PlayStation®2 EE Library Overview
Release 2.4.3

Sound Libraries

Publication date: January 2002

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

Summary Table of Contents

About This Manual

This is the Runtime Library Release 2.4.3 version of the *PlayStation®2 EE Library Overview - Sound Libraries* manual.

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

Changes Since Last Release

Chapter 1: CSL Overview

- In the "SE Messages" section of "CSL Standards", the following descriptions have been added to “Voice Control : 0xb?”

  “Time-Panpot-Clockwise” and “Time-Panpot-Counterclockwise” commands

  Specifications for setting arrival volume, pitch, and panpot

Related Documentation

This manual should be read in conjunction with the *PlayStation®2 EE Library Reference* manuals which define all structures and functions.

Additionally, library specifications for the IOP can be found in the *PlayStation®2 IOP Library Reference* manuals and the *PlayStation®2 IOP 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)

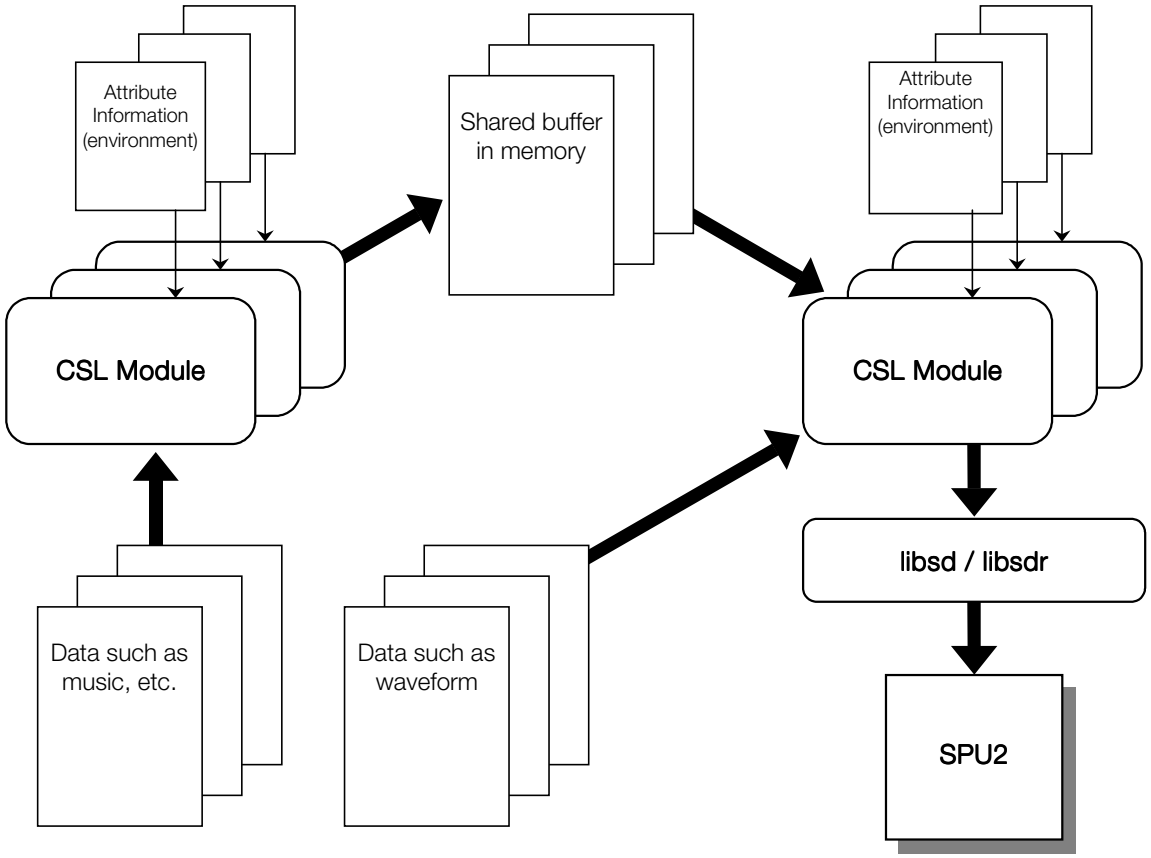
Chapter 1: CSL (Component Sound Libraries) Overview

CSL Overview

The PlayStation 2 sound library is designed around a component-based architecture. The architecture provides a flexible system for generating sound through software support.

The various sound generation processes and functions are divided into modules according to defined data delivery modes. The system is flexible because a given function can be replaced by a module with a different implementation, and new effects can be added simply by adding modules.

Figure 1-1



CSL is the generic name for the sound development environment that is based on this interface. Although CSL is highly flexible, it has a simple structure that is designed for realtime sound rendering. The component-based architecture design adds virtually no overhead to this simple structure.

A CSL module is a library having an interface that conforms to the CSL specification. A CSL module can be implemented for either the IOP or the EE. Although a CSL module is assumed to operate as a dynamic link library (.irx) on the IOP, a module can also be statically linked (.a). Static linking is always used on the EE.

CSL Standards

Rules related to CSL modules are described below.

CSL standardization concepts stress the importance of making stream data compatible between modules, while defining other aspects of the interface as loosely as possible, so that flexibility and performance are not lost.

For the convenience of user programs, the recommended specifications of the API and internal data structures are defined to provide a unified feel to CSL, but these are not compulsory, and specific details are left up to the creator of the CSL module.

A CSL module must satisfy the following three required conditions:

- It must have one module context structure per instance.
- It must conform to a Buffer Group structure.
- Sound stream data must conform to a category definition data structure.

Category Classification According to Input/Output (I/O)

CSL modules are classified into categories according to the combination of the input output data formats. The supported data formats are shown below.

Table 1-1

Data Format	Description
midi-stream	MIDI data only. No delta time.
pcm-stream	SPU-specified 16-bit PCM data. L/R interleaving every 512 bytes.
se-stream	SE stream. Structure for SE messages.
raw-stream	1 sample interleave or monaural 16-bit PCM data.
adpcm-stream	vag-format ADPCM data
original	Module-specific data
any	Any kind of data

For example, a module that inputs midi-stream data and outputs pcm-stream data would be in the "midi-pcm" category.

Common Data Structures

Some data structures defined in CSL are common to all categories and others are specific to a particular category. The common data structures are described first.

CSL Context Structure (sceCslCtx)

The CSL context structure mainly describes the structure of the I/O data buffers. A CSL module is defined so that it has one CSL context structure per instance.

```
Structure

  typedef struct {
  int           buffGrpNum;
  sceCslBuffGrp *buffGrp;
  void*         config;
  void*         callBack;
  char          *extmod[];
  } sceCslCtx;

Members

  buffGrpNum    Required:  Number of buffGrp arrays
  buffGrp       Required:  Pointer to buffGrp array
  config        Optional:  Structure for settings related to the entire
                           module
  callBack      Optional:  Callback function for asynchronous execution
  extmod        Optional:  Name of the module when directly accessing an
                           external module
```

CSL Buffer Group Structure (sceCslBuffGrp)

The CSL buffer group structure is used to gather together I/O data buffers and manage them as a group. A CSL module is defined so that it has data buffers that conform to this format.

```
Structure

  typedef struct {
  int           buffNum;
  sceCslBuffCtx *buffCtx[n];
  } sceCslBuffGrp;

Members

  buffNum       Required:  Number of buffers belonging to that group
  buffCtx       Required:  Pointer to buffer array
```

CSL Buffer Context Structure (sceCslBuffCtx)

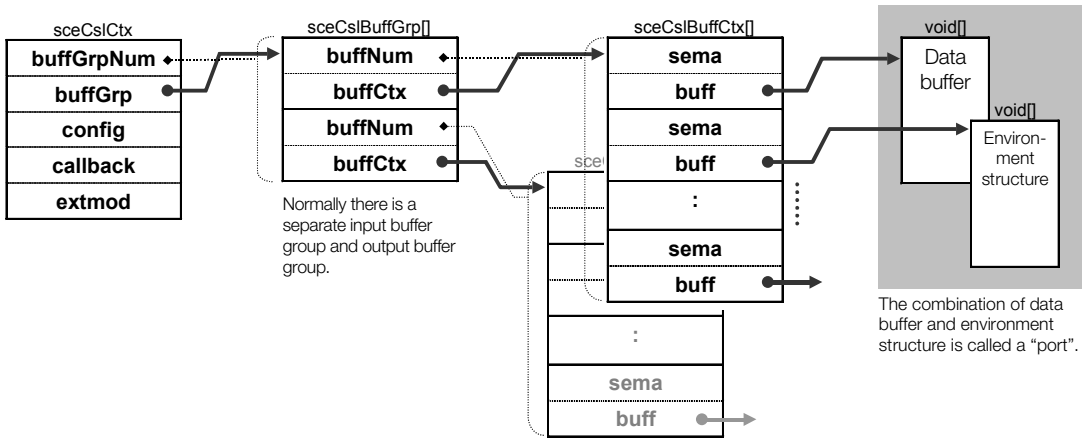
CSL buffer context structures are associated in a one-to-one correspondence with individual I/O data buffers.

```
Structure

  typedef struct {
  int    sema;
  void   *buff;
  } sceCslBuffCtx;

Members

  sema          Optional:  Valid semaphore for that buffer
  buff          Required:  Pointer to buffer
```

Buffer Group Structure

The data buffer structures used by a module are described by the CSL context structure, CSL buffer group structure, and CSL buffer context structure. Multiple groups are configured to have multiple buffers each as shown in the figure.

Figure 1-2



A CSL module can have a structure for describing data attributes at the same position as the input buffer or output buffer. This is called an environment structure (env), and a set consisting of data together with the environment structure is called a port.

