The RISC OS Sound System.

Introduction

RISC OS (RO) first appeared on ARM-based hardware with the OS and its hardware designed with each other in mind. In those early days the main interest in audio was in terms of 8-bit 'trackers', 'VoiceGenerators', etc, for purposes like games or simple interactive sounds. Over the following decades the RO Sound System (ROSS) evolved and expanded. During 2013 it took another step forward when RO was ported onto hardware like the PandaBoards. This has much better sound hardware than the early bespoke machines. It is quite capable of playing even 96k/ 24bit 'high resolution' audio. And as the decades have passed, computer users have come to expect steadily higher audio quality.

The ROSS has continued to develop, and now forms a basis for being able to use the new RO hardware to play 16-bit audio with superb quality. However one of the consequences of the decades of development is that the system has become more complex and harder to understand – both for programmers and users. This document is written to try and bring together a clearer description of how the ROSS operates.

The existing Programmer's Reference Manuals (PRMs) provide many details of the ROSS. But they omit modern developments. And they don't always explain clearly how all the details fit together. They also spend much time and effort on dealing with 8-bit 'Voice Generators' – an approach based on a now-ancient era. Alas, this also makes some of what the PRM says confusing to a modern reader. e.g. the use of the term "channel" to mean a Voice Generator (i.e. waveform synthesiser in software) 'channel'. Whereas today we tend to think of audio channels in terms like stereo having two channels – left and right.

Hence one of the purposes of this document is to change the focus, pay particular attention to 16-bit audio and more recent developments, and describe behaviour in terms appropriate for modern audio. The aim being to provide up-to-date documentation for those interested in high audio quality applications. In doing so I will largely ignore the 'legacy' side of 8-bit and voice generators that make beeps and twangs! If you wish to know more about those I'd recommend reading the PRMs and older documentation as they cover them in gory detail. I shall avoid duplicating that effort!

Overview

If you open a TaskWindow on a modern RO machine, type in the command **`*mod.`** and press return you will get a list of all the modules present. This will include mentions of the following modules

```
SoundDMA
SoundControl
SoundChannels
SoundScheduler
SharedSound
```

etc.

These provide the main user and program interface with the ROSS.

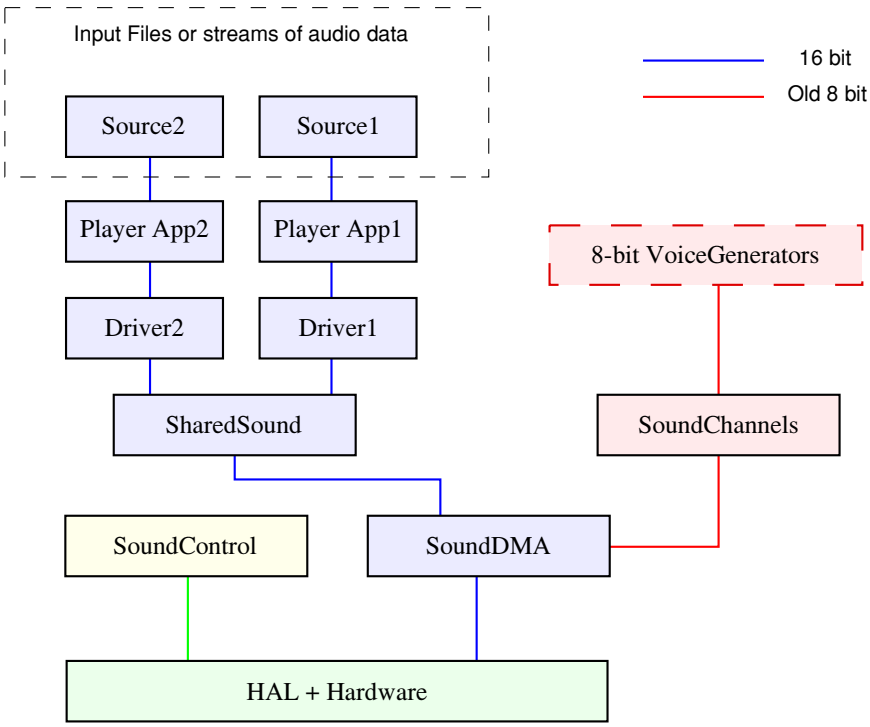


Fig 1 Simplified overview of audio playback pathways

Figure 1 gives a very simplified overview of the sound replay system. As you might expect for any system which many programmers have kept 'improving' over decades the reality is more complex. There are various other things which may be going on that I've omitted or ignored above for the sake of making the basic approach clearer at this point. Also some functions can be provided in more than one way.

Note that in this document the terms 'Driver' and 'Handler' will often be used. These are sometimes used with inconsistent meanings when you compare various audio documents on aspects of the ROSS. Here I will use 'Driver' to mean some item of software that is 'transmitting' a sequence of audio sample values and 'Handler' to mean an item that is 'receiving' the sequence. Usually the Handler (receiver) decides when a fresh batch of sample values is required, etc.

The early Acorn RO systems essentially had 8-bit VoiceGenerators that were combined using the **`SoundChannels`** module and the data then passed to the playback hardware via **`SoundDMA`**. Since I won't say much more about VoiceGenerators or 8-bit I've placed them to one side of the diagram. The **`SharedSound`** and **`SoundControl`** modules are relatively recent additions. **`SoundControl`** provides an interface to read and adjust the audio hardware settings. As such, the details of its operation depend on the hardware you may be using. **`SharedSound`** allows more than one sound source (e.g. from a file or internet stream) to be played simultaneously. Prior to the addition of **`SharedSound`** it was assumed that only one 16-bit source would be played at a time.

One real complication for understanding the ROSS is that various functions can be performed in more than one way, and indeed, can sometimes be applied at more than one stage in the 'chain' from source to output! This makes any initial explanations more difficult as it is easy to get befogged by a series of "on the other hand..." digressions to explain alternatives and their relationship. The most common confusion that arises for this reason is the issue of 'volume'

controls. But there are other less obvious issues. To try and make an initial understanding easier, in the next few sections I'll use some specific examples of how audio can be played, and explain some of the main features of each section of the ROSS.

SharedSound

The **SoundDMA** module is essentially a 'doorway' between user software + modules and the audio hardware. When you play audio the other desktop applications, modules, etc, are working to deliver the data samples you wish to hear to this doorway and making the arrangements that allow them to be understood and played as you wish. However in practice it is now usual for audio players to drive **SharedSound** and leave that with the task of passing the series of audio samples on to **SoundDMA**.

Using **SharedSound** provides various features. In particular the ability to 'mix' sound streams so you can hear more than one sound source simultaneously. Software that wishes to play audio can still bypass **SharedSound** if this is what is required. But the **SoundDMA** 16-bit linear handler can only accept one input stream at a time. As a result, most software will now choose to drive **SharedSound** and leave it with the task of driving **SoundDMA**. For that reason I'll start a detailed examination of the ROSS with **SharedSound**.

There is an excellent 'player' demo program written by Jeffrey Lee which includes the source code in 'C'. This provides a clear example of how to use **SharedSound**.[1] So I'll base the following explanation on that example. It can play 16-bit stereo LPCM wave files which have the common 44-byte header. The following description gives the main steps in the process. For clarity I've ignored various error checking processes, etc, and only describe the main steps required to actually play though the audio file.

- Firstly, the program finds the input Wave file and loads the series of sample values it contains into a 'buffer' store in memory. The samples are loaded as Right-Left pairs, using 16-bits per value. i.e. each stereo sample pair occupies a 32-bit word in memory.
- The program then calls the SWI **SharedSound_InstallHandler** giving it the address of the part of the program that will supply it with the required series of samples. This also tells **SharedSound** the locations of some other values which the module and main program can then use to exchange control information. The SWI call returns with a **sound_handler_id** which can then be used by the program to identify which handler it is using in any communications with the **SharedSound** module. This is necessary as **SharedSound** may be playing more than one series of sound samples provided by different client programs. The returned value will be **0** if no handler was provided. So a non-zero value confirms that a driver-handler connection has been made successfully.
- The program then calls **SharedSound_SampleRate** quoting the **sound_handler_id** value it was given. This is so **SharedSound** can tell the program the sample rate which will be used.
- The audio is then played by **SharedSound** calling the part of the program it has been told to use, and that feeds the sample values to **SharedSound**. This part of the process acts as a loop. As the playing process continues one of the locations provided by the program is regularly updated by **SharedSound** to report how many of the provided sample pairs have been played.
- The main program runs its own loop, monitoring this value until it sees that **SharedSound** has now played all the sound samples that were provided. It then calls **SharedSound_RemoveHandler** quoting the **sound_handler_id value**. This

[1] http://www.iconbar.com/Building_the_Dream_2_-_The_RISC_OS_Sound_System/news1209.html

terminates the audio playing process and allows **SharedSound** to remove the connection and forget the handler.

- The player program now quits.

To understand in more detail I'd recommend examining the source code which Jeffrey Lee provided because the program is intended as a tutorial demonstration of how audio can be played using **SharedSound**. More generally, a longer file (or something like a stream) may be too large to load as a complete item into memory. In such cases there will need to be added complications to keep supplying new blocks of sample values to be played as a continuous sequence by **SharedSound**. Note also that **SharedSound** leaves it to the driver/program providing the samples to do so at the required sample rate.

SharedSound allows you – if you wish – to have more than one audio file or stream playing simultaneously. So, for example, you can have one player (!PlaySound + PlayIt) play one sound file whilst another (e.g. !DigitalCD + DiskSample) plays another. Even if the two files being played are of different formats and sample rates, they will be combined and both play out at the correct speed and pitch. **SharedSound** simply sums the values from each input stream. Note that this means that there is a risk that the result will be too loud and clip the available 16-bit range.

SharedSound provides a SWI **SharedSound_SampleRate** which allows you to change the ROSS sample rate. So for example, on an ARMiniX which can support a sample rate of 88,200 samples/sec, a BASIC program:

```
SYS"SharedSound_SampleRate",0,88200*1024
END
```

will set the machine's sample rate to 88.2 ksamples/sec. Sample rate values are given in 1024th's of a Hz and assumes stereo. i.e. it is the rate for each individual stereo channel (L and R). Any change in rate made in this way will be forgotten over a shutdown and reboot. Note that various documents use the term "frequency" in this context to mean a sample *rate*. However the correct term in audio engineering is "rate", so I have adopted that word here.

SoundControl

Jeffrey Lee's 'player' demonstration program has no user controls or interface. However most user software – like !PlaySound and Playit – does provide various interactive controls which allow the user to make adjustments – e.g. to the volume (gain) at which the output is played. They also usually report or display some relevant details. e.g. The duration of a file and how much has been played.

In addition to these there are a number of control settings and adjustments provided by the ROSS. A relatively new addition providing some of these arrangements is the **SoundControl** module. This essentially provides a series of settings which allow the user to alter some of the audio hardware behaviour. For early RO hardware this may not do a great deal. But more modern hardware may have a a number of different audio inputs and outputs, and provide greater flexibility. The aim of **SoundControl** is to give a standard ROSS interface for audio hardware control.

The *command **MixVolume** can be used to adjust the operation of the hardware. It treats the hardware as if it were a 'mixer desk' with a set of inputs and outputs whose gains (volumes) may

be altered, or which can be muted, etc.

You can examine the main sound settings of your RO machine by using a small !SoundCheck application which is available from http://www.audiomisc.co.uk/software/SoundCheck.zip. When this application is run it will list a series of settings whose details will depend on the machine hardware, OS version, and if you are playing any audio at the time or not. These results appear in a TaskWindow. When I ran !SoundCheck on an ARMiniX running RO5.19, part of what it reports showed the MixVolume settings as listed below.

```
adj stereo  on  [-2] Headphones   gain =   0.00    Min -30.00  Max   0.00  Step   2.00 [dB]
fix stereo mute [-1] Speaker      gain =   0.00    Min -24.00  Max   6.00  Step   2.00 [dB]
fix stereo mute [-3] Line out     gain =   0.00    Min -52.00  Max   6.00  Step   2.00 [dB]
fix stereo mute [-4] Aux out      gain =   0.00    Min -52.00  Max   6.00  Step   2.00 [dB]
adj stereo  on  [ 0] System       gain =   0.00    Min -18.00  Max  24.00  Step   6.00 [dB]
fix  mono  mute [ 1] Microphone   gain =   0.00    Min   6.00  Max  30.00  Step   6.00 [dB]
adj stereo mute [ 2] Line in      gain =   6.00    Min   6.00  Max  30.00  Step   6.00 [dB]
fix  mono  mute [ 3] Aux in or CD gain =   0.00    Min   6.00  Max  30.00  Step   6.00 [dB]
```

Eight hardware 'devices' are shown. But in practice only one audio output 'port' (Headphones) was actually connected and working at the time, using the version of RO5.19 on an ARMiniX. The Headphones output is the hardware device which in this case provides the audible output from a socket on the back of the ARMiniX. The purpose of the MixVolume command can now be understood from its syntax.

```
*MixVolume <mixer> <category> <index> <mute> [<gain>]
```

- **<mixer>** specifies which item of hardware is providing the inputs and outputs. For the ARMiniX (based on the PandaBoardES) these are all provided by a TWL6040 chip. As the machine's main audio hardware this is treated as mixer **0**.
- **<category>** is the number representing the individual device/port. The relevant values are shown by !SoundCheck as the number in square braces. So the Headphones are category **-2**.
- **<index>** is a value representing the channel and defaults to **0** meaning both.
- **<mute>** indicates if the device/port is muted or not: **1** = muted / **0** = will pass audio.
- **[<gain>]** sets the nominal gain (volume) for the device/port.

Each line for a device/port listed by !SoundCheck gives various details. This starts with:

- **adj/fix** = indicates if the gain can be changed by MixVolume or not.
- **stereo/mono** = indicates if the device is stereo or mono. (surprise!)
- **on/mute** = indicates if the device is muted or working.
- **[<category>]** = device/port number. (Note an output has a negative category number.)

followed by the name assigned to the device/port. then followed by gain setting details.

