversely, if a 24-bit DAC is in use, and only 16 bits are provided, then it can use zeros for the remaining 8 bits, allowing it to play back 16-bit samples.

In a left-justified system, the word select changes its state, and the DAC will use the next bit as the most significant bit of the value on the next clock – as shown in figure 1 [WIK001]:

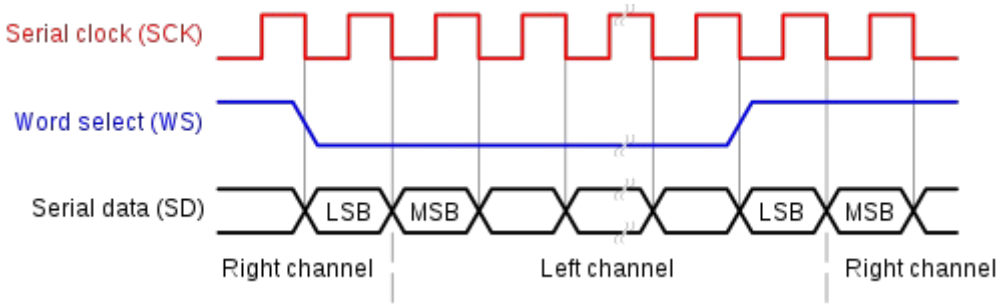

Figure 1: I²S Timing Diagram (image courtesy of Wikipedia)

In a 24-bit I2S system, the serial clock will be running at 24 times the sampling frequency per channel – so at 44.1 kHz, this would mean a clock speed of 2.12 MHz.

Note that two channels are the minimum as the word select clock is necessary for it to synchronise with the first bit.

I²S can support simultaneous playback and record – playback uses one data line, and record uses a second data line. The serial clock and word select signals can be used in both directions simultaneously. In this mode of operation, the playback and recording are intrinsically tied together (although some systems permit a different clock rate for transmit and receive – this is assumed not to be the case for simplicity and minimum compatibility).

Note that I^2S devices can also have a master clock – this normally runs at a multiple of the serial clock, such as 256x. This allows a DAC to perform hardware oversampling of the data.

S/PDIF

S/PDIF, or Sony/Philips Digital InterFace, is a unidirectional two-wire or optical interface that has a similar encoding scheme to AES3. It can support two channels of LPCM data, or more channels with compressed data.

The data encoding scheme is fairly complex, and outside the realms of this document. In essence, command words are interspersed with the data and used to denote the format in use. The commands and data are sent via a Differential Manchester Encoding scheme that negates the need for a clock (the bit stream itself forms the clock).

If audio input and output are required, then two S/PDIF connectors are required.

USB

USB is the dominant standard for off-board sound interfaces, and replaced FireWire with the addition of Isochronous transfers (these permit transfers of data at known time intervals – something that was not present in early USB).

While there is a standard for USB audio devices that manufacturers can follow, many have their own communications standards that require special drivers.

The underlying operation is similar to a buffered system such as I^2S – a block of sample data is transferred over the USB ready for playback or recording. This block will be for a certain number of samples (this number will depend on the negotiation with the USB device and the USB host).

Compression may be available on the audio device to reduce the amount of data being transferred between the device and the host.

Bluetooth

For wireless audio, Bluetooth has become the de-facto standard.

There are several different ways audio data can be transferred – and it is up to the devices as to which they support. Some schemes are protected via patent and have associated licencing costs.

An encoding known as SBC is supported by all devices. This uses a compression scheme to reduce the number of bits needed to transfer the data. Such compression is typically done in the Operating System [VAL001].

Getting sound into RISC OS

As indicated earlier, RISC OS currently has support for a single 16-bit stereo output channel, with sound input provided with custom hardware needing custom APIs.

Figure 2 shows broadly how the RISC OS sound system is structured in a 16-bit only system:

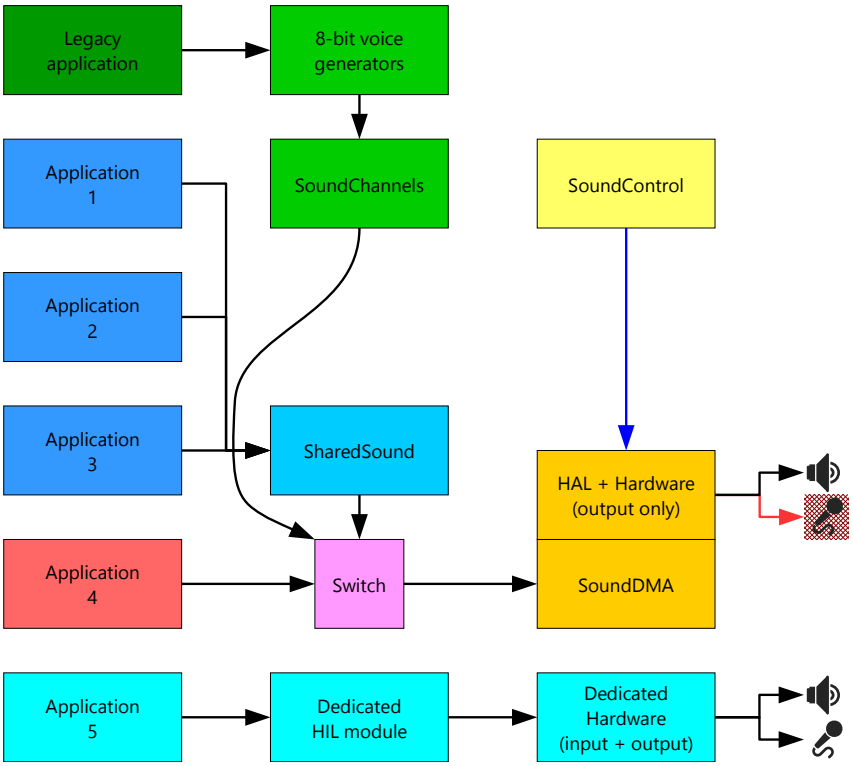

Figure 2: Current RISC OS Sound Options

Legacy applications that need to use the 8-bit µ-Law sound system go through an 8-bit voice generator that passes the audio to the *SoundChannels* module. This provides data that a *LinearHandler* (such as *Application 4*), or *SharedSound* can merge with (if they choose to).