Required Specification: Buffer Group arrangement

If a module has an input buffer group, the input buffer group should be assigned Group 0. If a module also has an output buffer group, the output buffer group should be assigned Group 1.

If the module does not have an input buffer, the output buffer group should be assigned Group 0.

Required Specification: Buffer arrangement

Arrange CSL buffer context structures in the input buffer array or output buffer array by individual ports. For example, if A and B are two sets of input data with environment structures, arrange the buffers such that buffer 0 is port A data, buffer 1 is port A's environment structure, buffer 2 is port B data, and buffer 3 is port B's environment structure.

When a module contains only data and no environment structure, simply arrange the data buffers sequentially.

Recommended Specifications: Keeping control information

Keeping control information that is needed for module operation as data associated with the Buffer Group structure is recommended whenever possible. Maintaining it within the module as a global variable or something similar, should be minimized.

It is recommended that control information related to each I/O port be collected together in the environment structure for that port. For control information that applies to the entire module (not just individual ports), it is recommended that this information be kept in a structure that is associated with the config member of the CSL context structure. All of these formats are arbitrary.

Recommended Specification: Memory allocation

It is recommended that memory allocation for I/O buffers, control information, or Buffer Group structures be done by the module and ensured by the user application. Since it is safe to entrust memory management to the user, dynamic allocation within the module should be avoided.

Category Definition Data Structures

This section explains data structures that are defined for each category (I/O format). The data structure for "original" can be arbitrarily defined.

midi-stream buffer structure (sceCslMidiStream)

```
Structure

  typedef struct {
   u_int  buffsize;           // include Header size
   u_int  validsize;          // valid data size
   u_char data[0];            // data max is data[buffsize]
  } sceCslMidiStream;

Members

  buffsize      Required:  Size of entire buffer
  validsize     Required:  Size of valid data
  data          Required:  midi data
```

This is a variable length structure in which buffsize and validsize are headers, and are followed by MIDI data. validsize is used to successively record the position of data that the module has processed.

Although MIDI messages themselves are entered for data, there is no delta time. Also, running time is not supported. CSL-specific extended MIDI messages are defined (these are described later). Support for extended MIDI messages is not required, but it is recommended.

se-stream buffer structure (sceCslSeStream)

```
Structure

  typedef struct {
   u_int  buffsize;           // include Header size
   u_int  validsize;          // valid data size
   u_char data[0];            // data max is data[buffsize]
  } sceCslMidiStream;

Members

  buffsize            Required:  Size of entire buffer
  validsize           Required:  Size of valid data
  data                Required:  SE message data
```

This is a variable length structure in which buffsize and validsize are headers, and are followed by SE message data. validsize is used to successively record the position of data that the module has processed.

The SE message (details given later) is stored in the data member. Support for SE messages is not required but is a recommended specification.

pcm-stream buffer structure (sceCslPcmStream)

```
Structure

  typedef struct {
   u_int  pcmbuf_size;
   u_int  validsize;
   void   *pcmbuf;
   u_int  pad;
  } sceCslPcmStream;

Members

  buffsize      Required:  Size of pcmbuf
  validsize     Required:  Size of valid data within pcmbuf
  pcmbuf        Required:  Pointer to pcmbuf
  pad           Unused
```

This structure differs from midi-stream in that the actual data is in an area that is pointed to by pcmbuf. validsize is used to successively record the position of data that the module has processed.

The data, which is 16-bit, signed, little endian PCM data, is structured so that L and R are interleaved every 512 bytes.

raw-stream structure

This is currently undefined (under investigation).

adpcm-stream structure

This is currently undefined (under investigation).

Extended MIDI Messages

MIDI messages, devised for music, have the following problems when controlling sound effects.

- Since generated voices are managed according to a channel number and key number only, key off operations cannot be performed separately by consecutively striking the same key.
- Panpot and other controls are performed for all channels.

Extended MIDI messages, which were uniquely defined by SCE to address these problems, are described below. These extended MIDI messages are used only by CSL and do not obey MIDI rules. Therefore, they should not be passed to a general MIDI device via the MIDI interface.

Pre Voice Control Message

Immediately after this message, the Expression, Panpot, and PitchBend will be set, but only for the generated voice. The message will have no effect on other voices. The Pre Voice Control Message consists of the following five bytes.

Figure 1-3

0xF9	op. code	ch.	1st data	2nd data

op. code, 1st data, and 2nd data

Specify these bytes as follows, according to the content of the message.

Table 1-2

Contents	op. code	1st data	2nd data
Expression	0x00	0-127	Invalid
Panpot	0x01	0-127	Invalid
PitchBend	0x02	LSB (0-127)	MSB (0-127)

ch.

Specify the channel in the low-order 4 bits.

Voice Control Message

This message controls a voice according to a key number and ID number. It enables more flexible control over the voice.

The Voice Control Message consists of the following seven bytes.

Figure 1-4

0xFD	op. code	mode & ch.	key	id	1st data	2nd data

op. code, 1st data, and 2nd data

Specify these bytes as follows according to the contents of the message.

Table 1-3

Contents	op. code	1st data	2nd data
Expression	0x00	0-127	Invalid
Panpot	0x01	0-127	Invalid
PitchBend	0x02	LSB (0-127)	MSB (0-127)
Note Control (Note On/Off)	0x10	velocity (0-127) If 0 is specified, this becomes a Note Off message	Invalid

mode and ch.

Specify channel according to bits 0-3 and mode according to bits 4-6.

If bit4=1, the key field is ignored, and channels for all voices are targeted, regardless of the key number. Always specify 0 for this bit for a Note Control Message.

key

Specify the key number (0-127).

id

Specify the ID number (0-126 or 127). If 127 is specified, all voices are targeted regardless of the ID number. Do not specify 127 for a Note Control Message.

SE Messages

An SE message is a data format that is provided for controlling sound effects. Its purpose is to control parameters in realtime from the start of sound generation to the end. An ID is used to identify a voice for an individual control, at the time sound is generated.

An SE message consists of the following byte string.

Table 1-4

Value	Contents
0xfe	Message Prefix
version	Version: Currently, 0x01
id0	ID: 32 bit / little endian: bit 0 - 7
id1	bit 8 - 15
id2	bit 16 - 23
id3	bit 24 - 31
status	SE status: bit 7 is always 1
data	SE data: length depends on SE status

Message Prefix

Indicates the beginning of the SE message.

Assigned the value 0xfe.

Version

Indicates the version of the SE message.

Assigned the value 0x01.

ID

Distinguishes SE messages. Data organization is 32-bit little endian.

The value of ID can range from 0x10000000 to 0xefffffff. The ranges from 0x00000000 to 0x0fffffff and 0xf0000000 to 0xffffffff are reserved for use by SCE modules.

SE Status

Indicates the contents of the SE message.

Currently, includes the following items.

Table 1-5

Contents	value	Number of consecutive data
Note on/off	0xa?	6
Voice Control	0xb?	Depends on individual command
Voice Group Control	0xc0	Depends on individual command

Note on/off : 0xa?

Status indicating the starting or ending of a sound. The low-order 4 bits specify the bank number for that port.

This SE message consists of the following 7 bytes, including the status.

Table 1-6

Value	Contents
0xa?	SE status: note on/off + bank number
prog_num	Program number (0x00 - 0x7f)
note_num	Note number (0x00 - 0x7f)
velocity	Velocity (key strike intensity) (0x00 - 0x7f, where 0x00 is reserved for note off)
panpot	Panpot (-127 - +127, where a number < 0 means inverted phase)
pitch_LSB	Pitch specification LSB (0x00 - 0x7f)
pitch_MSB	Pitch specification MSB (0x00 - 0x7f)

- Waveform data for which the sound is ultimately generated is determined by the bank number, program number, and note number (the decision rules depend on the format of the waveform data).
- Sound is generated when the specified velocity is not equal to 0x00. Sound is silenced when the specified velocity is equal to 0x00.
- Stereo balance is determined with the panpot. +64 is for the positive phase sector. When a negative value is specified, the phase will be inverted.
- The actual musical interval that is generated is determined by the specified note number and the attributes of the waveform data.
- The actual musical interval that is generated can be directly specified by specifying the pitch. The value used for pitch is equivalent to that for the SPU2 hardware. The 14-bit range (0x0000 - 0x3fff) is specified 7 bits at a time. If the musical interval that is determined by the specified note number and the attributes of the waveform data is given precedence and there is no pitch specification, both the LSB and MSB of the pitch specification will be 0x00.

Voice Control : 0xb?

Status for controlling the voice of the generated sound. This status, which includes status, consists of the following byte string.

Table 1-7

Value	Contents
0xb?	SE status: Voice Control + bank number
prog_num	Program number (0x00 - 0x7f)
note_num	Note number (0x00 - 0x7f)
command	Voice Control command
data	Voice Control data: length depends on the Voice Control command

Currently, the following commands are implemented.

Table 1-8

Command	Value	No. of consecutive data	Consecutive data
Time-Volume	0x07	3 or more	Time, arrival volume (0-0x80), 0
Time-Panpot-Clockwise	0x0c	3 or more	Time, arrival panpot (0-0xff), 0
Time-Panpot-Counterclockwise	0x0d	3 or more	Time, arrival panpot (0-0xff), 0
Time-Pitch+	0x0e	3 or more	Time, arrival pitch LSB (0-0xff), Arrival pitch MSB (0-0xff)
Time-Pitch-	0x0f	3 or more	Time, arrival pitch LSB (0-0xff), Arrival pitch MSB (0-0xff)
Pitch LFO+ Depth	0x10	2	Depth LSB (0 - 0xff), Depth MSB (0 - 0xff)
Pitch LFO- Depth	0x11	2	Depth LSB (0 - 0xff), Depth MSB (0 - 0xff)
Pitch LFO Cycle	0x12	2	Cycle LSB (0 - 0xff), Cycle MSB (0 - 0xff)
Amp. LFO+ Depth	0x20	2	Depth (-0x80 - 0x7f), 0
Amp. LFO- Depth	0x21	2	Depth (-0x80 - 0x7f), 0
Amp. LFO Cycle	0x22	2	Cycle LSB (0 - 0xff), Cycle MSB (0 - 0xff)

Time is in units of milliseconds, and is represented in delta time format for standard MIDI files.

