Run-Time Library Overview

Summary Table of Contents

List of Figures

List of Tables

CONFIDENTIAL

About This Manual

This manual is the latest release of the *Library Overview* for Run-Time Library 4.7. The purpose of this manual is to provide overview-level information about the PlayStation® libraries. For related descriptions of the PlayStation run-time library functions and structures, please refer to the *Run-Time Library Reference.*

Changes Since Last Release

Since release 4.6 of the Run-Time Library Overview, the following changes have been made:

Ch. 13:   Controller/Peripherals Library

- In the "Number of actuators that can be used simultaneously" section of "Precautions" in the Controller Library, an incorrect description of usage without a Multi tap has been deleted.

Ch. 20:   Memory Card GUI Library

- In the "Module and Header Files", "Module function" and "Language setting" sections of the "Overview", descriptions of the new language options (French, German, Italian and Spanish) have been added.

Related Documentation

This manual should be read in conjunction with the *Run-Time Library Reference*, which defines all structures and functions available in the Run-Time Library.

Manual Structure

Chapter	Description
Ch. 1: Overview of the PlayStation OS	Summarizes the PlayStation operating system and the run-time libraries.
Ch. 2: Kernel Library	Describes the Kernel library (libapi), which provides an interface between applications and the PlayStation OS.
Ch. 3: Standard C Library	Describes the PlayStation's subset of the standard C library (libc/libc2). This library includes character functions, memory operation functions, character class tests, non-local jumps, and utility functions.
Ch. 4: Math Library	Describes the Math Library (libmath), which contains ANSI/IEEE754 compliant math functions, including a software floating point computation package.
Ch. 5: Memory Card Library	Describes the Memory Card Library (libcard) for reading/writing to the PlayStation Memory Card and calling the Memory Card BIOS service.
Ch. 6: Extended Memory Card Library	Describes the Extended Memory Card Library (libmcard), which provides a high-level interface for accessing the Memory Card.

Chapter	Description
Ch. 7: Data Compression Library	Describes the Data Compression Library (libpress) for compressing (encoding) and expanding (decoding) image and sound data.
Ch. 8: Basic Graphics Library	Describes the Basic Graphics Library (libgpu) for drawing primitives such as sprites, polygons, and lines.
Ch. 9: Basic Geometry Library	Describes the Basic Geometry Library (libgte), which uses the PlayStation GTE co-processor to handle high-speed geometry operations.
Ch. 10: Extended Graphics Library	Describes the Extended Graphics Library (libgs) which uses the libgpu and the libgte to construct a 3-dimensional graphics system. It deals with more abstract entities such as objects and background surfaces.
Ch. 11: CD-ROM Library	Describes the CD-ROM Library (libcd), for controlling the PlayStation CD-ROM drive. It also contains functions for "streaming" real-time data such as movies, sounds or vertex data stored on high-capacity media.
Ch. 12: Extended CD-ROM Library	Describes the Extended CD-ROM Library (libds), which builds a new interface to the libcd kernel. It has the same capabilities as libcd, and places further emphasis on reliable CD-ROM controls such as recovery from system errors.
Ch. 13: Controller/Peripherals Library	Describes the Controller/Peripherals Library, which consists of a group of libraries (libetc, libgun, libtap, and libpad) for performing low-level interrupt processing and controller-related functions.
Ch. 14: Link Cable Library	Describes the Link Cable Library (libcomb), which provides services for connecting PlayStation units via a "link" cable.
Ch. 15: Extended Sound Library	Describes the Extended Sound Library (libsnd), which provides functions for working with sound data on the PlayStation.
Ch. 16: Basic Sound Library	Describes the Basic Sound Library (libspu) for controlling the SPU (sound processing unit).
Ch. 17: Serial Input/Output Library	Describes the Serial Input/Output Library (libsio) for supporting communications between the PlayStation and a PC.
Ch. 18: HMD Library	Describes the HMD Library (libhmd), which provides functions and definitions for handling the HMD format, which integrates modeling, animation, texture, and MIMe data.

