PlayStation®2 IOP Library Overview
Release 2.4

Sound Libraries

Summary Table of Contents

About This Manual

This is the Runtime Library Release 2.4 version of the *PlayStation®2 IOP Library Overview - Sound Libraries* manual.

The purpose of this manual is to provide overview-level information about the PlayStation®2 IOP sound libraries. For related descriptions of the PlayStation®2 IOP sound library structures and functions, refer to the *PlayStation®2 IOP Library Reference - Sound Libraries.*

Changes Since Last Release

Chapter 3: CSL Hardware Synthesizer (modhysn)

- In the "Buffer Structure" section of "Usage", a description restricting the number of currently implemented input ports has been added.
- In "Function Overview", a "MIDI Status" section has been added.
- In " Control Changes and NRPN" of "Function Overview", descriptions of the corresponding control change and NRPN have been added.
- In "Specifying the Information to Use Within Bank Binary Data" in "Function Overview", the description about bank binary data specification has been corrected.

Chapter 4: CSL MIDI Sequencer Module (modmidi)

- A "MIDI Messages" section has been added in "Function Overview".

Related Documentation

Library specifications for the EE can be found in the *PlayStation®2 EE Library Reference* manuals and the *PlayStation®2 EE Library Overview* manuals.

Note: the Developer Support Web site posts current developments regarding the Libraries and also provides notice of future documentation releases and upgrades.

Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

Convention	Meaning
`courier`	Indicates literal program code.
italic	Indicates names of arguments and structure members (in structure/function definitions only).
medium bold	Indicates data types and structure/function names (in structure/function definitions only).
blue	Indicates a hyperlink.

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In North America:	*In North America:*
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: PS2_Support@playstation.sony.com Web: http://www.devnet.scea.com/ Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees in Europe only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In Europe:	*In Europe:*
Attn: Production Coordinator Sony Computer Entertainment Europe 30 Golden Square London W1F 9LD, U.K. Tel: +44 (0) 20 7859-5000	E-mail: ps2_support@scee.net Web: https://www.ps2-pro.com/ Developer Support Hotline: +44 (0) 20 7859-5777 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT)

Chapter 1:
Low Level Sound Library

Library Overview

libsd is a low level library for controlling the PlayStation 2 sound device (SPU2 and peripheral circuits). Always use libsd to access the sound device.

libsd provides functions for accessing public SPU2 registers, transferring waveform data or sound data to or from SPU2 local memory, setting up processing for interrupts from the SPU2, and setting or getting effect attributes. It also provides batch functions for collecting together accesses to a series of SPU2 registers.

Register Macros

libsd uses register wrapper APIs and register macros to access SPU2 registers. SPU2 registers are classified into three types, namely basic parameter registers, voice control parameter registers, and address value registers. Register wrapper APIs for setting or getting values for each of these types of registers are provided separately. Pseudo registers have been defined for some SPU2 functions, and APIs that can be used in a similar manner as the other register wrapper APIs are provided.

Table 1-1

Register Type	Register Macro	Set Value	Get Value
Basic parameter register	SD_VP_*/SD_P_*	sceSdSetParam()	sceSdGetParam()
Voice control parameter register	SD_S_*	sceSdSetSwitch()	sceSdGetSwitch()
Address value register	SD_VA_*/SD_A_*	sceSdSetAddr()	sceSdGetAddr()
Pseudo register	SD_C_*	sceSdSetCoreAttr()	sceSdGetCoreAttr()

Since each core has registers with the same names, register macros must be used by taking the bitwise OR with a core-specific macro (SD_CORE_0 or SC_CORE_1). Also, macros that have registers with the same names for each voice (SD_V*) are used by further taking the bitwise OR with a voice-specific macro (SD_VOICE_00 to SD_VOICE_23).

(Example)

```
sceSdSetParam( SD_CORE_0|SD_P_MVOLL , 0x3fff ) ;
sceSdSetParam( SD_CORE_0|SD_VOICE_05|SD_VP_PITCH , 0x1000 );
```

List of Register Macros

Register macros are listed below.

Basic Parameter Register Macros

Table 1-2

Set: sceSdSetParam()

Get: sceSdGetParam()

Register Macro	Core Specification	Voice Specification	Function
SD_VP_VOLL	SD_CORE_?	SD_VOICE_??	Voice volume (left)
SD_VP_VOLR	SD_CORE_?	SD_VOICE_??	Voice volume (right)
SD_VP_PITCH	SD_CORE_?	SD_VOICE_??	Pitch when sound is generated
SD_VP_ADSR1	SD_CORE_?	SD_VOICE_??	Envelope
SD_VP_ADSR2	SD_CORE_?	SD_VOICE_??	Envelope (2)
SD_VP_ENVX	SD_CORE_?	SD_VOICE_??	Envelope current value
SD_VP_VOLXL	SD_CORE_?	SD_VOICE_??	Volume current value (left)
SD_VP_VOLXR	SD_CORE_?	SD_VOICE_??	Volume current value (right)
SD_P_MMIX	SD_CORE_?	---	Output specification after voice mixing
SD_P_MVOLL	SD_CORE_?	---	Master volume (left)
SD_P_MVOLR	SD_CORE_?	---	Master volume (right)
SD_P_EVOLL	SD_CORE_?	---	Effect return volume (left)
SD_P_EVOLR	SD_CORE_?	---	Effect return volume (right)
SD_P_AVOLL	SD_CORE_?	---	Core external input volume (left)
SD_P_AVOLR	SD_CORE_?	---	Core external input volume (right)
SD_P_BVOLL	SD_CORE_?	---	Sound data input volume (left)
SD_P_BVOLR	SD_CORE_?	---	Sound data input volume (right)
SD_P_MVOLXL	SD_CORE_?	---	Master volume current value (left)
SD_P_MVOLXR	SD_CORE_?	---	Master volume current value (right)

Voice Control Parameter Register Macros

Table 1-3

Set: sceSdSetSwitch()

Get: sceSdGetSwitch()

Register Macro	Core Specification	Voice Specification	Function
SD_S_PMON	SD_CORE_?	---	Pitch modulation specification
SD_S_NON	SD_CORE_?	---	Allocation to noise generator
SD_S_KON	SD_CORE_?	---	Key on (start voice generation)
SD_S_KOFF	SD_CORE_?	---	Key off (end voice generation)
SD_S_ENDX	SD_CORE_?	---	Endpoint passed flag
SD_S_VMIXL	SD_CORE_?	---	Voice output mixing specification (Dry left)
SD_S_VMIXR	SD_CORE_?	---	Voice output mixing specification (Dry right)
SD_S_VMIXEL	SD_CORE_?	---	Voice output mixing specification (Wet left)
SD_S_VMIXER	SD_CORE_?	---	Voice output mixing specification (Wet right)

Address Value Register Macros

Table 1-4

Set: sceSdSetAddr()
Get: sceSdGetAddr()

Register Macro	Core Specification	Voice Specification	Function
SD_VA_SSA	SD_CORE_?	SD_VOICE_??	Waveform data starting address
SD_VA_LSAX	SD_CORE_?	SD_VOICE_??	Loop point address
SD_VA_NAX	SD_CORE_?	SD_VOICE_??	Waveform data address that should be read next
SD_A_ESA	SD_CORE_?	---	Effect processing work area starting address
SD_A_EEA	SD_CORE_?	---	Effect processing work area final address
SD_A_TSA	SD_CORE_?	---	Transfer starting address
SD_A_IRQA	SD_CORE_?	---	Interrupt address specification

Entries (Pseudo Register Macros)

Table 1-5

Set: sceSdSetCoreAttr
Get: sceSdGetCoreAttr

Macro	Core Specification	Voice Specification	Function
SD_C_EFFECT_ENABLE	SD_CORE_?	---	Enable writing to effect area
SD_C_IRQ_ENABLE	SD_CORE_?	---	Enable IRQ interrupt
SD_C_MUTE_ENABLE	SD_CORE_?	---	Mute
SD_C_NOISE_CLK	SD_CORE_?	---	Noise generator M series shift frequency
SD_C_SPDIF_MODE	---	---	Set SPDIF (mask)

Caution: Setting SPDIF (Optical Digital) Output

Signals that are output from the SPU2 to the SPDIF must comply with the IEC958 standard. Although most required settings are made when the library is initialized, the application is responsible for the following three items. Therefore, be sure to set these correctly.

Table 1-6

Item	Setting	Meaning
Media (category code)	SD_SPDIF_MEDIA_DVD	DVD
	SD_SPDIF_MEDIA_CD	CD (default)
Output contents	SD_SPDIF_OUT_OFF	Do not output anything to the SPDIF.
	SD_SPDIF_OUT_PCM	Output the same sound as analog output by using PCM (default).
	SD_SPDIF_OUT_BITSTREAM	Output the data that was entered in the Core0 input block as a bitstream.
Digital recording	SD_SPDIF_COPY_NORMAL	Normal mode (first generation recordable) (default).
	SD_SPDIF_COPY_PROHIBIT	Prohibit digital recording

sceSdSetCoreAttr() is used to set attributes as follows. If a setting is made for either core, it becomes effective for both cores.