The arrival volume is the value that is ultimately reached when the upper limit is set to 0x80.

For the arrival panpot, +64 is the positive phase center. When a negative value is specified using two's complement, the phase will become inverted. If a specification is made so that the phase is inverted while a change is in progress, the result will become folded back using the same phase.

The arrival pitch is a relative value from the current pitch with units of 1/128 of a half-step.

Pitch LFO depth is obtained by dividing a signed 16-bit value from -0x8000 to 0x7fff into two unsigned bytes.

Cycle is obtained by dividing an unsigned 16-bit value from 0 to 0xffff into two unsigned bytes

For details, please refer to the "Sound Data Formats" document.

Voice Group Control : 0xc?

Status for controlling the voices of generated sounds which have the same ID. This status, which includes "status", consists of the following byte string.

Table 1-9

Value	Contents
0xc?	SE status: Voice Group Control + currently specified as 0x0 only
command	Voice Group Control command
data	Voice Control data: length depends on the Voice Group Control command

Currently, the following command is implemented.

Table 1-10

Command	value	No. of consecutive data	Consecutive data
All Note off mask	0x1e	8	Target base ID (0x00000000-0xffffffff), mask (0x00000000-0xffffffff)
All Note off	0x1f	0	----

All Note off mask performs Note Off processing for all voices for which the result of the bitwise AND of the specified target base ID and the mask is the same as the result of the bitwise AND of the ID held by the voice, and the mask. Both data sequences are 32-bit little endian.

All Note off performs Note Off processing for all voices that have the same ID.

Recommended API Format

There are no required rules for the APIs of a CSL module. An API can be freely implemented to suit the needs of the module. However, since all information required for module operation can be described in the CSL context structure, including a pointer to the CSL context structure as an API argument is recommended.

Also, implementing the following two APIs according to the format shown below is particularly recommended.

Recommended Specification: int vendername???_Init(sceSdModCtx *module_ctx, u_int interval)

```
Arguments:    module_ctx    Pointer to CSL context structure
interval      1 Tick interval (microseconds)
```

This API initializes the module's internal environment. It initializes the environment structure that is passed from the user program.

A Tick, which is the time interval between successive calls of the ATick function shown below, is specified by the user program.

Recommended Specification: int vendername???_ATick(sceSdModCtx *module_context)

```
Argument:     module_ctx    Pointer to CSL context structure
```

This API is routinely called by the user program at fixed intervals. When received, the module performs sound processing for 1 Tick. The user program is responsible for routinely calling this API.

Low Level Library

To enable a CSL module to access the SPU2, use the low level library libsd (for the IOP). If the libsd register wrapper API is used, the publicly available SPU registers can practically be accessed directly.

For a CSL module implemented on the EE, use libsdr to remotely call libsd. An API for batch processing register wrapper APIs is provided for performing remote access efficiently.

Using CSL

SCE-provided CSL Modules

SCE plans to provide the following CSL modules. The Category column indicates the I/O data formats.

Table 1-11: EE

Module	Category	Function
MIDI stream generation	Original -> MIDI	Module for generating a MIDI stream from a program
Software synthesizer	MIDI -> PCM	Multifunction Wave Table sound source module that performs calculations on the EE
Lineout	PCM -> Original	Module that generates sounds by sending a PCM stream to the SPU2
SE stream generation	Orignal -> SE	Module for generating an SE stream from a program

Table 1-12: IOP

Module	Category	Function
MIDI stream generation	Original -> MIDI	Module for generating a MIDI stream from a program
SE stream generation	Original -> SE	Module for generating a SE stream from a program
MIDI sequencer	Original -> MIDI	Sequencer module for reading an sq file and outputting a MIDI stream
Sound effect sequencer	Original -> SE	Sequencer module specifically for sound effects (not provided)
MIDI delay	MIDI -> MIDI	Latency adjustment module

Module	Category	Function
MIDI monophonic	MIDI -> MIDI	Module for assigning a MIDI stream as monophonic
Hardware synthesizer	MIDI -> Original	Sound source module for using SPU2 voices

Sample Programs

Sample programs that use CSL modules are listed below for reference.

sce/iop/sample/sound/sqhard

This program uses the following modules to perform a musical composition. This is the most basic example of CSL use.

a. MIDI sequencer (modmidi) [IOP]
b. Hardware synthesizer (modhsyn) [IOP]

sce/iop/sample/sound/ezmidi

This program uses the following modules to perform a musical composition and to generate sound effects from the EE.

a. MIDI sequencer (modmidi) [IOP]
b. MIDI stream generation (modmsin) [EE]
c. Hardware synthesizer (modhsyn) [IOP]

sce/iop/sample/sound/sqsoft

This program uses the following modules to perform a musical composition. The performance is executed by the software synthesizer on the EE, and music is output as PCM data, then transferred from the EE to the IOP and on to the SPU2.

a. MIDI sequencer (modmidi) [IOP]
b. Software synthesizer (modssyn) [EE/IOP]
c. Lineout (liblout) [EE]

sce/iop/sample/sound/sebasic

This program uses the following modules to start and stop sound effects. This is the most basic example of how to use SE modules.

a. SE stream generation (modsein) [IOP]
b. Hardware synthesizer (modhsyn) [IOP]

sce/iop/sample/sound/semidi

This program uses the following modules to start and stop sound effects while performing a MIDI chunk within an SQ file.

a. MIDI sequencer (modmidi) [IOP]
b. SE stream generation (modsein) [IOP]
c. Hardware synthesizer (modhsyn) [IOP]

Creator Authoring Environment

JAM

JAM is provided as an authoring tool that supports SCE-provided CSL modules.

JAM, which runs on a Macintosh, can perform integrated waveform conversion, bank editing, MIDI conversion, and preview operations. It supports both hardware and software synthesizers.

For more information, please refer to the documentation provided with JAM.

Chapter 2:
CSL Line-out Library

Library Overview

liblout is a library that outputs PCM Streams from the EE to the SPU2 through libsdr. The library conforms to CSL (Component Sound Library).

Up to four input ports can be processed, and these ports can be freely assigned to Left or Right channel 0 or 1 of SPU2.

Usage

Buffer Structure

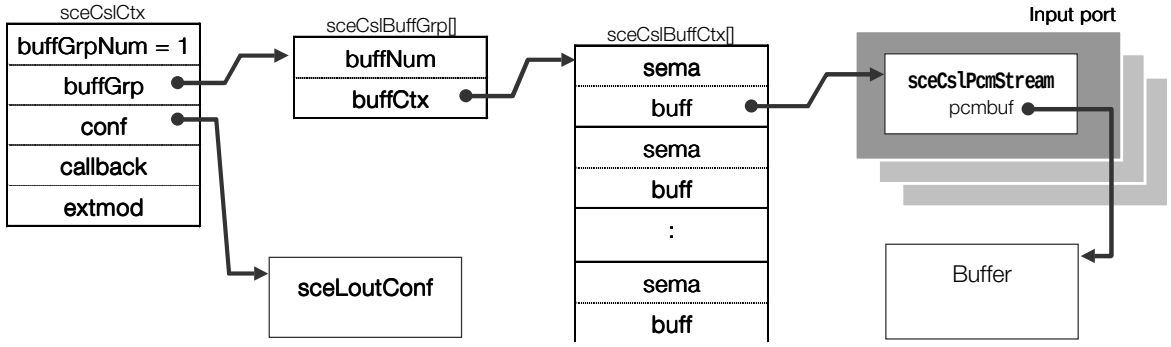
Buffer Group 0: input port group

Table 2-1

BufCtx Index	Contents	Data Structure
N	Input port Nth Input buffer	sceCslPcmStream

The range of N is 0 - 3.

Figure 2-1



Alignment, etc. must be performed for areas indicated by sceCslPcmStream and pcmbuf, since non-cached accesses will occur.

Initialization

To set up librsd and allocate IOP memory, the following initialization steps must be performed before calling sceLout_Init():

```
sceSdRemoteInit();
sceSdRemote(1, rSdInit);
sceSifInitIopHeap();
```

Chapter 3:
CSL MIDI Stream Creation

Library Overview

libmsin is an EE library that conforms to the CSL (Component Sound Library). libmsin creates and outputs MIDI streams in response to API calls from an application program. Unlike typical CSL modules, this library works using settings made through an API. There is no ATick() function.

libmsin is compatible with extended MIDI messages defined by SCE for sound effects.