Applications that are written to use the *SharedSound* module will have their audio outputs merged together, and then sent to the *SoundDMA* module, which interacts with the HAL and hardware to provide the sound output.

Applications which require direct access to *SoundDMA* will replace the output of *SharedSound* with their own sound samples – the switch simply permits either *SharedSound* or *Application 4* to use the *SoundDMA* module depending on who last registered as a linear handler.

The *SoundControl* module allows access to some of the underlying hardware features, such as enabling oversampling, or volume levels.

Sound playback or recording using dedicated hardware need their own Hardware Interaction Layer to provide access to the audio streams.

Controller choice in RISC OS Sound System

To support the ARMX6 machine, support was added to allow the audio to be switched between different hardware drivers [LES001].

This meant that a switch was put into place between the *SoundDMA* and the hardware itself, as shown in figure 3:

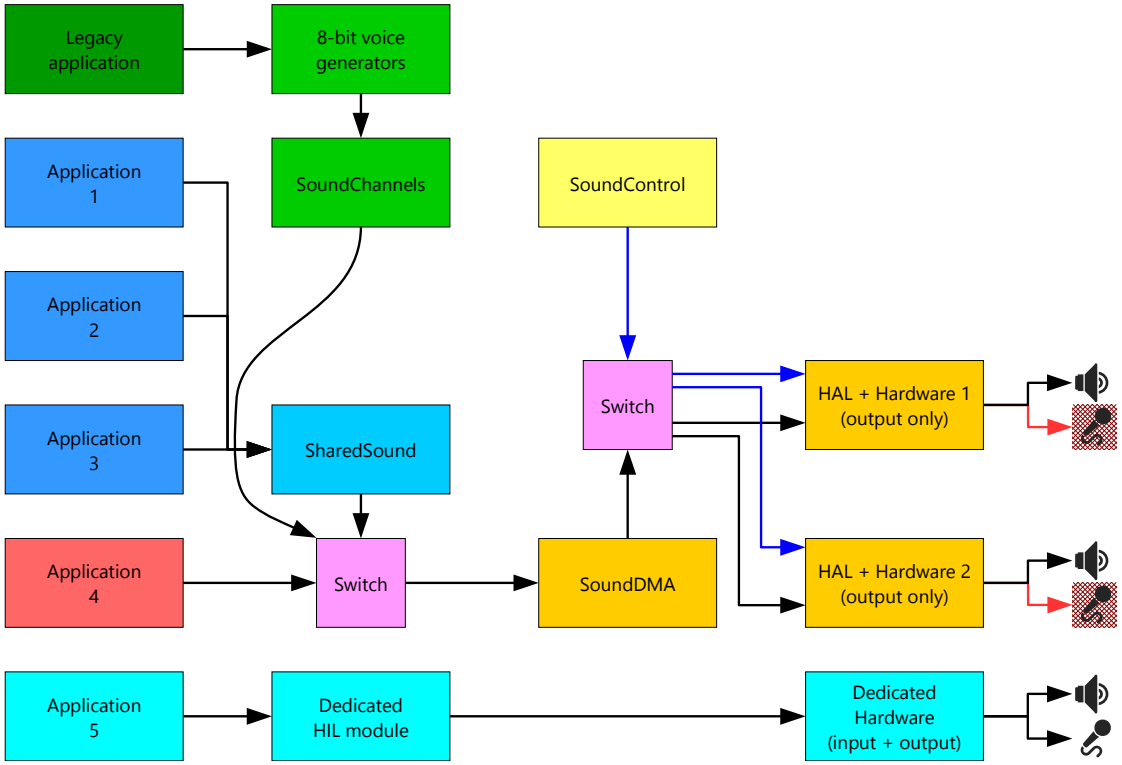

Figure 3: RISC OS Sound system with output switch

This new switch allowed the motherboard hardware to be used, or the sound output to the HDMI controller for presentation on a TV or monitor.

The **Audio* command was added to allow the default to be switched, and also a number of SWI calls added to support the switch:

- Sound_DeviceInfo
- Sound_EnumerateDevices
- Sound_ReadSysInfo
- Sound_SelectDefaultDevice

However, while it did not permit the playback by applications on different hardware at the same time, it did introduce the concept of an *audio controller ID string*, as well as a *device name*.

The audio controller ID string string is of the format:

```
SoundDMA#HAL_0007_1600000
```

Sound devices themselves have a name of the format:

```
i.MX6 HDMI audio controller
```

The new RISC OS Sound System

A view of what the RISC OS sound system could become is shown in figure 4:

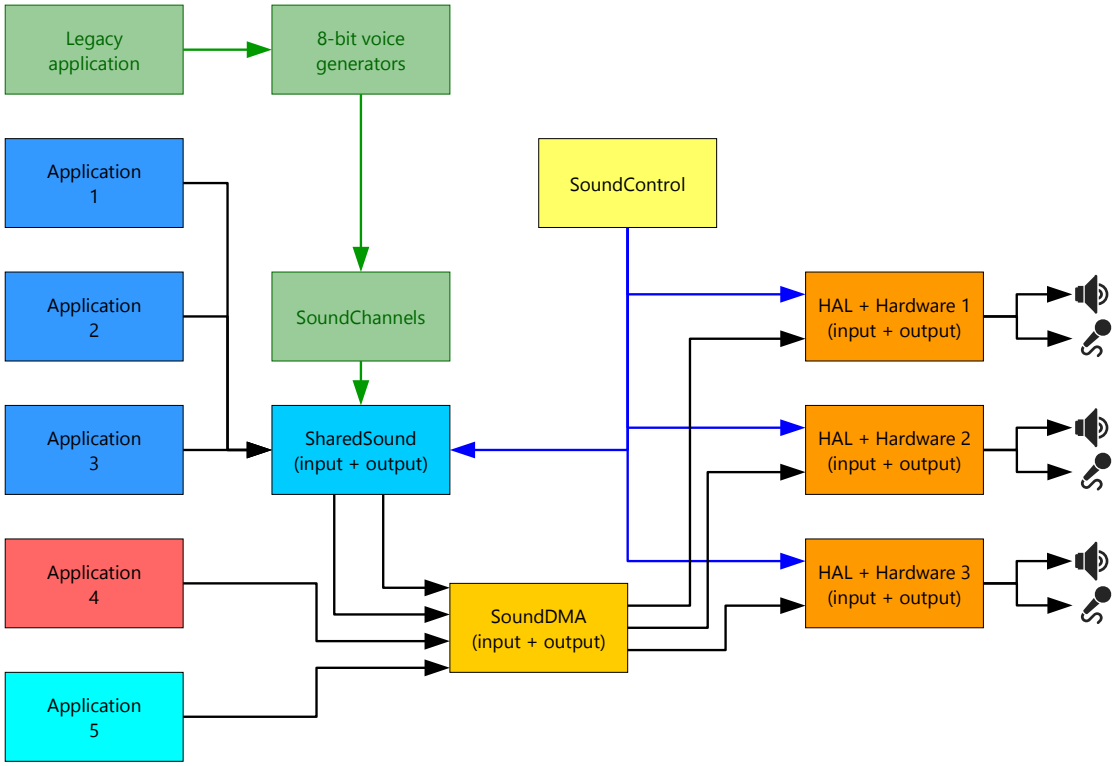

Figure 4: Proposed RISC OS Sound System

There are two main changes in this new model.

Firstly, *SoundDMA* and *SharedSound* have been extended to permit both input and output – this is because the underlying hardware typically will support both input and output using the same sample rates and resolutions.

Secondly, multiple hardware streams are supported. This permits several hardware sound 'cards' to be used at the same time – for example, a SoC could support multiple I^2S streams, and/or you could have several USB interfaces connected to the computer.

Other changes include the potential to drop support for the legacy µ-Law audio capabilities. This would reduce ongoing support for the code – although the code is unlikely to need to change. An alternative is that it could also be rewritten as a module that uses *SharedSound*, so it does not need any special treatment by applications.

Note that *Application 5* could still interface directly to its hardware, but the hope is that it will be modified to use the new API.

Getting to that stage

In order to get to this model, several steps will need to be taken – but note that the central idea is that existing applications will still function as expected.

Sound input with SoundDMA

Adding support for sound input to *SoundDMA* would be the first stage. A chosen hardware platform with support for a bidirectional audio system would be chosen (such as any with an I^2S interface with appropriate hardware).

Applications such as *Application 4* and *Application 5* can register to receive the sounds being sampled while the output is being played (if there is any). In order to pre-empt the next step, a hardware identifier would be used to say which hardware is being used (‘0’ being ‘default’).

Multiple hardware with SoundDMA/SoundControl

The next stage would be to allow support for multiple hardware.

This would require applications such as *Application 4* and *Application 5* to register which of the hardware interfaces they want to send and/or receive data over.

One hardware entity would be chosen as the ‘default’ one (probably by a configuration setting), and any application not providing the interface number would communicate with the default one – as would *SharedSound*.

Multiple channel devices (for example, a 16-channel audio I/O controller) can be created with each channel (or pair of channels) having their own hardware handler – although there are limitations with this approach which means that this would need to be looked at when a demand for such hardware is present.

Extending sound format capabilities

RISC OS currently assumes that 16 bit sound is being produced. *SoundDMA* would be extended to support both 16 bit and 32 bit (24 bit adds a level of complexity for linear sound handlers; storing the data as 32-bit values is much more efficient).

Hardware that requires support for 24 bit audio can still provide this via bit shifting when passing through the buffers – although with left-justified I^2S, this is not necessary.

Multiple hardware with SharedSound

With applications requiring exclusive access to hardware, *SharedSound* can be extended so that applications can share the sound output on different hardware.

The *SharedSound* module would only register itself with the hardware if an application has requested it. This means *SoundDMA* would be in one of three states for each hardware:

1. Idle (not playing any sound)
2. Exclusive (‘owned’ by a single application)
3. Shared (‘owned’ by *SharedSound*)

The hardware itself will not know whether it is being used exclusively, or in a shared environment – it will simply be idle or active.

SharedSound will only allow 16-bit sample formats to be used.

Support for legacy 8-bit µ-Law sounds

Rewriting SoundChannels to provide a 1st tier SharedSound output handler would simplify the handlers code (they no longer need to mix the sounds together – some did not permit this anyway).

This can be done at any point in time.

Note that if it is to be removed, then a new way to produce a system beep will need to be created – via *SharedSound* would be the best approach.

Sound input with SharedSound

This is actually trivial: *SharedSound* would simply call a list of handlers when the hardware indicates that a receive buffer has been filled.

In theory this support could be added to *SoundDMA* – however, it is better for applications to interact with a single module.