(Example) DVD, PCM output, prohibit digital recording

```
sceSdSetCoreAttr(
SD_C_SPDIF_MODE,
SPU_SPDIF_MEDIA_DVD | SD_SPDIF_OUT_PCM | SD_SPDIF_COPY_PROHIBIT
);
```

Bypass Function and Bitstream Voice Output

The SPU2 is equipped with a function to bypass its internal processing and output 3D sound (non-PCM) bitstream voice data for consumer use to the PlayStation 2's built-in optical digital output. The SPU2 can also bypass its internal processing and output PCM voice data.

The data formats and processing methods used for these functions are described below using actual code.

Summary of Bitstream Voice Output

Bitstream voice data is different from ordinary PCM voice data. Because it is highly encoded, it must be transmitted to a device that can decode the data, such as an AV amplifier, without changing a single bit.

When bitstream voice data is transferred to the SPU2 sound data input in the same manner as ordinary voice data, the value of each bit cannot be precisely guaranteed by volume calculations in the SPU2.

Consequently, the SPU2 is equipped with a bypass function for the SPU2 CORE0 sound data input that bypasses internal processing and outputs the data directly to the optical digital output. Bitstream voice data can also be handled using this function.

The following three procedures are performed for the processing in a program.

1. Mute: Sets the volume to 0 so no noise is output when settings are changed.
2. Optical digital output settings: Uses a function to change core attributes to set the switching between the optical digital output and bypass processing, and changes to a state that allows bitstream voice data to flow.
3. Data transfer: Transfers data to the SPU2 CORE0 sound data input that had been previously arranged in a suitable format.

Data transferred to the sound data input must be in a prescribed format, regardless of whether bypass processing is used. For bypass processing, a special data format is also required.

Precautions

Licensing

Generally speaking it is necessary to have an agreement or be certified with each license provider who plans, provides or manages the bitstream voice data format in order to use bitstream voice data in title applications. For details please consult with license providers of the bitstream voice data format you want to use.

Use with Analog Output

The SPU2 does not have a function to decode bitstream voice data. When you want to output sound output both to the optical digital output and the analog output as well, data in PCM format must be prepared in advance or the data decoded by software in the EE, down mixed to two channels, then output to the analog output (for example, input to the SPU2 CORE1 sound data input).

Pause / Resume Processing

When a state occurs in which data transfer is not continuous, such as when bitstream voice data pauses and resumes, the operation of a device that receives the optical digital output, such as an AV amplifier, will be different depending on the device. The SPU2 only transfers data and does not execute processing that resets the state when reissuing a transfer. Be sure to support suitable software according to the characteristics of the bitstream voice data.

For example, when pausing a transfer, the following types of representative processing are necessary: pausing only at fixed units (data block size), acquiring header information such as type of bitstream voice data and remaining size when paused, and resending this information when reissuing the transfer.

For details on the format when transferring bitstream voice data using a digital interface, refer to International Standard IEC61937 specified by the IEC (International Electrical Standards Association) and related documentation. Also refer to the documentation of the encoder being used. You should also consult with the license provider of the bitstream voice data format.

Amount of Data

From the standpoint of restrictions on data formats, the amount of data per unit time is twice that of a data transfer to an ordinary sound data input. Be sure to take note of the transfer rate as well as the amount of IOP memory being used.

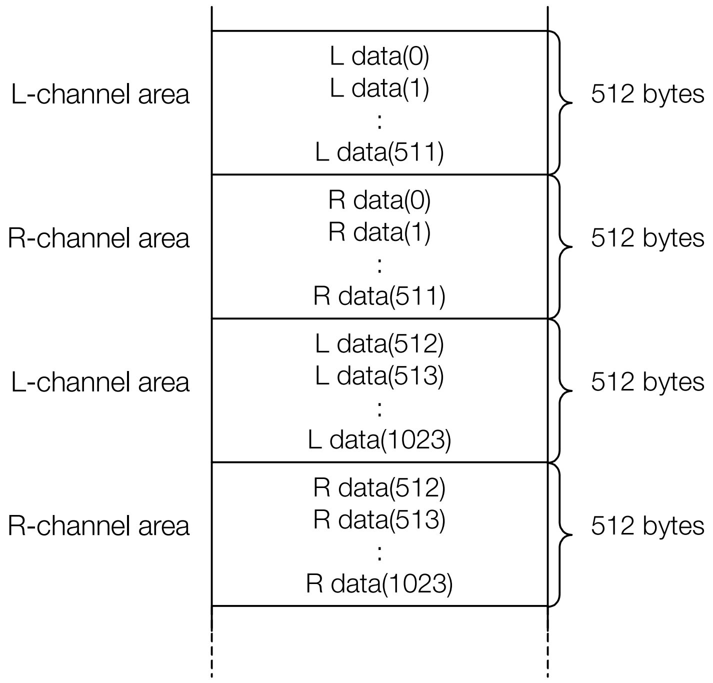
Data Format of Bitstream Voice Data

The data format when working with bitstream voice data will be described here.

Data transferred to the SPU2 sound data input generally must be in the following format.

- Should be little-endian.
- L-channel and R-channel data should be interleaved every 512 bytes.

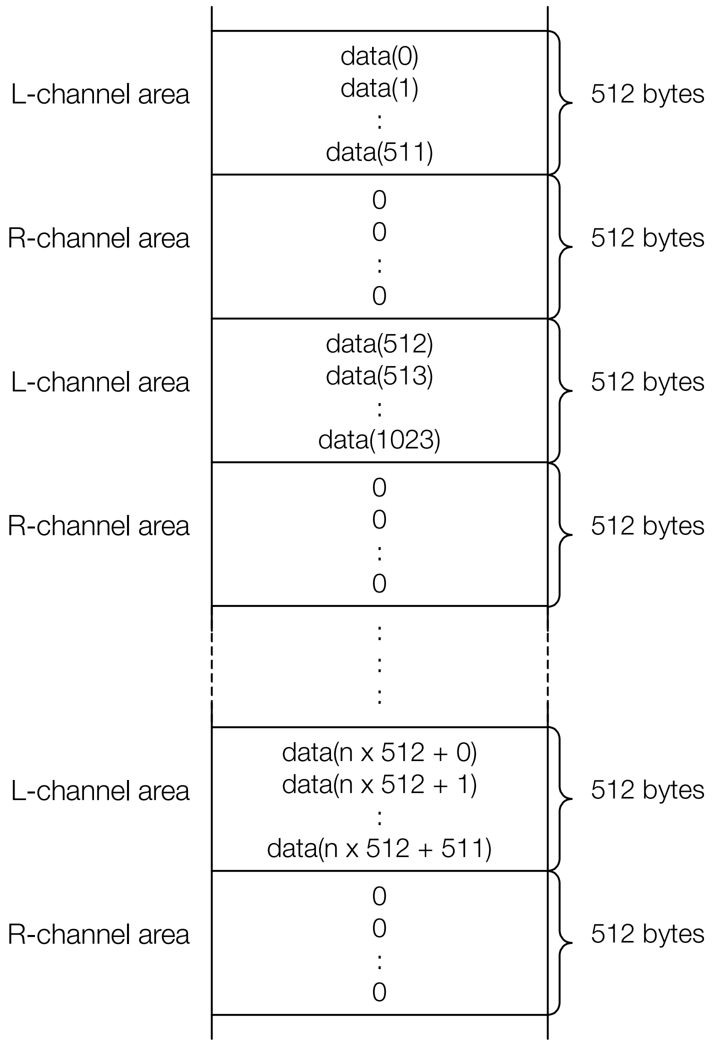
Figure 1-1: Data format for SPU2 sound data input



The following condition is added for bitstream voice data.

- The actual data should be sequentially arranged on the L-channel side and all values on the R-channel side should be 0.

Figure 1-2: Bitstream voice data format



Actual Processing

The data processing method that handles bitstream voice data will be described here using actual code.

Mute

Uses the register wrapper function sceSdSetParam() used for basic parameter setting, to set the value of volume BVOL of the SPU2 CORE0 sound data input to 0. If this is not done, noise will be output. Be sure to perform this operation.

Code example:

```
/* Set SPU2 CORE0 BVOL to 0 on both L/R sides */
sceSdSetParam (SD_CORE_0 | SD_P_BVOLL, 0);
sceSdSetParam (SD_CORE_0 | SD_P_BVOLR, 0);
```

Optical Digital Output Settings

Uses the core attribute setting function sceSdSetCoreAttr() to specify SD_SPDIF_OUT_BITSTREAM in the "type to output" of SD_C_SPDIF_MODE. Bypass processing becomes valid at the same time through the use of this specification.