Chapter	Description
Ch. 19: PDA Library	Describes the PDA Library (libmcx) used to provide access to PDA functions when it is inserted into a Memory Card slot.
Ch. 20: mcgui Module	Describes the module for saving and loading data in game titles, as well as support for the user interface

Developer Reference Series

This manual is part of the *Developer Reference Series*, a series of technical reference volumes covering all aspects of PlayStation development. The complete series is listed below:

Manual	Description
PlayStation Hardware	Describes the PlayStation hardware architecture and overviews its subsystems.
PlayStation Operating System	Describes the PlayStation operating system and related programming fundamentals.
Run-Time Library Overview	Describes the structure and purpose of the run-time libraries provided for PlayStation software development.
Run-Time Library Reference	Defines all available PlayStation run-time library functions, macros and structures.
Inline Programming Reference	Describes in-line programming using DMPSX, GTE inline macro and GTE register information.
SDevTC Development Environment	Describes the SDevTC (formerly "Psy-Q") Development Environment for PlayStation software development.
3D Graphics Tools	Describes how to use the PlayStation 3D Graphics Tools, including the animation and material editors.
Sprite Editor	Describes the Sprite Editor tool for creating sprite data and background picture components.
Sound Artist Tool	Provides installation and operation instructions for the DTL-H800 Sound Artist Board and explains how to use the Sound Artist Tool software.
File Formats	Describes all native PlayStation data formats.
Data Conversion Utilities	Describes all available PlayStation data conversion utilities, including both stand-alone and plug-in programs.
CD Emulator	Provides installation and operation instructions for the CD Emulator subsystem and related software.
CD-ROM Generator	Describes how to use the CD-ROM Generator software to write CD-R discs.
Performance Analyzer User Guide	Provides general instructions for using the Performance Analyzer software.

Manual	Description
Performance Analyzer Technical Reference	Describes how to measure software performance and interpret the results using the Performance Analyzer.
DTL-H2000 Installation and Operation	Provides installation and operation instructions for the DTL-H2000 Development System.
DTL-H2500/2700 Installation and Operation	Provides installation and operation instructions for the DTL-H2500/H2700 Development Systems.

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).

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., 2nd floor Foster City, CA 94404 Tel: (650) 655-8000	E-mail: DevTech_Support@playstation.sony.com Web: http://www.scea.sony.com/dev Developer Support Hotline: (650) 655-8181 (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 Waverley House 7-12 Noel Street London W1V 4HH Tel: +44 (0) 171 447 1600	E-mail: dev_support@playstation.co.uk Web: https://www-s.playstation.co.uk Developer Support Hotline: +44 (0) 171 447 1680 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT or BST/BDT)

Chapter 1:
Overview of the PlayStation OS

Table of Contents

The PlayStation OS

The PlayStation OS is a flexible and powerful operating system, which allows developers to take maximum advantage of the PlayStation's capabilities.

The OS has been developed for the R3000, which is the PlayStation's CPU. The efficiency of program development relies heavily on the environment and services provided by the OS. If the CPU and peripheral devices are fast enough, you won't need to spend your time trying to maximize the hardware's capabilities. You can concentrate on programming using the services the OS provides for you.

The PlayStation OS is designed to give the game program developer an environment in which interrupts can be easily controlled. Based on this concept, the kernel of the PlayStation OS provides services to control PlayStation hardware and the R3000.

Each service is provided as a C language function. By using C, your programs can be more readable and maintainable, and you can program more easily using block structure description and function calls.

Features of the PlayStation OS

This section describes the characteristics of the PlayStation's design concept.

Programming in C

Most services, such as controlling the R3000 CPU and the PlayStation hardware, are provided as C language functions. Therefore, programs can be written completely in C.

Easy Access to the Features of the R3000

Interrupt control in the R3000 is said to be complicated, but the PlayStation OS uses a substitute "dispatcher" system which has a simple interface. The dispatcher's overhead is kept very low, and it provides low-level support not available in ordinary operating systems. Because of this, the chip's capabilities can be fully exploited and high quality tuning can be achieved. And because everything can be done in C, it is not necessary to learn the intricacies of R3000 assembler.

Small Size, Emphasis on Performance

Because of the importance of an application's performance, the PlayStation OS was designed so that its RAM usage (64K bytes) and use of the CPU are kept to a minimum. In addition, the OS system tables are disclosed, and consideration given to future expansion of the OS.

To achieve greater speed, the PlayStation OS doesn't carry out many checks of prohibited items that other operating systems would. This policy allows applications to achieve a higher level of tuning. However, to avoid the risk of prohibited operations being performed, applications may need to perform some checks that would normally be carried out by the operating system.

Provision for Hardware Functions

Previously, to control video game machine hardware, one has had to analyze hardware driver code and painstakingly write one's code in assembler. The PlayStation OS lightens this burden by providing C language functions to control hardware. The overhead of each function is kept to a minimum.