By default the Headphones details above are listed as:

```
gain =   0.00    Min -30.00  Max   0.00  Step   2.00 [dB]
```

This means that the current gain (volume) setting for the Headphones output is 0dB. And that this can be changed in steps of 2dB over a range from 0dB down to -30dB.
When using the MixVolume command, gain values are specified in 16th's of a decibel. So if I use the command **`*MixVolume 0 -2 0 0 -64`** I can hear the level of music played out of the

Headphones output reduce. And when I use !SoundCheck again the relevant line in its report changes to

```
adj stereo on [-2] Headphones   gain =  -4.00    Min -30.00 Max   0.00 Step   2.00 [dB]
```

which shows that the command has changed the Headphones gain to -64/16 = -4 dB. Any change made in this way will be lost over a shutdown and reboot as !Boot will then re-impose the default settings.

SoundDMA and the main OS Sound Settings

From quite early versions, the ROSS has included the **SoundDMA** module and a set of commands and SWIs. Note that some of the *command/swi names which were originally provided for 8-bit VoiceGenerators now have names which may be confusing in a modern context. e.g. the SWI **Sound_Stereo** which sets the stereo position between Left and Right of the output from a specified 8-bit VoiceGenerator! Here I will largely neglect the commands and SWIs which are really relevant only for such situations, and will focus on the 16-bit arrangements.

Volume control.

The command ***Volume** is described in various documents as if primarily provided for 8-bit audio. Indeed, is sometimes assumed that it "should" have no effect on 16-bit audio. However in practice it generally does! The reason for this seems sensible enough. i.e. long before the addition of modules like **SharedSound** and **SoundControl** people would be wanting to play 16-bit audio alongside 8-bit and wished/expected to have a 'system' volume control that let them adjust the overall output level regardless of their choice of audio replay software, etc.

As a result it has become the norm for replay applications and drivers like !PlaySound + PlayIt or !DigitalCD + DiskSample to allow the user to control the sound replay level using the ***Volume** command. Hence amongst its preferences !PlaySound has a PlayIt Configuration option 'Scaled volume'. Ticking this causes the ***Volume** system setting to affect the output sound level. Similarly, !DigitalCD offers an 'Ignore system volume' option.

By default a ROSS setup will begin with a ***Volume** value of '**127**'. This nominally represents a unity gain – i.e. amplitude factor of x 1.00 (0dB). You can issue the command

```
*Volume <N>
```

where **<N>** is a value in the range 1 to 127 to specify the system volume you require. The amount by which a value affects the actual output level is rooted in ye ancient lore of the 8-bit semi-log-law samples era. Over most of the range a change in **<N>** of 16 causes a change in the amplitude scaling of the output of a factor of a half. e.g. ***Volume 111** will give an amplitude scaling factor of x 0.5 (-6dB), and ***Volume 95** gives an amplitude scaling of x 0.25 (-12dB). The result is that the setting behaves a bit like a 'log law' volume control.

This 'system' volume/gain can also be adjusted using a SWI, **Sound_Volume**. Calling this with a value in the range 1 - 127 in **r[0]** sets the system volume level as above. Calling it with '0' in **r[0]** causes the SWI to return the current system volume setting value in **r[0]**.

Turning sound output on or off

The command ***Audio** can be use to enable or disable the entire ROSS sound replay operation.

***Audio ON** enables output, and ***Audio OFF** disables it. Calling the equivalent SWI **Sound_Enable** with **r[0]=1** enables sound output, **r[0]=0** disables output. If the sound output has been disabled by this command you may find that you get no output regardless of what other settings you may make!

The command ***Speaker** may be used to switch on or off the connection to any internal loudspeaker(s). i.e. ***Speaker ON** connects the speaker(s) and ***Speaker OFF** disconnects them. There is an equivalent SWI **Sound_Speaker** which can also read the current state of the speaker connection. Note that on some hardware there may be no physical speaker or speaker connection, so this command may do nothing. In such cases any output may come via a different route – e.g. via a 'Headphones' output on an ARMiniX.

There is also a ***Configure** setting
***Configure SoundDefault <speaker> <volume> <voice_number>**
which will alter the defaults.

- **<speaker>** = **On** or **Off** and acts as the ***Speaker** command.
- **<volume>** = Value in the range **0** (quietest) to **7** (loudest).
- **<voice_number>** = Number of the VoiceGenerator used as the 'system beep'.

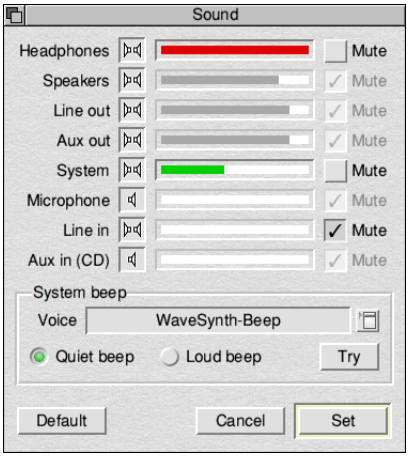
These values will persist over a shutdown-restart cycle as they store the relevant values in CMOS memory for re-loading at bootup. Note that here the **<volume>** value has a much more limited range than ***Volume** and gives fewer steps.

The command ***Voices** will list the installed VoiceGenerators and their numbers. By default, the first of these (WaveSynth-Beep) will have an extra '1' at the start of its line in the list. This indicates that it is the currently default system beep.

In general I would recommend using the !Boot Sound interface to adjust settings rather than use the above configuration command.

!Boot Sound settings

!Boot provides an interface for controlling the basic sound defaults. To access this double-click on !Boot and then select the 'Sound' option. You should then see a window similar to the one in the illustration below.



The window displays a series of 'thermometer bars' for various possible input and output devices. Note that the presence of a named device does not always mean one is present and accessible!

The sliders allow you to adjust the gains (volumes) of each useable device. Devices that are not usable have their slider greyed out. You can drag a usable slider to choose a new default gain (volume) setting for each accessible device.

Each device also has a 'mute' icon which can be used to disable its input or output.

The window also allows you to select your preferred VoiceGenerator as the 'system beep'. You can then test ('Try') the results of any changes you have made and either set them as the new default, or cancel any changes. Alternatively you can revert to the standard default settings.

An example – PlayIt

An examination of PlayIt serves two useful purposes

- It shows how the ROSS can be used to play a variety of types of audio input and how the ROSS may be controlled and employed in practice.
- PlayIt also provides a convenient access point to the ROSS for more purposes than simply being used by applications like !PlaySound or !DigitalCD. In effect it provides a convenient and flexible interface as well as being a neat example.

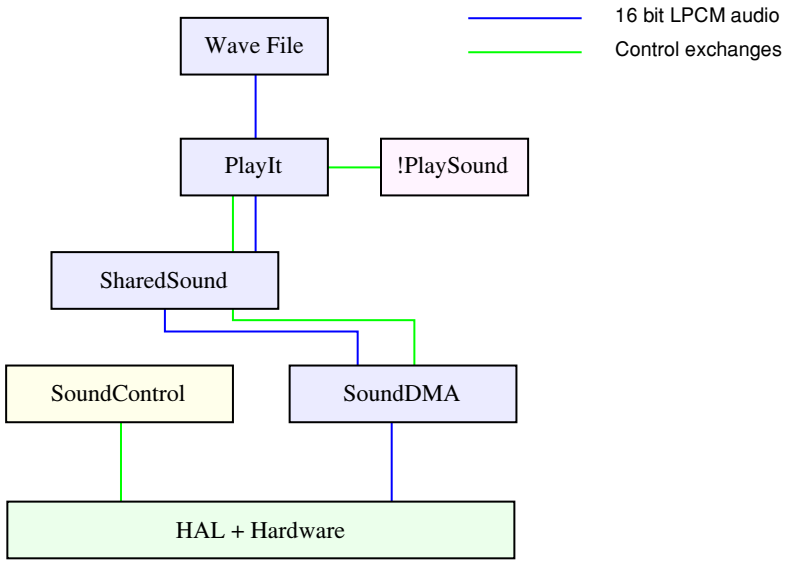


Fig 2 Using !PlaySound + PlayIt to play a Wave file

For the sake of example I'll assume initially that PlayIt is being controlled via !PlaySound, but this could be replaced by !DigitalCD or some other program.

Figure 2 shows an overview of what happens when !PlaySound + PlayIt are used to play the content of an LPCM Wave ('wav') file. The application, !PlaySound, provides the user with a convenient desktop interface for the process, providing controls for stopping and starting the replay, etc. But the actual process of playing is done by the PlayIt module. In effect, The user tells !PlaySound what to play, and it gets PlayIt to do the work, monitoring the process and giving new instructions when required. !PlaySound also passes on details like changes in the required Volume (gain) setting and if the system **`*Volume`** value is be applied or not.

Note that the above diagram shows the audio path in a slightly different way to Figure 1. This is because the desktop application doesn't need to process the actual audio data in this example. More generally, though, the data may be passed though an application if it requires processing or monitoring in some way. But if this isn't required then the arrangement shown in Figure 2 is more direct and efficient, so is preferred.

In fact if you have the PlayIt module loaded you don't actually need a specific desktop application to be able to play an audio file. The PlayIt module also provides a set of *commands which can be used to control playing audio files. Although it tends to be more convenient for desktop users to have PlayIt controlled using an application like !PlaySound.

In machines using an up-to-date ROSS the PlayIt module reads the data from the source file and,

by default, sends it to the **SharedSound** module. These two modules exchange control data as this process takes place. In effect, PlayIt gives **SharedSound** a new chunk of data when **SharedSound** says it wants the next chunk of samples from the source file to have them played. **SharedSound** then processes the data given to it (e.g. combining it with any other inputs it has to 'mix' and have played) and passes the result on to **SoundDMA** which puts them into DMA (Direct Memory Access) buffers. These are areas of memory from which the sound hardware will take the values and play them out. The details of how the hardware then deals with the audio data samples depends on the **SoundControl** module and its settings.

On older systems that lack **SharedSound**, PlayIt will send the data direct to **SoundDMA**. Indeed, if you wish, you can alter !PlaySound and !PlayIt so they bypass **SharedSound** and use **SoundDMA** directly if you wish. Such a change is generally deprecated. But it can be done by removing the "SSound" file from !PlayIt.Modules. It then uses the "Acorn 16-bit" interface. (On ancient systems which lacks 16-bit support it will use the old 8-bit interface via **SoundChannels**.)

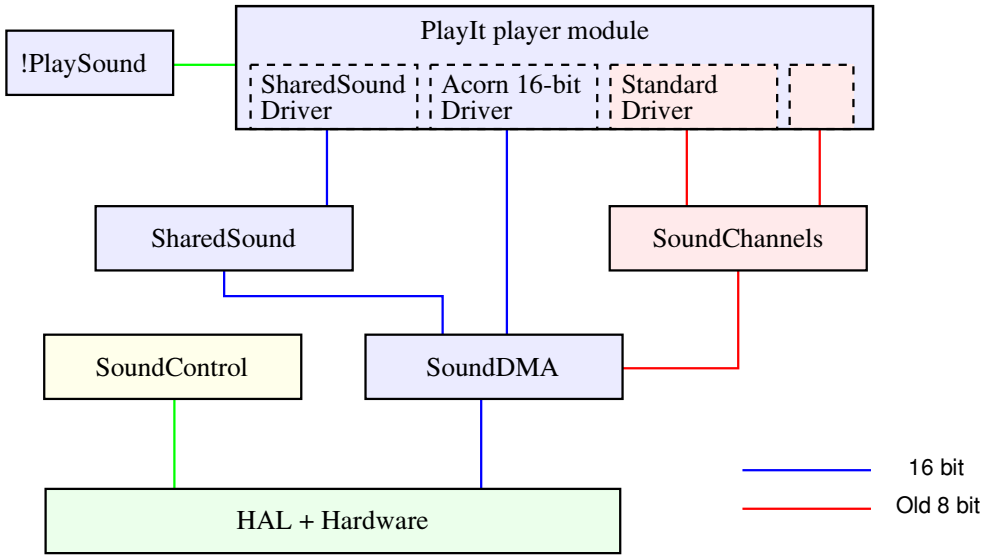


Fig 3 More detailed PlayIt overview

If you experiment with Jeffrey Lee's demonstration 'player' program you can see that it works, but it doesn't operate as a multitasking GUI Desktop application. In general, the process that plays audio and feeds **SharedSound** (or **SoundDMA**) has to run in a way that allows it to bypass the usual WIMP polling, etc, in order that it can respond promptly when required by **SharedSound**. This means that being able to play audio is more conveniently handled at the module or single-tasking level.

PlayIt is an audio player module that works well and is widely used. It can be used in a number of ways. It provides both a *command interface and a set of SWIs which virtually make it into a higher layer of the ROSS. To illustrate this I will outline how it can be used as a player via its * command interface. For clarity I will make some simplifying assumptions which allow me to omit some steps. e.g. I'll assume that we can start when nothing is already being played, no audio player application is running, and that the PlayIt module isn't already loaded. The following steps would usually be amongst the actions carried out for you by a playing application like !PlaySound. Here I will describe them so you can follow the process.

You can load the PlayIt module in the usual ways. e.g. by opening up the !PlayIt.Modules

application directory and double-clicking on the **PlayIt** module file. Before it can play a file you need to tell PlayIt which route to use to play audio. Usually you will use SharedSound. The command ***PlayIt_Driver PlayIt:SSound** tells PlayIt to load its SharedSound driver.

You can then play a wave file by using the ***PlayIt_Play** command followed by the file's full name. e.g. if you have a file called 'music/wav' on your ramdisc you can commence playing by using the command ***PlayIt_Play ram:music/wav** . If your ROSS is set up appropriately you should then hear the file's content being played. You can then use a set of commands which include

- **PlayIt_Pause** = pause the replay (then use **PlayIt_Play** to resume playing)
- **PlayIt_Volume <vol>** = changes PlayIt's replay gain to alter the volume
- **PlayIt_Stop** = stop playing the file.