Code example:

```
/* Media is DVD, output is bitstream, digital recording is
   prohibited */
sceSdSetCoreAttr(SD_C_SPDIF_MODE,
     (SD_SPDIF_MEDIA_DVD | SD_SPDIF_OUT_BITSTREAM |
SD_SPDIF_COPY_PROHIBIT));
```

Data Transfer

Uses the function sceSdBlockTrans() that performs data transfers (block transfers) to the sound data input, to transfer bitstream voice data in a format indicated in advance.

Code example:

```
sceSdBlockTrans(SD_CORE_0, (SD_TRANS_MODE_WRITE | SD_BLOCK_LOOP),
                  buffer_top_addr, buffer_size);
```

End Transfer of Bitstream Voice Data

Stops data transfers (block transfers) to the sound data input just like ordinary sound data.

Code example:

```
sceSdBlockTrans (SD_CORE_0, SD_TRANS_MODE_STOP, NULL, 0);
```

Thereafter, the settings of the optical digital output should be returned to normal when you want to output ordinary sound to the optical digital output. In addition, the value of volume BVOL of the SPU2 CORE0 sound data input should be set as necessary.

Code example:

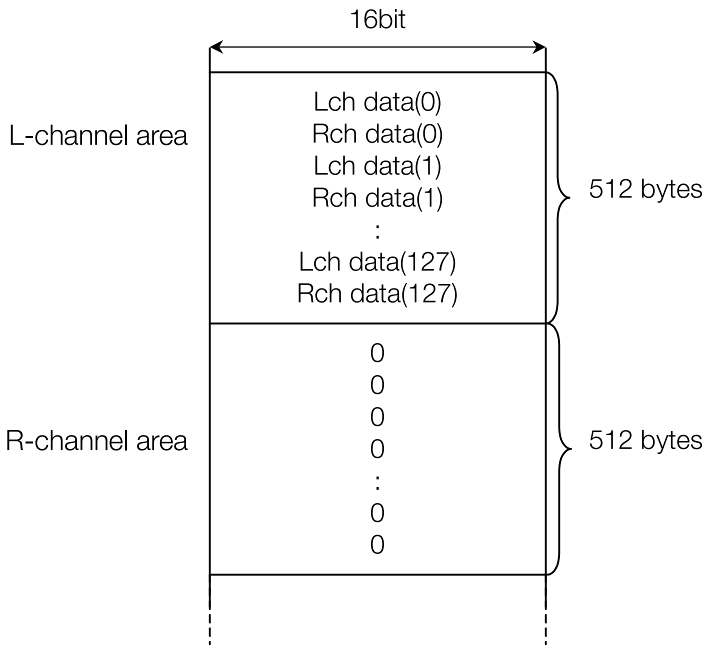
```
/* Media is DVD, output is PCM, digital recording is permitted */
        sceSdSetCoreAttr (SD_C_SPDIF_MODE,
                    (SD_SPDIF_MEDIA_DVD | SD_SPDIF_OUT_PCM |
SD_SPDIF_COPY_NORMAL));
        /* Set both L/R sides of SPU2 CORE0 BVOL to 0x7fff */
        sceSdSetParam (SD_CORE_0 | SD_P_BVOLL, 0x7fff);
        sceSdSetParam (SD_CORE_0 | SD_P_BVOLR, 0x7fff);
```

Operation for Bypass Processing Only

In the SPU2 CORE0 sound data input, not only can bitstream voice data be subject to bypass processing, but ordinary PCM voice data can as well.

The data format for this type of operation is similar to bitstream voice data with required data arranged in the L-channel area and the entire R-channel area being set to 0. In the L-channel area, 16-bit PCM voice data is alternately arranged L/R every 128 samples for a total of 256 samples.

Figure 1-3: PCM voice data format for bypass processing



In a program, processing is similar to that for bitstream voice data except that SD_SPDIF_OUT_BYPASS should be specified for the "type to output" for the argument SD_C_SPDIF_MODE of sceSdSetCoreAttr().

Code example:

```
/* Media is DVD, output is bypass processing, digital recording is
   prohibited */
sceSdSetCoreAttr(SD_C_SPDIF_MODE,
            (SD_SPDIF_MEDIA_DVD | SD_SPDIF_OUT_BYPASS |
SD_SPDIF_COPY_PROHIBIT));
```

If the bypass function is used, the amount of data transferred will normally be two times as indicated above. Also, the data is not normally output to the analog output and the volume cannot be adjusted either. Given these characteristics, be sure to use this feature only if you have a special need to handle bitstream data and only after thorough examination.

SPU2 Overview and Differences with the SPU

This section presents an overview of the hardware for the SPU2, which is the PlayStation 2 sound processor, and briefly explains the differences between the SPU2 and the SPU, which is the PlayStation sound processor.

SPU2 Overview

The SPU2, which consists of two cores, is a sound synthesizing processor that has local memory and external input/output. Each core (CORE0 and CORE1) performs the equivalent processing of the basic functions of the SPU that had been provided by the PlayStation, ranging from voice processing to effects processing and master volume processing, and each core also has sound generation functions for 24 voices.

Various types of switching functions and sound data input/output functions for 16-bit PCM data have also been added.

CORE0 and CORE1, which are functionally equivalent, operate independently. They are connected so that CORE0 output is input to CORE1, and the final mixed sound is output from CORE1.

Differences Between the SPU2 Cores and the SPU

Each core of the SPU2 differs from the SPU for the items described below.

Processing resolution and sampling frequency

Core processing is performed in units of 48 kHz. The resolution and sampling frequency of sound generation processing are also 48 kHz.

Read position available during waveform data processing

The waveform data that voice processing will use next can be read as an address for each voice.

End of loop can be confirmed

The core, which has a flag for each voice that indicates that voice processing has reached the end of a loop, can confirm that the end of a loop was reached for waveform data having loop information.

Mixing can be switched separately for dry and wet output

In addition to panpot control according to the voice volume, mixing for the wet and dry, left and right channels can be turned on and off for each voice.

16-bit or 24-bit straight PCM sound data input

16-bit or 24-bit straight PCM data can be input (transferred) as sound data from the host and mixed with the voice processing mixing results. Although local memory is used as the input buffer, this corresponds in form to the core external input of the SPU.

16-bit sound data input without volume processing

Encoded 16-bit sound data can be directly output (transferred) from the host to digital output.

Sound data output of mixing results

The 24-voice final mixed output sound data of a total of four channels (wet and dry, left and right channels) can be transferred to the host.

Auto DMA Transfer

Auto DMA Transfer, which reduces the host-side interrupt load due to data transfers during sound data processing, has been added as a DMA transfer mode.

Effects work area can be restricted and moved

The ending position of the effects area can be specified in units of 128 Kbytes.

Differences Between the SPU2 and SPU

The entire SPU2 differs from the SPU for the items described below.

Enlarged local memory size

The SPU2 has a 2Mbyte local memory (sound buffer). Only one local memory exists for the two cores, and the entire 2Mbyte area can be accessed from both cores.

Additional reserved area within local memory

Since there are two cores and since local memory is used as the sound data input/output buffer area, the reserved area at the beginning of local memory has been enlarged.

Two DMA transfers can be executed at the same time

Each core has its own independent DMA transfer functions, and these functions can operate in parallel.

Two interrupts can be generated at the same time

Each core has its own independent hardware interrupts, and these interrupts can operate in parallel.

Two reverbs can be set separately

Each core has its own independent reverb processing functions, and reverb processing can be performed with different settings.

CORE0 output is connected as CORE1 input

The 24-voice + reverb processing sound that was generated by CORE0 is input to CORE1. CORE1-side reverb processing can also be added to this sound. The CORE1 input to which CORE0 output is linked corresponds to the CD input of the SPU.

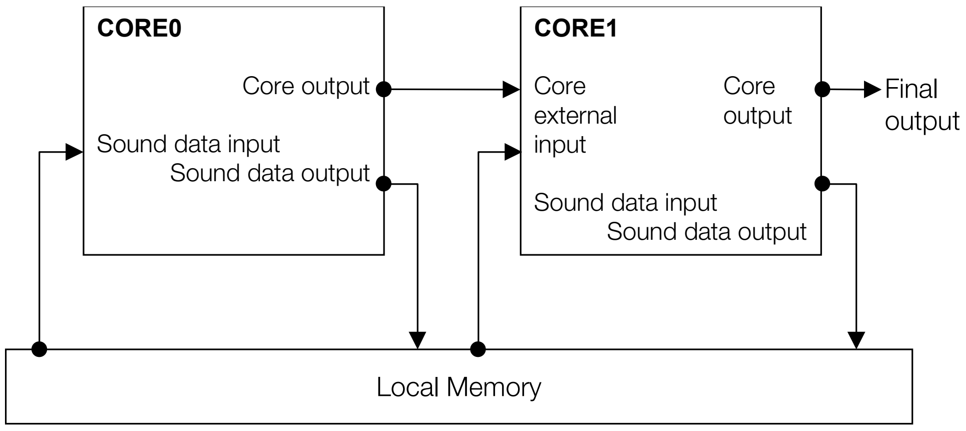
Single-system output for each of analog and digital

Analog and digital each have single-system output, which can be switched on and off.