Adding support for USB

USB support would be best performed using the standard USB audio protocols. With a non-encoded format, figure 5 shows the communications flow with the USB device:

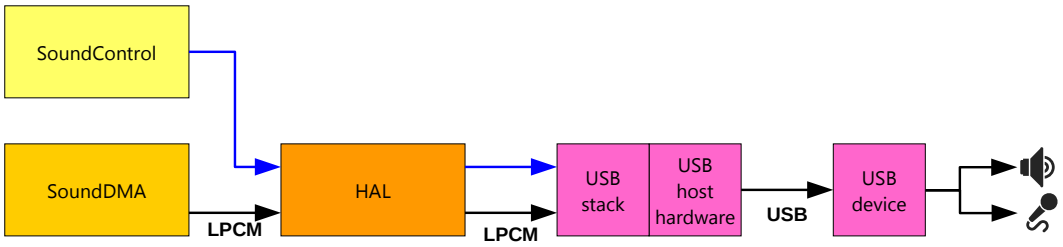

Figure 5: USB Hardware with no encoding

If a USB device needed encoding, then the HAL would have a CODEC to encode and decode the data that the device is expecting, as shown in figure 6:

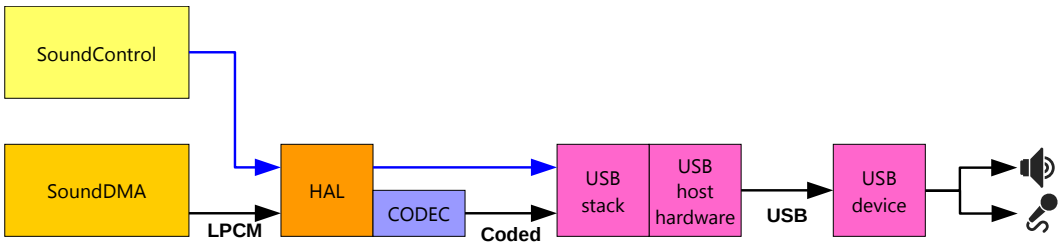

Figure 6: USB Hardware with encoding

The data sent to the HAL is still encoded as LPCM, so applications do not need to know about the encoding mechanism being deployed.

In addition, this means that *SharedSound* can also use this device without any modification.

Adding support for Bluetooth

This is almost identical to the USB case where an encoder is required, and is shown in figure 7:

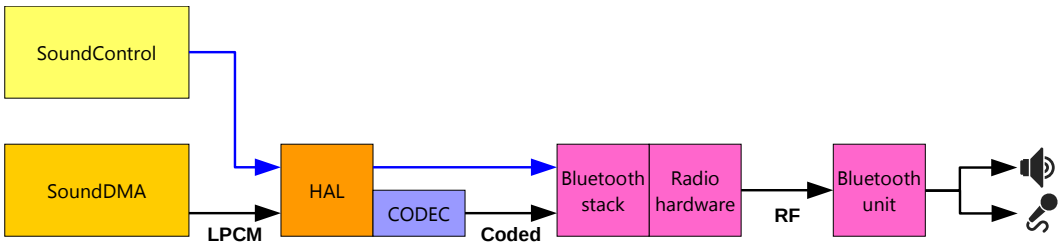

Figure 7: Bluetooth Hardware

Again, the applications do not need to know how the data is going to be encoded. The selection of the CODEC could be given by calls to *SoundControl*.

However, a Bluetooth stack would need to be developer first of all!

Adding support for non-LPCM streams

Some transfer mechanisms can cope with non-LPCM streams, such as MP3. The mechanism for this would be considered later.

Additional calls to *Sound_LinearHandler* would allow the sound player or recorder to send or receive the encoded stream data. The HAL would pass this through to the sound device (with any appropriate framing or control information).

Sound_SampleRate could be used to allow knowledge of the transfer encoding mechanism to be known to the applications.

API calls

The extensions to the *SoundDMA* module would be as follows. Note that some of these are extensions on top of the ones added for ARMX6.

Device identification

New API elements would be needed to support multiple hardware devices from an identification perspective – as service calls, or SWI calls.

Service_Sound 8, 9, 10

These service calls are only called if the default sound interface settings are changed.

Service_Sound 12

Requests modules register as handlers for sound hardware

On entry

R0 = 12
R1 = &54 (reason code)
R2 = 0 (for future expansion)

On exit

All registers preserved

Use

This is sent by *SoundDMA* to ask all HAL modules to register themselves with the *SoundDMA* module as a hardware handler, and is sent during initialisation of the *SoundDMA* module.

Service_Sound 13

Informs modules that a new sound device is present

On entry

R0 = 13
R1 = &54 (reason code)
R2 = pointer to the null-terminated audio controller ID string
R3 = pointer to the null-terminated string of the hardware name
R4 = bit flag for device capabilities (TBD)
R5 = pointer to handler control block (TBD)

On exit

All registers preserved

Use

This is sent by *SoundDMA* to inform other modules that a new sound device is present.

Service_Sound 14

Informs modules that a sound device has been removed

On entry

R0 = 14
R1 = &54 (reason code)
R2 = pointer to the null-terminated audio controller ID string

On exit

All registers preserved

Use

This is sent by *SoundDMA* to inform other modules that a sound device has been removed.

Sound_AddDevice

Registers a handler for a new sound hardware device

On entry

R0 = pointer to null-terminated audio controller ID string
R1 = pointer to the null-terminated string of the hardware name
R2 = bit flag for device capabilities (TBD)
R3 = pointer to handler control block (TBD)

On exit

Registers preserved

Use

This SWI is called by a module to register a new handler for sound hardware. A module can register for multiple sound hardware units.

This would normally be called upon reception of *Service_Sound 12*, but could be at the initialisation of a module, when a USB device is plugged in, or when a Bluetooth audio unit is paired.

The hardware name is of the form “i.MX6 HDMI audio controller”. While it does not need to be unique, it is advisable for it to be so that users can be presented with an identifiable list of hardware devices.

The audio controller ID string is of the form “SoundDMA#HAL_0007_16000000”, and must be unique within a system (across reboots if possible).

Sound_RemoveDevice

Deregisters a handler for a sound hardware device

On entry

R0 = pointer to null-terminated audio controller ID string

On exit

Registers preserved

Use

This SWI is called by a module to inform *Sound* that the device has been removed – for example, if the USB device is unplugged, or a Bluetooth audio unit has lost communication.

Registering as a sound playback provider

Currently this is done in RISC OS with *Sound_LinearHandler*. The changes are as follows (note that some of the changes are purely wording):

Sound_LinearHandler 0

Returns the current 16 bit linear stereo **output** handler **for the default sound hardware**.

