

---

# RISC OS Audio input and output

---

## Introduction and Overview

This document describes the new Audio interface module – SoundSystem – and how it relates to the existing APIs available in earlier versions of RISC OS.

### Audio capabilities – a brief history

#### Audio output

The first versions of RISC OS had a 1-8 channel audio sound output system that supported an 8-bit  $\mu$ -Law based audio playback.

Later on, a 2-channel (stereo) 16-bit linear PCM (LPCM) was added to the hardware capabilities, with existing audio supported via a mechanism to translate the  $\mu$ -Law audio into the 16-bit linear PCM format.

At a later stage, a mechanism was created that allowed different audio playback applications to share the sound system by merging the audio playback data together.

#### Audio input

No version of RISC OS has ever supported a generic audio input capability. Independent vendors had their own solution, and some published their mechanisms to allow other vendors to use their hardware within audio capture software

#### Audio devices

Within the legacy ARM hardware, audio was part of the video hardware; the VIDC1 and VIDC20 had methods to read data (via the MEMC), and play it back via audio hardware.

With more modern system on a chip (SoC) hardware, audio is separated from the video, and has dedicated registers controlling the flow of the data. These are generally I<sup>2</sup>S-based, which is a simple digital mechanism for transferring audio data to a digital to analogue converter (DAC). They also have the capability of providing audio input, using an analogue to digital converter (ADC).

Vendors of RISC OS hardware have replaced the VIDC interfaces with on-device audio output – although the underlying mechanism has not changed.

#### Audio APIs

All existing APIs follow the same basic method – a buffer is provided for the audio playback software to populate with data that will be output via the hardware interface.

A double-buffer approach is taken, where the software is filling one sound buffer while the hardware is emptying another. When the hardware has emptied the buffer, an interrupt is used to tell the operating system that it needs new data, and the operating system gives the hardware the buffer the software has just filled.

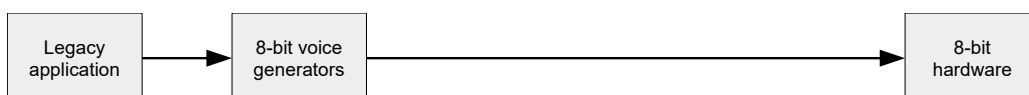
The software then fills the second buffer with data in preparation for the hardware to play.

## 8-bit $\mu$ -Law audio

This API dates back to the original Archimedes computers. The number of sound channels is configured by software that is performing the playback.

The sound system had the capability of providing up to 32 different ‘voices’, or instruments, and these could be scheduled to play with the BASIC SOUND command (or via SWI calls). These voices filled the audio buffers with the data that is needed to play the audio.

Some software intercepted the built-in sound scheduler in order to provide more functionality, or improved performance when playing multiple sounds.



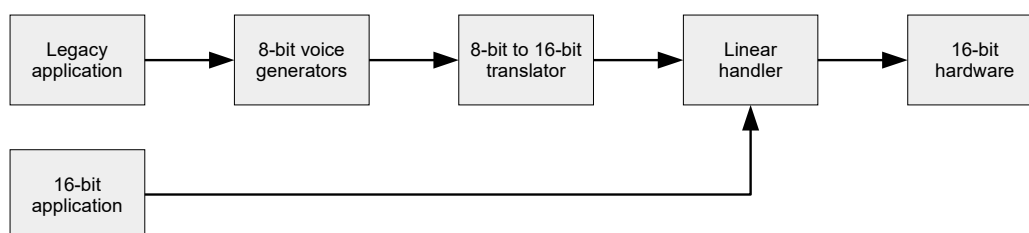
Few programmes use this – !Maestro is about the only one, although the system error “beep” still relies on this mechanism.

## 16-bit single application audio

With the later RiscPC hardware, 16-bit stereo audio capability was added. In order to support this, two changes were made:

- The 8-bit  $\mu$ -Law audio worked as before – voices fill a buffer with the sound data. However, when complete, the data is translated to 16-bit stereo data.
- A 16-bit buffer fill API was provided that applications can provide the data with 16-bit stereo audio data.

Applications using the 16-bit API are informed if there is 8-bit data present in the buffer in order to merge the sound – however, some applications ignore this and overwrite their data all of the time.



These applications implement a *Linear Handler* that is called when more audio data is required. The sound system is triggered by a hardware event that indicates that data is required, and this is passed to the Linear Handler so that it can fill the buffers.

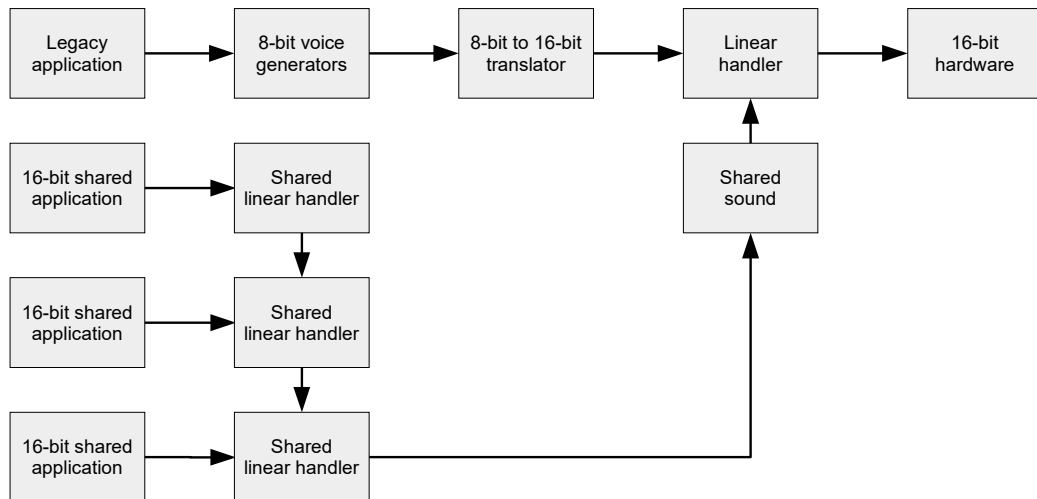
## Shared audio

In order to accommodate multiple audio streams using the hardware at the same time, an API was developed to allow applications to share the audio hardware output.

This has similar concepts to a Linear Handler, except that applications must:

- Cope with other applications using the audio system
- Work at (potentially) a different sample rate to what it is expecting

The Shared Audio system is implemented as a Linear Handler, and can be replaced by a different Linear Handler if an application desires it.



An application that has been written as a standard linear handler, rather than a shared linear handler, can hook into the sound system, and this stops all shared applications from being able to use the sound system.