Internal Signals and Function Connections

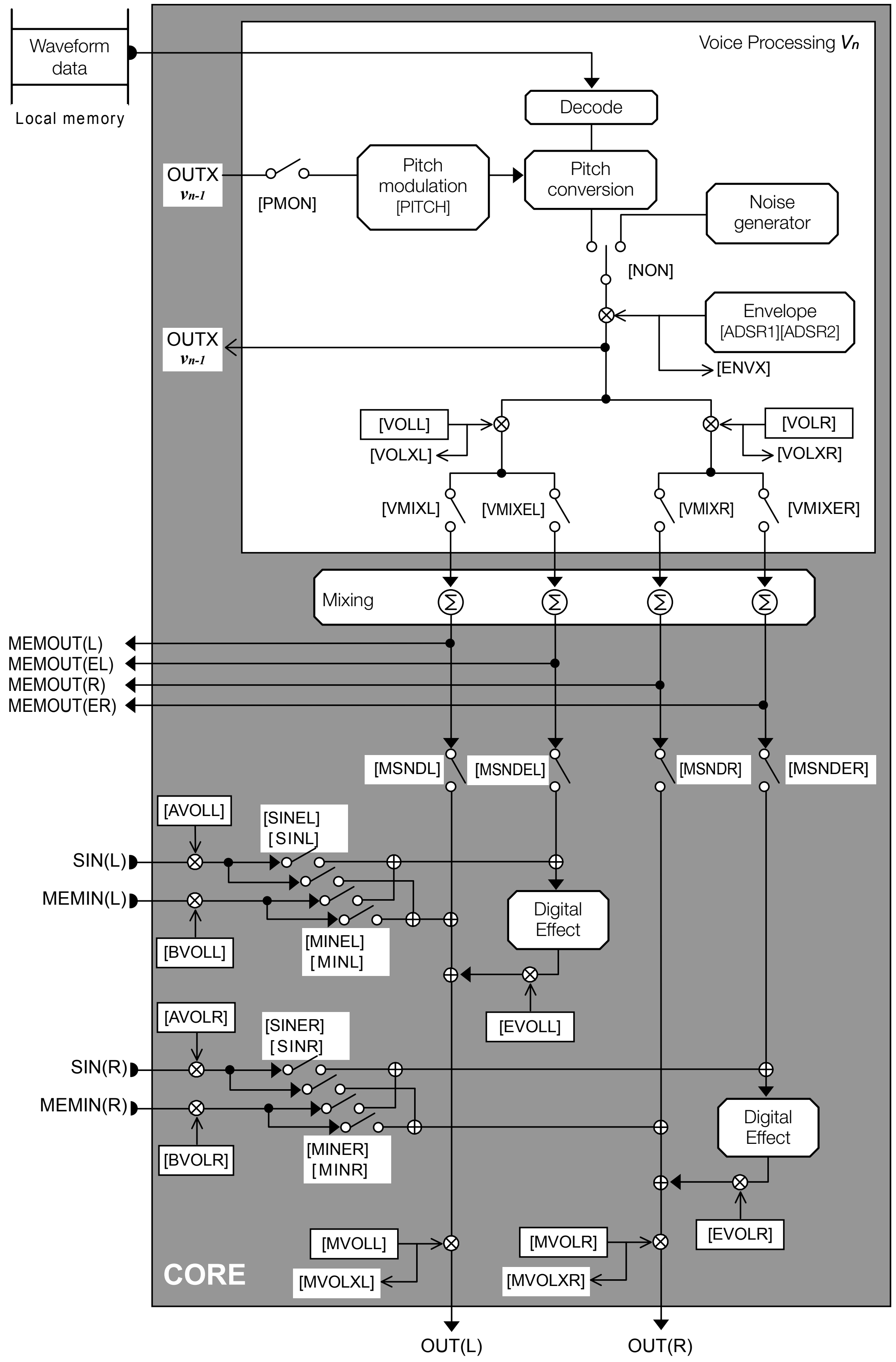
The SPU2 core and local memory connections are shown below.

Figure 1-4



The core internal connections are shown below.

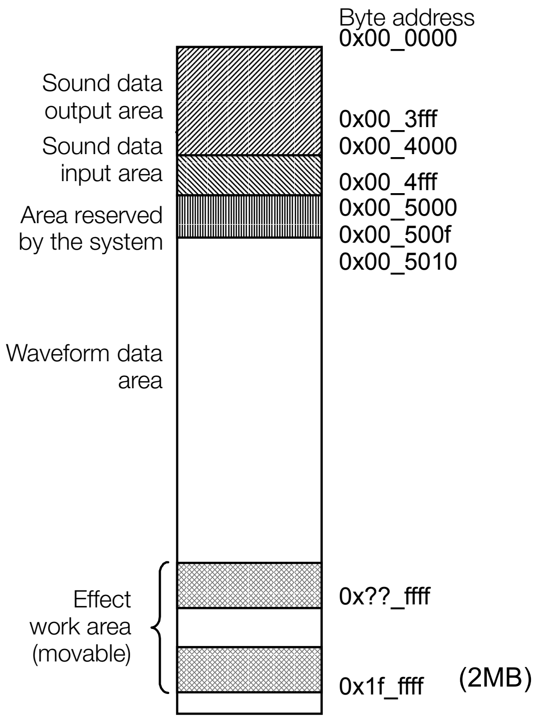
Figure 1-5


Waveform data
Local memory
Voice Processing Vn
Decode
OUTX vn-1
[PMON]
Pitch modulation [PITCH]
Pitch conversion
Noise generator
[NON]
Envelope [ADSR1][ADSR2]
OUTX vn-1
[ENVX]
[VOLL]
[VOLXL]
[VOLR]
[VOLXR]
[VMIXL]
[VMIXEL]
[VMIXR]
[VMIXER]
Mixing
MEMOUT(L)
MEMOUT(EL)
MEMOUT(R)
MEMOUT(ER)
[MSNDL]
[MSNDEL]
[MSNDR]
[MSNDER]
[AVOLL]
[SINEL] [SINL]
SIN(L)
MEMIN(L)
[BVOLL]
[MINEL] [MINL]
Digital Effect
[EVOLL]
[AVOLR]
[SINER] [SINR]
SIN(R)
MEMIN(R)
[BVOLR]
[MINER] [MINR]
Digital Effect
[EVOLR]
[MVOLL]
[MVOLXL]
[MVOLR]
[MVOLXR]
CORE
OUT(L)
OUT(R)

Memory Map

The memory map of local memory is shown below.

Figure 1-6



Chapter 2:
CSL MIDI Delay Module (moddelay)

Overview

moddelay is an IOP module that is implemented to conform with the Component Sound Library (CSL) specification. Moddelay has functions for outputting the input buffer MIDI Stream data such that it can be delayed by a specified number of Ticks. It can be used, for example, to absorb the latency difference between the software and hardware synthesizers.