On entry

R0 = 0 (reason code)

On exit

R0 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed in R0 to current handler, or –1 if no handler is installed

Use

This call returns the current 16 bit linear stereo **output** sound handler, giving the address of the handler code, and the parameter passed to it in R0.

Note that this expects that a 16 bit stereo sound format has been selected

Sound_LinearHandler 1

Registers or removes the 16 bit linear stereo sound **output** handler **for the default sound hardware.**

On entry

R0 = 1 (reason code)
R1 = pointer to new handler code, or 0 to remove the handler
R2 = parameter passed in R0 to handler, or –1 if removing the handler

On exit

R0 preserved
R1 = pointer to previous handler code, or 0 if no handler was installed
R2 = parameter passed in R0 to previous handler, or –1 if no handler was installed

Use

This call registers or removes the 16 bit linear stereo **output** sound handler. When registering, you give the address of the handler code – which is called to fill the sound DMA buffer – and a parameter passed to the handler in R0. The address and parameter of the previous linear handler (if any) are returned.

Only one linear **output** handler can be registered with the *SoundDMA* module for each sound hardware. You should therefore only register your linear handler immediately before starting to play sound, and should re-register the previous handler as soon as you have finished.

Sound_LinearHandler 2

Returns the current linear stereo output handler for the given sound hardware

On entry

R0 = 0
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed into R0 to current handler, or -1 if no handler is installed

Use

This call returns the current linear stereo output sound handler, giving the address of the handler code, and the parameter passed into R0.

Note that the format does not need to be 16 bits.

Sound_LinearHandler 3

Registers or removes the linear stereo sound output handler for the given sound hardware

On entry

R0 = 0
R1 = pointer to new handler code, or 0 to remove the handler
R2 = parameter passed into R0 to handler, or -1 if removing the handler
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed into R0 to current handler, or -1 if no handler is installed

Use

This call registers or removes the linear stereo sound handler. When registering, you give the address of the handler code – which is called to fill the sound DMA buffer – and a parameter passed to the handler in R0. The address and parameter of the previous linear handler (if any) are returned.

Only one linear output handler can be registered with the *SoundDMA* module for each sound hardware. You should therefore only register your linear handler immediately before starting to play sound, and should re-register the previous handler as soon as you have finished.

Note that the format does not need to be 16 bits.

Adding support for sound sampling

This is done by extending the *Sound_LinearHandler* SWI call with further options.

Sound_LinearHandler 4

Returns the current linear stereo input handler for the given sound hardware

On entry

R0 = 0
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed into R0 to current handler, or -1 if no handler is installed

Use

This call returns the current linear stereo input sound handler, giving the address of the handler code, and the parameter passed into R0.

Note that the format does not need to be 16 bits – although it does need to be the same as the output format.

Sound_LinearHandler 5

Registers or removes the linear stereo sound input handler for the given sound hardware

On entry

R0 = 0
R1 = pointer to new handler code, or 0 to remove the handler
R2 = parameter passed into R0 to handler, or -1 if removing the handler
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed into R0 to current handler, or -1 if no handler is installed

Use

This call registers or removes the linear stereo sound input handler. When registering, you give the address of the handler code – which is called to fill the sound DMA buffer – and a parameter passed to the handler in R0. The address and parameter of the previous linear handler (if any) are returned.

Only one linear input handler can be registered with the *SoundDMA* module for each sound hardware. You should therefore only register your linear handler immediately before starting to record sound, and should re-register the previous handler as soon as you have finished.

Note that the format does not need to be 16 bits – although it does need to be the same as the output format.

Sound linear handlers

The sound linear handlers will need to be modified slightly. By either removing the *SoundChannels* module, or making it a 1st class interface to *SharedSound*, the handler can be simplified.

Note that these have already been extended to support different sample formats.

Sound output handler

This is the entry point for sound playback code

On entry

R0 = workspace (as specified in R2 to *Sound_LinearHandler 1* or *Sound_LinearHandler 3*)
R1 = base address of buffer (word aligned)
R2 = end address of buffer (exclusive)
R3 = bit flags:

Bits 0..2	0 = Data in buffer is invalid and must be completely overwritten
	1 = Reserved (was µ-Law data present)
	2 = Reserved (was data in buffer is zeroes)
	3..7 = Reserved
Bits 3..31	Reserved

R4 = sample rate (as per *Sound_SampleRate 5*)

On exit

R0-R10 can be corrupted
R11-R13 preserved

Use

The linear sound output handler is the main point of entry for sounds within the sound system.

Each time *SoundDMA* processes an output buffer fill interrupt from the hardware, the sound output handler will be called in order to allow it to send its data to the buffer.

On entry, R1 will point to the base of a 16-byte aligned buffer. The output buffer will need to be filled with data that will be sent to the sound interface. The format of the data depends on the selected sample format.

16-bit data is organised with bits 0..15 holding the signed 16-bit integer for the right channel, and bits 16..31 holding the signed 16-bit integer for the left channel.

32-bit data is organised as pairs of 32-bit words, with first word holding the signed 32-bit integer for the right channel, and the second word holding the signed 32-bit integer for the left channel.

While the hardware can support 24-bit samples, for efficiency, only 16-bit or 32-bit data formats can be generated by the linear output handler. The HAL will ignore the bottom 8-bits from the 32-bit data when transferring to the hardware (the data is left-justified in memory). This means that if a 24-bit format is selected, the linear handler will still need to generate 32-bits of data.

The buffer is filled up until the current output address is equal to the end address.

Note that the linear output handler does not need to cope with oversampling nor mono conversion. These are performed by *SoundDMA* upon receipt of this data.

Also note that the code should not determine whether it needs to generate 16-bit or 32-bit data each time – when the handler is registered, it should pick the right code.

Sound input handler

This is the entry point for sound recording code

On entry

R0 = workspace (as specified in R2 to *Sound_LinearHandler 5*)
R1 = base address of buffer (word aligned)
R2 = end address of buffer (exclusive)
R3 = reserved (0)
R4 = sample rate (as per *Sound_SampleRate 5*)

On exit

R0-R10 can be corrupted
R11-R13 preserved

Use

The linear sound input handler is the main point of entry for incoming sounds within the sound system.