## SoundSystem API

The SoundSystem API is designed to provide a future-resistant mechanism for audio playback and capture whilst providing a degree of backward compatibility for earlier applications. The level of backward compatibility may change in the future as more applications support the SoundSystem mechanisms.

It has a number of design goals, including:

- Higher audio precision
- More than one output device
- Multi-channel audio devices
- Non-LPCM encoding

### Higher audio precision

When the sound system was changed to allow 16-bit digital audio, this was the quality that was expected – 24-bit and 32-bit was in its infancy (and expensive).

With more modern technology, inexpensive audio devices are available that support 24-bit audio and the higher resolution available cannot be used by RISC OS applications.

The SoundSystem API uses 32-bit audio data paths for LPCM formatted data. This is higher than most audio devices can cope with, but has been chosen for the following reasons:

- It is unlikely to need to go to a higher resolution as the human ear has its limitations
- It is the ARM processor's native word size, so is a programmatically efficient encoding

## More than one output device

With the advent of USB and Bluetooth, computers can often have several devices that are capable of audio playback and capture.

Users would tend to use a single audio device, and the user would configure which of their devices they want to use.

Some users have multiple audio devices, and would switch between them depending on circumstances. For example:

- Using a USB headset when making voice calls
- Using a Bluetooth headset for privacy
- Using a high quality, multi-channel device for music composition

## Multi-channel audio devices

While 2-channel (stereo) is “multi-channel”, multi-channel audio devices refer to devices that have more than 2 channels – for example, surround sound devices, and professional audio devices whose outputs are destined for a mixing desk.

## Non-LPCM encoding

Previous RISC OS sound system implementations required the audio to be in a linear PCM encoded format – however, some devices now support (or even require) non-LPCM formatted data, which offloads the conversion of compressed audio files from the CPU to the audio device.

## Procedural interfaces

The SoundSystem API defines four interfaces that applications and hardware developers can use to provide and consume audio resources.

### Discovery interfaces

This interface is used by both applications and hardware.

Applications use the discovery interface to find out about the audio capabilities of the system; hardware uses the discovery interface to inform SoundSystem that audio hardware is available, and its capabilities.

### Control interfaces

This interface is used by both applications and hardware.

Applications use the control interface to configure aspects of the audio system (such as mixer volumes and input selectors); hardware uses the interface to receive configuration requests from applications.

### Buffer manager

This interface is used by both applications and hardware.

Applications use the buffer manager to provide and retrieve data for audio playback and capture; hardware uses the buffer manager to request data from the applications for playback, and inform applications that data is available for capture.

## Codec interface

The codec interface is primarily used by applications, although it is envisaged that hardware may use it at a later date.

The codec interface provides a method of converting one audio format into another audio format – for example, a frame of MP3 data into LPCM data. It can even be used for more basic LPCM format conversions (such as sample rate conversion, audio mixing or channel selection),

Hardware may use this interface to convert LPCM data into the hardware-native format.

## Audio format descriptors

With many different audio formats available, a method to describe the format is required. The simplest format – and the one that is currently supported by the sound system – is LPCM. Any audio device driver should support 32-bit 2-channel LPCM format in order to maximise support for applications, but other formats can be supported, if the applications support them.

The sample rate is not considered to be part of the format descriptor, although it is part of the overall device configuration.

## Format lists

When registering, a device informs SoundSystem what formats it is capable of supporting. This is done via a pointer to a list of formats, each notionally 1 word in length.

If the word in the list is zero, then this denotes the end of the list.

If the word in the list has either of the bottom two bits set, then it is a simple format that consists of a single word.

If the word in the list has both of the bottom two bits clear, then it is a pointer (or an offset) to a value held in memory for an extended format.

## Simple formats

The simple format descriptor consists of single 32-bit word, with one, or both, of the bottom two bits set.

The format of this word is &nnnnnnxx where “nnnnnn” are bits used to describe the format, and &xx is the format.

The values for &xx are as follows:

01	2-channel 32-bit LPCM format
02	2-channel 24-bit LPCM format
03	2-channel 16-bit LPCM format
05	2-channel 16-bit SoundDMA LPCM format
06	$n$ -channel, $m$ -bit LPCM format
07	$n$ -channel, 8-bit $\mu$ -Law format
Others	Reserved

Note that the format descriptor of &FFFFFFFF is reserved, and used to indicate the end of a list of formats.

### 2-channel 32-bit LPCM format (format &01)

This format should be supported by all SoundSystem devices, as this is the native format that applications must provide.

The “nnnnnn” bits are reserved and set to zero.

The format of the data itself consists of two signed 32-bit integers per sample, with the first integer being the left channel’s data, and the second integer being the right channel’s data.

Each integer is stored in little endian format, so a sample value of &12345678 is stored as &78, &56, &34, &12.

Any buffer **must** be word aligned, and have a length of a multiple of words.

### 2-channel 24-bit LPCM format (format &02)

This format can only be used by applications registering exclusive access to the audio device where the audio device supports it.

The “nnnnnn” bits are reserved and set to zero.

The format of the data itself consists of a pair of 3-byte signed integers per sample, with the first of the three bytes giving the left channel’s data, and the last of the three bytes giving the right channel’s data.

Each 3-byte value is formatted in little endian format, which means a sample of &123456 is encoded as the bytes &56, &34, &12.

This format does not require the buffers to be word aligned.

### 2-channel 16-bit LPCM format (format &03)

This format can only be used by applications registering exclusive access to the audio device where the audio device supports it.

The “nnnnnn” bits are reserved and set to zero.

The format of the data itself consists of a pair of 2-byte signed integers per sample, with the first of the two bytes giving the left channel’s data, and the last of the two bytes giving the right channel’s data.

Each integer is stored in little endian format, which means a sample of &1234 (left) paired with &5678 (right) is encoded as the byte sequence &34, &12, &78, &56.

This format requires the buffers are word aligned.

### 2-channel 16-bit SoundDMA format (format &05)

This format can only be used by applications registering exclusive access to the audio device where the audio device supports it. It is also the format that is supported by SharedSound and SoundDMA interfaces.

The “nnnnnn” bits are reserved and set to zero.

The format of the data itself consists of a pair of 2-byte signed integers per sample, with the first of the two bytes giving the right channel’s data, and the last of the two bytes giving the left channel’s data.

Each integer is stored in little endian format, which means a sample of &1234 (left) paired with &5678 (right) is encoded as the byte sequence &78, &56, &34, &12.

This format requires the buffers are word aligned.

### N-channel, m-bit LPCM format (format &06)

This format can only be used by applications registering exclusive access to the audio device where the audio device supports it.