Usage Method

Buffer Structure

Table 2-1: Buffer Group 0: Input port group

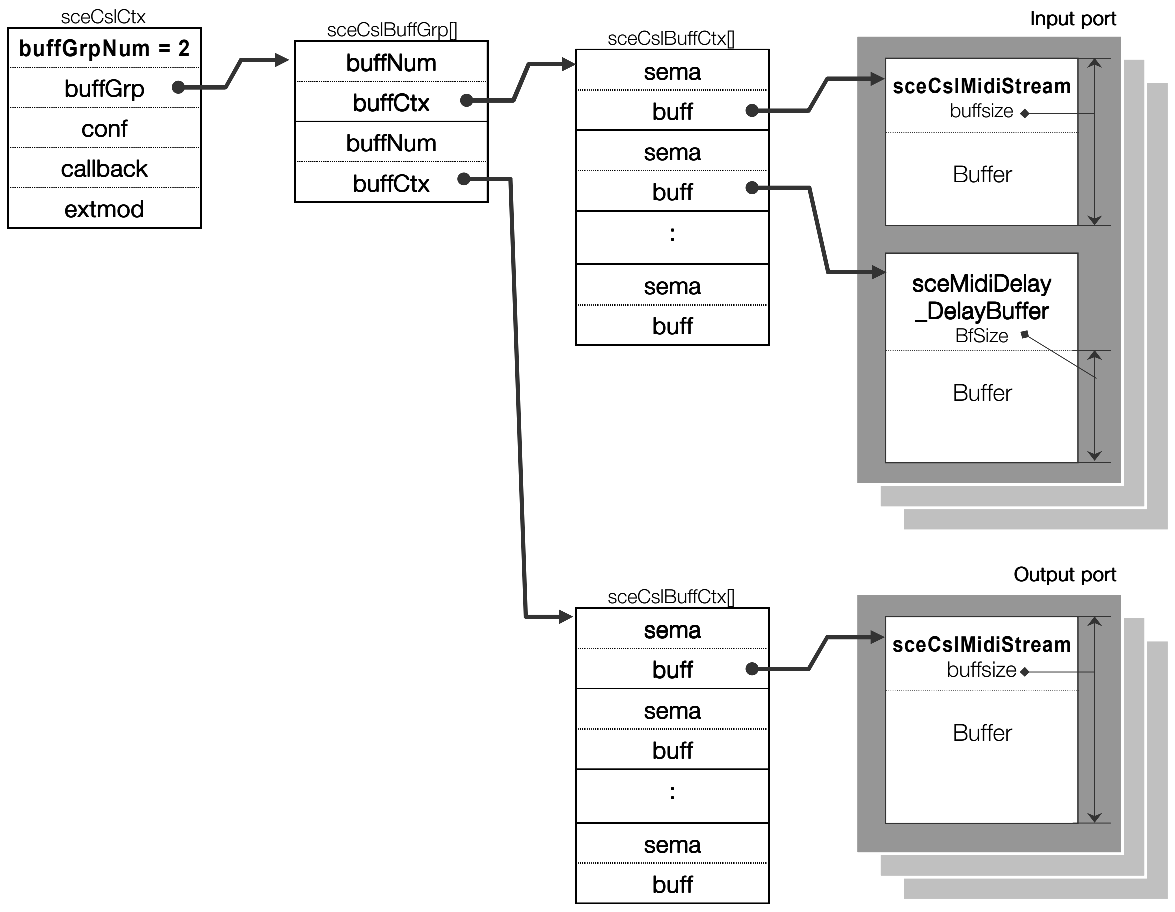
BufCtx Index	Contents	Data Structure
N*2	Input port Nth input buffer	sceCslMidiStream void[buffsize-sizeof (sceCslMidiStream)]
N*2+1	Input port Nth delay buffer	sceMidiDelay_DelayBuffer

Table 2-2: Buffer Group 1: Output port

BufCtx Index	Contents	Data Structure
N	Output port Nth output buffer	sceCslMidiStream void[buffsize - sizeof(sceCslMidiStream)

N can be in the range (0 - 15).

Figure 2-1



Input buffers and their corresponding delay buffers must exist in pairs. If one is missing, either an error will occur during initialization or that input buffer will become unusable.

Initialization Processing

Moddelay initialization is performed by calling sceMidiDelay_Init(). Before initializing moddelay, you must reserve the buffer structures described above in memory and set values for the members of the sceMidiDelay_DelayBuffer structure to be placed in the delay buffer, as indicated below.

delayBfSize	Number of bytes in the delay buffer data area (delayBf[])
delayTime	Delay time (number of times ATick() is called)

Chapter 3:
CSL Hardware Synthesizer (modhysn)

Overview

modhsyn is a sound source module that is implemented in conformance with the Component Sound Library (CSL). Input buffer MIDI Stream data and SE Stream data are interpreted, and bank binary files (*.bd, *.hd) are used as phoneme data to generate sound using the SPU2. Modhsyn also has a debugging function that can be used to obtain SPU2 access history.

modhsyn supports SE messages and extended MIDI messages that are defined for controlling sound effects. For detailed information about these kinds of messages, please see the "CSL Overview" document.

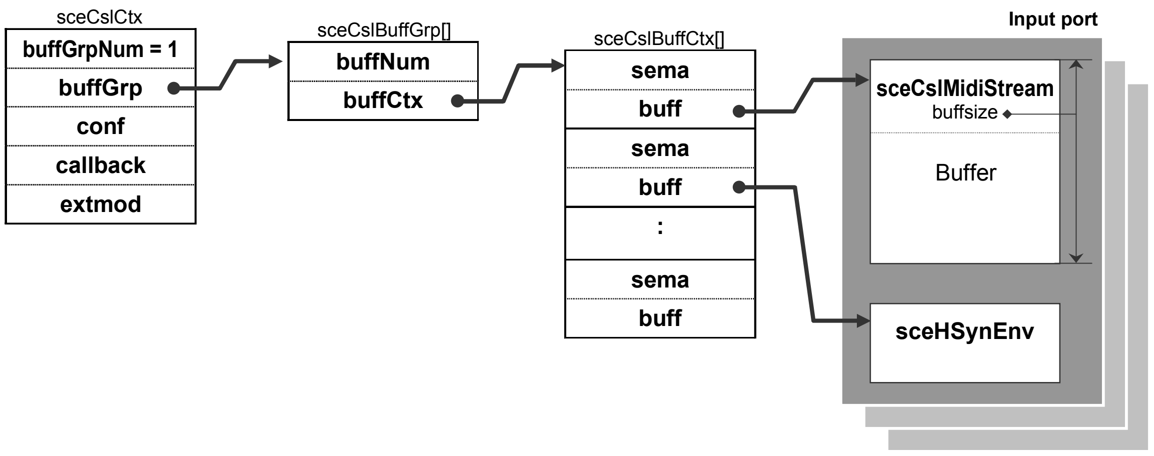
Usage

Buffer Structure

Table 3-1: Buffer Group 0: Input port group

BufCtx Index	Contents	Data Structure
N*2	Input port Nth input buffer	sceCslMidiStream void[buffsize-sizeof (sceCslMidiStream)] or sceCslSeStream void[buffsize-sizeof (sceCslSeStream)]
N*2+1	Input port Nth environment	sceHSynEnv

Figure 3-1



In the current implementation, the 16 ports between input ports 0 and 15 can be used.

The input buffers and their environments must exist as pairs. If one is missing, either an error will occur during initialization or that input buffer will become unusable.

Bank Binary Files

Bank binary files that are to be used as phoneme data can be created using an SCE-provided tool (JAM).

Function Overview

External Specifications

External specifications for modhsyn sound source module functions, in particular, specifications for functions specific to modhsyn are shown below.

For information about extended MIDI messages and SE messages, please refer to the "CSL Overview" document.

MIDI Status

modhsyn supports the following MIDI statuses.

Table 3-2

MIDI Status	Contents
$8n	Note OFF
$9n	Note ON (when velocity=0 Note OFF)
$Bn	Control Change
$Cn	Program Change
$En	Pitchbend Change

Also, in addition to the above, in exclusive messages, data between $F0 and $F7 are skipped. Extended MIDI messages $F9 and $FD are also supported.

Control Changes and NRPN

modhsyn currently supports the following control changes.

Table 3-3

Control No.	Contents in modhsyn
#00	Bank Select
#01	Pitch Modulation Depth
#02	Amp Modulation Depth
#05	Portament Time
#06	Data Entry (Effect Parameter)
#07	Channel Volume
#10 ($0A)	Pan
#11 ($0B)	Expression
#64 ($40)	Damper Pedal
#65 ($41)	Portament Switch
#84 ($54)	Portament Control
#98 ($62)	NRPN LSB
#99 ($63)	NRPN MSB
#120 ($78)	All Sound Off
#121 ($79)	Reset All Controller
#123 ($7B)	All Note Off

Currently, modhsyn supports the following NRPN.

Table 3-4: Effect type setting

NRPN	Function
$B0,$63,$02	Effect type command
$B0,$62,$00	• Core specifications (Core0)
$B0,$62,$10	• Core specifications (Core1)
$B0,$06,$00	: Off
$B0,$06,$01	: Room
$B0,$06,$02	: StudioA
$B0,$06,$03	: StudioB
$B0,$06,$04	: StudioC
$B0,$06,$05	: Hall
$B0,$06,$06	: Space
$B0,$06,$07	: Echo
$B0,$06,$08	: Delay
$B0,$06,$09	: Pipe

Table 3-5: Effect depth and other settings

NRPN	Function
$B0,$63,$03	Effect depth command
$B0,$62,$00	Effect depth positive phase (Core0)
$B0,$62,$01	Effect depth negative phase (Core0)
$B0,$62,$02	Effect delay (Core0)
$B0,$62,$03	Effect feed (Core0)
$B0,$62,$10	Effect depth positive phase (Core1)
$B0,$62,$11	Effect depth negative phase (Core1)
$B0,$62,$12	Effect delay (Core1)
$B0,$62,$13	Effect feed (Core1)
$B0,$06,$nn	Parameters ($nn=$00 to $7F)

MIDI Streams and SE Streams

modhsyn supports MIDI streams and SE streams as streams that are input to the input port. The stream that is input to that input port can be specified by setting the portMode member of the input port environment.