Each time *SoundDMA* processes an input buffer full interrupt from the hardware, the sound input handler will be called in order to allow it to send its data to the buffer.

On entry, R1 will point to the base of a 16-byte aligned buffer. The input buffer will contain the data received by the sound hardware. The format of the data depends on the selected sample format.

16-bit data is organised with bits 0..15 holding the signed 16-bit integer for the right channel, and bits 16..31 holding the signed 16-bit integer for the left channel.

32-bit data is organised as pairs of 32-bit words, with first word holding the signed 32-bit integer for the right channel, and the second word holding the signed 32-bit integer for the left channel.

24-bit data can be retrieved from the 32-bit sequence by dropping the bottom 8-bits.

The buffer is emptied until the current input address is equal to the end address.

Note that oversampling and mono conversion has no impact on sound recording.

Additional support for multiple devices and formats

Some existing calls will need to be extended for multiple devices, as well as multiple sound formats.

Service_Sound 15

Informs modules that a sound device has been reconfigured

On entry

R0 = 15
R1 = &54 (reason code)
R2 = Bits 0..27 = New sample rate, in units of 1/1024 Hz
Bits 28..29 = Reserved (0)
Bit 30..31 = 00 => 16-bit
01 => 24-bit
10 => 32-bit
11 => Reserved
R3 = New buffer size (samples per buffer)
R4 = pointer to the null-terminated audio controller ID string

On exit

All registers preserved

Use

This is sent by *SoundDMA* to inform other modules that a sound device has been reconfigured. If the default device has been reconfigured, then this is sent in addition to *Service_Sound 8*.

The sample rate is as per *Sound_SampleRate 5*.

Service_Sound 16

Informs modules that a sound device is about to start output or input

On entry

R0 = 16
R1 = &54 (reason code)
R4 = pointer to the null-terminated audio controller ID string

On exit

All registers preserved

Use

This is sent by *SoundDMA* to inform other modules that a sound device has been reconfigured. If the default device has been started, then this is sent in addition to *Service_Sound 9*.

Service_Sound 17

Informs modules that a sound device is about to stop output or input

On entry

R0 = 15
R1 = &54 (reason code)
R4 = pointer to the null-terminated audio controller ID string

On exit

All registers preserved

Use

This is sent by *SoundDMA* to inform other modules that a sound device has been reconfigured. If the default device has been stopped, then this is sent in addition to *Service_Sound 10*.

Sound_SampleRate

Old applications would use *Sound_SampleRate* with reason codes 0-3, and new applications would use *Sound_SampleRate* with reason codes 4-7.

Sound_SampleRate 0

Reads the number of available **16-bit** sample rates **on the default sound hardware**

On entry

R0 = 0 (reason code)

On exit

R0 preserved
R1 = number of available sample rates, or *nsr*

Use

This call reads the number of available sample rates, or *nsr*.

You need to know this value to ensure that the sample rate index you must pass to most other *Sound_SampleRate* reason codes is in the required range 1 – *nsr*.

Sound_SampleRate 1

Reads the current sample rate index, and the corresponding **16-bit** sample rate **for the default sound hardware**

On entry

R0 = 1 (reason code)

On exit

R0 preserved
R1 = current sample rate index, in the range 1 – *nsr*
R2 = current sample rate, in units of 1/1024 Hz

Use

This call reads the current **16-bit** sample rate index, and the corresponding **16-bit** sample rate **for the default sound hardware**, measured in units of 1/1024 Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000.

Sound_SampleRate 2

Reads the sample rate corresponding to a **16-bit** sample rate index **for the default sound hardware**

On entry

R0 = 2 (reason code)
R1 = sample rate index to be read, in the range 1 - *nsr*

On exit

R0, R1 preserved
R2 = sample rate corresponding to the given sample rate index, in units of 1/1024 Hz

Use

This call reads the **16-bit** sample rate corresponding to a **16-bit** sample rate index, in units of 1/1024 Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000.

Once you have called *Sound_SampleRate* 0 to find the number of available **16-bit** sample rates **on the default sound hardware** (*nsr*), you can then:

- Enumerate the available sample rates by repeatedly making this call with R1 set to all valid indexes (ie 1 - nsr inclusive).
- Find a particular sample rate (or the closest approximation, if acceptable) by using this call in a ‘binary chop’ algorithm, since sample rates increase monotonically with increasing sample rate index.

Sound_SampleRate 3

Sets the current **16-bit** sample rate index **for the default sound hardware**

On entry

R0 = 3 (reason code)
R1 = new sample rate index, in the range 1 - *nsr*

On exit

R0 preserved
R1 = previous sample rate index
R2 = previous sample rate, in units of 1/1024 Hz

Use

This call sets the current sample rate index **for the default sound hardware.**

It returns the previous **16-bit** sample rate index, and the corresponding **16-bit** sample rate measured in units of 1/1024 Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000.

Sound_SampleRate 4

Reads the number of available sample rate and formats on the given sound hardware

On entry

R0 = 4 (reason code)
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = number of available sample rates, or *nsr*

Use

This call reads the number of available sample rates, or *nsr*.

You need to know this value to ensure that the sample rate index you must pass to most other *Sound_SampleRate* reason codes is in the required range 1 – *nsr*.

Sound_SampleRate 5

Reads the current sample rate index, and the corresponding sample rate for the given sound hardware

On entry

R0 = 5 (reason code)
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = current sample rate index, in the range 1 – *nsr*

R2 =	Bits 0..27 =	current sample rate, in units of 1/1024 Hz
	Bits 28..29 =	Reserved (0)
	Bit 30..31 =	00 => 16-bit
		01 => 24-bit

10 => 32-bit
11 => Reserved

Use

This call reads the current sample rate index, and the corresponding sample rate for the given hardware, measured in units of 1/1024 Hz. For example a 16-bit sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000. A 32-bit 192 kHz sample would be returned in R2 as &8BB80000 (&BB80000 being 192000 x 1024).