The top-24 bits are split into the top 16-bits giving a bit field indicating the channels present, the next 4 bits the number of channels, and the last 2 bits the number of bytes per sample:

Bit(s)	Description
0 – 7	&06 (format descriptor)
8 – 11	Number of bytes per sample 0 = reserved, 1 = 8-bit, 2 = 16-bit, 3 = 24-bit, 4 = 32-bit, 5 – 15 = reserved
12 – 15	Number of channels (1-15, 0 = reserved)
16	If set, left front data is present
17	If set, right front data is present
18	If set, centre front data is present
19	If set, low frequency enhancement data is present
20	If set, left surround data is present
21	If set, right surround data is present
22	If set, left of centre data is present
23	If set, right of centre data is present
24	If set, surround data is present
25	If set, side left data is present
26	If set, side right data is present
27	If set, top data is present
28 – 31	Reserved (0)

Note that the number of bits set in 16 – 31 does not need to equal the number of channels – although it must not exceed the number of channels. This permits data for spacial locations that are not defined in the above list to be present. The interpretation of such data depends on the audio device.

The data is presented in the order of the bits set in 16 – 31, with any non-spatially aware data after the spatially aware data. This means that if there are 5 channels present, with bits 16, 17 and 18 set, then the first data will be the left front channel, the second data will be the right front channel, the third data will be the centre front data. The remaining two pieces of data are for the non-spatially aware channels.

The buffer that holds the data does not need to be word aligned for 8-bit and 24-bit formats, but must be for 16-bit and 32-bit formats.

### 8-bit $\mu$ -Law format (format &07)

This format is only aimed at backward compatibility with earlier RISC OS systems.

Bits 8 – 11 of the format identifier are the number of channels (1, 2, 4 and 8 – the remainder are reserved). The remaining bits (12 – 31) are reserved.

Stereo positioning of the data is not included in the channel data, but is included as part of the 8-bit  $\mu$ -Law configuration.

Each channel is presented as a single byte, as per the VIDC1 format (bit 0 = sign, bits 2 – 7 = logarithmic value).

The buffer that holds the data does not need to be word aligned.

### Other formats

Other format descriptors are yet to be defined, but are anticipated to be non-LPCM data, such as MP3, or Bluetooth's SBC.

These formats tend to be compressed, and will only be useable when a device is being used in exclusive access – merging with existing data will not be possible.

For these formats, the data is likely to consist of frames, and buffers will be large enough to contain one or more whole frames. The amount of data may be variable.

## Audio handlers

In order for applications to send and receive data to an audio device, audio buffer handlers are registered by the applications with SoundSystem.

Data is transferred to the audio device with a playback handler, and received from the device with a recording handler.

Applications can either register for shared access to the audio devices, or they can register for exclusive access.

### Shared access

Shared access allow multiple applications to share an audio device, by mixing the audio playback streams together. Audio capture data is passed to all applications interested in recording audio.

All applications implementing shared audio playback must support 2-channel (or higher), 32-bit linear PCM formatted data at any sample rate (that can change at any point) and mixing the sounds from other applications.

Audio devices must also support this format, converting the data to or from the device's native format where necessary.

### Exclusive access

Exclusive access precludes the ability for applications to share an audio device – but an application can reconfigure the audio device to support different audio formats.



As there is no need for applications to share the audio device, mixing and variable sample rates do not need to be considered for the audio playback handler.

If an application already has exclusive access to an audio device, and another application requests exclusive access, then the new application will have exclusive access. When the new application relinquishes its exclusive access, then the previous application will regain the access.

When the last exclusive application relinquishes its access, then the shared system will be given control back again.

If a shared application joins the shared system while exclusive access is in use by another application, then it will lie dormant until all exclusive applications have relinquished their access.

In order to maintain the exclusive access, when an application registers for exclusive access on a device, it is allocated a unique number. In order to perform operations on the device on an exclusive basis, the application must pass this through to the relevant SWI calls. This also prevents one application from overriding another application's settings, which could cause undefined results.

## Playback handler code

The audio playback handler's function is to provide data for the sound output. The code for all formats is essentially the same.

### On entry

R0 = pointer to buffer to fill  
 R1 = length of buffer (in bytes)  
 R2 = bit field (undefined for exclusive access)

Bit	Description when set
0	Buffer contains existing data to be merged
1	Buffer will be merged down to a mono-channel output
2 – 23	Reserved
24 – 31	Volume (0 = silent, 255 = maximum)

R12 = value given when the playback handler was registered

### On exit

R0 = pointer to byte after buffer was filled  
 Other registers must be preserved

### Processor mode

Processor is in SVC or interrupt mode

### Use

The current format and sample rate are not passed into this handler, as they can be queried via a SWI call, or via the SoundSystem Service Call.

Note that if bit 0 of R2 is zero on entry, then the whole buffer must be filled in, even if there is not enough data in the application to fill it. In this case, the application must fill the rest with zeros.

If bit 0 of R2 is set on entry, then the application can stop at its last sample, and does not need to fill in the rest.

## Recording handler code

The audio recording handler is called when there is data received from the audio device. The code for all formats is essentially the same.

### On entry

R0 = pointer to buffer containing data

R1 = length of data in buffer (in bytes)

R12 = value given when the recording handler was registered

### On exit

Registers must be preserved

### Processor mode

Processor is in SVC or interrupt mode

### Use

The current format and sample rate are not passed into this handler, as they can be queried via a SWI call, or via the SoundSystem Service Call.

## Codec interface

Codecs can be provided by relocatable modules that have registered the source and destination format with SoundSystem.

The codec interface is performed exclusively via Service Calls. Modules that implement codecs need to handle the Service Calls and provide pointers to code accordingly.

## Codec configuration handler

The codec initialisation handler is called to prepare the codec for data it needs to convert, or to reconfigure itself for a different sample rate.

### On entry

R0 = reason code

Other registers depend on reason code

### On exit

R0 preserved

Other registers depend on reason code

**Use**

This call is used to configure, reconfigure and terminate the codec.

**Codec configuration 0**

Get the size of the contextual data.

**On entry**

R1 = pointer to codec configuration

- +0 Bit field for codec configuration
- +4 Source format descriptor
- +8 Source sample rate (in Hz, multiplied by 1024)
- +12 Destination format descriptor
- +16 Destination sample rate (in Hz, multiplied by 1024)

R2 = length of the destination buffer (in bytes), or 0 for undefined

The bit field is as follows:

Bit	Value when set
0 – 3	Quality indicator (0 = low quality, 15 = highest quality)
4	Codec will need to merge with existing data
5	Source sample rate is the maximum sample rate (0 is average sample rate)
6	Destination sample rate is the maximum sample rate (0 is the average sample rate)
9 – 23	Reserved (0)
24 – 31	Volume (0 = silent, 255 = maximum)

**On exit**

R1 = the length of data required for the codec to store its contextual data

**Use**

This call is used to get the length of data the codec needs for its contextual data. The codec will return the number of bytes that the codec user will need to allocate in order for it to function.