Table 3-6

Mode	Contents
SceHSynModeHSyn	MIDI stream
sceHSynModeSESyn	SE stream

Specifying the Information to Use Within Bank Binary Data

modhsyn supports multiple waveform data information sets included within bank binary data (waveform data) that was assigned to the input port. By setting the waveType member of the input port environment, you can specify which waveform data information set is to be used from the bank binary data that was assigned to that input port.

Table 3-7

Mode	Contents
sceHSynModeHSyn	Normal waveform data information set
sceHSynModeTimbre	Waveform data information set for SE stream (Do not specify for MIDI sequences)

Assigning Voices Used by Each Stream

The sceHSyn_SESetMaxVoices() function can be used to set an upper limit for the total number of voices that an SE stream will use to generate sound.

This upper limit is used as follows when sound is being generated between MIDI streams and SE streams.

When the number of voices of the SE stream is below the upper limit

- When there are voices that are not being used by either stream

  MIDI stream sound generation uses all remaining voices that are not being used by the SE stream, regardless of the SE stream upper limit.

  SE stream sound generation also uses remaining voices within a range that does not exceed the upper limit.

- When no voices are available

  MIDI stream sound generation only compares the priorities of only voices that are being used by the MIDI stream, frees a voice, then assigns a voice.

  Similarly, SE stream sound generation compares the priorities of only voices that are being used by the MIDI stream, frees a voice, and assigns a voice.

When the number of voices of the SE stream has reached the upper limit

MIDI stream sound generation operates in a similar manner as described above for when the number of voices of the SE stream has not reached the upper limit.

Even when there are available voices, SE stream sound generation compares the priorities of only voices that are being used by the SE stream, frees a voice, then assigns a voice.

Also, if the upper limit is set to 0, no distinction is made for MIDI and SE stream sound generation with respect to the voices that are used. All available voices are used for both types of sound generation, and voices are freed using the priorities of all voices.

Debug Support

You can obtain SPU2 access history using int sceHSyn_SetDebugInfoBuffer(sceHSyn_DebugInfo *debuginfo_buffer) to register a debug information buffer.

sceHSyn_DebugInfo is used as a ring buffer. The arguments used when calling libsd are entered for arg of sceHSyn_SdCall and the return value is entered for retVal. The number representing the libsd function is entered for func, and after function is called, the sceHSyn_SdCall_inProcess bit of func becomes 1 until that function terminates.

Chapter 4:
CSL MIDI Sequencer Module (modmidi)

Overview

modmidi is an IOP module that is implemented to conform with the Component Sound Library (CSL) specification. Modmidi has functions for inputting and interpreting sq data (sequence data) and outputting MIDI Stream data.

The sq data that is input to modmidi is a proprietary format converted from smf. It can be created using an SCE-provided tool (smf2sq or JAM). Since modmidi can have multiple input buffers, multiple sq data streams can be performed simultaneously.

The MIDI Stream data that is output by modmidi becomes the input data of the sound source module (hardware synthesizer or software synthesizer). Since modmidi can have multiple output buffers, multiple sound sources can be driven. The correspondence between a track and an output destination buffer is specified by the Bank Change LSB (32) of a MIDI message.

Usage

Buffer Structure

Table 4-1: Buffer Group 0: Input port group

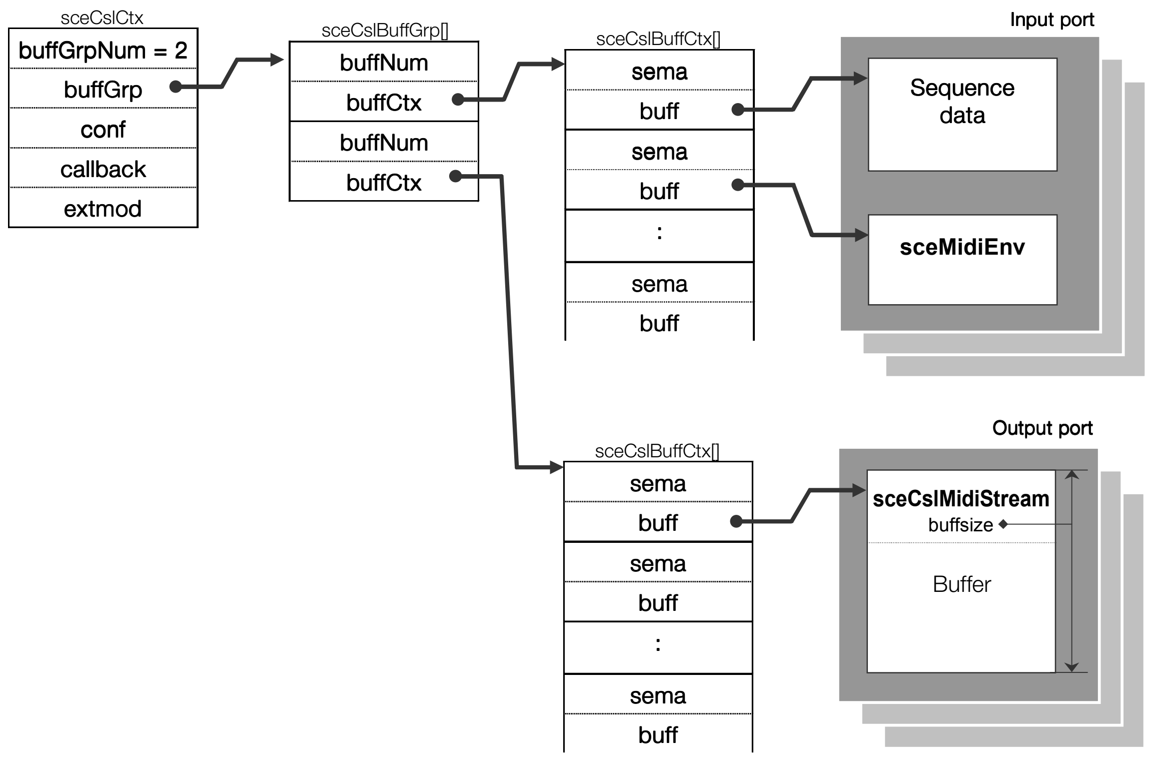
BufCtx Index	Description	Data structure
N*2	Input port Nth input buffer	void[]
N*2+1	Input port Nth environment	sceMidiEnv

Table 4-2: Buffer Group 1: Output port group

BufCtx Index	Description	Data structure
N	Output port Nth output buffer	sceCslMidiStream void[buffsize - sizeof (sceCslMidiStream)]

N can be in the range (0 - 15).

Figure 4-1



Input buffers and their corresponding environments must exist in pairs. If one is missing, either an error will occur during initialization or that buffer will become unusable.

MIDI Message Filter Callbacks

The following callbacks are provided to allow MIDI messages output from modmidi to be changed at any time.

Table 4-3

Trigger	Callback Function	Filter Function
Channel message	chMsgCallBack	Replace message
Exclusive message	excMsgCallBack	Suppress message output
Meta-event	metaMsgCallBack	Suppress message output
Loop	repeatCallBack	Suppress loop

Callback functions are registered in sceMidiEnv. When a message, etc., appears in the sequence data as a trigger, the associated callback function will be called.

The callback function examines the message, etc., which is passed as a parameter, and suppresses the message output. The function can then specify a new message to be output in place of the original message.

Function Overview

MIDI Messages

modmidi can process sequence data format (SQ) data. For information about the sequence data format, please see the "Sound Data Formats" document.

Although most MIDI messages are output as is to the output port, the following MIDI messages are supported by modmidi.

Table 4-4

MIDI Message	Contents
$Bn, $20, $nn	Set output port of MIDI channel n to $nn
$B0, $63, $00/$01	Loop control (see next section)
$FF, $51, $03, $n1, $n2, $n3	Meta event: Set tempo in microseconds
$FF, $2F, $00	Meta event: End of data

For "Meta event: Set tempo in microseconds," although modmidi performs its processing based on that value, keep in mind that it is ultimately quantized according to the ATick processing time interval.

Loop Control

modmidi can embed a loop performance as in the following control change messages.

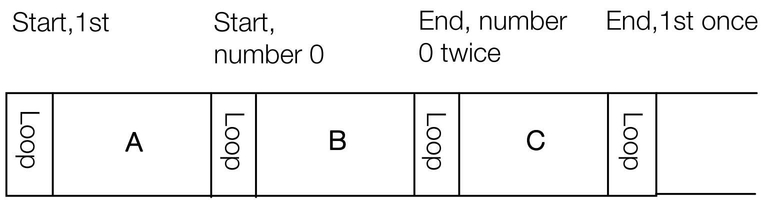
Table 4-5

Function	Message	Explanation
Loop start	$B0,$63,$00,$B0,$06,$nn	nn is the loop number
Loop end	$B0,$63,$01,$B0,$06,$nn, $B0,$26,mm	nn is the loop number mm is the loop count

If the loop count is 0, an endless loop will occur.