Usage

Buffer Structure

Buffer Group 0: (None)

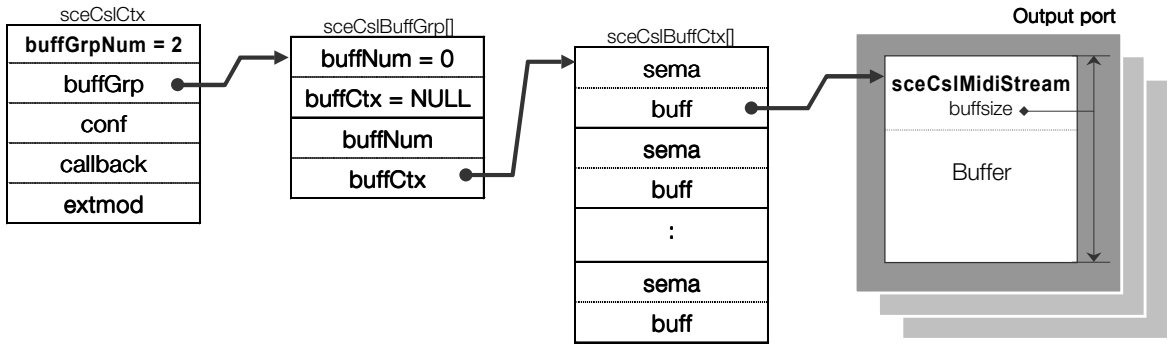
Buffer Group 1: Output port group

Table 3-1

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

The range of N is 0 - 15.

Figure 3-1



Packing MIDI Messages

In libmsin, 2-3 byte MIDI messages are packed into 32-bit words, then output. This is accomplished as follows: first, a macro such as sceMSIn_MakeMsg() is used to prepare a packed MIDI message, then the sceMSIn_PutMsg() function is used to write the message to an output stream.

Extended MIDI Messages

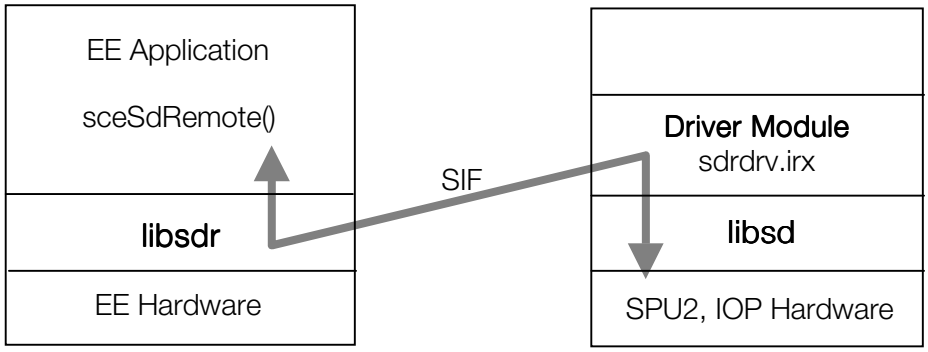
In libmsin, the sceMSIn_PutHsMsg() function can be used to embed extended MIDI messages defined by SCE for sound effects into the output stream. A macro is also provided to prepare the different parameters used by an extended MIDI message. For information on the contents of extended MIDI messages, please see the "CSL Overview" document.

Chapter 4:
Low-Level Sound Library

Library Overview

libsdr is a low-level sound library that allows the EE to remotely control the SPU2, which is connected to the IOP. The libsdr routines work together with the driver module on the IOP side and provide the EE with features similar to the IOP low-level sound driver library libsd. Most of the processing takes place in the IOP driver module. On the EE side, non-blocking operations, where a command is issued and then control returns immediately, are supported.

Figure 4-1



Related Files

The following files are required to use libsdr:

Table 4-1

Category	Filename
Library	libsdr.a
Header file	libsdr.h
Run-time module (IOP-side)	sdrdrv.irx libsd.irx

Description of Features

List of libsd Functions

The following is a list of libsd functions that can be called through libsdr.

Please refer to the description of sceSdRemote() for how these functions are called and to the libsd document for details on individual functions.

Table 4-2: libsd functions

libsd Function Name	Description
sceSdInit	Initialize sound device
sceSdSetParam	Write to basic parameter register
sceSdGetParam	Read from basic parameter register
sceSdSetSwitch	Write to voice control parameter register
sceSdGetSwitch	Read from voice control parameter
sceSdSetAddr	Write to address register
sceSdGetAddr	Read from address register
sceSdSetCoreAttr	Write to core setting pseudo-register
sceSdGetCoreAttr	Read from core setting pseudo-register
sceSdProcBatch	Batch process
sceSdProcBatchEx	Batch process with batch voice processing
sceSdVoiceTrans	Transfer to SPU2 local memory
sceSdBlockTrans	Transfer to input/output block
sceSdVoiceTransStatus	Get voice transfer status
sceSdBlockTransStatus	Get input/output block transfer status
sceSdSetTransCallback	Set up interrupt callback for transfer
sceSdSetIRQCallback	Set up IRQ interrupt callback
sceSdSetEffectAttr	Set up effect attributes
sceSdGetEffectAttr	Get effect attributes
sceSdClearEffectWorkArea	Clear effect work area
sceSdNote2Pitch	Convert from note to pitch
sceSdPitch2Note	Convert from pitch to note

List of Batch Commands

A series of batch commands can be sent all at once to the IOP for execution using sceSdProcBatch() and sceSdProcBatchEx(). The following is a list of batch commands. For details, refer to the libsd document.

Table 4-3: Batch commands

Batch Command	Description
SD_BSET_PARAM	Run sceSdSetParam()
SD_BGET_PARAM	Run sceSdGetParam()
SD_BSET_SWITCH	Run sceSdSetSwitch()
SD_BGET_SWITCH	Run sceSdGetSwitch()
SD_BSET_ADDR	Run sceSdSetAddr()
SD_BGET_ADDR	Run sceSdGetAddr()
SD_BSET_CORE	Run sceSdSetCoreAttr()
SD_BGET_CORE	Run sceSdGetCoreAttr()
SD_WRITE_IOP	Write to IOP memory
SD_WRITE_EE	Write to EE memory
SD_RETURN_EE	Transfer returns to EE memory

List of Register Macros

In libsdr, SPU2 registers are specified using register macros defined in libsd. The following is a list of register macros. For more information, refer to the libsd and SPU2 documents.

Table 4-4: Basic parameter register macros

Register macro	Core	Voice	Description
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_??	Playing pitch
SD_VP_ADSR1	SD_CORE_?	SD_VOICE_??	Envelope
SD_VP_ADSR2	SD_CORE_?	SD_VOICE_??	Envelope (2)
SD_VP_ENVX	SD_CORE_?	SD_VOICE_??	Current envelope value
SD_VP_VOLXL	SD_CORE_?	SD_VOICE_??	Current volume (left)
SD_VP_VOLXR	SD_CORE_?	SD_VOICE_??	Current volume (right)
SD_P_MMIX	SD_CORE_?	---	Output 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_?	---	Current master volume (left)
SD_P_MVOLXR	SD_CORE_?	---	Current master volume (right)

Table 4-5: Voice management parameter register macros

Register Macro	Core	Voice	Description
SD_S_PMON	SD_CORE_?	---	Set pitch modulation
SD_S_NON	SD_CORE_?	---	Assign noise generator
SD_S_KON	SD_CORE_?	---	Key on (start voice playback)
SD_S_KOFF	SD_CORE_?	---	Key off (end voice playback)
SD_S_ENDX	SD_CORE_?	---	Endpoint passed flag
SD_S_VMIXL	SD_CORE_?	---	Voice output mixing (Dry left)
SD_S_VMIXR	SD_CORE_?	---	Voice output mixing (Dry right)
SD_S_VMIXEL	SD_CORE_?	---	Voice output mixing (Wet left)
SD_S_VMIXER	SD_CORE_?	---	Voice output mixing (Wet right)

Table 4-6: Address register macros

Register macro	Core	Voice	Description
SD_VA_SSA	SD_CORE_?	SD_VOICE_??	Start address of wave data
SD_VA_LSAX	SD_CORE_?	SD_VOICE_??	Loop point address
SD_VA_NAX	SD_CORE_?	SD_VOICE_??	Address of next wave data to be read
SD_A_ESA	SD_CORE_?	---	Start address of effect processing work area
SD_A_EEA	SD_CORE_?	---	End address of effect processing work area
SD_A_TSA	SD_CORE_?	---	Transfer start address
SD_A_IRQA	SD_CORE_?	---	Interrupt address

Table 4-7: Entry (pseudo-register macros)

Macro	Core	Voice	Description
SD_C_EFFECT_ENABLE	SD_CORE_?	---	Effect area write permission
SD_C_IRQ_ENABLE	SD_CORE_?	---	Enable IRQ interrupts
SD_C_MUTE_ENABLE	SD_CORE_?	---	Mute
SD_C_NOISE_CLK	SD_CORE_?	---	Noise generator M-series shift frequency
SD_C_SPDIF_MODE	---	---	SPDIF setting (mask)

Changing the Priority of a Thread

The priorities of the sdrdrv.irx driver module threads running on the IOP can be changed. The following three methods are available:

- Modify the source code
- Specify the priority at module load time
- Change the priority during execution

To modify the source code, change the value specified for param.initPriority in the thread creation sections of sdd_main.c (main thread) and sdd_com.c (callback thread) in the sdrdrv source code.

To specify the priorities at module load time, pass the thread priorities as arguments to sceSifLoadModule() in the following manner.

```
char arg[] = "thpri=24,24";
sceSifLoadModule( "host0:/usr/local/sce/iop/modules/sdrdrv.irx",
strlen( arg ) + 1, arg );
```

There are two thread priorities in sdrdrv.irx--the first is the priority used for the main thread, and the second is the priority used for the callback thread. These values are both 24 by default. The priority value of the callback thread must be greater than or equal to the priority value of the main thread (i.e., the callback thread must have a lower priority than the main thread).

To change the priorities during execution, use sceSdRemote() to call rSdChangeThreadPriority. This also involves specifying the two arguments in the same order as in the case when the priorities are changed at module load time. Both priorities will be changed simultaneously. For more information, refer to the function reference.