Note that the maximum permitted sample rate is 262.143 kHz.

Applications must ignore any sample rates that are using the reserved bit values (bits 28..39 non-zero, or bits 30..31 equal to 11). These are likely to be used at a later date to support features such as non-LPCM transport mechanisms, and / or devices with more than 2 channels.

Sound_SampleRate 6

Reads the sample rate and format corresponding to a sample rate index for the given sound hardware

On entry

R0 = 6 (reason code)
R1 = sample rate index to be read, in the range 1 – *nsr*
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R1, R3 preserved

R2 =	Bits 0..27 =	current sample rate, in units of 1/1024 Hz
	Bits 28..29 =	Reserved (0)
	Bit 30..31 =	00 => 16-bit
		01 => 24-bit
		10 => 32-bit
		11 => Reserved

Use

This call reads the sample rate corresponding to a sample rate index, in units of 1/1024 Hz. For example a 16-bit sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000. A 32-bit 192 kHz sample would be returned in R2 as &8BB80000 (&BB80000 being 192000 x 1024).

Once you have called *Sound_SampleRate* 4 to find the number of available sample rates on the given sound hardware (*nsr*), you can then:

- Enumerate the available sample rates by repeatedly making this call with R1 set to all valid indexes (ie 1 - nsr inclusive).

- Find a particular sample rate (or the closest approximation, if acceptable) by using this call in a ‘binary chop’ algorithm, since sample rates increase monotonically with increasing sample rate index.

Sound_SampleRate 7

Sets the current sample rate and format index for the given sound hardware

On entry

R0 = 7 (reason code)
R1 = new sample rate index, in the range 1 – *nsr*
R3 = pointer to audio controller ID string (or 0 for default)

On exit

R0, R3 preserved
R1 = previous sample rate index
R2 = Bits 0..30 = previous sample rate, in units of 1/1024 Hz
Bits 28..29 = Reserved (0)
Bit 30..31 = 00 => sample format was 16-bit
01 => sample format was 24-bit
10 => sample format was 32-bit
11 => Reserved

Use

This call sets the current sample rate index for the given sound hardware.

It returns the previous sample rate index, and the corresponding sample rate measured in units of 1/1024 Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000 x 1024, which is 20480000. A 32-bit 192 kHz sample would be returned in R2 as &8BB80000 (&BB80000 being 192000 x 1024).

Sound_ControllerInfo 3

Gets the list of supported 16-bit sample rates

On entry

R0 = pointer to audio controller ID string (or 0 for default)
R1 = pointer to buffer for result, or 0 to check required length
R2 = length of buffer
R3 = 3 (reason code, a.k.a. property number)

On exit

R0 = 0 or pointer to error block
R1 = pointer to buffer updated with value of property
R2 = length of data in buffer

Use

This call gets the list of sample rates (1 word per entry), as per *Sound_SampleRate 2*.

Sound_ControllerInfo 4

Gets the list of all supported sample rates (all formats)

On entry

R0 = pointer to audio controller ID string (or 0 for default)
R1 = pointer to buffer for result, or 0 to check required length
R2 = length of buffer
R3 = 4 (reason code)

On exit

R0 = 0 or pointer to error block
R1 = pointer to buffer updated with value of property
R2 = length of data in buffer

Use

This call gets the list of sample rates (1 word per entry), as per *Sound_SampleRate 5*.

Sound_Speaker

Turns the audio on and off

On entry

R0 = reason code
R1 = value dependent on reason code
Reason code on entry is as follows:

0	Check audio output on default controller
1	Turn audio output off for the default controller
2	Turn audio output on for the default controller
3	Check audio output for the audio controller ID string pointed to by R1
4	Turn audio output off for the audio controller ID string pointed to by R1
5	Turn audio output on for the audio controller ID string pointed to by R1
6	Check audio input for the audio controller ID string pointed to by in R1
7	Turn audio input off for the audio controller ID string pointed to by in R1
8	Turn audio input on for the audio controller ID string pointed to by in R1

On exit

If R0 = 0, 1 or 2 on entry, then R0 = 1 to indicate audio output was off for the default audio device, 2 to indicate audio was on for the default audio device
If R0 = 3, 4 or 5 on entry, then R0 = 4 to indicate audio output was off, 5 to indicate audio output was on
If R0 = 6, 7 or 8 on entry, then R0 = 7 to indicate audio input was off, 8 to indicate audio input

was on
R1 preserved

Use

This call allows audio input and output to be turned off.

Sound_Volume

Sets the volume level

On entry

R0 =	Bits 0..6	Volume level (1..127, or 0 to read the volume)	
	Bits 8..29	Reserved (0)	
	Bits 30..31	00	Default controller
		01	Audio output on controller given in R1
		10	Audio input on controller given in R1
		11	Reserved

R1 = value dependent on bits 30..31 of R0
If bits 30..31 == 00 on R0, then R1 is ignored
Otherwise R1 = pointer to audio controller ID string (or 0 for default)

On exit

R0 = previous volume level for the given controller (or default if bits 30..31 of R0 == 00 on entry)

Use

This call allows the audio output and input volume levels to be set

Sound_Mode 0

Detects and controls features of the audio system

On entry

R0 = 0 (reason code)

On exit

R0 = 1 (16-bit present)
R1 = bit flags:

Bit 1 set	16-bit linear output (44.1 kHz, 22.05 kHz, 11.025 kHz clock rates)
Bit 2 set	16-bit linear output, internal clock
Bit 3 set	16-bit linear output, external clock
Bit 4 set	Oversampling mode (1 = enabled)
Bit 5 set	Mono conversion status (1 = enabled)
Bit 6 set	Multiple device support enabled (1 = enabled)
Bits 7..31	Reserved (0)

Use

This call reads the current sound system configuration. R0 will always equal 1 on exit, since there are no 8-bit only sound systems. Bits 1 through 5 will give values for the default sound interface.