**Codec configuration 1**

Configures and initialises the codec.

**On entry**

- R1 = pointer to codec configuration (as per Codec configuration 0)
- R2 = length of the destination buffer (in bytes), or 0 if undefined
- R3 = pointer to codec contextual data (if any was needed)

**On exit**

R1 = pointer to codec conversion code

**Use**

This is used to initialise a codec for the given source and destination configuration

A codec may offer more than one ‘quality’ for its conversion, for example a codec that simply converts from one sample rate to another may have a simple conversion where samples are repeated or skipped, and another where some filtering is applied pre- or post- conversion to reduce aliasing artefacts that can occur with the simplistic approach. The quality would normally increase the amount of CPU activity required to perform the conversion.

Note that for non-LPCM formatted data, the sample rate may be variable – in which case the sample rate will be the average or maximum sample rate depending on bits 5 and 6 of the bit field.

## Codec configuration 2

Reconfigures the codec.

### On entry

Registers as per Codec configuration 1

### On exit

R0 = 0 for codec needs to be reinitialised from the beginning

= 1 for codec has been reconfigured

Other values reserved

R1 = pointer to codec conversion code

### Use

This is used by the codec user to indicate that some of the parameters have changed, and the codec needs to work differently.

If R0 is 0 on exit, then the codec is no longer relevant at all, and the codec needs to be destroyed, and a new codec created (with memory allocated).

Note that the codec conversion code pointer may change when the codec is reconfigured.

## Codec conversion handler

The codec conversion handler is called to perform the conversion from one format to another format.

### On entry

R0 = the pointer to the source buffer

R1 = the length of the data in the source buffer

R2 = the pointer to the destination buffer

R3 = the length available in the destination buffer

R12 = the pointer for the codec contextual data

### On exit

R0 = the pointer to the byte after the source buffer that has been used

R1 is bit field

R2 = the pointer to the byte after the destination data that has been filled

R3 can be corrupted

Other registers preserved

## Use

The codec conversion handler takes data passed into it, and writes it to the destination buffer. The bit field is as follows:

Bit	Value when set
0	Not all of the source data was used
1	Not all of the destination data was filled
2 – 31	Reserved (0)

If the codec does not use all of the source buffer, then the caller must add new data to the end of the previous data (or copy the previous data and then append it) ready for the next call. R1 will have bit 0 set on return.

If the codec requires the destination buffer to be completely filled, then the caller must add more data to the source and call the codec again until bit 1 of R1 is set on return.

If there is not enough data in the source or the destination for the codec to function, then R0 and R2 will be preserved. The caller will need to increase the amount of source data, and/or the output buffer size.

The contextual data is used to maintain information required by the codec between calls.

## Codec finalisation handler

This is an optional handler that allows a codec to tidy up any data it may have held outside its contextual data.

### On entry

R12 is the pointer to the codec contextual data

### On exit

Registers preserved

### Processor mode

Processor is in SVC or interrupt mode

## Use

Not all codecs will require special finalisation code. Ordinarily, all that needs to happen is the codec is not called any more – and SoundSystem simply deallocates the memory that had been assigned to its contextual data.

Codecs should not store anything that is not in its contextual block, but if they do, then this handler is used to inform them that this instance is about to be terminated.

## Hardware interface

The hardware interface is used by hardware drivers to interact with SoundSystem. Hardware drivers need to ask SoundSystem to get data to play back, and tell SoundSystem that there is data that has been recorded.

During registration, a hardware driver will pass in a control block, with the data at offset +12 being a pointer (or an offset from the module start) being “control code” that is called by SoundSystem to provide the driver with configuration details, and the data at offset +16 being a pointer to the driver's private word.

Hardware interfaces must support 32-bit 2-channel audio if they want to support the shared access system. Otherwise, they can only be used by applications that support their audio formats.

### Control code

This performs various functions depending on the reason code passed in to R0. Unused and unsupported reason codes are to be ignored.

#### Control code 0

Sets up the addresses that the hardware driver needs to call in order to request or provide sample data

##### On entry

R0 = 0 (reason code)  
R1 = pointer to playback data request  
R2 = pointer to recording data provision  
R3 = value that must be passed into R12 on entry to either routine above

##### On exit

Registers preserved

##### Use

The hardware driver is provided with the addresses that it must call in order to request the playback data, or to provide the recorded data.

The code that is called is essentially the same for both playback and recording, and is detailed later.

Note that both pointers will be provided even if the device does not support playback or recording.

#### Control code 1

Sets or queries the sample format for use for playback and recording.

##### On entry

R0 = 1 (reason code)

R1 = new sample format descriptor (or -1 to query)

**On exit**

R1 = previous sample format descriptor

**Use**

This call is used by SoundSystem to inform the hardware driver that the sample format needs to be changed.

**Control code 2**

Sets or queries the sample rate for playback and recording.

**On entry**

R0 = 2 (reason code)

R1 = new sample rate or -1 to query (in Hz, multiplied by 1024)

**On exit**

R1 = previous sample rate (in Hz, multiplied by 1024)

**Use**

This call is used by SoundSystem to get or set the sample rate for use in playback or recording.

**Control code 3**

Sets or queries the overall output volume or output mixer volume level.

**On entry**

R0 = 3 (reason code)

R1 = mixer channel number (0 for overall)

R2 = new output volume level (0 = silent, 255 = maximum volume, -1 to query)

**On exit**

R2 = previous output volume level

**Use**

This call is used by SoundSystem to get or set the overall output volume level, or the output mixer level for the given channel number.

If R1 is 0 on entry, then the overall output volume level is being set (or queried).

Otherwise, R1 is set to the mixer channel number. The channels are defined by the hardware capability bit flags, with mixer channel 1 indicating the channel with the lowest bit set in bits 16 – 27, mixer channel 2 indicating the channel with the second lowest bit and so forth.

**Control code 4**

Sets or queries the overall input volume or input mixer volume level.

### On entry

R0 = 4 (reason code)  
R1 = mixer channel number (or 0 for overall)  
R2 = new input volume level (0 = silent, 255 = maximum volume, -1 to query)

### On exit

R2 = previous input volume level

### Use

This call is used by SoundSystem to get or set the overall input volume level, or the input mixer level for the given channel number.

If R1 is 0 on entry, then the overall input volume level is being set (or queried).