When changing the priority of a thread, care must be taken with regard to priorities of other modules. It is not recommended that the priority of IOP threads be changed casually.

Notes

Non-blocking Operations and Callbacks

sceSdRemote(), which calls libsd functions, can specify a blocking or non-blocking operation as a parameter. For blocking operations, sceSdRemote() issues a command to the IOP driver module, waits for the operation to finish, and returns the result as the return value. For non-blocking operations, sceSdRemote() issues the command but does not wait for the IOP operations to finish and returns immediately with a return value of 0.

Problems may occur if a command is sent before the previous IOP operation is finished. For non-blocking operations, the completion of the IOP operation can be determined by setting up a terminating callback function using sceSdRemoteCallBack() or by polling using the SD_WRITE_EE and the SD_RETURN_EE batch commands. Use of a terminating callback may result in context switching, so polling is more effective if EE performance is to be emphasized.

Calling from Multiple Threads (Re-entry)

libsdr is not re-entrant. If multiple threads call libsdr functions asynchronously, care must be taken to prevent overlapping calls.

RPC re-entry issues are discussed in the "SIF System" document. Please use this as a reference for how re-entry takes place and how to avoid it.

Memory Alignment in Transferred Data

When using sceSdRemote() to set up or get effect attributes or to execute batch commands (SD_WRITE_EE / SD_RETURN_EE) accompanying EE memory writes, SIF DMA transfers will be used internally by libsdr. Thus, the following general issues relating to DMA transfers must be kept in mind.

First, note alignment in the memory being transferred. The attribute setting in the compiler must be set to provide 16-byte alignment on the EE and 4-byte alignment on the IOP.

Also keep consistency issues between the main memory and cache in mind. The EE has no bus-snooping functions, so if a DMA transfer takes place from the IOP to the EE, then the EE accesses another variable over the same cache line, the region that was DMA-transferred could be overwritten by a write-back. To prevent this, make sure to consider memory alignment. The size of the EE cache line is 64 bytes, so using 64-byte alignment will be safe (although memory use will be less efficient).

Chapter 5: Standard Kit Library/ Sound System Overview

Library Overview

The standard kit library (libsk) is a utility library whose main function is to allow the environment on the IOP to be easily controlled from the EE.

The various functions of libsk work together with support modules that are provided as standard modules on the IOP, and provide IOP functions to EE applications, thereby enabling IOP functions to be used without requiring any IOP-side programming. Currently, these functions are provided only for the sound environment (sound system).

The functions provided by the standard kit/sound system library (SKSS) include functions for performing score data, generating / muting one-shot sounds, and simplified versions of functions for configuring the SPU2 environment and transferring waveform data. Furthermore, since the low-level sound libraries (libsdr/sdrdrv) are also used in the internal implementation, libsdr functions can also be used in an auxiliary capacity.

Related Files

The following files are required to use libsk. Some of these files may not be needed depending on the functions that are used by the IOP runtime module (see section 2.3).

Table 5-1

Category	Filename
Library	libsk.a
Header file	sk/sk.h
	sk/common.h
	sk/sound.h
Runtime module	sk/sksound.irx
(IOP side)	sk/skhsynth.irx
	sk/skmidi.irx
	sk/sksesq.irx
	sk/skmsin.irx

In addition to these files, -lsdr must be specified as a link-time option. Also, during execution, the CSL modules (mod???.irx) that are required by sdrdrv.irx as well as each sk???.irx must be loaded on the IOP (see section 2.3).

Sample Programs

The following sample programs use the standard kit/sound system library.

- **sce/ee/sample/sk/playsq/**

  These samples show basic usage of the standard kit/sound system library functions.
- **sce/ee/sample/sk/ctrlsq/**

  These samples call standard kit/sound system library functions and use controller instructions to control the sound performance.

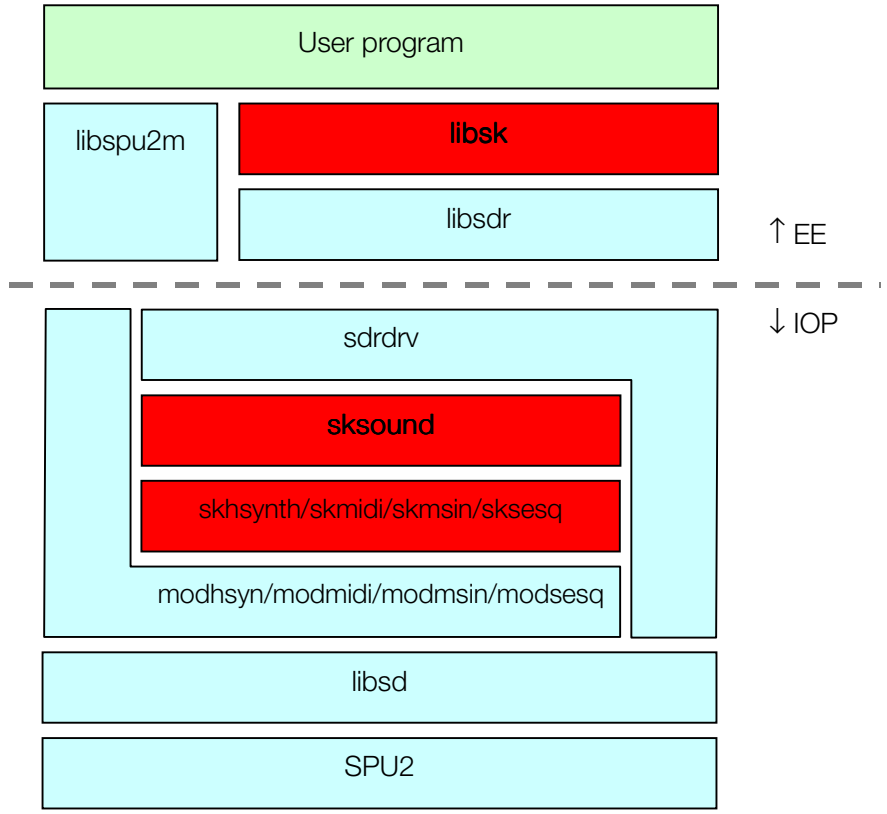
Description of Standard Kit/Sound System Library Functions

Library Configuration

A software hierarchy diagram of the standard kit/sound system library is shown below together with existing library modules.

The software that is included in the standard kit/sound system library is shown in red.

Figure 5-1



libsk	Standard kit
libspu2m	SPU2 local memory management
libsdr	Low-level sound library/EE
sdrdrv	Low-level sound driver
sksound	Standard kit/sound system/sound basic
skhsynth	Standard kit/sound system/hardware synthesizer
skmidi	Standard kit/sound system/MIDI sequencer
skmsin	Standard kit/sound system/MIDI stream generation
sksesq	Standard kit/sound system/sound effect sequencer
modhsyn	CSL hardware synthesizer
modmidi	CSL MIDI synthesizer
modmsin	CSL MIDI stream generation
modsesq	CSL sound effect sequencer
libsd	Low level sound library/IOP

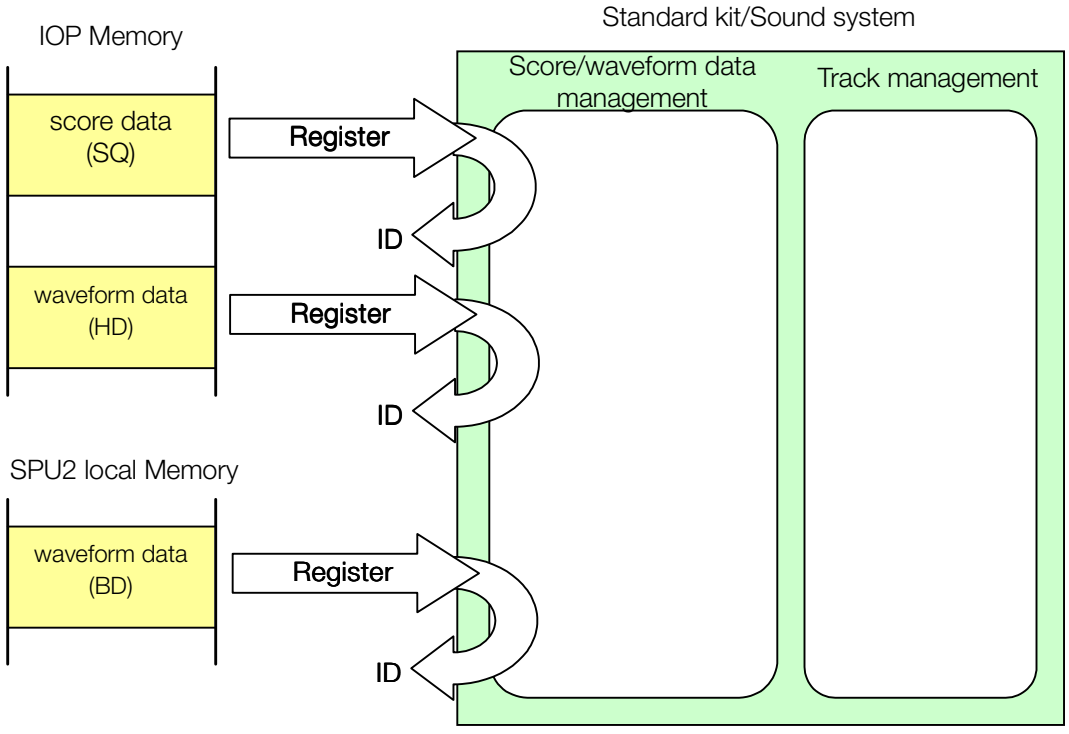
All programming for the standard kit/sound system is performed on the EE. The IOP-side software consists of modules which require dynamic linking. When an application is launched, the required modules are selected and loaded into memory.