Single and Multitasking

The PlayStation OS can carry out many tasks asynchronously while executing code, such as controlling a CD-ROM drive, which is a comparatively slow device, and playing background music.

When the OS starts, it is in single-task mode. If desired, you can specify that your application will have multiple tasks or threads. See Chapter 2, "Kernel Library", for information on threads.

The File System Device Driver

The PlayStation's file system (i.e., files of data on CD-ROM) is accessed via a device driver. This allows multiple file systems to coexist, and improves development time by avoiding low level file manipulation problems.

Starting and Operating the OS

The PlayStation OS provides a game program developer's environment. Therefore, there is fundamentally no interface for the user to operate directly (excluding the debug monitor in the debug environment). Applications must provide the user interface.

Activation of the OS

When the system starts, it jumps first a special address in ROM and performs a check on connected hardware (such as a CD-ROM drive).

It then checks for a suitable disk in the CD-ROM drive. If it finds one, it reads the system configuration file (`SYSTEM.CNF`).

If there is no disk, a ROM-resident demonstration program plays repeatedly.

Boot Sequence

The boot sequence is as follows:

CONFIDENTIAL

Figure 1-1: Boot Sequence

```
┌─────────────────────────────────────────┐
│   Initialize System  Operation Memory    │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│            Set System  Stack             │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│          Initialize Kernel Memory        │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│ Initialize System  Memory Control Capability │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│          Initialize IO  Manager          │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│            Set Error Handler             │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│  Read System  Setting File (system.cnf)  │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│        Analyze System  Setting File      │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│      Maintain All System  Table Types    │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│         Read Boot Execution File         │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│                Set Stack                 │
└─────────────────────────────────────────┘
                    ↓
┌─────────────────────────────────────────┐
│        Execute Boot Execution File       │
└─────────────────────────────────────────┘
```
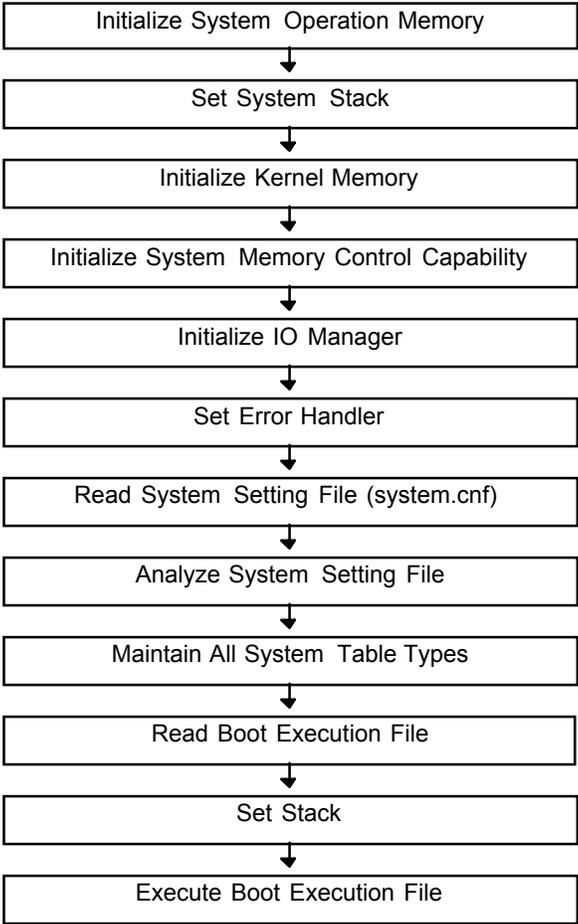
PlayStation OS Library Components

PlayStation libraries can be thought of as low-level libraries or high-level libraries, depending on their relationship to the PlayStation OS. They form a two-level library structure. Programs may use any level as needed, and, with some exceptions, may use both levels concurrently.