Otherwise, R1 is set to the input mixer channel number. The channels are defined by the hardware capability bit flags, with mixer 1 indicating the channel with the lowest bit set in bits 8 – 11, mixer 2 indicating the channel with the second lowest bit and so forth.

## Control code 5

Enables or disables the playback and / or recording

### On entry

R0 = 5 (reason code)  
R1 = bit field

Bit	Value when set
0	Enable playback
1	Enable recording
2 – 31	Reserved (0)

### On exit

Registers preserved

### Use

This call is used by SoundSystem to tell the hardware driver if it needs to perform a request for playback data, or to provide recording data.

For example, if there are no playback or recording handlers attached to a device, then the device does not need to request or provide any data.

## Control code 6

Request the list of format descriptors that the device supports

### On entry

R0 = 6 (reason code)  
R2 = pointer to block of sample format descriptors to fill in, or 0 to query length  
R3 = length of block of data (in bytes)



**On exit**

The block pointed to by R2 is filled with the sample format descriptors, if non-zero on entry.

R3 is set to the number of format descriptors available.

**Use**

See `SoundSystem_GetDeviceFormats` for details on the returned data.

**Control code 7**

Request the list of sample rates for the given format indicator

**On entry**

R0 = 7 (reason code)

R1 = sample format descriptor

R2 = pointer to block of sample rates to fill in, or 0 to query length

R3 = length of block of data (in bytes)

**On exit**

The block pointed to by R2 is filled with the available sample rates (in Hz, multiplied by 1024), if non-zero on entry.

R3 is set to the number of sample rates available for the given sample format.

**Use**

This is used to query the list of sample rates that a device can support for a given sample format.

Some devices may have a bandwidth threshold which means that a higher number of channels or bits per channel may reduce the maximum sample rate supported by the device.

If all sample rates are supported by all sample formats, then the same data will be returned.

See `SoundSystem_GetDeviceFormatSampleRates` for details on the returned data (including the divisor-based mechanism).

**Control code 8**

Gets or sets the buffer size for playback and recording

**On entry**

R0 = 8 (reason code)

R1 = Requested buffer size in bytes (must be a multiple of 4 \* channel count for 32-bit LPCM samples), or:

0 = Query the current buffer size

-1 = Query the device optimal or default buffer size

**On exit**

R1 = Buffer size (in bytes)

### Use

This is used to get or set the current buffer size in bytes. Setting the buffer size can only occur if exclusive access has been granted on the device.

Note that the buffer size for non-LPCM formats relates the amount of data that can be potentially used – a variable bit-rate format may result in less data being passed.

If the buffer size is outside its bounds, then the device driver can override the requested buffer size with values that are in the range. For example, if the requested buffer size is larger than the hardware buffers.

## Playback data request

The hardware driver calls the playback data request to get data.

### On entry

R0 = pointer to playback buffer to fill in

R1 = length of playback buffer (in bytes)

R12 = value of R3 when given the playback request code pointer in Control code 0.

### On exit

R0 = pointer to byte after filled data

Other registers preserved

### Use

When the hardware device needs some data, the driver makes a call to the playback data request passing in the pointer to a buffer in R0, and the length of the buffer in R1.

SoundSystem then goes through the list of audio playback handlers for them to fill the data, merging with the previous data if necessary.

## LPCM formats

It is guaranteed that the whole buffer will be filled in by the playback handler(s), so the return value of R0 can be ignored by the hardware driver.

## Non-LPCM formats

Not all of the data may be filled in when dealing with non-LPCM formats, and the value of R0 allows the hardware driver to determine how much of the buffer has actually been filled.

If the hardware driver requires more data, then it can call the playback data request code repeatedly until it has enough data – although if the value of R0 does not change, then it must assume that there is no more data ready, and cope with this accordingly.

## Recording data provision

The hardware driver informs SoundSystem that there is data that has been captured.

### On entry

R0 = pointer to buffer containing data

R1 = length of buffer (in bytes)

R12 = value of R3 when given the recording provision code pointer in Control code 0

### **On exit**

Registers preserved

### **Use**

When the hardware driver has recorded some data, it calls the recording data provision code to inform SoundSystem that new data is present, with R0 pointing to the start of the data, and R1 being the length of the data.

Any applications that are receiving this data must be able to handle all of this data, and when the call returns, the hardware driver can assume that the data is no longer required.

This is true for both LPCM and non-LPCM formats.

## Service calls

### Service\_SoundSystemInitialised (Service Call &81140)

The SoundSystem module is initialising, and applications and hardware devices can start to register with it.

#### On entry

R0 = 0 for initialisation; 1 for finalisation; all other values reserved (these must be ignored)

#### On exit

All registers preserved

#### Use

Hardware drivers can use this service call to register their hardware devices with SoundSystem so that applications can start to use them.

Finalisation is called so that hardware devices can stop providing their interrupts for audio playback and recording – hardware drivers and codec providers do not need to deregister the services they are supporting.

This service call must not be claimed.

## Service\_SoundSystemHardwareAdded (Service Call &81141)

A new hardware has been registered with SoundSystem.

### **On entry**

R0 is the pointer to the hardware device identifier that has been added

### **On exit**

All registers preserved

### **Use**

This service call is issued when a new hardware device has been registered with SoundSystem\_RegisterDevice.

This service call must not be claimed.

## Service\_SoundSystemHardwareRemoved (Service Call &81142)

Hardware has been removed from the system.

### **On entry**

R0 is the hardware device identifier that has been removed

### **On exit**

All registers preserved

### **Use**

This service call is issued when hardware has been removed with SoundSystem\_DeregisterDevice.

All playback and recording handlers associated with the given device will automatically be deregistered.

This service call must not be claimed.

## Service\_SoundSystemExclusiveAccessChanged (Service Call &81143)

Exclusive access to a hardware device has been changed.

### **On entry**

R0 = the hardware device identifier that exclusive access has been registered

R2 = the new identifier (nominally WIMP handle) for the exclusive access (as passed into SoundSystem\_RegisterExclusiveAccess), or 0 if shared access is returned

### **On exit**

All registers preserved

### **Use**

This service call is issued when an application has requested exclusive access to an audio device, and allows applications to indicate that they no longer need to play their audio.

This call is also issued when an application deregisters exclusive access, and control is passed back to the previous application.

If an application deregisters exclusive access, and there were no previous exclusive registrations, then -1 is returned to indicate that shared access is returned.

## Service\_SoundSystemRequestCodec (Service Call &81144)

Used by applications and/or hardware drivers to request a codec.