Note that bit 6 will be set even if only one device is present.

Sound_Mode 1

Enables or disables automatic oversampling **for the default sound device**

On entry

R0 = 1 (reason code)
R1 = new state of automatic linear 2x oversampling: 0 = disabled, 1 = enabled

On exit

R0 preserved
R1 = previous state of automatic linear 2x oversampling:
0 = disabled,
1 = enabled

Use

This call enables or disables automatic linear 2x oversampling for the default sound device

Sound_Mode 3

Enables or disables mono conversion mode **for the default sound device**

On entry

R0 = 3 (reason code)
R1 = new state of mono conversion: 0 = disabled, otherwise = enabled

On exit

R0 preserved
R1 = previous state of mono control:
0 = disabled,
1 = enabled

Use

This call enables or disables mono conversion mode for the default sound device

Sound_Mode 4

Enables or disables automatic oversampling level for the given sound device

On entry

R0 = 4 (reason code)
R1 = new state of automatic linear oversampling:
    0 = no oversampling
    1 = 2x oversampling,
    2 = 4x oversampling, …
    -1 = query current oversampling level
R2 = pointer to audio controller ID string (or 0 for default)

On exit

R0 preserved
R1 = previous state of automatic linear oversampling:
    0 = disabled,
    1 = 2x oversampling,
    2 = 4x oversampling, …
R2 preserved

Use

This call enables or disables automatic linear oversampling for the given sound device.

The oversampling selected will be the highest quality available up to the oversampling rate required. For example, if a device only supports 4x oversampling, and R1 has requested 8x oversampling, then 4x oversampling will be used. If subsequently R1 = -1 on entry, then R1 will reflect 4x oversampling is in place.

Note that oversampling has no effect on audio input.

Sound_Mode 5

Enables or disables mono conversion mode for the given sound device

On entry

R0 = 5 (reason code)
R1 = new state of mono conversion: 0 = disabled, otherwise = enabled
R2 = pointer to audio controller ID string (or 0 for default)

On exit

R0 preserved
R1 = previous state of mono control:
    0 = disabled,
    1 = enabled
R2 preserved

Use

This call enables or disables mono conversion mode for the given sound device.

Note that mono conversion has no effect on audio input.

Changes to SoundControl

SoundControl needs no modifications – it already has support for the audio controller ID string in all SWI calls.

Changes to SharedSound

SharedSound is likely to remain 16-bit only – however, adding support for multiple hardware devices and input can be achieved. These changes are only outlined here. Note that some of these may not be implemented in the current *SharedSound* module (as indicated in [LES001] and [ROO001]).

SharedSound handlers

These should need no modification – the sample rate will always be given as a 16-bit sample rate, and the mix bit will be determined if it is the first handler in the chain to be called.

SharedSound_InstallHandler

R2 is extended to permit a flag to state that an audio controller ID string is present in R5. Otherwise, the default sound hardware is used.

R2 would also be extended to permit a flag to indicate that the input handler is being assigned. This can be with or without the device identifier flag.

SharedSound_RemoveHandler

Needs no modification. Handler number is enough to identify if it is on a specific device or not.

SharedSound_HandlerInfo

R1 is extended to permit a flag to state that a device specific identifier is present in R6. Otherwise, the default sound hardware is used.

SharedSound_HandlerVolume

Needs no modification.

SharedSound_HandlerSampleType, SharedSound_HandlerPause

Undocumented at present, so unknown.

SharedSound_SampleRate

Needs no modifications.

SharedSound_InstallDriver

Only linear drivers will be permitted. R0 is extended to permit other reasons that allow an audio controller ID string to be present in R4.

SharedSound_RemoveDriver

R0 is extended to permit other reasons to allow other reasons that allow an audio controller ID string to be present in R4.

SharedSound_DriverInfo

R0 extended to allow a bit set to add the audio controller ID string for the handler in R5. If this bit is clear, then only information on drivers that are present on the default sound handler is returned.

SharedSound_DriverVolume

Needs no modifications.

SharedSound_DriverMixer

Needs no modifications.

SharedSound_CheckDriver

Only supports the default sound hardware. Additional SWI required to check the driver on a specific hardware interface.

SharedSound_ControlWord

Unknown at present as to what this does.

SharedSound_HandlerType

Needs no modifications

SharedSound_CheckSoundDriver

New SWI to check the driver on a specific hardware interface.

R0 = pointer to audio controller ID string

Bibliography

WIK001: Wikipedia, I2S, 2020, https://en.wikipedia.org/wiki/I%C2%B2S

VAL001: ValdikSS, Audio over Bluetooth: most detailed information about profiles, codecs, and devices, 2019, https://habr.com/en/post/456182/

LES001: J.C.G. Lesurf, The RISC OS Sound System, 2016, http://jcgl.orpheusweb.co.uk/temp/ROSSDocument.pdf

ROO001: RISC OS Open, Sound SWI Calls, 2017, https://www.riscosopen.org/wiki/documentation/show/Sound%20SWI%20Calls

About the author

I (Jason Tribbeck) was the author of several RISC OS applications for sound sampling, and designer for sound output hardware sold under the Vertical Twist and VTi brands. I have been creating synthesizers for many years – although none commercially since the VTX2000.

I attended the meetings at Acorn in the mid-1990s, and took home some documentation at the time – but this has become lost (I recall seeing it in 2017, but I think I was having a clear-out at the time).

I would like to thank J.C.G. Lesurf in particular for his excellent guide as to what has happened with the RISC OS sound system since I last did any serious work (16-bit RiscPC days!), and to ROOL for their website that contains a fair bit of documentation.

And an exceptional thank you to Stefan Fröhling for prompting me to look at this.

I would really like to finish something I started many years ago – !Sonor2 – which was meant to be a better version of my sound editing software with support for 16-bit data, but the lack of hardware dented my enthusiasm. Now, with today’s hardware, it should be very possible to achieve this...