Figure 1-2: PlayStation library structure

```
                                                                    ┌──────────┐
                                                                    │  libpad  │
                                                                    │ libcomb  │
                                                                    │  libsio  │
          ┌──────────────────┐                                      │  libgun  │
          │     libhmd        │                                      │  libtap  │
          │    ┌──────┐       │   ┌────────┐   ┌──────┬────────┐   ┌────────┐ │libc/libc2│
          │    │libgs │       │   │ libsnd │   │libcd │ libds  │   │libmcrd │ │ libmath  │
          │    └──────┘       │   └────────┘   └──────┴────────┘   │        │ │  libsn   │
┌─────────┬─────────┬─────────┬──────────┬──────────┬──────┬──────┼────────┤ libmcx │          │
│ libgpu  │ libgte  │ libapi  │ libspu   │ libpress │      │libetc│ libcard │        │          │
└─────────┴─────────┴─────────┴──────────┴──────────┴──────┴──────┴─────────┴────────┴──────────┘
```
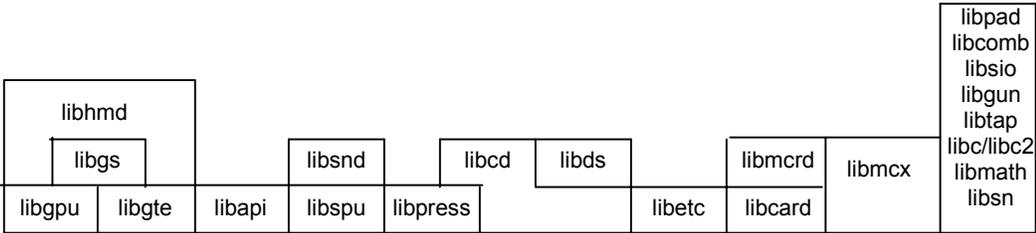
A summary of the libraries follows:

libapi (Kernel Library)

Provides an interface (API) between the PlayStation OS and applications.

libc/libc2 (Standard C Libraries)

A subset of the standard C library, including character functions, memory operation functions, character class tests, non-local jumps, and utility functions.

libmath (Math Library)

Contains ANSI/IEEE754 compliant math functions and a software floating point computation package.

libcard (Memory Card Library)

Provides functions for controlling the Memory Card, which preserves data after reset and power off. It includes the Memory Card, the file system, and drivers.

libmcrd (Extended Memory Card Library)

Provides a high-level interface to the Memory Card.

libpress (Data Compression Library)

Provides functions for compressing (encoding) and expanding (decoding) image and sound data.

libgpu (Basic Graphics Library)

Contains commands for the drawing engine and for building a drawing command list. Handles data for simple entities such as sprites, polygons and lines.

libgte (Basic Geometry Library)

A library for controlling the GTE (geometry transformation engine). Handles data such as matrices and vertices.

libgs (Extended Graphics Library)

A three-dimensional graphics system which uses libgpu and libgte. Handles larger entities such as objects and background surfaces.

libcd (CD/Streaming Library)

Reads program, image, and sound data from a CD-ROM drive, and performs playback of DA (digital audio) and XA sound. Also includes fast disk access through a file name key, and a support function for simultaneous data reading and processing streaming techniques.

libds (Extended CD-ROM Library)

Builds a new interface to the CD-ROM library kernel. Has the same capabilities as libcd and places further emphasis on reliable CD-ROM controls such as performing error recovery .

CONFIDENTIAL

libetc (Peripherals Library)

Performs callback control for using controllers and other peripheral devices and processing low-level interrrupts.

libtap (Multi Tap Library)

Allows access to 3-8 controllers and memory cards through the optional peripheral multi-tap.

libgun (Gun Library)

Provides access to Light Pen type input equipment which can connect to the controller connector.

libpad (Controller Library)

A library for accessing the controller. Supports extended protocol controllers such as DUAL SHOCK.

libcomb (Link Cable Library)

Communicates between the link cable and PlayStation. It includes an 8-bit block size communication driver.

libsnd (Extended Sound Library)

Plays as background, sound production sequences that have been prerecorded as score data.

libspu (Basic Sound Library)

Controls the SPU (sound processing unit).

libsio (Serial Input/Output Library)

Sets the standard input/output on the debugging station to SIO 1.

libhmd (HMD Library)

Provides functions and definitions for handling the HMD format, which integrates modeling, animation, texture, and MIMe data.

libmcx (PDA Library)

The PDA library provides access to various functions of the PDA when it is inserted into a Memory Card slot.

mcgui (Memory Card GUI module)

This module provides support for loading and saving data in game titles, as well as support for the user interface.

Chapter 2:
Kernel Library

Table of Contents

CONFIDENTIAL

Overview

The Kernel library (libapi) provides an interface (API) by which applications can control basic aspects of the PlayStation OS, including the CPU and other hardware features.