### On entry

R0 = source format descriptor  
 R2 = source sample rate (in Hz, multiplied by 1024)  
 R3 = destination format descriptor  
 R4 = destination sample rate (in Hz, multiplied by 1024)  
 R5 = bit flags

Bit	Value when set
0 – 3	Quality indicator (0 = low quality, 15 = highest quality)
4	Codec will need to merge with existing data
5	Source sample rate is the maximum sample rate (0 is average sample rate)
6	Destination sample rate is the maximum sample rate (0 is the average sample rate)
7	Codec must support volume levels
8 – 31	Reserved (0)

### On exit

If the codec is supported, then this service call is claimed, with the following registers set:

R0 = bit flags

Bit	Value when set
0 – 1	Reserved (0)
2	Codec can support changing of sample rates without reconfiguring
3 – 31	Reserved (0)

R2 = pointer to codec provider name  
 R3 = pointer to codec initialisation code  
 R4 = pointer to codec finalisation code (zero if none)  
 R5 = size of contextual data needed

### Use

This service call is used by an application or hardware device to request code to convert from one format to another format, and/or one sample rate to another sample rate. For example:

- Applications can use this to convert a format that is present on a hard disk to another format before passing it through to SoundSystem.
- Hardware devices can use this to convert the standard 32-bit, 2-channel LPCM format to a format the device itself supports

If the codec is needed, then the application needs to allocate the amount of memory as indicated by R5 (in an area of memory that will be available when the codec conversion code is called).

When the codec is no longer needed, the finalisation code needs to be called (if present)



## Service\_SoundSystemCodecRemoved (Service Call &81145)

A codec has been removed from SoundSystem.

### **On entry**

R0 = the source format descriptor

R2 = the destination format descriptor

R3 = the pointer to the codec provider name

### **On exit**

All registers preserved

### **Use**

This service call is issued when a codec is deregistered. Applications should check the values passed in – if R0 is 0, then all codecs supported by the provider are no longer valid; otherwise the given source and destination formats are no longer supported.

If an application finds that its codec is no longer supported, then it must stop using that codec, and either select a different codec, or stop any audio playback (or ignore incoming audio).

Applications do not need to deregister their codecs upon receipt of this service call.

## Service\_SoundSystemSampleRateChanged (Service Call &81146)

Informs applications that the shared sample rate has been changed.

### **On entry**

R0 = old sample rate (in Hz, multiplied by 1024)  
R2 = new sample rate (in Hz, multiplied by 1024)  
R3 = device identifier

### **On exit**

All registers preserved

### **Use**

This service call is issued when an application has requested the change of the sample rate of a shared access hardware device. The sample rate affects both playback and recording, so applications should adjust their playback and recording to accommodate the change.

Applications needing a codec to change the sample rate from one to another may need to change their codec.

This service call is not issued when exclusive access has been given on the hardware device, as only the application with exclusive access can change the sample rate.

## SWI calls

### SoundSystem\_RegisterDevice (SWI &5A080)

Registers a hardware device with SoundSystem.

#### On entry

R0 = pointer to user-orientated device name

R1 = pointer to device identifier (zero-terminated string)

R2 = pointer to device information block

+0	Bit flags of capabilities
+4	Maximum number of channels
+8	Pointer or offset to control code
+12	Value to pass in as R12 to the device's Control Code

#### On exit

Registers preserved

#### Interrupts

Interrupt status is unaltered

#### Processor mode

Processor is in SVC mode

#### Re-entrancy

SWI is not re-entrant

#### Use

This call is used by hardware to register a device with SoundSystem, providing it with a user-orientated device name, and a device identifier (which must be unique across a system).

Applications should use the user-orientated device name when providing a list of options to the user, but the device identifier when registering to transfer data with the audio device.

The bit flags are:

Bit	Value when set
0	Device supports playback
1	Device supports recording
2	Device supports LPCM playback / recording
3	Device supports non-LPCM playback / recording
4 – 7	Reserved (0)
8	Device supports microphone input
9	Device supports line input

- 10 Device supports digital input
- 11 – 15 Reserved (0)
- 16 Left front mixer is present
- 17 Right front mixer is present
- 18 Centre front mixer is present
- 19 Low frequency enhancement mixer is present
- 20 Left surround mixer is present
- 21 Right surround mixer is present
- 22 Left of centre mixer is present
- 23 Right of centre mixer is present
- 24 Surround mixer is present
- 25 Side left mixer is present
- 26 Side right mixer is present
- 27 Top mixer is present
- 28 Reserved (0)
- 29 Microphone input mixer is present
- 30 Line input mixer is present
- 31 Digital input mixer is present

## SoundSystem\_DeregisterDevice (SWI &5A081)

Deregisters a device that has been registered with SoundSystem.

### **On entry**

R1 = pointer to device identifier (zero-terminated string)

### **On exit**

Registers preserved

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is not re-entrant

### **Use**

This call is used to de-register a device with SoundSystem.

## SoundSystem\_EnumerateDevices (SWI &5A082)

This call is used to find out the list of audio devices available on the system.

### On entry

R1 = pointer to previous device identifier (or 0 to start)

### On exit

R1 = pointer to device identifier (or 0 for end of list)

Other registers preserved

### Interrupts

Interrupt status is unaltered

### Processor mode

Processor is in SVC mode

### Re-entrancy

SWI is re-entrant

### Use

This call is used by applications to determine the available audio devices that are available on the system.

The first call an application would make starts with R1 set to 0. Upon return, R1 is updated to point to the new device identifier name. A subsequent call to SoundSystem\_GetDeviceInformation can be made to read the device's user-orientated name and capabilities

The application repeatedly calls this SWI until R1 is equal to zero.

If R1 is zero on entry and exit, then there are no audio devices present.

Note that the value returned R1 is the copy of the value as passed into SoundSystem\_RegisterDevice.

## SoundSystem\_GetDeviceInformation (SWI &5A083)

This call is used to find out the information about a particular device.

### **On entry**

R1 = pointer to previous device identifier (or 0 to start)

### **On exit**

R0 = pointer to user-orientated device name

R1 preserved

R2 = pointer to device information block (as per SoundSystem\_RegisterDevice)

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is re-entrant

### **Use**

This call is used to get the device name and device information block for a particular device identifier. It can be called with SoundSystem\_EnumerateDevices to get a list of all of the devices and their capabilities.

Note that the values returned in R0 and R2 are copies of the values as passed into SoundSystem\_RegisterDevice.