The **PlayIt_Volume <vol>** value works in a similar way to the system ***Volume.** but in this case it is applied by PlayIt rather than the ROSS. So **PlayIt_Volume 127** means 100% volume, which is the default. However unlike ***Volume** the PlayIt value can be up to 256, and values larger than 127 will amplify the audio.

Note that this means that you may have at least *three* volume controls along the way between the source and the output: PlayIt's volume control, the system volume, and **SharedSound's** MixVolume. Also note that PlayIt can be told to apply the system volume setting or ignore it.

While using PlayIt, the command **PlayIt_Status** will list some information about the current situation. By default, when **SharedSound** is present PlayIt will normally be expected to use it. However you can choose to bypass SharedSound if you wish by using ***PlayIt_Driver PlayIt:Acornl6bit** or the equivalent SWI to have PlayIt drive **SoundDMA** directly. There is also a **:Standard** driver module compatible for use with 8-bit material on older machines. However in general modern playing applications will tend to have PlayIt use **SharedSound**.

There are a number of PlayIt commands, and their equivalent SWIs, which can be used. The result is that PlayIt provides a convenient interface for the user to be able to control audio replay, either by typing in commands, or via Obey files. or by writing their own desktop application as a front-end for the process. In effect, !PlaySound is just one example of such a desktop front-end that allows the user to play files via drag-and-drop and control the process via a GUI. Similarly, other applications – e.g. !DigitalCD – also use PlayIt to play some types of files. So for many purposes the user or programmer can choose to regard a player/driver module like PlayIt as their interface with the ROSS if they wish to avoid diving into the details of SharedSound, etc.

However a point to bear in mind is that PlayIt can only play a specific set of audio formats. Hence if you have some other format you wish to play, you either need to have the data converted before sending it to PlayIt, or use an alternative to PlayIt.

A bigger picture

A number of applications and programs have been ported from other platforms (mainly Linux) to RO. These generally make use of a common 'UnixLib' which makes implementation easier. In addition a specific module **DigitalRenderer** (DR) has also been developed to handle the output from media players and other Linux programs that output audio. The DR module has therefore primarily been of interest to those wanting to port, and the programs that require them. The diagram below shows how DR fits into the ROSS at present. Note that as things stand DR

sends 16bit audio to **SoundDMA**. i.e. it uses the 'Acorn16bit' route. This may change at some future point to allow DR to send its audio output via **SharedSound**.

Linux and other similar unix-like operating systems adopt a common 'everything is a file' philosophy. In effect, when audio is played on a Linux system the operating system regards the hardware that plays the stream of values as a 'file' into which the samples are written. Conforming to this and supporting it for Linux programs leads to an interesting consequential side-benefit provided by DR. The **DigitalRenderer** module provides a set of swis that can also be used by programs which have *not* been ported from another platform. It also provides a new 'filing system' "**DRender:**". This makes it possible to play 16-bit audio by using a RO program or command to 'write the samples to a file'. As such it provides quite a simple-to-use route as a basis for simple playback! I will therefore use this approach to illustrate how DR works and can be used to play audio using the ROSS.

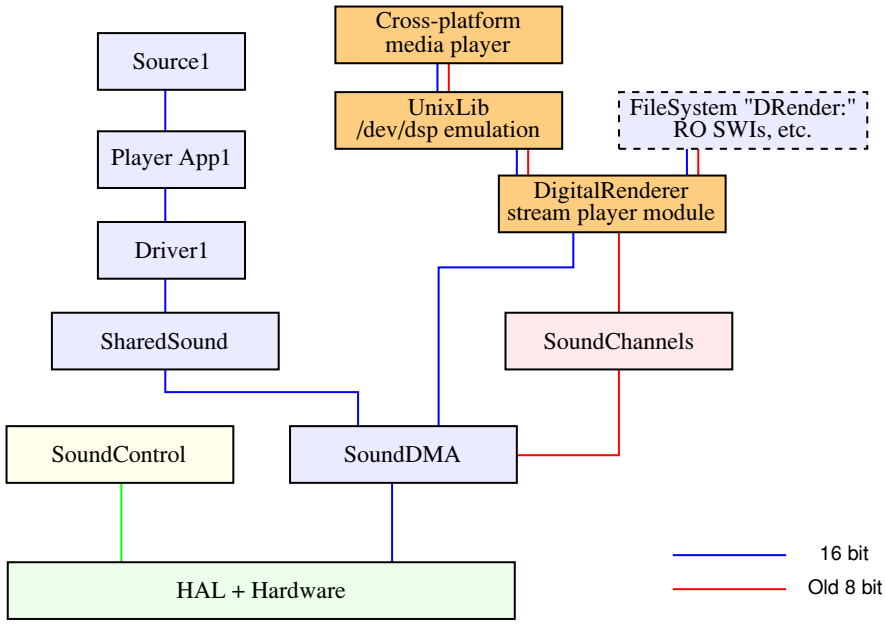


Fig 4 DigitalRender and crossplatform audio input

The above diagram shows how DR links into the ROSS. To illustrate the operation of DR I'll start by assuming you have already loaded the **DigitalRender** module. This provides three commands: ***DRenderStatus**, ***DRenderDefaults**, and ***DRenderOff**.

The ***DrenderOff** command does what it says. It switches off the module's activity and closes the **DRender:** filing system if it was open.

If DR is not being used then ***DRenderStatus** will simply return the statement **Inactive**.

***DRenderDefaults** will summarise the default values the module will use unless they are changed by the user or a program. If no changes have been made these defaults will typically be:

```
Channels: 1. Format: 16bit signed linear (little-endian, right-left order). Period:
45. Frequency: 22050. Buffer size: 512. Num buffers: 43
```

You can change these settings by using ***DRenderDefaults** follows by one or more specific parameter values as specified below:

```
-c #: set number of channels (1,2,4,8)
-f #: set sample format,
      1 for 8 bit ulaw
      2 for 16 bit signed linear little-endian right-left
      3 for 16 bit signed linear little-endian left-right
-p #: set sample period in usecs, i.e. 1e6 / frequency
-q #: set sample frequency in Hz. Specifying a positive value
      automatically tries 16bit sound first, otherwise the old 8bit sound
      is used.
-s #: set buffer size per channel (keep it small)
-n #: set number of buffers to use in streaming interface. You should
      make this large enough to buffer enough data to keep playback
      stable (default is 43).
```

There is also an equivalent **SWI DigitalRenderer_SetDefaults** which allows you to read or change the default playback parameters. Note that the currently set defaults will be used to play out any audio samples written to any 'file' in the **DRender:** filing system.

The module having been loaded the first step in the process of playing audio via **DRender:** is therefore to use either ***DRenderDefaults** or **DigitalRenderer_SetDefaults** to check the defaults and then change them if necessary, telling the module what number of channels, sample rate, etc, to employ when playing sample values written to **DRender:**. Having done this I can use two simple examples using programs in 'C'.

Example 1 - Mono 44.1k

Here the process starts by using the **DigitalRenderer_SetDefaults** SWI to set the sample rate for DR to 44100. When giving a sample rate we also have to set the sample period (interval) in microseconds, rounded by integer calculation. So we require something like:

```
nch = 1; /* number of channels ( 1 means mono ) */
fin = 44100; /* sample rate */
pin = 1000000/fin; /* sample period (interval between samples) */
rin.r[0]=nch;
rin.r[1]=0;
rin.r[2]=pin;
rin.r[3]=0;
rin.r[4]=0;
rin.r[5]=fin;
_kernel_swi(0x4F70E,&rin,&rout);
```

The zeros for other inputs mean that their existing defaults are not changed. One of these defaults is 16-bit little endian right-left order which is fine for these test purposes. Note that in general DR will expect 16-bit input and will normally drive it into the 'Acorn16bit' handler of **SoundDMA**. However DR can also drive the older 'Standard' 8bit handler and will use this on older systems which lack the 16bit capability.

Having setup DR to know how input is to be played we can generate a test waveform. Since the input is a series of 16-bit integers the test program defines an array, **short wave[44100];**, to hold the test waveform. This can then be filled by a calculation like

```
i=0;
fw=300.0;  /* test frequency in Hz */
dp=2.0*3.1415927*fw/44100.0; /* step in phase between samples */
phase=0.0;
do
```

```
{
  wave[i]=(int)( 5000.0*sin(phase) );
  phase+=dp;
  i++;
} while (i<44100);
```

When played correctly this should produce an output sinewave with a frequency of 300 Hz and an amplitude of 5000/32768 = 0.152 of max volume. i.e. a 'beep' of moderate volume.

The result could be played in various ways using DR. For the sake of example here this can be done by defining a char pointer, **`char* bpoint;`** and the result can then be played using

```
FILE* outfile;
outfile=fopen("DRender:test\0","wd");
bpoint=(char*)wave;
i=0;
do
{
  fprintf(outfile,"%c",bpoint[i]);
  i++;
} while (i<88200);
fclose(outfile);
```

This reads through the bytes in the **`wave[]`** array and 'saves' them to a file we have opened in the **`DRender:`** filing system. Note that since there are two 8-bit bytes ('chars') per 16bit short value this loop has to output 88200 bytes, in sequence. Here I've called the file 'test' but in fact the name is irrelevant. Any stream written to the **`Drender`**: filing system will be treated as a series of values to be played according to the current DR settings.

Once finished we can close the 'file' and the program can be ended. A simple program of this kind is most safely run in a TaskWindow. This is because a TaskWindow allows the DR module to prevent its buffers being over-filled by using an UpCall to pause the writing process if and when necessary. When you run the program you should hear a 1-second duration 300Hz beep from the sound output of the ROSS.

Example 2 - Stereo 88.2k

This example is generally similar to the previous one, but the changes can be used to bring out a few points which affect stereo replay or the use of more modern hardware – e.g. the PandaBoards, etc. Some modern hardware can support higher sample rates than the old machines. So for example the ARMiniX (PandaBoardES) hardware can actually handle 88.2k and 96k, and indeed, 24 bit sample values. Indeed, the use of these higher rates may lead to improved performance for various reasons even when playing 44.1k material! So it is worth giving an example of how these can be used for stereo.

As before we can use either the **`*DRenderDefaults`** command or the relevant SWI to set DR to treat input as 16-bit stereo to be played at 88200 samples/sec (per channel). However the ordering of the data now has to alternate right and left channels rather than give a mono series that will be played out of both channel simultaneously.

A beep one second long now requires 88200 samples per channel. For test purposes the example only plays a beep out of one of the two channels, and the other is kept silent. Given this, the array whose values define the beep should now be **`short wave[88200]`**. This is filled much as before, but with half the previous step in phase between samples. i.e. we now have

```
i=0;
fw=300.0;  /* test frequency in Hz */
dp=2.0*3.1415927*fw/88200.0; /* step in phase between samples */
phase=0.0;
do
{
  wave[i]=(int)( 5000.0*sin(phase) );
  phase+=dp;
  i++;
} while (i<88200);
```

to fill the array we wish to play out.

Since for test purposes we want the output to *only* appear on *one* of the stereo channels we can write the output using

```
czero=(char)0;
FILE* outfile;
outfile=fopen("DRender:test\0","wd");
bpoint=(char*)wave;
i=0;
do
{
  fprintf(outfile,"%c%c",czero,czero)
  fprintf(outfile,"%c",bpoint[i]);
  i++;
} while (i<176400);
fclose(outfile);
```

Each loop sends two 16-bit sample values and DR sends the first value to one channel and the second to the other. Hence each pass round the loop sends a left-right pair to be played. One channel will be silent, and the other plays the test sinewave. By default DR will assume right-left order, but this can be reversed if you change the DR setting that controls the ordering. Note, though, that some hardware may have its left-right connections swapped over!

In the above examples I simply generated a 1-second block of values to play a test tone. And then used an elementary 'one byte at a time' method to play the results. However various other approaches can be employed. So it is possible to read in chunks of data from a file, manipulate it, and play out the results. But more complex playing arrangements will need to take extra steps to be able to play safely and reliable. For example, you should regularly check how many DR buffers are available. If this isn't done problems may arise either because you are trying to write data too quickly and overrun the buffers, or too slowly, and the buffers empty, causing gaps in the replay. Hence it is generally advisable to regularly monitor the state of DR and its buffers whilst using it to play audio. Similarly you'd normally need to check when starting that DR was loaded and, if not, to either load it or quit with a suitable error of it wasn't available.

The above said, DR does have a useful feature which makes audio replay easier when running a program in a TaskWindow. In such a situation DR will use the **`UpCall6`** signal to pause or 'sleep' the playing program's TaskWindow process when DR's buffers are full. Hence in such cases, DR will avoid the program trying to give it audio data faster than it can be played. You may still need to take care to ensure your playing program can keep up, though!

Audio CD

The above sections consider audio outputs. However a RO system may also have various types of audio input device. One of the most common is an optical drive than can read Audio CDs. As a result the **CDFS** module provides a set of commands and SWIs for reading Audio CDs. **CDFS** also provides ways to access 'data' discs, but I will ignore the non-audio side.

Many of the SWIs that are provided by **CDFS** pass values in a CD Data Block (CDDB) that essentially allocates a block of five 32-bit ints (i.e. 20 bytes) as a block for data transfers during SWI calls. This has the format:

- +0 : **device** number (0 - 7)
- +4 : **card** number (0 - 3)
- +8 : **LUN** or Logical Unit number (0 - 7)
- +12 : driver **handle**
- +16 : currently unusued (?)

This is based on each CD (or other optical drive that provided CD functionality) being identified by CDFS in terms of this set of values. To illustrate their use we can use a simple 'C' example program that employs two SWIs – **CD_Inquiry** and **CD_Identify**.