Software Architecture

To perform sound processing, the system must be directed to bind score and waveform data to the actual performance of that data. To control score and waveform data, registration (BIND) processing is performed using information on the score and waveform data (the address and size of the area where the data is located), and an ID representing that information is returned. The data can subsequently be identified by specifying the registered ID.

Score data (SQ data) must be placed in IOP memory. For waveform data, attribute information data (HD data) must be loaded in IOP memory and phoneme data (BD data) must be placed in SPU2 memory. The addresses are also used to represent the addresses for each of the spaces. For the waveform data ID, the same ID number is assigned to both the attribute information data (HD data) and phoneme data (BD data), which are grouped together. Although score data isn't needed when a one-shot sound is generated, registration processing is still performed so that an ID can be obtained to provide the required management.

Figure 5-2

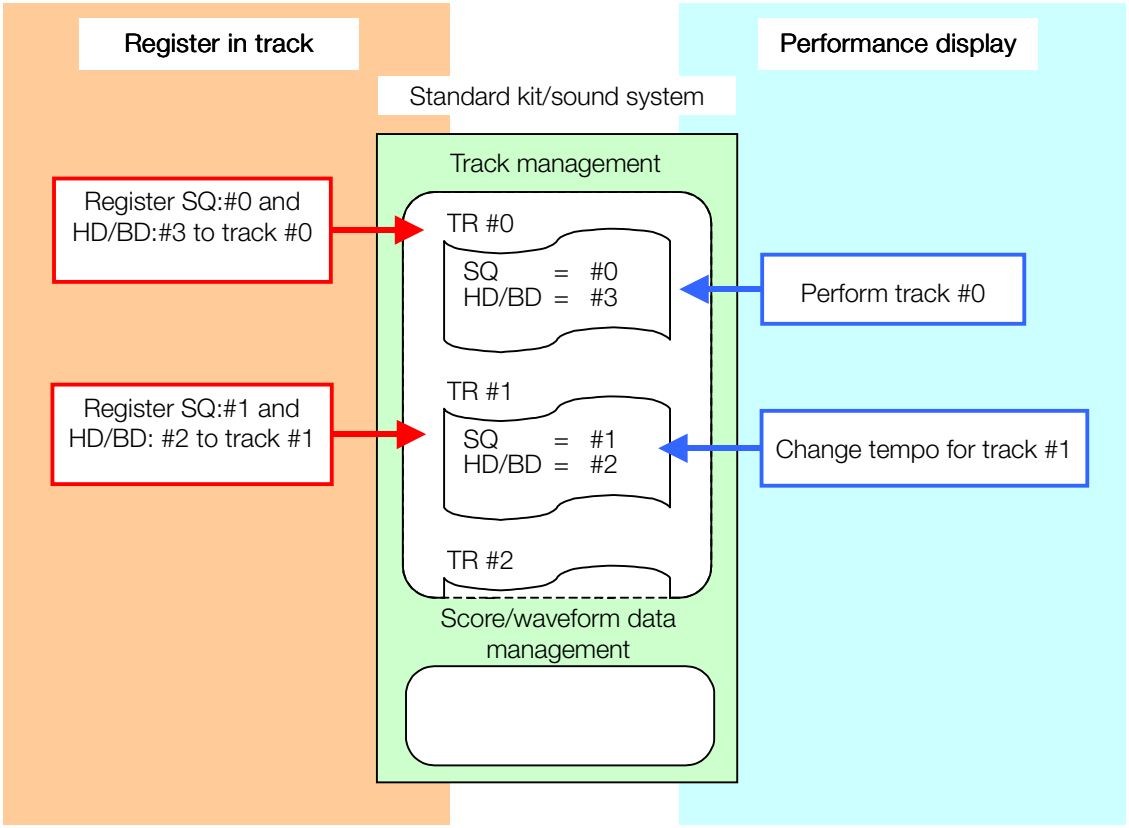


To represent the combination of score and waveform for a performance, each data ID is registered in a "track", enabling the score and waveform to be joined together. This operation is also known as a BIND. The data can be performed by executing instructions with the track as the target.

Each track contains several IDs. The ID of the waveform data must be registered in the track. Then, to perform score data, the score data ID should be registered in the track. To generate a one-shot sound, the ID of the one-shot sound generation should be registered in the track. The waveform data, score data, and one-shot sound generation IDs can be registered in multiple tracks.

A performance can be executed for a track in which a waveform data ID and a score data or one-shot sound generation ID was registered, by specifying the track ID as an argument to the function that will execute the performance.

Figure 5-3



Each ID serves as an identifier that maintains information on the specified address and size; the actual data itself is not included. Consequently, the data area that was specified when the ID was assigned must not be deleted, cleared, or overwritten by other data while the registered ID is valid. When data is no longer needed, the registration of its ID is deleted (this operation is known as an UNBIND), then the memory area where it had been placed is nullified.

IOP Modules

The following IOP modules are provided.

Table 5-2

Module Name	Name	Requirement	Low-level Required Module
sksound	Sound basic	Required	sdrdrv, libsd
skhsynth	Hardware synthesizer	Required	sksound, modhsyn
skmidi	MIDI sequencer	Optional	skhsynth, modmidi
skmsin	MIDI stream generation	Optional	skhsynth, modmsin
sksesq	Sound effect sequencer	Optional	skhsynth, modsesq

Sound Basic (sksound)

Provides basic processing functions for sound, such as periodic interrupt processing, SPU2 environment configuration/data transfer processing, and management of track information.

Hardware Synthesizer (skhsynth)

Performs management of waveform data and SPU2 voice control.

MIDI Sequencer (skmidi)

Performs management of score data and performances.

Sound Effect Sequencer (sksesq)

Performs management of sound effect sequence data and performances.

MIDI Stream Generation (skmsin)

Performs management and control of one-shot sound generation.

These IOP modules are loaded together with the various required modules according to the desired functionality.

Example: When only performing a score

skmidi + modmidi + skhsynth + modhsynth + sksound + sdrdrv + libsd

Example: When generating one-shot sounds + performing a score

skmsin + modmsin +
skmidi + modmidi + skhsynth + modhsynth + sksound + sdrdrv + libsd

Example: When executing a sound effect sequence + generating one-shot sounds + performing a score

sksesq + modsesq +
skmsin + modmsin +
skmidi + modmidi + skhsynth + modhsynth + sksound + sdrdrv + libsd

Example: When generating only one-shot sounds

skmsin + modmsin + skhsynth + modhsynth + sksound + sdrdrv + libsd

Options When Loading IOP Modules

The attributes of various IOP modules can be changed by specifying options when loading the modules.

Below are examples of options specified in the source code. For details, see the description of sceSifLoadModule() in the library reference.

```
sceSifLoadModule ("cdrom0:\\SKMIDI.IRX",
                        10 + 1 + 10 + 1, "maxtrack=8\000maxentry=8")
```

- tickresolution (sksound)

Specifies time resolution (= time for 1 tick) for sound processing

Default: 240 ... 1/240 second

Example: Specify 1/480 second as the time resolution

tickresolution=480

- thpri (sksound)

Specifies the thread priority for sound processing in the IOP

Default: 32

Example: Specify 48 as the thread priority

thpri=48

- maxtrack (sksound, skmidi, sksesq, skmsin)

Specifies the upper limit for registering data to be managed by these modules as tracks

Default: 8 (sksound) / 4 (skmidi, sksesq, skmsin); Maximum: 16 (common)

Example: Specify 16 as the number of tracks to be used

maxtrack=16

Remarks:

- The following relationship with maxentry must hold in a module: maxentry >= maxtrack. Although modules will operate correctly even if maxentry < maxtrack, the upper limit of the number of registration tracks in that module will be limited to the maxentry value.
- The following relationship must hold between the maxtrack value of the sksound module and the maxtrack values of the other sk modules (excluding skhsynth).

  Sumof(maxtrack values of modules to be used) <= (maxtrack value of sksound module)
- The maxtrack value of the skhsynth module will automatically become equal to the maxtrack value of the sksound module internally when skhsynth is loaded.

- maxentry (skhsynth, skmidi, sksesq, skmsin)

Specifies the upper limit on the number of data entries that can be registered as IDs

Default: 8 (skhsynth) / 4 (skmidi, sksesq, skmsin); Maximum: 16 (common)

Specification example: Specify 16 as the upper limit on the number of data entries to be registered

maxentry=16

Remarks:

The following relationship with maxtrack must hold in a module: maxentry >= maxtrack. Although modules will operate correctly even if maxentry < maxtrack, the upper limit of the number of registration tracks in that module will be limited to the maxentry value.

Precautions

Blocking/Non-blocking Processing

The standard kit/sound system library (SKSS) is implemented internally by calling libsd functions (mainly sceSdRemote()) from the EE.

An argument can be used to specify blocking or non-blocking processing using sceSdRemote(). However, only blocking processing is currently implemented and used with SKSS. As a result, to prevent an IOP-side sound processing instruction from interrupting graphics processing, processing must be divided into threads and dedicated sound threads must be created.

Calls From Multiple Threads (Reentry)

Internally, since SKSS uses libsdr as described above and libsdr is not reentrant, SKSS also has the same restrictions. When SKSS functions are called asynchronously from multiple threads, be sure that these calls do not mutually overlap.

RPC reentry is explained in the libsdr documents and "SIF System" document, so please refer to the use of reentry and its restrictions in these documents.

Chapter 6:
SPU2 Local Memory Management

Library Overview

The SPU2 local memory management library (libspu2m) is a utility library that simplifies the management of SPU2 local memory areas.