## SoundSystem\_GetDeviceFormats (SWI &5A084)

Used to iterate over the list of device formats

### On entry

R0 = pointer to block of memory to hold format descriptors, or 0 to query length

R1 = device identifier

R2 = length of data available in the block (if R0 is non-zero on entry)

### On exit

R2 = length needed to hold all of the available sample rates (if R0 was zero on entry), or length filled in (if R0 was non-zero on entry)

Other registers preserved

The data block passed into R0 is filled in with the device formats supported by the device if R0 was non-zero on entry

### Use

This call is used to iterate over the list of formats the device can support.

An application will typically call this with R0 set to zero to get the amount of memory that is needed to get the list of all of the available formats, as returned in R2.

With the memory allocated, the application can then make a second call to fill in the memory block with the list of available formats.

Note that if there is not enough memory in the block, then the memory will still be filled in – up to the number of available formats.



## SoundSystem\_GetDeviceFormatSampleRates (SWI &5A085)

Used to get the list of sample rates offered by a device for a particular format

### On entry

R0 = pointer to block of data to hold sample rates, or 0 to query number of sample rates

R1 = device identifier

R2 = sample format indicator

R3 = length of the block (if R0 is non-zero on entry)

### On exit

R3 = length needed to hold all of the available sample rates (if R0 was zero on entry), or length filled in (if R0 was non-zero on entry)

Other registers preserved

The data block passed into R0 is filled in with the available of sample rates for the given format indicator if R0 was non-zero on entry. Each sample rate is the rate given in Hz, multiplied by 1024.

Note that some devices can support an arbitrary divisor of a particular clock frequency or frequencies.

### Use

An application can use this to query the list of sample rates available on a particular device for a given sample format indicator.

The application should initially call this with the required sample format indicator, and R0 set to zero. This will return with R3 set to the length of data required to fill in all of the information.

After allocating the required amount of memory, the application should then call this again with R0 pointing to the memory block, and R3 set to the size of the block.

If an application wants to use a fixed block, then it can simply pass R0 as pointing to the memory block, and R3 the size of the block.

The block will be filled in with as many sample rates as it can up to the available space.

### Divisor-based sample rates

Some devices have the capability of generating a sample rate based on a particular clock frequency (or frequencies) with a divisor. As this can result in a large number of sample rates if there are a large number of divisors available, an alternative method is provided to support these devices.

The same mechanism to determine the size is used as before (R0 is set to zero on entry, and R3 is set to the data size on exit).

If the first sample rate is zero, then the divisor mechanism is active – the remaining data consists of a number of blocks consisting of 3 words:

- +0 Clock frequency given in Hz (note: not multiplied by 1024).
- +4 Minimum divisor value.
- +8 Maximum divisor value.

This is repeated for each clock frequency.

For example, if a device supports two clock frequencies of 32MHz and 48MHz, with a divisor of between 128 and 1023, then the following block will be returned (R3 will be 28):

R3 = 28 (7 words of data)

- +0 0 (divisor mechanism)
- +4 32000000 (32MHz)
- +8 128 (minimum divisor)
- +12 1023 (maximum divisor)
- +16 48000000 (48MHz)
- +20 128 (minimum divisor)
- +24 1023 (maximum divisor)

The available sample rates are  $32,000,000 / 1023$  to  $32,000,000 / 128$  (31.280KHz to 250.000KHz), and  $48,000,000 / 1023$  to  $48,000,000 / 128$  (46.920KHz to 375.000KHz).

A sample rate of 44.1KHz can be specified as a clock speed of 32MHz, and a divisor of 726 (this would actually give a sample rate of 44.077KHz). 48KHz can be represented as a clock speed of 48MHz, and a divisor of 1,000.

## SoundSystem\_RegisterExclusiveAccess (SWI &5A086)

This call is made by an application to register exclusive access to the audio device.

### **On entry**

R1 = device identifier

### **On exit**

R0 = unique identifier for the exclusive access

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is re-entrant

### **Use**

The value returned by R0 is a unique number that is allocated for this particular exclusive access request. Applications must use it when de-registering their exclusive access.

If more than one application registers for exclusive access to a device, the most recently registered application will have the access. When the application deregisters, the previous application will have exclusive access again.

## SoundSystem\_DeregisterExclusiveAccess (SWI &&5A087)

This call is made by an application to deregister exclusive access to the audio device.

### **On entry**

R0 = unique identifier for the exclusive access (as given by  
SoundSystem\_RegisterExclusiveAccess)

### **On exit**

Registers preserved

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is re-entrant

### **Use**

Applications will call this SWI when deregistering their exclusive access – for example, at application termination. Control will pass back to the previous application that had exclusive access – unless there are no exclusive applications remaining, in which case control passes back to the shared sound system.

Note that an application can deregister itself when another application has exclusive access, and it is removed from the list of applications having exclusive access to that device.

## SoundSystem\_SetFormat (SWI &5A088)

This call is used to set the audio playback and recording format.

### On entry

R0 = basic format identifier, or -1 if querying the current playback format  
R1 = pointer to device identifier (zero terminated string)  
R2 = pointer to additional format information (0 if none provided)  
R3 = unique identifier for exclusive access

### On exit

R0 = previous basic format identifier  
R1 preserved  
R2 = pointer to previous additional format information (0 if none)

### Interrupts

Interrupt status is unaltered

### Processor mode

Processor is in SVC mode

### Re-entrancy

SWI is re-entrant

### Use

This call is used to set or query the audio playback and recording format.

If R0 = -1 on entry, then the other registers are ignored on entry, and the return values refer to the current audio playback format. Otherwise, R0 and R1 specify the required audio playback format.

This call can only be used when an exclusive access has been claimed on an audio device.

## SoundSystem\_SetSampleRate (SWI &5A089)

This call is used to get or set the current sample rate.

### On entry

R0 = sample rate (in Hertz multiplied by 1024), or -1 to query

R1 = pointer to device identifier (zero terminated string)

R3 = unique identifier for the exclusive access (or 0 for using shared access)

### On exit

R0 = previous sample rate

R1, R3 preserved

### Interrupts

Interrupt status is unaltered

### Processor mode

Processor is in SVC mode

### Re-entrancy

SWI is re-entrant

### Use

If R0 = -1 on entry, then R0 is set to the current sample rate (in Hertz multiplied by 1024).

Otherwise, R0 is the new sample rate, and is set to the previous sample rate on exit.

Note that this affects both playback and recording sample rate.

If an application is getting or setting the sample rate when using shared access, this will return the sample rate of the shared access system; for exclusive access, it is the sample rate for that exclusive access.

## SoundSystem\_AttachPlaybackHandler (SWI &5A08A)

This call is used to attach a playback handler.