CD_Identify tells us if a CD drive is connected with a given set of device, card, and lun values. So we can use this as follows

```
int identify(void)
{
 int answer;
 handle=0;
 cb[0]=device;
 cb[1]=card;
 cb[2]=lun;
 cb[3]=handle;
  rin.r[7]=(int)cb; /* Note that cb is a CDDB of 5 (or 4) ints */
 _kernel_swi(CD_Identify,&rin,&rout);
 answer=rout.r[2];
 handle=answer;
 return answer;
}
```

If a CD drive is connected with given device, card, and lun values the swi will return with the drive's **handle** value in **rout.r[2]**. Otherwise **rout.r[2]** will return **-1**. We can therefore scan a RO machine for the presence of drives by calling the **CD_Identify** swi with device values in the range 0 to 7, card values 0 - 3, and lun values 0 - 7 using something like

```
result=identify();
if(result!=(-1))
{
  printf(" %d %d %d = %d  is  ",device,card,lun,result);
  inquire();
  printf("\n");
}
```

to show the results. Here the **inquire()** proceedure uses **CD_Inquiry** as follows

```
int inquire(void)
{
 int answer,iq;
 rin.r[0]=(int)inquire_buffer;
 rin.r[7]=(int)cb;
 _kernel_swi(CD_Inquiry,&rin,&rout);
 iq=8;
 do
 {
  printf("%c",(char)inquire_buffer[iq]);
   iq++;
  } while (iq<36);
 return answer;
}
```

where the CDDB **cb** contains the relevant values for the drive, and **inquire_buffer** is a 36-byte (i.e. char) buffer which the SWI will use to return its response. This will print any identifying string provided by the drive. On my ARMiniX machine as an example, this gives me

```
1 0 0 = 0  is  Optiarc DVD RW AD-7740H 1.00
```

i.e. it tells me that device = 1, card = 0, lun = 0 finds my CDFS drive number 0 which identifies itself as an Optiarc DVD RW of a specified model and firmware revision. By default if you only have one CD / optical drive it will usually be device 1 and assigned CDFS 0. The card = 0 value indicates that it is connected via the motherboard. Other card values are nominally associated with additional expansion cards, etc.

Having established what drive(s) are present you can then find out if a CD is loaded by using the **CD_DiscUsed** SWI.

```
char ubuffer[8];
ubuffer[0]=0; ubuffer[1]=0; ubuffer[2]=0;

rin.r[0]=1;
rin.r[1]=(int)ubuffer;
rin.r[7]=(int)cb;
_kernel_swi(CD_DiscUsed,&rin,&rout);
frames=(int)ubuffer[0];
secs=(int)ubuffer[1];
mins=(int)ubuffer[2];
total_frames=frames+75*(secs+60*mins)-151;
```

If no Audio CD is present in the drive specified by the CDDB **cb** then the returned values in the three integer int array **ubuffer** will all be zero. If an Audio CD is present the results returned tell you the total duration of the Audio Tracks in terms of the time in mins/secs/frames and the complete number of accessible frames of audio data. Note that there are 151 'inaccessible' frames of nominal audio data. (The details of these frames are explained below.)

You can obtain information on Audio Tracks using the **CD_EnquireTrack** SWI. To use this you require some extra buffer arrays for values to be passed.

```
char tb[8];
int tintbuffer[2];
```

If you then call the SWI with zero in **rin.r[0]** it will return the numbers of the first and last

track on the Audio CD. e.g.

```
rin.r[0]=0;
rin.r[1]=(int)tb;
rin.r[7]=(int)cb;
_kernel_swi(CD_EnquireTrack,&rin,&rout);
printf("First track = %d\n",(int)tb[0]);
printf("Last  track = %d\n",(int)tb[1]);
ntracks=(int)tb[1];
```

Once you know the number of tracks you can re-use the same SWI to determine the start-point (in CD Audio frames) of a track by calling the SWI with the track number in **rin.r[0]** as follows

```
rin.r[0]=tin  /* track number */
rin.r[1]=(int)tintbuffer;
rin.r[7]=(int)cb;
_kernel_swi(CD_EnquireTrack,&rin,&rout);
printf(" Track %3d starts at %6d \n",tin,tintbuffer[0]);
```

Note that in practice you could use the same buffer for both the above calls because in each case the output is written to the 8 bytes that follow the address pointer given to **rin.r[1]**.

From the above you can see that Audio CD uses a sequential 'frame' method to store audio. Audio discs have no 'filing system' in the normal sense the term is applied in computing. Instead they have a Table Of Contents (TOC) in a dedicated area at the start of the disc, all 'times' and 'track numbers' are then provided as being look-up values for the relevant frame count into the stream of audio sample frames on the CD.

Having determined the presence of a disc and the details of the tracks it contains we can proceed to read audio data from the disc. Each CD Digital Audio or Philips/Sony CDDA 'Red Book' format disc 'frame' contains the stereo sample pairs for an interval lasting 1/75th of a second. i.e. for 44100 ÷ 75 = 588 16-bit sample pairs, equivalent to 2352 bytes.

To read a specified number of sequential frames of Audio data you can use the **CD_ReadAudio** SWI as shown in the following example which reads a block of 5 seconds duration

```
char dbuffer[883200];       /* 5 second's worth of bytes */

printf("Type in start time [mm ss ff] => ");
scanf("%d %d %d",&mins,&secs,&frames);

seek_frame=frames+75*(secs+60*mins);
printf("Seek frame %d\n",seek_frame);

rin.r[0]=0;
rin.r[1]=seek_frame;
rin.r[2]=375;               /* five second's worth of CDDA frames */
rin.r[3]=(int)dbuffer;
rin.r[4]=883200;
rin.r[7]=(int)cb;

_kernel_swi(CD_ReadAudio,&rin,&rout);
```

where as before **cb** is a CDDB that specifies the drive, card, etc, for the drive containing the disc

being read. You can also see from this example how to convert between the number of frames from the disc start to a time in mins/secs/frames.

The result is a block of values in **dbuffer** in Stereo 16-bit ('short int') little-endian format to be played at 44100 samples/sec. This can be saved to a file and played using applications and players like !PlaySound or !DigitalCD.

In practice if you wish to read long sequences of audio to 'rip' tracks it makes more sense to grab smaller portions during each grabbed 'block', and then loop around, reading successive blocks and saving them to a file or playing them with appropriate buffering, etc. But the principle of how to read the data from the disc is as illustrated above.

It is worth bearing in mind that the CD Audio format and players/readers don't guarantee to read frames with precise alignment. Hence when reading an Audio CD there is always a chance your drive may read the correct number of successive sample values in sequence, but that the actual start and end points *don't* perfectly align with the frame boundaries! This will depend on the details of your drive and the disc. It is one of the reasons that programs like CDparanoia do repeated overlapping reads, compare the results to check for any misalignments, and then try to correct for them. The Audio CD format was not designed to work with the sample-perfect precision we have come to expect from Data/Computer discs with their filing arrangements. This is also the reason for two properties of CD Audio that can lead to confusion.

The disc is assumed to begin with 151 (or 150) 'inaccessible' frames. This is a consequence of the Philips/Sony "Red Book" specifications that allocate a 2-second 'lead in' to each track. The assumption is that when you ask a CD Audio player to play a disc it may not be able to start exactly with the first sample pair of the start-frame of the track. So the playing hardware is allowed a 'lead in'. This is analogous as the old 'lead in spirals' between tracks and at the start of a Vinyl LP. A good player may be able fairly accurately start from near the first samples of the track. But it should tend to err on the side of starting 'early'. The 'inaccessible' frames are meant as a space for this to occur when you start playing track 1. For the same reasons, discs or players may assume there is a 2-second (i.e. 151 or 150 frame) allowance at the start of each track. Hence you sometimes see statements that tracks must be seperated by at least 2 seconds 'gap'. When writing code to read an Audio CD you should therefore allow for such read misalignments when testing your code. Having given these warnings, in practice a decent drive and commercial Audio CD should usually be readable without such problems. But this can't be guaranteed.

USB Audio

Modern home computer systems have in recent years made increasing use of USB Audio devices. 'Hi Fi' enthusiasts are most familiar with this in the form of 'USB DACs'. Those interested in home or small recordings studios encounter them as boxes to connect to their computer via USB and which allow recording as well as play output. There are now established sets of standard USB protocols and transfer methods for such devices. This means that for many Linux and MacOS users many such devices 'just work' although Windows users may require a 'driver'. However the standardisation now allows many USB Audio devices to now become usable with RO hardware. This is due to work by Colin Granville and Dave Higton. Colin has added the required support for the transfer methods, to the RO USB modules. Dave has produced a new, **USBAudio**, module that provides a convenient API and set of SWI calls to make writing programs easier.

Note that as I write this the new modules are not, yet, a part of the standard ROSS or included in the standard ROOL ROM releases. It is expected that they will be included soon, but initially the relevant modules have to be softloaded. However I am documenting and explaining their use to

help programmers and others make use of the new abilities the modules provide.

The **USBAudio** module essentially provides an API that is similar to other RO program interfaces for 'streaming' audio via a buffer. As usual. the basic approach is to 'open' the device, then transfer data, then 'close' the device. To illustrate this I will use some examples in Acorn/ROOL 'C'.

A great practical advantage of USB Audio is that it seperates the audio hardware from the main computer hardware. This means you have a wider choice of audio devices and performance and can bypass some of the mainboard hardware problems and limitations that have affected some types of machine. However the drawback is that you need to have a suitable device connected, and the RO USB Audio system has to be able to identify it and establish some details needed for transfers to be possible. As a result, the **USBAudio** module provides a set of swis which can be used to find what USB Audio devices have been conncted, and their relevant details. I'll give details of their use in the Reference section near the end of this document. For the sake of simplicity I'll assume here that you only have one USB Audio device connected, and that you wish to use it to play the contents of an LPCM file (Wave format). Many devices can accept the audio data payloads from such files without any conversions. You then just need to transfer data from file to device. However some other files may need prior processing. Here I will assume that the file's data can be played without any alterations. The complications which arise when you have multiple devices, etc. are discussed in the Reference section.

The first task facing the program is to find the USB device number of the Audio device. This can be done using the swi **USBAudio_EnumerateDevices**. e.g.

```
rin.r[0]=(int)device_list;  /* buffer start in memory */
rin.r[1]=256;               /* length of buffer in bytes */
_kernel_swi(USBAudio_EnumerateDevices,&rin,&rout);
printf("List of devices by ID = %s\n",device_list);
```

where **device_list** is the address of the start of a buffer 256 bytes long. The swi will zero-terminated text string into this buffer listing the USB numbers for any Audio devices it finds. In the above example I just have the code print out the string which is returned.

The buffer doesn't have to be 256 bytes long, but you need to ensure it is long enough for the returned string. The USB device numbers are what the underpinning USB system will use to identify the device. However you should bear in mind that the number assigned to a device may change. For example, if you unplug a device and plug it in again. So you should normally use the above at the start of any USB Audio program to ensure you find the current USB number for the device you wish to use.

If you only have one device connected the result will be a string like **USB12** but if there more than USB Audio device is present their numbers will be given as a comma-separated list like **USB15,USB17,USB16,USB12** and you would then have to use other USBAudio swis to determine which USB device number refers to the specific device you may wish to use.

You can now open the USB Audio connection using the **USBAudio_OpenOut** swi, e.g.

```
char sblock[32];
int* iblock;
int playhandle;
```

```
iblock=(int*)sblock;

iblock[0]=48000;     /* sample rate */
sblock[4]=16;        /* audio resolution (bits per sample) */
sblock[5]=2;         /* bytes per channel for transfer */
sblock[6]=2;         /* number of channels */
sblock[7]=1;         /* 1 for LPCM */
sblock[8]=0;

iblock[3]=osize_b;   /* audio data buffer size (bytes) */

rin.r[0]=(int)dev_chosen;  /* pointer to USB device number string */
rin.r[1]=(int)sblock;

_kernel_swi(USBAudio_OpenOut,&rin,&rout);

playhandle=rout.r[0];
```

In this example I'm using char as an equivalent to byte and using two offset pointers so I can put values into the input buffer as either bytes or integers as relevant. For the sake of example I'm giving the for a file whose data has a sample rate of 48000 samples/sec, each sample has 16 bits, and there are two channels (i.e. stereo). The value for **`osize_b`** specifies the size of the buffer in memory into which data will be placed to be fed via USBAudio to the device. In practice it is suggested you choose a data buffer size adequate for holding a chunk of audio data that will take around 0·2 sec to play. But other durations/sizes may be fine. This data buffer has to contain *both* channels. So for the above example the buffer size would need to be $48000 \times 2 \times 2 \times 0{\cdot}2$ bytes for 0·2 sec per full buffer transfer.

Note that dev_chosen is the address of the start of a string of characters that specify the device's USB device number. it represents a char array *not* an integer. So - for example - having obtained a result **`USB12`** from using **`USB_AudioEnumerateDevices`** the above would require a char array like char dev_chosen[32] which contains a zero terminated **`USB12`**. Note also that for many class 1 USB Audio devices we can expect the required number of bytes per channel for transfers to equal the number of bytes by sample for the input file. However this isn't always the case, particularly for class 2 devices. So it may be necessary to use other USBAudio swis to check what value is required for **`sblock[5]`**. See the Reference section for more details.

The abive SWI call will return a value for **`playhandle`**. If this value is zero it tells us that the attempt to open a connection for transfer has failed for some reason. A non-zero value tells us the buffer handler address we can use to transfer data. We can then start data transfers and play out the audio using the chosen device.