Loop settings can be nested. For example, if the loop start and loop end are set as follows, the performance order will be A B B B C A B B B C.

Figure 4-2



As with SEQ loops, these loops are performed by sequence. Loops cannot be performed by track. Consequently, loop settings should be made for a single track.

Chapter 5:
CSL MIDI Monophonic (modmono)

Overview

modmono is an IOP module that is implemented to conform with the Component Sound Library (CSL) specification. modmono has functions for outputting the input buffer's MIDI Stream data as MIDI Stream data that has been monophonically assigned. The existence of monophonic assignment can be specified for each channel.

Usage

Buffer Structure

Table 5-1: Buffer Group 0: Input port group

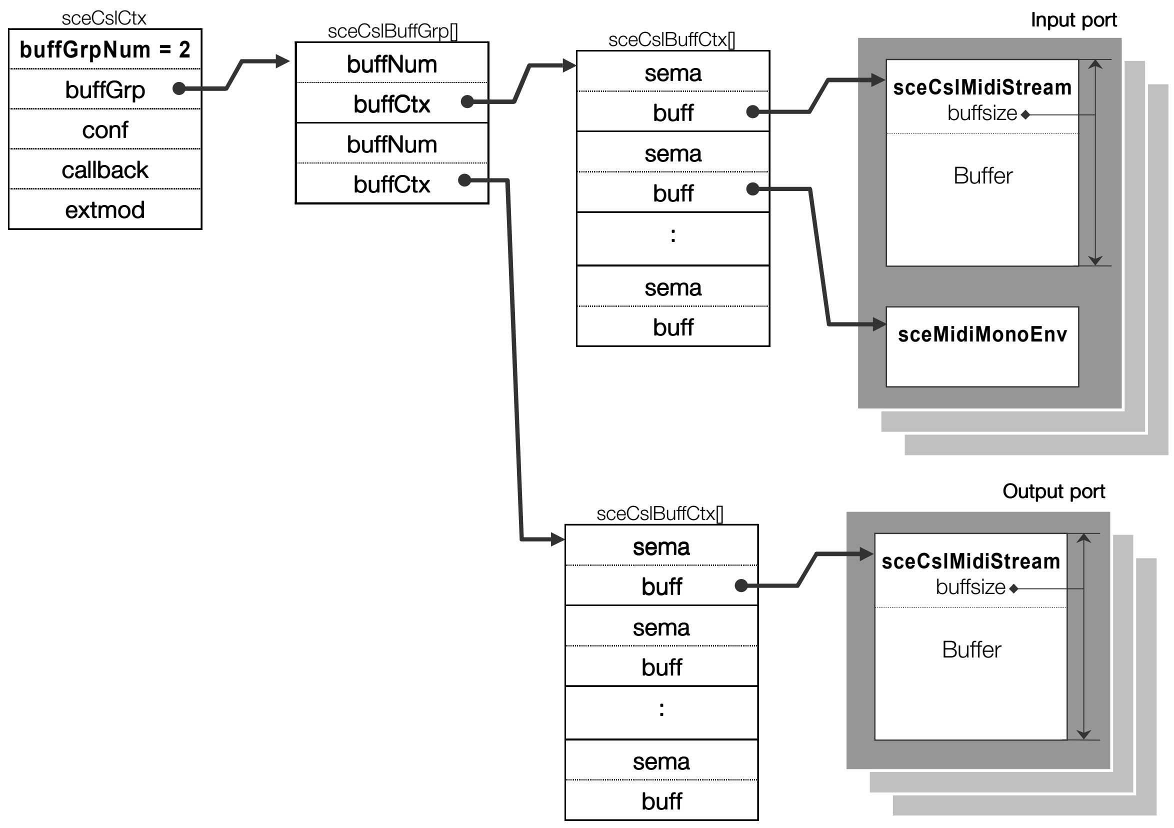
BufCtx Index	Contents	Data Structure
N*2	Input port Nth input buffer	sceCslMidiStream void[buffsize - sizeof(sceCslMidiStream)]
N*2+1	Input port Nth environment	sceMidiMono_Env

Table 5-2: Buffer Group 1: Output port group

BufCtx Index	Contents	Data Structure
N	Output port Nth output buffer	sceCslMidiStream void[buffsize - sizeof(sceCslMidiStream)]

N can be in the range (0 - 15).

Figure 5-1



Input buffers and their corresponding output buffers must exist in pairs. If one is missing, either an error will occur during initialization or that input buffer will become unusable.

Chapter 6: CSL MIDI Stream Generation Module (modmsin)

Overview

modmsin is an IOP module that is implemented to conform with the Component Sound Library (CSL) specification. Modmsin generates a MIDI stream by an API call from an application program, and outputs that data to an output buffer. It differs from a typical CSL module in that it has no input buffers and the ATic() function is not implemented.

modmsin supports extended MIDI messages. For information about extended MIDI messages, please refer to the "CSL Overview" document.

Usage Method

Buffer Structure

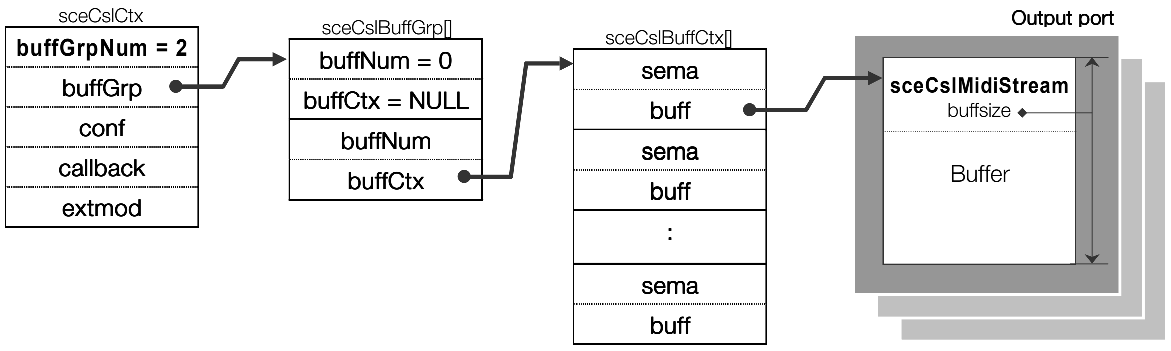
Buffer Group 0: (None)

Table 6-1: Buffer Group 1: Output port group

BufCtx Index	Contents	Data Structure
N	Output port Nth buffer	sceCslMidiStream void[buffsize - sizeof (sceCslMidiStream)]

N can be in the range (1-15).

Figure 6-1



MIDI Message Packing

modmsin packs a 2- to 3-byte MIDI message into a 32-bit word before outputting it.
The procedure is to first use a macro such as sceMSIn_MakeMsg() to prepare a packed MIDI message and then to use the sceMSIn_PutMsg() function to write it to the output stream.

Function Characteristics

Extended MIDI Message

modmsin can embed extended MIDI messages defined by SCE for sound effects in the output stream using the sceMSIn_PutHsMsg() function. A macro is also provided for restoring the various parameters of the extended MIDI message.

For information about the contents of extended MIDI messages, please refer to the "CSL Overview" document.

Chapter 7: CSL SE Stream Generation (modsein)

Overview

modsein is an IOP module that is implemented in conformance with the Component Sound Library (CSL). Modsein generates an SE stream via an API call from an application program and outputs that data to the output buffer.

modsein differs from a typical CSL module in that it has no input buffer, and the ATick() function is only provided formally. Modsein supports SE messages. For information about SE messages, refer to the "CSL Overview" document.

Using Modsein

Buffer Structure

Buffer Group 0: (none)

Buffer Group 1: Output port group

Table 7-1

BufCtx Index	Contents	Data Structure
N	Buffer of output port N	sceCslSeStream void[buffsize - sizeof (sceCslSeStream)]

N can range from 1 to 15.

Generating SE Messages

modsein can generate an SE message consisting of a number of bytes from the argument specification, and output it directly to the output buffer. Or, the message can be temporarily packed into units of 32-bit words and output to the output buffer. Headers added to already generated SE messages can also be output to the output buffer.

Normally, a function specially prepared for each message is used.

Buffers which create SE messages in advance are also provided to allow the messages to be output to the output buffer using sceSEIn_PutSEMsg(). For Note ON / Pitch ON messages, utility macros which correspond to each message are also provided.

Function Characteristics

SE Messages

An SE message is a data format that is provided for controlling sound effects. It is used for handling parameter controls in realtime from the time sound is generated until it is cancelled. The ID that is specified when a sound is generated is used for identifying a voice in an individual control.

For details about SE messages, refer to the "CSL Overview" document.