Related Files

The following files are required to use libspu2m.

Table 6-1

Category	Filename
Library	libspu2m.al
Header file	libspu2m.hl

Sample Programs

The following sample programs use libspu2m.

- **sce/ee/sample/sk/playsq/**
- **sce/ee/sample/sk/ctrlsq/**

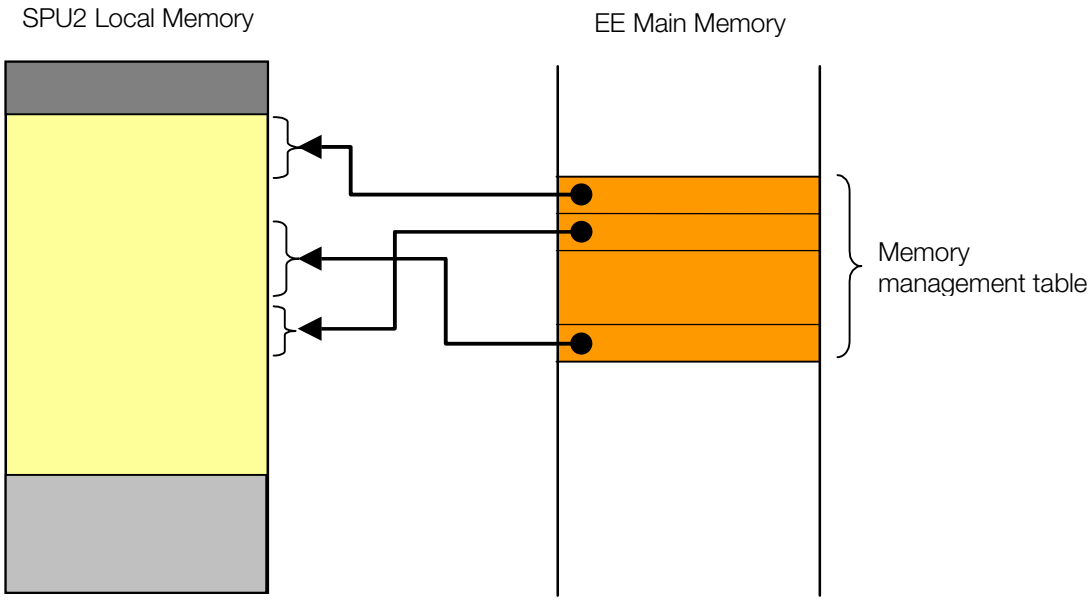
  These are standard kit/sound system library sample programs which use libspu2m for SPU2 local memory management.

Software Architecture

Memory Management Table

SPU2 local memory cannot be directly accessed from both the EE and IOP, therefore, the EE uses information on memory allocation and freeing to manage the state of memory areas. The table for managing this information (known as the "memory management table") is allocated by a user program, and the starting address and size are specified as arguments to the initialization routine. The initialization routine initializes and maintains the specified memory management table, which is then used for the allocation and freeing of SPU2 local memory areas.

Figure 6-1



Reserving the Effect Work Area

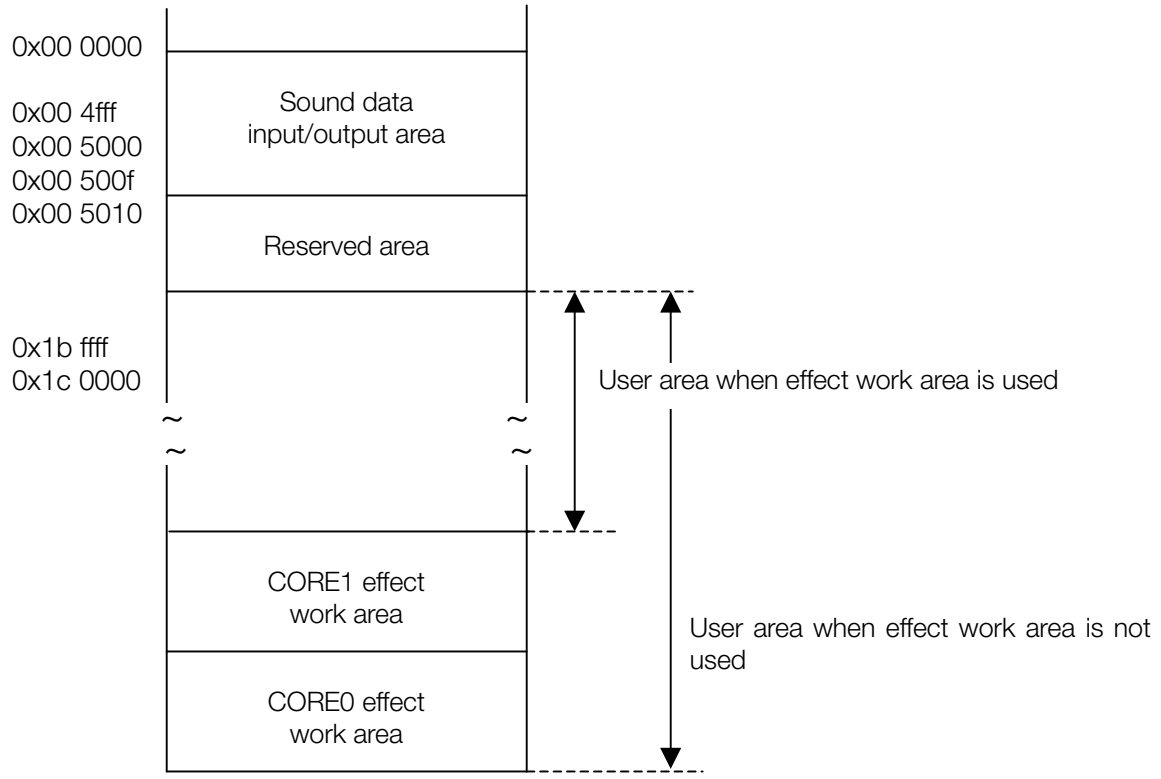
Not only does SPU2 local memory have an area for user waveform data, but it also contains an area for sound data input/output as well as a partial reserved area. These areas are at fixed locations, and libspu2m allocates them as user areas that are not subject to management.

SPU2 local memory can also be used as a work area for effects, however, the size of the work area will vary depending on the effect. Whether or not a work area will be used can be specified in libspu2m using an argument of the initialization routine.

With the current implementation, 128KB from each SPU2 core (total of 256KB) is reserved from all of SPU2 local memory when an effect work area is used. This reserved area is not subject to management.

Since the sound data input/output and partial reserved areas are also not subject to management as described above, the amount of memory available to the user is 1,814,512 bytes.

Figure 6-2



- Size of user area when effect work area is used
  2 MB - (0x5010 + 2 * 128 KB) = 1,814,512 (0x1baff0) bytes
- Size of user area when effect work area is not used
  2 MB - (0x5010) = 2,076,656 (0x1faff0) bytes

Allocating and Freeing Memory Areas

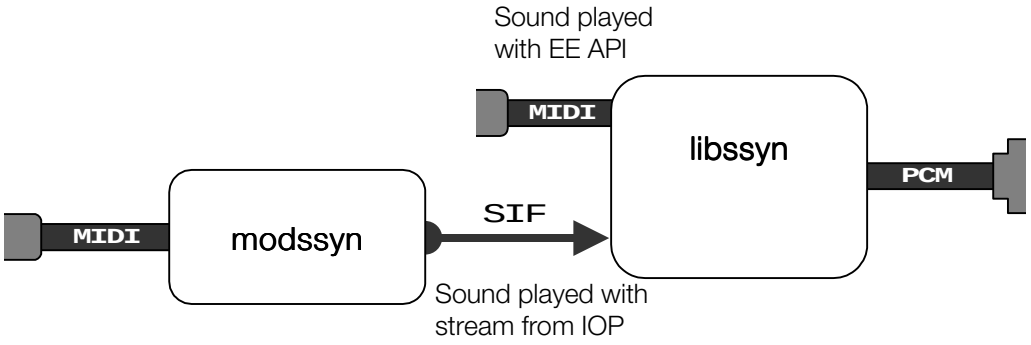
Dedicated functions are provided for allocating and freeing memory areas. The number of area allocations, that is, the number of sections into which SPU2 local memory can be subdivided and managed is determined by the size of the memory management table. If the size of the memory management table is exceeded when the area allocation function is called, a value indicating that the "area cannot be allocated" is returned, even if there exists an unallocated area having the specified size. Therefore, be sure to allocate a memory management table of the required size and pass it to the initialization routine.

Chapter 7:
CSL Software Synthesizer

Library Overview

libssyn is a software sound source library that conforms to the CSL (Component Sound Library) specification. IOP MIDI Stream data is received via modssyn (software synthesizer communication module), and PCM Stream data is output using phoneme data in EE memory. Alternatively, instead of using MIDI Stream data from the IOP, sound can also be played through MIDI messages (EE MIDI stream) using API calls from an EE application.

Figure 7-1



A maximum of 8 input ports and 8 output ports can be used. Each input port is capable of serving as an independent sound source (the maximum number of generated sounds is the total of the upper limit of the number of generated sounds set up for each of the input ports).

Each input port has 16 channels of parts. Each part has four output ports, and an output port of a part can be freely assigned to an actual output port.

Libssyn works in conjunction with the IOP module "modssyn.irx". See modssyn for additional information.

Since libssyn uses vu0 internally, to protect the vu context in a multithreaded environment, threads that are currently using vu0 should be grouped together as a single thread.