When playing a wave file should have already read in its metadata header and checked the values given there for the required sample rate, etc. Then used those in the above **`USBAudio_OpenOut`** call. Having successfully opened the transfer connection we can now read a chunk of the file's data payload into the data buffer we be using. We then have the program perform the usual repeat looping process of reading in successive chunks of data and sending them to the device until all the file has been played. The following gives an example of this process. It assumes you have already opened the Wave file for data reads with the file handle **`infile`**, and placed the position for the next read at the start of the data payload. (i.e. 44 bytes from the start of the file for most simple Wave files.)

```
playstop=0;
do
{
  sizeread=(int)fread(outblock,sizeof(char),osize_b,infile);
  if(sizeread!=osize_b)
  {
     playstop=1;
  }
  rin.r[0]=2;               /* send data to device */
  rin.r[1]=(int)playhandle;
  rin.r[2]=(int)outblock;   /* address of data block start */
  rin.r[3]=sizeread;        /* data block size in bytes  */
  do
  {
      _kernel_swi(OS_GBPB,&rin,&rout);
      rin.r[2] = rout.r[2];
      rin.r[3] = rout.r[3];
  } while (rout.r[3] != 0); /* repeat until block done */
} while (playstop==0);      /* repeats until file done */
```

Each loop uses **fread** to read a block of audio date from the Wave file and place the result into **outblock**. It then calls the general purpose SWI **OS_GBPG** with reason code **2** for the transfers from this buffer to the USB device. The inner loop then calls this SWI until the returned values confirm all the data has been read and the process can be repeated for the next block of audio data from the file. The program monitors the value returned by fread and when this differs from the size of the block which was requested it tells us that all the file has now been read, so once this final block has been read the process can end. We can then close the connection using

```
rin.r[0]=(int)playhandle;
_kernel_swi(USBAudio_Close,&rin,&rout);
```

Capturing audio from a suitable device uses essentially the same approach, but to read in audio data from the USB Device we use **USBAudio_OpenIn** to initiate a recording/capture process. The registers are used exactly as for **USBAudio_OpenOut** to specify the required sample rate. resolution, etc. In this case a **recordhandle** value is then returned in **r[0]**, or this value will be a zero if an input from the device could not be opened. We can then use a loop equivalent to the above audio play example, based on **OS_GBPB** but in this case we give the swi reason 4. i.e. we set **rin.r[0]=4** for the calls to **OS_GBPB** and the captured series of audio values will be saved to the databuffer whose start address we have given **OS_GBPB** in **r[2]**. Otherwise the registers are set as for playback.

In principle we can have the machine carry out more than one capture and/or play process in parallel, using more device if required. However clearly there will be a point where given hardware won't be anle to keep up with the demands this places upon it!

Multiple Audio Controllers and HDMI Audio

As of January 2016 changes were added into the ROSS which provide a significant extension in its capabilities. In general terms, the most significant alteration was to implement the ability to select alternative sound hardware. In immediate practical terms the most significant aspect of this was to allow the user to enable audio output to be directed out from the computer over an HDMI connection. This means the ability to export audio in *digital* format to a screen or other suitable HDMI device. The initial version of these new capabilities was for the ARMX6 machine,

but the intention is that they will become a standard part of the ROSS on any suitable hardware.

In the past, some audio cards were developed and could be used to play or capture audio computers made by Acorn. However these generally involved loading dedicated third party firmware and were never part of the standard OS provision.. So these changes represents a significant development. Here I will cover both of these new abilities. But it should be noted that any further extensions to other types of sound hardware in the future should also be able to be accessed using equivalent methods.

To avoid ambiguity I will refer to the choice of audio hardware in terms of selecting its audio 'controller'. This allows me to keep the term audio 'device' for referring to specific inputs and outputs like loudspeakers or a headphone socket which some controllers may allow to be independently used or controlled.

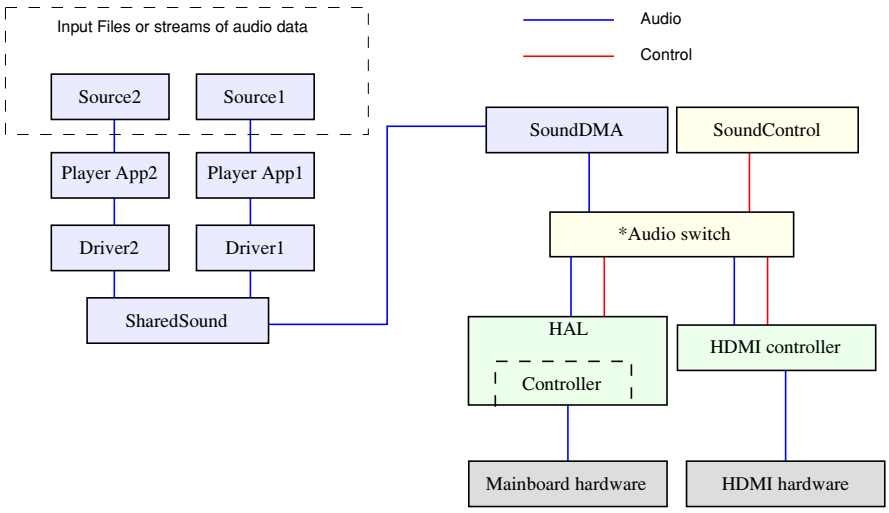


Fig 5 Overview of audio playback with controller choice

Figure 5 illustrates how these new abilities essentially sit 'underneath' the pre-existing audio and control pathways when playing audio. In practice, so far as any audio playing software is concerned it can operate as previously. It general it need not know which audio controller/ hardware is actually selected and in use. Provided the hardware can play at the required sample rate it just operates as before. The changes only really affect what happens once audio has been passed to the SoundDMA system.

When the user, or some software, needs to select or change the audio controller and hardware to be used they can make use of the established command, **`*Audio`**. This has now been extended so it can accept a parameter (string) which tells the HAL (Hardware Abstraction Layer) which audio controller and hardware should now be connected to SoundDMA.

For example, on the ARMX6 machine

```
*Audio SoundDMA#HAL_0005_41000000
```

would tell the sound system to use the mainboard hardware and its controller. Whereas

```
*Audio SoundDMA#HAL_0007_16000000
```

would tell the sound system to use the HDMI audio hardware and controller (assuming this was present and initiated.) The strings that follow the ***Audio** command are "ID strings" which indicate which controller is to be used. Note that these strings, and the details of the actual audio controllers, may vary depending on your hardware, etc. Also, in practice the 'switch' is a part of the SoundDMA module, and the HDMI controller is a part of the IMXVideo module. However these details should not matter for practical use of the system.

The SWI **Sound_SelectDefaultDevice** can also be used to choose the required controller and hardware . Hence,

```
sprintf(deviceIDstring,"SoundDMA#HAL_0007_16000000\0");
rin.r[0]=(int)deviceIDstring;
_kernel_swi(Sound_SelectDefaultDevice,&rin,&rout);
```

would also set the chosen audio controller and hardware to be HDMI on the ARMX6. If the call was successful it will return a zero in **rout.r[0]**. A non-zero value will be a pointer to an error block detailing the problem encountered. An obvious problem which may arise is that the machine running the program lacks the new ability to select/change the default audio hardware and controller. The SWI **Sound_ReadSysInfo** reads the features the sound system provides and thus can be used to determine the relevant information, as exampled below.

```
rin.r[0]=1;   /* reason code 1 */
_kernel_swi(Sound_ReadSysInfo,&rin,&rout);
result = 2&rout.r[1];
if (result == 2)
{
  printf("System supports choice of sound hardware/controller.\n");
}
else
{
  printf("Old OS, no choice of sound hardware!\n");
}
```

If you don't already know the ID string, the same SWI can also be used to obtain its value for the current audio hardware controller. For example:

```
char  devstring[1024];     /* set up a buffer for the result */

rin.r[0]=2;                /*  reason code 2 */
rin.r[1]=(int)devstring;   /*  tell the swi where the buffer is */
rin.r[2]=1023;             /*  max safe length for result  */
_kernel_swi(Sound_ReadSysInfo,&rin,&rout);
printf("ID string for the current device = %s\n",devstring);
```

will print out the ID string for the current sound hardware controller. Note that I chose 1024 bytes for the buffer size simply as a value I felt was likely to be large enough. Then told the SWI that the maximum available size was only 1023 to ensure no chance of it overflowing.

The SWI **Sound_EnumerateDevices** can be used to list all of the controllers available. For each controller that is found the SWI **Sound_DeviceInfo** can also obtain a 'name' string. For example:

```
char devstring[1024];
char namestring[1024];

rin.r[0]=0;
rin.r[1]=(int)devstring;
rin.r[2]=1023;
_kernel_swi(Sound_EnumerateDevices,&rin,&rout);
printf("ID string for audio device = %s\n",devstring);

dslen=0; dvcount=0;
do
{
  printf("\n");
  dvcount++;
  rin.r[0]=rout.r[1];
  rin.r[1]=(int)devstring;
  rin.r[2]=1023;
  _kernel_swi(Sound_EnumerateDevices,&rin,&rout);

  if(rout.r[2]>1)
  {
    printf("ID string for audio device = %s\n",devstring);
    nin.r[0]=(int)devstring;
    nin.r[1]=(int)namestring;
    nin.r[2]=1023;
    nin.r[4]=0;
    _kernel_swi(Sound_DeviceInfo,&nin,&nout);
    printf("Name of device = %s\n",namestring);
  }
  dslen=rout.r[2];
} while (dslen>1);
```

uses these calls to find out which controllers are present and list the name strings they provide on request. On my ARMX6 at present this gives the results:

ID string for audio device = SoundDMA#HAL_0007_16000000
Name of device = i.MX6 HDMI audio controller

ID string for audio device = SoundDMA#HAL_0005_41000000
Name of device = SGTL5000-compatible audio controller

Note that these name strings *aren't* the ID strings. The ID strings are required for the ***Audio** command or the related SWIs when you wish to select a controller. The name strings provide a more human-readable way to see the nature of each controller and its associated hardware. On the ARMX6 the mainboard audio 'codec' hardware is modelled on the SGTL5000 chip architecture. Hence the name string tells us this is the mainboard controller. The name which includes "i.MX6 HDMI" clearly identifies the HDMI audio controller.

The above assumes that the HDMI audio controller is available and can be used. In practice this requires initiation and selection. This can be done via two new commands.

The first of these is the ***ReadEDID** command. This prompts the OS to interrogate the attached

HDMI monitor and discover details like what screen and audio modes it supports. The most immediate and obvious effect of this command will be if you examine the monitor settings offered via the 'Display' icon on the iconbar. The screen modes offered will probably have changed because the OS has found out what screen modes your monitor supports and now offers these instead of whatever was listed in your old Monitor Definition File.

If you are using an HDMI monitor it is now recommended that you have an obey file containing the ***ReadEDID** command which is run during the bootup process. The main reason for this is that it will optimise the screen modes offered by the 'Display' application. Before the January 2016 developments the screen modes offered were based upon a Monitor Definition File (MDF) that had to be written to be suitable for your specific monitor. This approach is now replaced by using the new command to ask the monitor you are using to specify what screen modes it can accept. These then replace the ones from any MDF you had been using. This should mean optimised results. It should also mean that your computer will automatically adapt if you change HDMI monitor!

The second new command is ***HDMIOn**. This tells the OS to treat the monitor as a fully HDMI device, not a DVI device. The key difference here is that the HDMI protocols can convey audio, but DVI – even via an HDMI lead – cannot. In effect, this command sets up the computer to be able to send audio via HDMI. However if you play any audio at this point it will still go via the main computer output as before. Note that this command's name is perhaps slightly confusing because in practice what it does is switch on HDMI *audio*. It is worth being aware of this because there is in fact another similar command, ***HDMIOff**, which changes the output back to the DVI system and hence disables audio transfer via HDMI. Despite its name, an HDMI monitor should still function as your display following an ***HDMIOff**. However in practice you are unlikely to need to use ***HDMIOff**.

Having used the above commands to actually switch the audio output you then need to issue the relevant ***Audio** command to actually get HDMI audio working. Hence in practice if you wish to have this enabled from bootup you can include an obey file containing the commands:

```
*ReadEDID
*HDMIOn
*Audio SoundDMA#HAL_0007_16000000
```

This reads the relevant information from the HDMI monitor, enables audio being able to get to the monitor, and sets the sound system to use the HDMI audio controller as the current default.

The audio formats which the controller/hardware can accept will vary from one example to another. Two new SWI calls have been introduced to enable the details of the current HDMI device to be determined if required. The SWI **ScreenModes_EnumerateAudioFormats** can be used to determine what types of audio data stream can be accepted by the HDMI device connected to the machine. So for example:

```
hdmi_format=1;
hdmi_token=-1;
do
{
  rin.r[0]=1;
  rin.r[1]= hdmi_format;
  rin.r[2]= hdmi_token;
  _kernel_swi(ScreenModes_EnumerateAudioFormats,&rin,&rout);
```

```
    hdmi_format=rout.r[1];
    hdmi_token=rout.r[2];
    /* if greater than zero, modes found OK so loop though them */
    if(hdmi_token >=0)
    {
      modes_found=1;
      format_name[0]=(char)0;
      switch(hdmi_format)
        {
          case 1 : sprintf(format_name,"LPCM \0");
          break;
          case 2 : sprintf(format_name,"AC-3 \0");
          break;
          case 3 : sprintf(format_name,"MPEG1\0");
          break;
          case 4 : sprintf(format_name,"MP3  \0");
          break;
          case 5 : sprintf(format_name,"MPEG2\0");
          break;
          case 6 : sprintf(format_name,"AAC  \0");
          break;
          case 7 : sprintf(format_name,"DTS  \0");
          break;
          case 8 : sprintf(format_name,"ATRAC\0");
        }

      if (hdmi_format==1)
      {
        printf(" Format %d = %s, channels %d, sample rate %d, bits/
channel %d\n",hdmi_format,format_name,rout.r[3],rout.r[4]/
1024,rout.r[5]);
      }
      else
      {
        printf(" Format %d = %s, channels %d, sample rate %d, max
bitrate %d\n",hdmi_format,format_name,rout.r[3],rout.r[4]/
1024,rout.r[5]);
      }
    }
  } while (hdmi_token != (-1));
```

will list the audio formats which a HDMI device (e.g. monitor) can accept via its controller. As an example, an LG IPS235 monitor I tried listed the following modes:

Format 1 = LPCM , channels 2, sample rate 32000, bits/channel 16
Format 1 = LPCM , channels 2, sample rate 44100, bits/channel 16
Format 1 = LPCM , channels 2, sample rate 48000, bits/channel 16
Format 1 = LPCM , channels 2, sample rate 32000, bits/channel 20
Format 1 = LPCM , channels 2, sample rate 44100, bits/channel 20
Format 1 = LPCM , channels 2, sample rate 48000, bits/channel 20
Format 1 = LPCM , channels 2, sample rate 32000, bits/channel 24
Format 1 = LPCM , channels 2, sample rate 44100, bits/channel 24
Format 1 = LPCM , channels 2, sample rate 48000, bits/channel 24

This shows it will accept LPCM stereo audio at the common standard bit rates and sample sizes. Note that at present the ROSS is still limited to a maximum of 16 bits per sample. Audio CD material is 44100/16. and 48000/16 is the usual default for TV broadcasts, DVB Video stereo output, etc.