### On entry

R0 = pointer to playback handler code  
R1 = pointer to device identifier (zero terminated)  
R2 = bit flags:

Bit	Value when set
0 – 31	Reserved (0)

R3 = the unique access identifier, or 0 if registering as a shared playback handler  
R4 = value of R12 passed into playback handler code

### On exit

Registers preserved

### Interrupts

Interrupt status is unaltered

### Processor mode

Processor is in SVC mode

### Re-entrancy

SWI is re-entrant

### Use

This call is used by an application to register a playback handler to provide audio to a hardware device.

# SoundSystem\_AttachRecordingHandler

(SWI &5A08B)

This call is used to attach a recording handler.

**On entry**

- R0 = pointer to record handler code
- R1 = pointer to device identifier (zero terminated)
- R2 = bit flags:

Bit	Value when set
0 – 31	Reserved (0)

- R3 = the unique access identifier, or 0 if registering as a shared playback handler
- R4 = value of R12 passed into recording handler code

**On exit**

Registers preserved

**Interrupts**

Interrupt status is unaltered

**Processor mode**

Processor is in SVC mode

**Re-entrancy**

SWI is re-entrant

**Use**

This call is used by an application to register a recording handler to receive audio from a hardware device.



## SoundSystem\_DetachPlaybackHandler (SWI &5A08C)

This call is used to attach a playback handler.

### **On entry**

R0 = pointer to playback handler code

R1 = pointer to device identifier (zero terminated)

### **On exit**

Registers preserved

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is re-entrant

### **Use**

This call is used by an application to deregister a playback handler from a hardware device.

## SoundSystem\_DetachRecordingHandler (SWI &5A08D)

This call is used to attach a recording handler.

### **On entry**

R0 = pointer to record handler code

R1 = pointer to device identifier (zero terminated)

### **On exit**

Registers preserved

### **Interrupts**

Interrupt status is unaltered

### **Processor mode**

Processor is in SVC mode

### **Re-entrancy**

SWI is re-entrant

### **Use**

This call is used by an application to deregister a recording handler from a hardware device.

## SoundSystem\_SetVolume (SWI &5A08E)

Gets or sets the mixer settings

### On entry

R0 = bit field

Bit	Value when set
0 – 7	Mixer number, or 0 for overall volume
8 – 30	Reserved (0)
31	Change input mixer (if clear, change output mixer)

R1 = mixer value (0 = silent, 255 = maximum, or -1 to read)

R2 = device identifier

R3 = unique exclusive identifier (or 0 for shared)

### On exit

R1 = previous mixer value

Other registers preserved

### Processor mode

Processor is in SVC mode

### Re-entrancy

SWI is re-entrant

### Use

This SWI is used to get or set the mixer values. For shared access, only the left and right channels are available. For exclusive access, all channels can be changed.

# SoundSystem\_Configure (SWI &5A0BF)

Configures SoundSystem

**On entry**

R0 = reason code  
Other registers depend on reason code

**On exit**

R0 preserved  
Other registers depend on reason code

**Processor mode**

Processor is in SVC mode

**Re-entrancy**

SWI is re-entrant

**Use**

This SWI provides a number of configuration options for SoundSystem.  
The reason code values are as follows:

R0	Action
0	Mono mode

All other values are reserved

## SoundSystem\_Configure 0 (SWI &5A090)

Configures the shared access sound to be mono mode.

### On entry

R0 = 0 (reason code)

R1 = device identifier

R2 = 0 => stereo mode, 1 => mono mode

### On exit

R2 = previous stereo mode

Other registers preserved

### Use

This is used to configure the shared sound system to output mono audio, by providing a stereo to mono mix-down after all the audio playback handlers have filled their data.

Audio playback handlers are informed if the data is going to be mixed down to a mono playback when they need to fill their buffers so they can use different routines for optimisation.

Mono mode would normally only be needed if there was a single speaker output, such as that provided by an on-board sound system.

## SoundSystem\_EnumerateSharedHandlers (SWI &5A091)

Iterates over the list of shared playback or recording handlers

### **On entry**

R0 = previous shared handler identity, or 0 to start

R1 = device identity

### **On exit**

R0 = next shared handler identity, or 0 for no more

R1 preserved

R2 = WIMP task handle (if R0  $\neq$  0 on exit)

R3 = volume for the playback handler (if R0  $\neq$  0 on exit)

## SoundSystem\_SetSharedVolume (SWI &5A092)

Sets the playback volume for a shared handler

### **On entry**

R1 = device identity

R2 = WIMP task handle

R3 = new volume (0 = silent, 255 = maximum)

### **On exit**

R3 = old volume

Other registers preserved

---

# Index

---

## Table of Contents

Introduction and Overview.....	1
Audio capabilities – a brief history.....	1
SoundSystem API.....	3
Procedural interfaces.....	4
Audio format descriptors.....	5
Audio handlers.....	8
Playback handler code.....	9
Recording handler code.....	10
Codec interface.....	10
Codec configuration handler.....	10
Codec configuration 0.....	11
Codec configuration 1.....	11
Codec configuration 2.....	12
Codec conversion handler.....	12
Codec finalisation handler.....	13
Hardware interface.....	14
Control code.....	14
Control code 0.....	14
Control code 1.....	14
Control code 2.....	15
Control code 3.....	15
Control code 4.....	15
Control code 5.....	16
Control code 6.....	16
Control code 7.....	17
Control code 8.....	17
Playback data request.....	18
Recording data provision.....	18
Service calls.....	20
Service_SoundSystemInitialised.....	20
Service_SoundSystemHardwareAdded.....	21
Service_SoundSystemHardwareRemoved.....	22
Service_SoundSystemExclusiveAccessChanged.....	23
Service_SoundSystemRequestCodec.....	24
Service_SoundSystemCodecRemoved.....	25
Service_SoundSystemSampleRateChanged.....	26
SWI calls.....	27
SoundSystem_RegisterDevice.....	27
SoundSystem_DeregisterDevice.....	29
SoundSystem_EnumerateDevices.....	30
SoundSystem_GetDeviceInformation.....	31
SoundSystem_GetDeviceFormats.....	32
SoundSystem_GetDeviceFormatSampleRates.....	33
SoundSystem_RegisterExclusiveAccess.....	35
SoundSystem_DeregisterExclusiveAccess.....	36
SoundSystem_SetFormat.....	37
SoundSystem_SetSampleRate.....	38
SoundSystem_AttachPlaybackHandler.....	39
SoundSystem_AttachRecordingHandler.....	40
SoundSystem_DetachPlaybackHandler.....	41
SoundSystem_DetachRecordingHandler.....	42



SoundSystem_SetVolume.....	43
SoundSystem_Configure.....	44
SoundSystem_Configure 0.....	45
SoundSystem_EnumerateSharedHandlers.....	46
SoundSystem_SetSharedVolume.....	47