Usage

Buffer Structure

Buffer Group 0: Input port group

Table 7-1

BufCtx Index	Description	Data Structure
1 - N	Input buffer for input port N	sceSSynEnv

Buffer Group 1: Output port group

Table 7-2

BufCtx Index	Description	Buffer Contents
N	Output buffer for input port N	sceCslPcmStream

* Non-cached access is performed on the buffer area indicated by pcmbuf and sceCslPcmStream itself, so alignment, etc. must be taken into account.

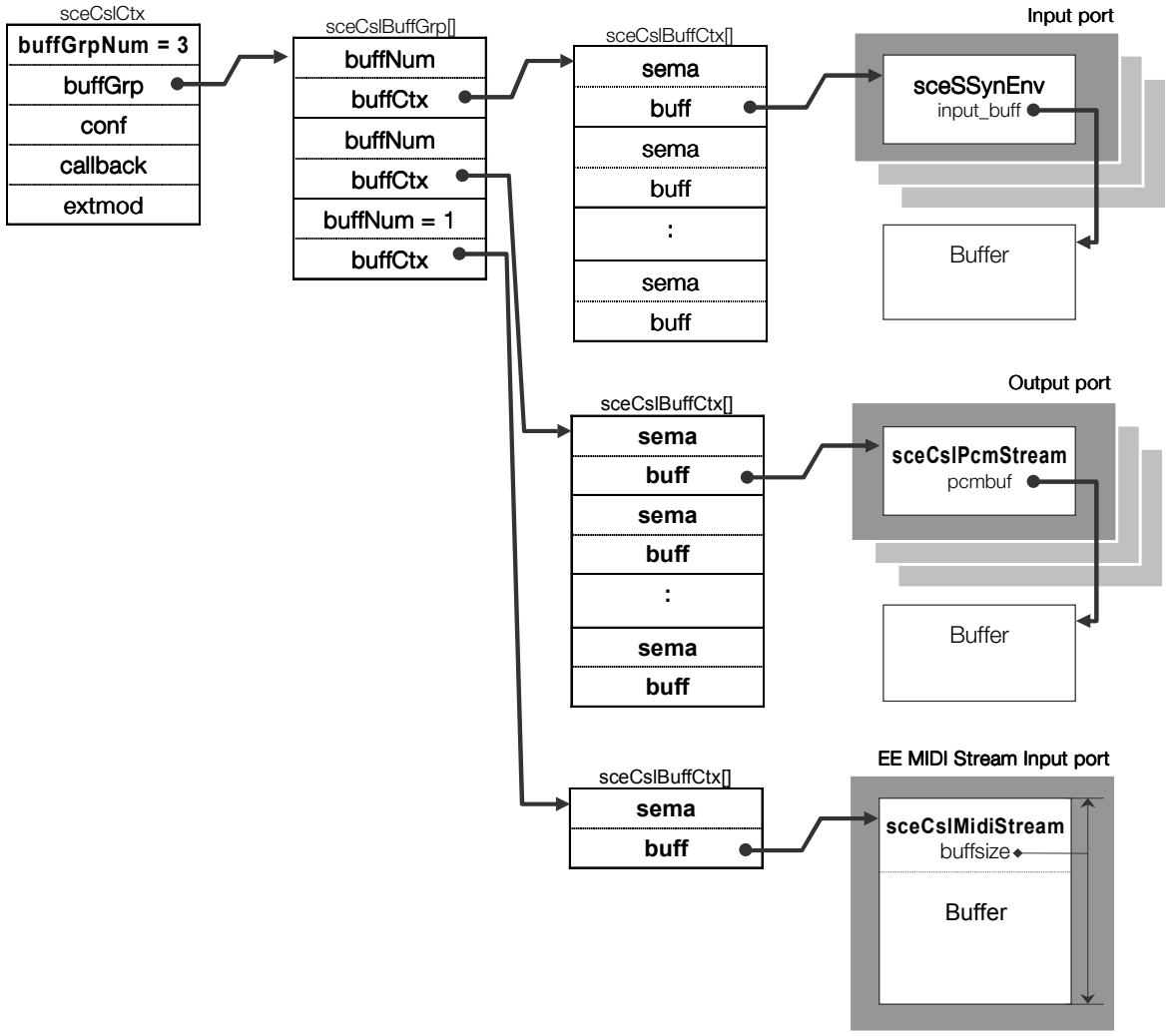
Buffer Group 2: EE MIDI Stream input port group

Table 7-3

BufCtx Index	Description	Data Structure
1	Sequence data generated through API calls	sceCslMidiStream void[buffsize - sizeof (sceCslMidiStream)]

* The validsize member of sceCslMidiStream indicates the valid data count in the buffer.

Figure 7-2



Buffer group 2 is used if EE MIDI Stream is used. Set buffGrpNum = 2 if EE MIDI Stream is not used.

If sceCslBuffCtx.buff of an input port is set to NULL, the port will be skipped during ATick processing. This can be used to temporarily disable sounds from a particular port.

Programming Procedures

Libssyn causes IOP communication to take place, so the following sequence of operations must be performed for initialization.

1. Call sceSSyn_Init() to initialize the synthesizer routine.
2. Call sceSSyn_RegisterRpc() to perform registration of the SIF RPC server.
3. Call sceSifLoadModule() to load the IOP module (moddssyn.irx).
4. Load sound phoneme data into EE memory, call sceSSyn_PrepareParameter() to resolve addresses, and call sceSSyn_Load() to register.

2. and 3. are not needed if sounds are played only with EE MIDI Stream, and not with IOP MIDI Stream.

MIDI Message Filter Callback

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

Table 7-4

Trigger	Callback Function	Filter Function
Channel message	msg_callback	Suppress message output
Exclusive message	excMsgCallBack	Suppress message output

Callback functions are registered in sceSSynEnv. When a message serving as a trigger, etc. appears in the sequence data, the associated callback function is called.

The callback function examines the message, etc., which is passed as a parameter, and can, in certain cases, suppress the message.

Overview of Features

The external specifications of the software sound source implemented by libssyn are shown below.

Tone Generator (Wave Table Reading Section): TG

- Wave Data is 8/16-bit PCM wave data with a 22.05K/44.1K/48K sampling rate. Wave data contains a read start position and loop start/end positions as address information.

  One loop can be specified and it must be a forward loop. To use a loop other than a forward loop, 1 shot can create a silent loop as a forward loop and the other loop is expanded into a forward loop. The maximum playback frequency depends on the loop size.
- The read start position can be modified with velocity, etc.
- The key region/velocity can be used to change the waveform being read. Up to two waves can be read at the same time.
- A dedicated ENV is provided and pitch can be modified.
- pitch can be modified using LFO.
- pitch can be modified using a MIDI pitch bend message.
- A pitch can be set up with keyfollow using key#.
- Portamento can be used.

Time Variant Filter: TVF

- One of LPF/BPF/HPF for 12db/oct. can be selected and used.
- Control can be performed using cutoff Freq. and resonance (1/Q).
- A dedicated ENV is provided and cutoff can be modified.
- cutoff can be modified with LFO.
- A cutoff freq. can be set up with keyfollow using key#.

Time Variant Amplifier: TVA

- Operates as an attenuator.
- Provides a dedicated ENV and level can be modified.
- level can be modified with LFO.
- level can be set up with keyfollow using key#.

Output Mixer: MIX

- Up to eight channels of output. 16 parts x 4 channels output assigned to 8-channel output port.
- Pan pot settings can be made for each voice.

Envelope Generator: ENV

- Envelope control through ADSR level 4 points, time 3 points.
- time is specified in milliseconds.
- The range for level is -1.0 - 1.0 for TG and TVF, 0.0 - 1.0 for TVA. The endpoint level for TVA ReRelease is fixed at 0.
- level and time can be modified with velocity, key#.
- Operating curves:

  TG: read frequencies can be modified in cents

  TVF: cutoff Freq. can be modified in cents

  TVA: linear

Low Frequency Oscillator: LFO

- Each voice has two LFOs.
- Supported waveforms are: sine, rectangular, triangular, sawtooth, and noise.
- Delay before transmission can be set.

Received MIDI Messages

The following is a list of MIDI messages that can be received by the software sound source.

8n kk note off (ignore velocity)

9n kk vv note on(velocity = 0: off)

Bn

- 0x00,0x20 Bank Change
- 0x01 Modulation
- 0x05 Portamento time
- 0x06,0x26 Data Entry
- 0x07 Volume
- 0x0a Panpot
- 0x0b Expression
- 0x40 hold1
- 0x41 portamento
- 0x54 portamento control
  - 0x63,0x62 NRPN
  - 0x01,0x08 vibrato rate
  - 0x01,0x09 vibrato depth
  - 0x01,0x0a vibrato delay
  - 0x01,0x20 TVF cut-off
  - (0x01,0x21 TVF resonance
  - 0x01,0x63 TVF,TVA attack time
  - 0x01,0x64 TVF,TVA decay time
  - 0x01,0x65 TVF,TVA release time
  - 0x18,rr drum pitch course
  - 0x1a,rr drum TVA level
  - 0x1c,rr drum pan-pot
- 0x65,0x64 RPN
  - 0x00,0x00 pitch bend sensitivity
  - 0x00,0x01 master fine tuning
  - 0x00,0x02 master coarse tuning
  - 0x7f,0x7f NULL
- 0x78,0x00 all sound off
  - 0x79,0x00 reset all controllers
  - Other Mode message is recognized as all notes off

Cn pp Program Change

Dn pp	Channel Pressure
En vv vv	Pitch Bend Change

Extended MIDI Messages

The software sound source implemented in libssyn can receive extended MIDI messages as defined by the CSL. For more information on extended MIDI messages, see the "CSL Overview" document.