Some better monitors will also show that they can support other formats, e.g. AC-3, for 6 or 8 channel 'surround sound' for Home Theatre uses. However at present the ROSS can only handle sending stereo LPCM audio. Your monitor may have an optical digital output, but note that it seems standard for these to always output at the 48000 samples/sec rate for LPCM *whatever* you LPCM rate you send it over HDMI.

Another SWI, **ScreenModes_ReadInfo** will ask the HDMI device how many 'speakers' it can provide. Note this is an optional part of the HDMI specifications. So not all monitors or devices will give a reply. By using:

```
 rin.r[0]=1;
 _kernel_swi(ScreenModes_ReadInfo,&rin,&rout);
 hdmi_speaker=rout.r[0];
 hdmi_valid=rout.r[1];
 if ((hdmi_speaker&hdmi_valid) == 0)
 {
   printf(" No HDMI Speakers found!\n\n");
 }
 else
 {
   hdmi_bitmasked=(hdmi_speaker&hdmi_valid);
   printf (" HDMI speaker arrangements reported as available:\n\n");
   if ((hdmi_bitmasked&1) == 1)   printf("Stereo Left and Right\n");
   if ((hdmi_bitmasked&2) == 2)   printf("LFE\n");
   if ((hdmi_bitmasked&4) == 4)   printf("Front  Center\n");
   if ((hdmi_bitmasked&8) == 8)   printf("Rear Left and Right\n");
   if ((hdmi_bitmasked&16) == 16) printf("Rear Center\n");
   if ((hdmi_bitmasked&32) == 32) printf("Front Left Center and Front
Right Center\n");
   if ((hdmi_bitmasked&64) == 64) printf("Rear Left Center and Rear
Right Center\n");
 }
```

you can list any speaker arrangements which the HDMI device reports. Because responding is optional you may get no response despite the device actually having speakers you can use!

The RISC OS Sound System - Reference.

Introduction

This documents the SWIs and commands used by various Sound modules, etc. You should refer to the general ROSS document for examples illustrating the uses of these SWIs/commands and explanations of how the various parts of the system connect and work together. In general it may be advisible to call the X versions of SWIs to trap errors as not all drives support all SWIs.

CDFS and CD Audio

Here the emphasis is mainly on the use of the CD module for Audio discs so I will omit the details of those SWIs that are only for data CDs . There are also some SWIs for which I have no details and others I can't check because they don't work on all hardware! However the full list of SWIs provided on my ARMX6 are:

```
0x041240 CD_Version             0x041241 CD_ReadData            0x041242 CD_SeekTo
0x041243 CD_DriveStatus         0x041244 CD_DriveReady          0x041245 CD_GetParameters
0x041246 CD_SetParameters       0x041247 CD_OpenDrawer          0x041248 CD_EjectButton
0x041249 CD_EnquireAddress      0x04124A CD_EnquireDataMode     0x04124B CD_PlayAudio
0x04124C CD_PlayTrack           0x04124D CD_AudioPause          0x04124E CD_EnquireTrack
0x04124F CD_ReadSubChannel      0x041250 CD_CheckDrive          0x041251 CD_DiscChanged
0x041252 CD_StopDisc            0x041253 CD_DiscUsed            0x041254 CD_AudioStatus
0x041255 CD_Inquiry             0x041256 CD_DiscHasChanged      0x041257 CD_Control
0x041258 CD_Supported           0x041259 CD_Prefetch            0x04125A CD_Reset
0x04125B CD_CloseDrawer         0x04125C CD_IsDrawerLocked      0x04125D CD_AudioControl
0x04125E CD_LastError           0x04125F CD_AudioLevel          0x041260 CD_Register
0x041261 CD_Unregister          0x041262 CD_ByteCopy            0x041263 CD_Identify
0x041264 CD_ConvertToLBA        0x041265 CD_ConvertToMSF        0x041266 CD_ReadAudio
0x041267 CD_ReadUserData        0x041268 CD_SeekUserData        0x041269 CD_GetAudioParms
0x04126A CD_SetAudioParms       0x04126B CD_SCSIUserOp
```

Note that the CD Data Block (CDDB) is explained in the main part of the ROSS documentation. It acts to identify the drive upon which a SWI will act.

What follows is a list of some of the main Audio and drive SWI APIs related to being able to read Audio CDs.

`CD_Version` Identifies the version of the module.
On exit **`r[0]`** points to a version and string for the CDFS module.

`CD_DriveStatus` Determines the drive's operational status
On entry **`r[7]`** = pointer to CDDB for the drive
On exit **`r[1]`** = **`1`** (CD in drive), **`2`** (busy), **`4`** (not ready), or **`8`** (unavailable)

`CD_OpenDrawer` Opens drawer.
On entry **`r[7]`** = pointer to CDDB.

`CD_EjectButton` Controls any eject button. (Allows the drive to be locked/unlocked.)
On entry **`r[0]`** = **`0`** (enable button) or **`1`** (disable button), **`r[7]`** = pointer to CDDB.

`CD_PlayTrack` Plays a chosen track.
On entry **`r[0]`** = track number, **`r[1]`** = **`&FF`**, **`r[7]`** = pointer to CDDB.

CD_AudioPause Pause or resume audio playback.

On entry **r[0]** = **0** (resume) or **1** (pause), **r[7]** = pointer to CDDB

CD_EnquireTrack Returns information on the tracks present on the Audio CD.

The action depends on the value given on entry in **r[0]**.

If **r[0]** = **0**, and **r[1]** = pointer to 2-byte buffer, **r[7]** = pointer to CDDB then it returns the first and last track numbers in the bytes of the buffer pointed to by **r[1]**.

If **r[0]** = track number (**1 - 99**), **r[1]** = pointer to 2-int buffer, **r[7]** = pointer to CDDB it returns the start frame in the first (4 byte) int of the buffer and a value identifying the disc type in the second.

CD_AudioStatus Discovers if a disc is playing, etc.

On entry **r[7]** = pointer to CDDB

On exit **r[0] AND &F** = **0** (playing), **1** (paused), **3** or **5** (stopped), **4** (error)

CD_DiscUsed Details of disc space usage.

On entry **r[0]** = **1**, **r[1]** points to an 8-byte buffer, **r[7]** = pointer to CDDB

On exit the buffer contains at offsets:

+0 = number of frames, +1 = number of seconds, +2 = number of mins

+3 = always zero, +4 always 2048

This specified the total duration of the recording on the Audio CD in mins, seconds, and frames (75ths of a second). It includes 151 'inaccessible' audio frames. So the total number of readable audio frames on the disc is such that:

```
(min*60) + sec)*75 + frames = readable + 151
```

CD_Inquiry Returns drive details

On entry **r[0]** = pointer to 36-byte buffer, **r[7]** = pointer to CDDB

On exit the buffer contains the device name in the bytes at offset 8 - 31 and its firmware version in bytes 32 - 36.

CD_CloseDrawer Closes drive drawer

On entry **r[7]** = pointer to CDDB

CD_Identify Returns driver handle for use with other SWIs.

On entry **r[7]** = pointer to CDDB with driver handle set to zero.

On exit **r[2]** returns **-1** (drive invalid) or drive handle.

I will add more Audio CD SWI details at such time as I can find out more about them, or can even determine which of them actually work!

Digital Renderer

The primary purpose of the DigitalRenderer (DR) module is to allow programs and applications ported from Linux to be able to play/render their output via the ROSS. It is not a standard part of RO but is widely used for such purposes. As outlined in the main ROSS document it is also a

convenient and easy way to play audio files. It provides a filing system for playback, a set of commands, and a set of SWIs. I have already explained the commands provided by DR. Here I will give more detail regarding the DRender: file system and the DR SWIs. Note that much of the comments given below were kindly provided by Andreas Dehmel or Chris Martin.

DRender:

From Version 0.40 onward the DR module has provided this filing system to allow the user to play audio by simple "writing the sample values to a file". This allows using samples like on Unix by writing them to a pseudo-device (/dev/dsp). All you have to do is first configure the defaults (see SWI **`DigitalRenderer_SetDefaults`** or the command **`DRenderDefaults`**), open the file and pipe the data in. If you're running in a TaskWindow you don't even have to take care you don't write too much data because the module will automatically sleep using **`UpCall 6`** when the buffer is full (that's the same UpCall the TaskWindow uses to yield control to other tasks and means that the computer doesn't freeze for the duration). However if you're using this interface from an application, you should check the number of free buffers using SWI DigitalRenderer_StreamStatistics, however.

The filing system uses the streaming interface. Only one file can be opened on this filing system. When it's opened, the default values are activated (number of channels, number of buffers, sample period, ...) and the data is streamed into the buffers (and the music starts). This might come in very handy if you're porting applications from Unix. Bear in mind, however, that the file opened on **`DRender:`** should *not* be buffered, otherwise you might get strange dropout effects.

NOTE: there is a problem with the **`*copy`** command which doesn't work with the **`DRender:`** filing system (at least not in a TaskWindow). I don't know exactly why that is, maybe copy can't stomach the UpCall or stops interrupts, I just advise you not to use that command but rather OS_Find / OS_GBPB combinations. DRender is officially registered as filing system number 167.

SWIs

```
0x04F700 DigitalRenderer_Activate              0x04F701 DigitalRenderer_Deactivate
0x04F702 DigitalRenderer_Pause                 0x04F703 DigitalRenderer_Resume
0x04F704 DigitalRenderer_GetTables             0x04F705 DigitalRenderer_ReadState
0x04F706 DigitalRenderer_NewSample             0x04F707 DigitalRenderer_New16BitSample
0x04F708 DigitalRenderer_BufferStatistics      0x04F709 DigitalRenderer_NumBuffers
0x04F70A DigitalRenderer_StreamSamples         0x04F70B DigitalRenderer_Stream16BitSamples
0x04F70C DigitalRenderer_StreamStatistics      0x04F70D DigitalRenderer_StreamFlags
0x04F70E DigitalRenderer_SetDefaults           0x04F70F DigitalRenderer_Activate16
0x04F710 DigitalRenderer_GetFrequency          0x04F711 DigitalRenderer_ActivateAuto
0x04F712 DigitalRenderer_SampleFormat
```

`DigitalRenderer_ActivateAuto` Ideal call for the impatient!

On entry:

`r[0]` = number of channels

`r[1]` = buffer size per channel

`r[2]` = sample rate

All registers preserved on exit.

This call first tries **`DigitalRenderer_Activate16`** with **`r[3]`** = **`1`** (i.e. restore the previous handler). If that call fails, it converts the sample rate to the nearest value available and calls **`DigitalRenderer_Activate`**. Only if that call fails does it return with an error.

DigitalRenderer_Activate Activates DR use of the (8-bit) system.

On entry:

r[0], bits 0-7 = number of channels (will be rounded up to 1/2/4/8)
bit 31: clear (i.e. = **0**) use polling. Otherwise call back buffer mode.

r[1] = sample buffer length (bytes per channel) must be a multiple of 16.

r[2] = Sample period (microseconds). i.e. 1000000/sample_rate

r[3] = pointer to buffer fill code if **r[0]**, bit 31 set to **1**.

Your application must call this to take over the 8-bit sound system. N.B. Once activated, things like the system beep won't work. This call activates the old 8-bit system. Use the following SWI to activate the 16-bit system.

DigitalRenderer_Activate16 Activate DR use of 16-bit sound system.

On entry:

r[0] = number of channels. Must be 1 or 2 (limitation of 16bit ROSS)

r[1] = buffer size per channel in samples, not bytes.

r[2] = Sample rate

r[3] = flags: If bit 0 set (i.e. = **1**) restore previous handler on exit.

Use this SWI to activate DR using the 16-bit sound system. All other DR SWIs automatically adapt to the system chosen without your needing to worry about internal details. The call will return with an error if no 16-bit system is available. In stereo mode, the samples are by default assumed to be in the correct right-left channel order for Acorn's 16-bit sound system. If this is not the case, you'll have to call **DigitalRenderer_SampleFormat** with format 3 (left-right).

DigitalRenderer_Deactivate Deactivates DR using sound system.

No values required on entry or returned on exit.

DigitalRenderer_Pause Pauses playback.

No values required on entry or returned on exit.

This SWI suspends sound playback (disabling DMA on sample buffers and so on). In contrast to deactivating DR this will not install the old handlers but merely mute the entire sound system.

DigitalRenderer_Resume Resumes playback.

No values required on entry or returned on exit.

The resumes playback after it has been paused.

DigitalRenderer_GetTables For 8-bit only.

On exit: **r[0]** = pointer to 8k LinToLog table.
r[1] = pointer to 256 byte table scaling 8-bit log samples according to the currently selected *volume value.

DigitalRenderer_ReadState Indicates DR's state

On exit: **r[0]** returns a value whose bits being set indicate:
bit 0: DigitalRenderer active

bit 1: New sample data required
bit 2: Buffer overflow occurred
bit 3: (reserved)
bit 4: Playback is paused
bit 5: 16bit sound hardware used

Call this SWI to poll DigitalRenderer in single-buffer mode. If Bit 1 is set you have to provide a new sample buffer by calling DigitalRenderer_NewSample. Bit 2 lets you check for buffer overflows (i.e. your application took too long to provide the next sample). If you're using the call-back method or the streaming interface, bits 1 and 2 are undefined. For streaming interface, use the SWI **DigitalRenderer_StreamStatistics**.

DigitalRenderer_NewSample Provides new samples via buffer.
On entry: **r[0]** = pointer to buffer holding new samples (8-bit ulaw)

DO NOT call this SWI if you're using the call-back method or the streaming interface. It is for single-buffer polling only. Call this when **DigitalRenderer_ReadState** returned with Bit 1 set. The buffer must have the correct size (number of channels times Sample length) and contain data that can be sent directly to the VIDC, i.e. 8-bit logarithmic data, interleaved for multiple channels.