Chapter 8: CSL Sound Effect Sequencer Module (modsesq)

Overview

modsesq is an IOP module that performs inputting and decoding of sq data (sequence data) and outputting of SE Stream data. modsesq is implemented in conformance with the Component Sound Library (CSL).

Input sq data is created with an SCE tool (smf2sq or JAM) and has a unique format that is used to represent playback sequences. Although sq data includes several types of data, modsesq supports only the SE sequence set format and the SE song format (currently undefined). Also, these formats are only supported when created with JAM.

An SE sequence set can have multiple SE sequences. An SE sequence describes the order in which sound effects will be played. For information about SE sequences, refer to the "Sound Data Format" document.

modsesq can perform multiple SE sequences simultaneously with one input port. Since modsesq can have more than one input buffer, it can also perform multiple sets of sq data simultaneously.

The SE stream data that is output by modsesq is the input data of the sound source module (hardware synthesizer). modsesq can have more than one output buffer and can drive multiple sound sources. An SE stream consists of SE messages. For information about SE messages, refer to the "CSL Overview" document.

Usage

Buffer Structure

Buffer Group 0: Input port group

Table 8-1:

BufCtx Index	Contents	Data structure
N*2	Input buffer of input port N	void[]
N*2+1	Environment of input port N	sceSESqEnv

Buffer Group 1: Output port group

Table 8-2:

BufCtx Index	Contents	Data structure
N	Output buffer of output port N	sceCslSeStream
		void [buffsize – sizeof (sceCslSeStream)]

An input buffer and its environment must exist as a pair. If either one is missing, an error will occur during initialization, and that buffer will be disabled.

Output Port Specification

The output of sq data that is played by modsesq can be specified for each SE sequence set by setting the environment.

Table 8-3

Member	Contents
defaultOutPort	Output port number when outPort is not specified
outPort[32]	The SE sequence set number, and the output port number for outputting the SE sequence that contains that SE sequence set.

When no outPort[0] output port has been specified (the port number is sceSESqEnv_NoOutPortAssignment) or when the SE sequence set number that contains the SE sequence is not in the outPort specification, that SE sequence is output to the defaultOutPort.

When the SE sequence set number is in the outPort specification, that SE sequence is output to the specified output port.

Function Overview

Volume, Panpot, and Timescale

The input port environment, overall SQ, SE sequence set, and SE sequence each have a volume, panpot, and timescale as attributes. The final SE stream is controlled with the value obtained by summing all of these attributes.

The SE sequence reading speed can be controlled by setting the timescale. Since time is managed in milliseconds for an SE sequence, the timescale is based on 1000 (realtime). As the value increases, the reading speed increases (if the timescale is 2000, the speed will be doubled). As the value decreases, the reading speed decreases (if the timescale is 500, the speed will be 1/2 as fast).

SE Sequence Sets, SE Sequences, and Sequence IDs

An SE sequence set is a grouping of multiple SE sequences.

An attribute (volume, panpot, or timescale) of an SE sequence set affects all SE sequences contained in that SE sequence set.

The final SE sequence is specified according to a combination of the SE sequence set number and SE sequence number. modeseq assigns and manages the combination of these two numbers as a single number called the sequence ID.

Up to 32 of these sequence IDs can be assigned simultaneously per input port. This sequence ID is used for specifying performances or expressing changes.

SE Messages Created by Playback of SE Sequences

SE sequences are successively generated by calls to sceSESq_Atick().

The ID that is kept by an SE message generated from an SE sequence has the format shown below.

Table 8-4

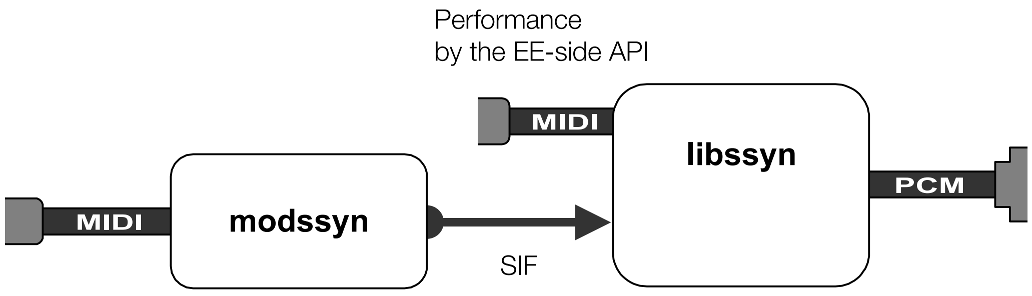
SE Message ID	Meaning
0xF1AABBCC	F1: sceSESqMessageIdPrefix: Currently 0xf1
	AA: Sequence ID assigned to the target SE sequence
	BB: Input port number in modsesq for which sq data is specified to which the target SE sequence belongs
	CC: Output port number in modsesq to which the target SE sequence is finally output

Chapter 9:
CSL Software Synthesizer (modssyn)

Overview

modssyn is a software synthesizer IOP module that is implemented to conform with the Component Sound Library (CSL) specification. modssyn works together with an EE-side library (libssyn) to interpret MIDI Stream data from the input buffer and generate sound using the software sound source. modssyn is responsible for sending MIDI Stream data from the input buffer to libssyn on the EE. For information about the functions and external specifications of the software synthesizer, please refer to the libssyn document.

Figure 9-1



Time Precision of a Performance

Generally, since software synthesis is performed on the EE in the intervals between other processing, the time precision during processing will be less than the corresponding precision on the IOP. Even if processing is performed every 1/240 second on the IOP, it is often the case that processing can only be performed every 1/60 second on the EE. To ensure that the precision of a performance is high even in these cases, time stamps are added to the data that is transferred from modssyn to libssyn. Since the waveforms are generated on the EE using those time stamps, the performance can have a time precision that corresponds to the precision of processing on the IOP.

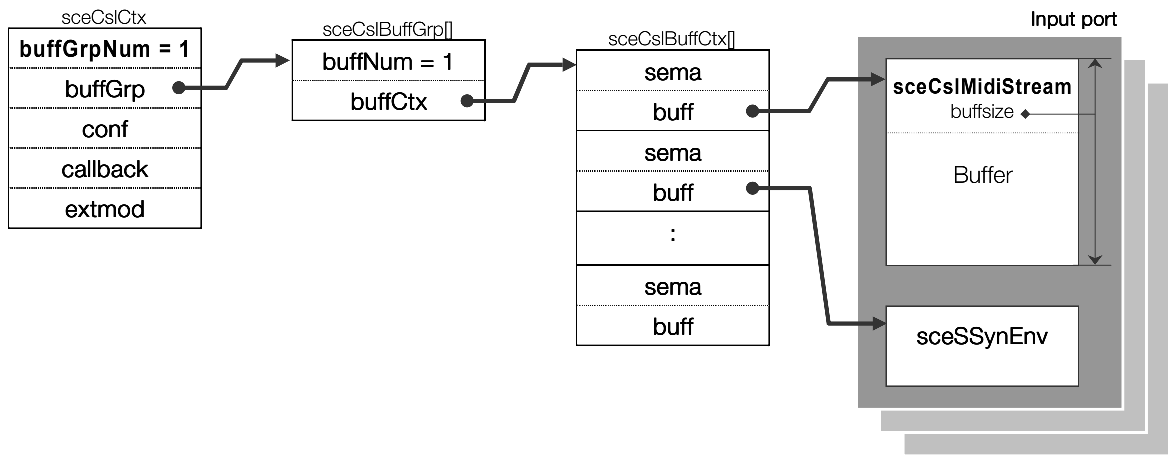
Usage Method

Buffer Structure

Table 9-1: Buffer Group 0: Input port group

BufCtx Index	Contents	Data Structure
N*2	Input port Nth input buffer	sceCslMidiStream void[buffsize-sizeof (sceCslMidiStream)]
N*2+1	Input port Nth environment	sceSSynEnv

Figure 9-2



The input buffers and their environments must exist as pairs. If one is missing, either an error will occur during initialization or that input buffer will become unusable.

The data buffer size (buffsize - sizeof (sceCslMidiStream)) must be an integral multiple of 16.

Chapter 10: SPU2 Waveform Data Encoding Module (spucodec)

Overview

SpuCodec is an IOP module for handling the encoding of waveform data for the SPU2. The waveform data is encoded with a function call from an application program, and the result is output to an area in IOP memory.

Functional Characteristics

Currently the only function provided by SpuCodec is encoding from 16-bit straight PCM data to waveform data that is supported by the SPU2. Options include a function for setting a loop point when encoding and a function for encoding by dividing the data into fixed areas or sizes to reduce overhead when encoding.

To use the function, the waveform data that is to be encoded should first be placed in an area in IOP memory, and the data area of the output destination should be reserved in IOP memory. The function should then be called from the application program while specifying information about both of these areas. Since the encoded output is in IOP memory, a suitable function such as one from the low-level sound library should be used to transfer the data to SPU2 local memory.