DigitalRenderer_New16BitSample Provides new samples via buffer.
On entry: **r[0]** = pointer to buffer holding new samples (16-bit linear)

DO NOT call this SWI if you're using the call-back method or the streaming interface. It is for single-buffer polling only. Call this when **DigitalRenderer_ReadState** returned with Bit 1 set. The buffer must have the correct size. Details generally as **DigitalRenderer_NewSample** but with a larger buffer due to the samples having two bytes per sample.

DigitalRenderer_NumBuffers Set or read the number of buffers to use.
On entry: **r[0]** = new number of buffers for streaming interface or **-1** to read.
On exit: If **r[0]** = **-1** on entry **r[0]** returns the number of buffers currently allocated.

This call must be issued before **DigitalRenderer_Activate** (or **DigitalRenderer_Activate16)** and initializes the streaming interface (if **r[0]** > **0**). In this mode you can pipe samples into a ring buffer containing **r[0]** buffers using the **DigitalRenderer_StreamSamples** (or **DigitalRenderer _Stream16BitSamples**) SWIs. Each buffer is as large as the one specified when activating DR (which ideally is also the size of the sound hardware's buffers). This system basically allows caching a large number of samples. You can switch off the streaming interface when the DR is inactive by calling with **r[0]** = **0**.

DigitalRenderer_StreamSamples Provides 8-bit samples for streaming.
On entry:
r[0] = pointer to buffer holding 8-bit ulaw samples.
r[1] = number of samples.

Provides 8-bit ulaw samples to the ring buffer in streaming mode. The format is as required for the sound hardware (see PRM4). The number of samples can be any size, but should normally not be larger than the ring buffer size or the call will either block or samples will be truncated,

depending on the stream flags (**DigitalRenderer_StreamFlags**). Note that the number of samples for all channels combined and not for an individual channel.

DigitalRenderer_Stream16BitSamples Provides 16-bit samples for streaming.

On entry:

r[0] = pointer to buffer holding 16-bit signed linear samples.

r[1] = number of samples. (Not bytes!)

Similar to **DigitalRenderer_StreamSamples**, but the samples are in 16bit signed linear rather than 8-bit ulaw. As in **DigitalRenderer_StreamSamples**, the number of samples is for all channels combined.

DigitalRenderer_StreamStatistics Discovers number of buffers in queue.

On exit: **r[0]** = Number of buffers filled and waiting to be played

DigitalRenderer_StreamFlags Read or change stream flags

On entry:

r[0] = **EOR** mask

r[1] = **AND** mask (applied first)

On exit: **r[0]** = old stream flags

Use this to read and change the stream flags. At the moment the following flags are defined, with the following meaning when set:

- **0**: (reserved)
- **1**: fill buffers with **0** on overruns, otherwise repeat last buffer
- **2**: use **UpCall6** if more data is written to **DRender:** than fits the buffer. That means the task sleeps, otherwise it has to do busy waiting.
- **3**: (reserved)
- **4**: block if more samples are streamed than fit into the buffer

Use r[**0] = 0**, **r[1] = -1** to read.

DigitalRenderer_SetDefaults Set or read DR current defaults.

On entry:

r[0] = number of channels (or **0** to read).

r[1] = format **1**: 8bit ulaw;
2: 16bit signed linear little-endian right-left;
3: 16bit signed linear little-endian left-right)
or **0** to read

r[2] = sample period or **0** to read

r[3] = buffer size per channel or **0** to read

r[4] = number of buffers for streaming interface

r[5] = sample rate; positive values mean: use 16bit RISC OS sound in preference

DigitalRenderer_GetFrequency Read the sample rate

r[0] = sample rate, or 0 if DR is inactive.

DigitalRenderer_SampleFormat Changes or reads format

On entry: **r[0]** = new format or 0 to read current format.

On exit: **r[0]** = current format.

See **DigitalRenderer_SetDefaults** for the details of how the format is represented. This call allows you to change the sample format from the default. The most important application of this call is to swap the left and right channels in 16bit stereo mode. Changing the format in 8bit mode doesn't have an effect. The call can be issued at arbitrary times even when the stream is open.

HDMI Audio and Screen Modes

Three new commands have been introduced with the ability to employ HDMI audio.

***ReadEDID**

This command prompts the computer to interrogate the monitor, screen, or other device attached via HMDI. It collects the relevant details supplied by the device which describe the screen and audio modes, etc, it can accept. It also updates the Display Manager so that it will list the relevant screen modes which can be used. If you were previously using a Monitor Definition File to specify the screen modes that could be selected via the Display Manager, this command will replace them with the modes reported by the actual screen.

***HDMIOn**

This sets the output from the computer's HDMI socket be in HDMI format. HDMI format includes the ability to carry digital audio.

***HDMIOff**

This sets the output from the computer's HDMI socket to be in DVI format. Unlike HDMI, the DVI format cannot convey audio. However the video formats used are essentially the same.

One command has had its options extended.

***Audio on | off | <deviceIDstring>**

As previously **on** or **off** turn the sound system on or off. The **deviceIDstring** is a string which specifies which audio controller is to be used. Hence it determines what audio hardware is connected for current use.

The available SWIS are:

```
0x0487C0 ScreenModes_ReadInfo       0x0487C1 ScreenModes_EnumerateAudioFormats
```

ScreenModes_ReadInfo Reads info from the HDMI screen.

The action depends on the value given on entry in **r[0]**.

If on entry, **r[0]** = **0** then on exit:

r[0] contains a pointer to a null-terminated name for the screen.

If on entry, **r[0]** = **1** then on exit:

r[1] contains a Speaker mask

r[2] contains a Validity mask

It is *optional* for a screen or other HDMI device to return this information. The Validity mask indicates if the data has been given or not. If both returned values are zero, no information was returned. Otherwise each bit of the values represents a specific Speaker arrangement. If a Validity bit = `1` the Speaker bit tells us if a given Speaker arrangement is supported. The arrangments for each bit in the masks are as follows:

Bit 0 = Front Left and Front Right
Bit 1 = LFE
Bit 2 = Front Center
Bit 3 = Rear Left and Rear Right
Bit 4 = Rear Center
Bit 5 = Front Left Center and Front Right Center
Bit 6 = Rear Left Center and Rear Right Center

Because responding is optional, an HDMI device may accept and play audio despite the above SWI returning zeros in `r[1]` and `r[2]`.

`ScreenModes_EnumerateAudioFormats` Lists supported audio formats.

On entry:
If `r[0]` = `0` Return data in 'raw' format
If `r[0]` = `1` Return data in 'friendly' format
`r[1]` = Format code number for this entry (`-1` for first)
`r[2]` = Enumeration token for format code (`-1` to start with first)
On exit:
`r[0]` = Preserved
`r[1]` = Format code for this entry (`-1` if no more entries)
`r[2]` = Enumeration token for this entry (`-1` if no more entries)

For 'raw' results:
`r[3]` = Max number of channels
`r[4]` = Audio short descriptor byte 2
`r[5]` = Audio short descriptor byte 3

For 'friendly' results:
`r[3]` = Max channels
`r[4]` = Sample rate in units of 1/1024 Hz
`r[5]` = For format code 1 (LPCM): Sample bit depth (size)
For format code > 1: Max bit rate in Hz.

Enumeration can start at the first supported format (by specifying `r[0]` and `r[1]` as `-1`) or at a specific format (Set `r[1]` to the required format code, and `r[2]` to `-1`). On return `r[1]` and `r[2]` will indicate the validity of the data; for valid data they will both *not* be `-1`, and for invalid data they will both be `-1`. Only if valid data is indicated will `r[3]` to `r[5]` be updated.

The format code specified in `r[1]` corresponds to the format codes defined by CEA 861, e.g. basic LPCM audio is format code 1. The audio formats can be enumerated in two modes: 'raw' mode and 'friendly' mode. 'raw' mode returns bytes 2 and 3 of the CEA short audio descriptor directly (byte 1 will have already been decoded to determine the values of `r[1]` and `r[3]`). 'friendly' mode interprets the bytes and converts them to standard units – however note this is currently only supported for format codes 1 through 8.

The results of the enumeration will be ordered by their format code, so if you are only interested in one format code you can simply begin enumeration with `r[1]` set to that value and end enumeration when on exit `r[1]` deviates from that value.

For monitors with complex EDID, it is possible that the same format will be reported twice when enumerating in 'friendly' mode.

Shared Sound

The Shared Sound module was originally developed by ESP (Expressive Software Projects) but has now become a standard part of the ROSS. Its purposes may be summarised as:

- Shared use of digital sound output (i.e. 'mixing').
- 16 bit linear facility independently from the hardware used.
- Software support to potential users of digital sound output.
- A call back facility for CPU intensive handlers allowing the hardware drivers to operate with interrupts disabled.
- Provision for effects and other processes to act on the sound

SharedSound allows multiple sound handlers to output symultaenously through one sound source. It can potentially provide a number of useful functions such as mixing, sample rate conversion via interpolation, and volume scaling. However, current versions of the module simply pass the relevant values onto the handlers and they are expected to do the work. Values are passed to the Handler fill routine which indicates the current state of play – in particular the current sample rate, along with associated values, as well as the current volume setting. The volume for a handler is calculated from the current system volume and the volume requested for the handler as well as scaling depending on the number of currently active handlers. This is passed as a Left and Right (Stereo) volume as two 16 bit words packed into a 32 bit word.

SharedSound acts as a universal interface to any supported hardware. This currently includes support for the Risc PC 16 bit sound system upgrade and the Audio Dynamics DMI50/S card.

The current version of SharedSound provides three levels of handler activity.

- The first is an immediate handler. This is called with interrupts disabled and must return as quickly as possible. Such handlers are likely to be simple 'fill buffer' handlers that do not need to process any data, or a timing device that is using the Sound interrupt as a timer.
- The second type of handler is a call back handler. This is called on a call back and can therefore operate with interrupts enabled and take more time to execute.
- The third is a process handler. This is called in the same way as a call back handler but is called at the end of the call back process to allow it to add effects or process the current buffer in some way.

SharedSound is currently used by the the Sound16 codecs for Replay, the PCSound support module and the MIDI Synthesiser.

SharedSound has the `SharedSound_HandlerVolume` SWI for setting the volume of the left & right channels for a specific client, and `SharedSound_DriverVolume` for setting the volume of the output as a whole. As with `*Volume`, it's the client's responsibility to obey the values. Judging by the implementation of the built-in buffer fill code, the volume values are simple multipliers applied to the 16-bit data, with a resolution of 1/65536. i.e. with both the

handler and driver volume set to maximum, you actually get a multiplier of 1/65534 due to the way the handler and driver volumes are combined into a 16-bit volume level that's passed to the client/fill code.

Note that when SharedSound clients produce their output, they mix it directly into the buffer containing the 16-bit data produced by SoundDMA's log-to-linear code. This means they must perform their own clipping on the values, and if there are lots of sounds playing then the multiple clipping stages could easily result in corruption.

Note also that SharedSound is currently still being developed so new features may be added and some intended features will be implimented. Hence the details I give below can be expected to change with time and may be out-of-date.

SharedSound handlers

The handlers of each type are called in the order in which they were installed. Process handlers are called after call back handlers. Each handler is passed a buffer to fill with the registers used as follows:

Input:

`r[0]` = parameter
`r[1]` = base address of buffer to fill
`r[2]` = end of buffer
`r[3]` = flags
`r[4]` = sample rate
`r[5]` = sample period
`r[6]` = fractional step as specified by `SharedSound_SampleRate`
`r[7]` = LR volume
`r[8]` = Pointer to fill code

Exit:

`r[3]`, `r[4]`, `r[5]`, and `r[6]` are updated at the start of a buffer fill to represent the current values.

The important requirement on exit is that bit `0` of `r[3]` should be set if any data has been written into the buffer. In fact, SharedSound cannot currently cope with no data being filled, so zeros should always be written. More generally, the details of the flag bits are as follows:

Flags

Bits

`0` Mix. If set then data must be mixed, otherwise overwrite.
`1` Quality. If set then use highest quality e.g. oversampling. *
`3-7` Overrun count *
  = `0` No overrun
  = `1` to 30 times buffer has been overrun
  = `31` Buffer overrun > 30
`29` Stereo. Reverse stereo if set. *
`30` Set if to use volume in R7, otherwise use max volume
`31` Set if volume in R7 is zero, ie. muted.

N.B. * = Not currently implimented (SharedSound version 1.11). Only bit 0 of the above flags is implemented by SharedSound. The other flags are currently used by the Sound16 drivers.

Fill code

On entry to the fill handler, `r[8]` points to the following table:

Offset	
`0`	Flags word, currently `0`
`4`	Pointer to "silence fill" routine
`8`	Pointer to "data fill" routine

This table may be extended in future to include more fill variants; the presence/absence of these routines will be indicated by bits in the flags word.

Silence fill code

On entry:

`r[1]` = Base address of buffer to be filled
`r[2]` = End of the buffer to be filled
`r[3]` = Flags (as on entry to the fill handler)
`r[6]` = Fractional step value
`r[7]` = LR volume
`r[9]` = Fractional accumulator

On exit:

`r[0]` = New flags (to be returned from fill handler in R3)
`r[1]` = Next byte to be filled
`r[4]` = Next byte to fill from
`r[9]` = New Fractional accumulator
`r[2] r[3] r[5] r[6] r[7] r[8] r[12]` preserved.
All other registers corrupted.

Data fill code

On entry:

`r[1]` = Base address of buffer to be filled
`r[2]` = End of the buffer to be filled
`r[3]` = Flags (as on entry to the fill handler)
`r[4]` = Base of buffer to fill from
`r[5]` = End of buffer to fill from
`r[6]` = Fractional step value
`r[7]` = LR volume
`r[9]` = Fractional accumulator

On exit:

`r[0]` = New flags (to be returned from fill handler in R3)
`r[1]` = Next byte to be filled
`r[4]` = Next byte to fill from
`r[9]` = New Fractional accumulator
`r[2] r[3] r[5] r[6] r[7] r[8] r[12]` preserved.
All other registers corrupted.

The SWIs provided are:

```
0x04B440 SharedSound_InstallHandler        0x04B441 SharedSound_RemoveHandler
0x04B442 SharedSound_HandlerInfo           0x04B443 SharedSound_HandlerVolume
0x04B444 SharedSound_HandlerSampleType     0x04B445 SharedSound_HandlerPause
0x04B446 SharedSound_SampleRate            0x04B447 SharedSound_InstallDriver
0x04B448 SharedSound_RemoveDriver          0x04B449 SharedSound_DriverInfo
0x04B44A SharedSound_DriverVolume          0x04B44B SharedSound_DriverMixer
0x04B44C SharedSound_CheckDriver           0x04B44D SharedSound_ControlWord
0x04B44E SharedSound_HandlerType
```

`SharedSound_InstallHandler` Used to install a handler.

On entry:

`r[0]` = handler address pointer

`r[1]` = parameter to pass

`r[2]` = flags

`r[3]` = pointer to (or name of) handler

`r[4]` = type of handler indicated by the value:

`0` = Immediate

`1` = Call back

`2` = Process

On exit: **`r[0]`** = handler number.

`SharedSound_RemoveHandler` Removes a handler.

On entry: **`r[0]`** = handler number.

`SharedSound_HandlerInfo` Returns details of handler.

On entry: **`r[0]`** = handler number or **`0`** for first call.

On exit:

`r[0]` = number of next handler on list or 0 for no more.

`r[1]` = flags

`r[2]` = pointer to name

`r[3]` = sample rate

`r[4]` = handler type

`r[5]` = volume

`SharedSound_HandlerVolume` Sets L/R volume (gain) values.

On entry:

`r[0]` = handler number

`r[1]` = volume

On exit:

`r[0]` preserved

`r[1]` = current volume

The volume value consists of two 16-bit values in a 32-bit int. Bits 0-15 set the Left channel volume (gain) and bits 16-31 set the Right channel volume (gain). SharedSound may pass this information on to the handler. At present SharedSound does not itself alter gain values to take mixing into account, although this may change in the future.

`SharedSound_HandlerSampleType` Not yet implimented?

SharedSound_HandlerPause Not yet implimented?

SharedSound_SampleRate Used to set a sample rate

On entry:

r[0] = handler number or **0** to set system rate for current output.

r[1] = sample rate required

On exit:

r[1] = sample rate

r[2] = sample period

r[3] = fractional step required to achieve this rate in 8.24 fixed point format

This SWI is used to set the sample rate used by a particular handler. Information is returned to indicate whether Shared Sound can provide this rate and if not what is the nearest rate it can provide. Along with this is a fractional step value to suggest how the handler can achieve it's required rate. Sample rate is always supplied in 1/1024ths of Hz.

SharedSound_InstallDriver Currently not implimented?

On entry:

r[0] = **1** for linear driver, or **2** for log driver, or a pointer to a driver table address.

r[1] = parameter to pass driver in **r[12]** when called.

r[2] = flags

r[3] = volume

This is to allow other output devices to be installed.

SharedSound_RemoveDriver Currently not implimented?

On entry:

r[0] = **1** to remove linear driver, **2** for log driver, or pointer to driver table address.

SharedSound_DriverInfo Currently not implimented?

On entry: **r[0]** = driver number

On exit:

r[0] = number of next available driver or **0** for none

r[1] = flags

r[2] = name

r[3] = volume

r[4] = overrun count.

SharedSound_DriverVolume Currently not implimented?

On entry:

r[0] = driver number

r[1] = volume

SharedSound_DriverMixer Currently not implimented?

On entry:

r[0] = driver number

r[1] = function (**0** = read, **1** = set)

r[2] = mixer number

r[3] = value

On exit:

r[0] preserved

r[1] = number of mixers

r[2] = value

r[3] = name

This sets or reads mixer values for a driver where relevant.

SharedSound_CheckDriver Check driver still active

No entry conditions or returns.

This is used to force SharedSound to make sure that current driver is still active. It is needed for situations where another application has taken control of the output Linear handler and has not returned the control to SharedSound.

SharedSound_ControlWord Replay Sound16 control

On entry: **r[0]** = control word

This is used by the Replay Sound16 codecs as part of the multi-tasking Replay implementation. It provides a control word to both Replay and the Sound16 code allowing for a "1 audible handler of many" support.

SharedSound_HandlerType Change handler type

On entry:

r[0] = handle

r[1] = type

This is used to change the type of a handler from, for example, an immediate handler to a call back handler.

Sound

The Sound module and the commands and SWIs it provides pre-date the addition of various other modules – in particular SharedSound – into the ROSS. A number of the SWIs and commands it provides have either been supplanted by newer alternatives or deal with 8-bit audio or the ancient 'Voice generator' system. For that reason, here I will ignore many of the SWIs and concentrate on the ones relevant for the modern 16-bit audio system and which don't have later alternatives which are to be preferred.

In the context of modern audio only a few commands provided by the Sound module are particularly relevant. These are the ***Audio**, ***Volume**, and ***Speaker**, commands which have already been discussed on pages 6-7 and page 34, along with the ***Configure SoundDefault** command. Note, however that the ***Audio** command now also allows the user to select the current audio controller/hardware as an extension to its original function. (See the section on HDMI Audio and Screen Modes for more details).

The full list of SWIs are grouped into three blocks:

```
0x040140 Sound_Configure          0x040141 Sound_Enable             0x040142 Sound_Stereo
0x040143 Sound_Speaker            0x040144 Sound_Mode               0x040145 Sound_LinearHandler
0x040146 Sound_SampleRate         0x040147 Sound_ReadSysInfo        0x040148 Sound_SelectDefaultDevice
0x040149 Sound_EnumerateDevices   0x04014A Sound_DeviceInfo

0x040180 Sound_Volume             0x040181 Sound_SoundLog           0x040182 Sound_LogScale
```

```
0x040183 Sound_InstallVoice         0x040184 Sound_RemoveVoice          0x040185 Sound_AttachVoice
0x040186 Sound_ControlPacked        0x040187 Sound_Tuning               0x040188 Sound_Pitch
0x040189 Sound_Control              0x04018A Sound_AttachNamedVoice     0x04018B Sound_ReadControlBlock
0x04018C Sound_WriteControlBlock

0x0401C0 Sound_QInit                0x0401C1 Sound_QSchedule            0x0401C2 Sound_QRemove
0x0401C3 Sound_QFree                0x0401C4 Sound_QSDispatch           0x0401C5 Sound_QTempo
0x0401C6 Sound_QBeat                0x0401C7 Sound_QInterface
```

`Sound_DeviceInfo` Reads the details of an audio controller/device

On entry:
r[0] = Pointer to the controller/device ID string.
r[1] = Pointer to a buffer for result, or 0 to check.
r[2] = Length of buffer.
r[3] = 0
On exit;
r[0] = 0, or pointer to error block.
r[1] = Buffer containing an identifying string given by the controlled device hardware.
r[2] = Length of data in r[1].

Note that the identifying string returned here in the r[2] buffer is *not* the same thing as the controller's device ID string. It is provided as a label which is more human-readable.

`Sound_Enable` Equvalent of **`*Sound On|Off`**

On entry:
If **`r[0]`** = **`0`** No action, just return current state.
If **`r[0]`** = **`1`** Disable sound output.
If **`r[0]`** = **`2`** Enable sound output.
On exit:
`r[0]` = Previous state [**`1`** = disabled, **`2`** = enabled]

`Sound_EnumerateDevices` List the audio controllers/devices

On entry:
`r[0]` = Pointer to space/control terminated ID string to start from, or **`0`** to start.
`r[1]` = Pointer to vuffer for result, or **`0`** to check required buffer length.
`r[2]` = Length of buffer pointed to by **`r[1]`**.
On exit:
`r[0]` = **`0`**, or pointer to error block.
`r[1]` = Buffer updated with null-terminated ID string of next device, or empty string if no more devices.

This call can be used to enumerate the available audio controllers and their hardware devices. If the supplied buffer was too small or the pointer was **`0`** an overflow error will be generated and no data copied into the buffer.

`Sound_Mode` Detect and control features of 16bit system

On entry:
If **`r[0]`** = **`0`** Reads presence of 16-bit sound system
On exit:
If **`r[0]`** = **`1`** 16-bit sound system available.

`r[1]` flag bits:
bit 1 set = 16-bit linear output inc. 44·1k, 22·05k, and 11·025k.
bit 2 set = 16-bit linear output, internal clock
bit 3 set = 16-bit linear output, external clock
bit 4 = oversampling mode (1 = enabled)
bit 5 = mono conversion status (1 = enabled)

On entry:
If `r[0]` = `1` Oversampling control:
If `r[1]` = `0` Disable oversampling
If `r[1]` = `1` Enable oversampling if possible
On exit:
`r[1]` = Previous oversampling state.

On entry
If `r[0]` = `3` Mono control
If `r[1]` = `0` Disable mono conversion
If `r[1]` != `0` convert to mono output
On exit:
`r[1]` = Previous mono conversion state.

In practice if you wish to use this call you should initially use it to check for the presence of a 16-bit system.

`Sound_ReadSysInfo` Examine audio features available

On entry:
If `r[0]` = `1` [reason code 1]
On exit:
`r [0]` = `0`
`r[1]` = flags
Bit 0 set = Supports generating service calls for Sound rate changing, Sound starting, and Sound stopping.
Bit 1 set = Supports the SWIs, Sound_SelectDefaultDevice, Sound_EnumerateDevices, and Sound_DeviceInfo
On entry:
If `r[0]` = `2` [reason code 2]
`r[1]` = Pointer to buffer for result, or `0` to check required length
`r[2]` = Length of buffer provided
On exit:
`r[0]` = `0` or error (e.g. buffer overflow)
`r[1]` = Buffer filled with null terminated string (if buffer was long enough)
`r[2]` = length of result inc null terminator

To detect if this SWI call is supported you should first check that r[0] = 0 on return.

Reason code 1 allows you to discover the features the system supports. If bit 2 of the value

returned in r[1] is set then the system supports the use of a choice of audio controller/hardware. You can then use Reason code 2 to read the ID string for the currently selected audio controller.

Sound_SelectDefaultDevice Selects current controller/device

On entry:

r[0] = Pointer to a space/control terminated controller device ID string.

On exit:

r[0] = **0**, or an error block.

This call allows you to select the audio controller and hardware to be used. Both 8-bit and 16-bit sound systems will then use this controller. This SWI provides the same functionality as provided by the extended command syntax, ***Audio <deviceIDstring>**.

Sound_Speaker Equivalent of ***Speaker**

On entry:

r[0] = **0** No Change, just return existing setting

r[0] = **1** Turn speaker off [Equivalent of ***Speaker Off**]

r[0] = **2** Turn speaker on [Equivalent of ***Speaker On**]

On exit:

r[0] = Previous speaker setting.

Sound_Volume Equivalent of ***Volume**

On entry:

r[0] = New volume [range **1** to **127**, or **0** to read volume]

On exit:

r[0] = Previous volume

This volume value uses a log scale with 127 representing a unity scaling. For every 16 you subtract from 127 the linear output will be halved.

SoundControl

The SoundControl module was added to the ROSS in RO 5. Its purpose is to act as a control interface for the sound hardware, thus providing an API that can be kept consistent for different examples of hardware. It provides the ***MixVolume** control discussed in the main part of the ROSS document. In addition it currently provides just three SWIs.

```
0x050000 SoundCtrl_ExamineMixer     0x050001 SoundCtrl_SetMix          0x050002 SoundCtrl_GetMix
```

SoundCtrl_ExamineMixer Examines all mixer settings

On entry:

r[0] = system

r[1] = pointer to a word-aligned buffer

r[2] = buffer size (may be **0**)

On exit: the buffer provided will be filled with a series of 16-byte blocks

with the following format:

bytes

0-1 flags: bit **0** set => fixed

bit `1` = `1` mono, or `0` stereo.

bit `2` = `1` muted by default

`2-3` category

`4-7` minimum gain (dB*16)

`8-11` maximum gain (dB*16)

`12-15` minimum step size/granularity (dB*16)

`r[0]`, `r[1]` preserved

`r[2]` decremented by number of bytes needed for complete description
(ie if 0 on entry, now holds negative of buffer size required)

`r[3]` = individual block size (bytes) - will be a multiple of 4

`r[4]` = number of blocks placed in buffer

Should be called twice, first to determine buffer size required, then to fill in buffer. This is to allow block size to be enlarged in future. Currently, valid block sizes are 4 bytes and 16 bytes. The source code provided with !SoundCheck provides an example in 'C' of how to do this.

`SoundCtrl_SetMix` SWI equivalent to **`*MixVolume`**

On entry:

`r[0]` = system

`r[1]` = category

`r[2]` = channel index

`r[3]` = mute (1 for mute, 0 for unmute)

`r[4]` = gain (16th of a dB)

This allows you to change the settings of active parts of the mixer. It acts in much the same way as *MixVolume as described in the main part of the ROSS documentation.

`SoundCtrl_GetMix` Reads mixer gain setting.

On entry:

`r[0]` = system

`r[1]` = category

`r[2]` = channel index

On exit:

`r[3]` = muted? (1 for yes, 0 for no)

`r[4]` = gain (16ths of a dB)

Acknowledgements

This document is based on a large number of items of advice and information kindly provided by many people. I would like to thank them for their help and knowledge. In particular, I'd like to acknowledge, for simplicity in alphabetical order:

Ben Avison, Colin Granville, Dave Higton, Jeffrey Lee, Chris Martin, Willi Theiss, André Timmermans, and Robin Watts.

and extend my thanks to them and the others. A full list of those who have helped would – to long-term RO users – reveal many of the 'usual suspects'! My apologies to anyone who I've forgotten because I've been swamped by all the details.

Jim Lesurf
8th Feb 2016

Please note that this is numbered Version 0.95 as I know that it still containes errors and has omissions. However I am releasing it for people to comment and provide feedback. I will then produce an improved version and make it available in due course. As and when there are developments I will also include them. I may even get around to correcting the typos! :-)