It includes the following services:

- Root-counter processing
- Event processing
- Thread processing
- IO processing
- Module processing
- Controller
- Other

Library and Header Files

Programs using kernel services must link with the library file `libapi.lib`.

Source code must include the header files `libapi.h` and `kernel.h`.

System Designation File

You use the system designation file `system.cnf` to reserve memory for stack, tasks and events. The system reads this file at boot time.

The format of each line of the file is "<KEYWORD> = <CONTENT>". The table below shows the available keywords. All characters must be uppercase (1 byte alphanumeric) and there must be a space on either side of the equal sign. (If there is more than one line with the same parameter within a file, the first one takes precedence.)

Table 2-1: SYSTEM.CNF Overview

Key word	Contents	Default	Minimum
BOOT	Device name:\Product number; version	cdrom:PSX.EXE;1	N/A
	Example: BOOT = cdrom:\SLUS_123.45;1		
STACK	Stack pointer value when booted	0x801FFF00	0x80010000**
TCB	Number of task control blocks (hex)*	4	1
	Example: TCB = 5		
EVENT	Number of event processing blocks (hex)*	0x10	0
	Example: EVENT = 5		

* The maximum number of task control blocks and event control blocks that you can allocate is shown by the following formula: `TCB * 192 + EVENT * 32 + 544<4096`.

** Since 0-0x0000FFFF is reserved as the system area (see the "Physical Memory Map" section of the "CPU and its Peripherals" chapter in the *PlayStation Hardware* manual) the minimum address of the user program work area should be at least 0x80010000.

System Table Information (ToT)

The kernel uses several types of tables, such as task control blocks and event control blocks. To access these tables in a uniform manner, system table information is represented by the structure ToT (Table of Tables), located at address 0x00000100.

Each entry in the ToT is defined by the following structure (defined in `kernel.h`). The member `head` is a pointer to the actual table.

```
struct ToT {                  /*system table table*/
   unsigned long *head;       /*system table initial address*/
   long size;                 /*system table size (in bytes)*/
};
```

The ToT entries are:

Table 2-2: ToT Entries

Entry	Corresponding Table
0	System Reserved
1	TCBH (pointer to execution TCB)
2	Task control block (TCB) array
3	System reserved
4	Event control block array
5-31	System reserved

The TCB (Task control block) structure contains information about a specific task. (See "Threads" for more information on using TCBs.) The TCBs are stored in an array, pointed to by the TCB entry of the ToT.

The TCBH structure contains a pointer to the currently executing TCB.

```
struct TCB {                         /*task control block*/
   long status;                      /*status*/
   long mode;                        /*mode*/
   unsigned long reg [NREGS];        /*register save area*/
                                     /*specify with register-specified macro*/
   long system[6];                   /*system reserved*/
};

struct TCBH {             /*task status queue*/
   struct TCB *entry;     /*pointer to execution TCB*/
   long flag;             /*system reserved*/
};
```

The ToT can be used as follows:

Example 1: Getting the Pointer to the Execution TCB

```
struct ToT *t = (struct ToT *)0x100; /* address of ToT */
struct TCBH *h = (struct TCBH *)((t + 1)->head); /* address of TCB status
      queue header, which contains a pointer to currently executing TCB */
struct TCB *tcb_exec = (struct TCB*)(h->entry); /* address of execution TCB
      */
```

Example 2: Getting the Pointer to the Start of the TCB Array

```
struct TCB *tcb_0 = (struct TCB *)((t + 2)->head;
```

Example 3: Getting the Pointer to the Start of the Event Control Block Array

```
struct EvCB *evcb_0 = (struct EvCB *)((t + 4)->head);
```

See the *Run-Time Library Reference* for the definition of EvCB.

CONFIDENTIAL

Descriptors

When you work with certain system resources such as files or threads, the kernel provides you with *descriptors* to refer to the resources. Descriptors are unsigned 32-bit integers, with the following bit assignments:

Table 2-3: Descriptor Bit Patterns

Bit Number	Contents
31-24	Descriptor classification
23-16	Reserved by system
15-0	System table number

The kinds of descriptors available are listed below. Each macro is defined in `kernel.h`.

Table 2-4: Descriptor Classification

Macro	Class contents	Notes
DescTH	Thread	
DescHW	Hardware	System internal use
DscEV	Event	
DescRC	Root counter	
DescUEV	User-defined flag	
DescSW	System call	System internal use

The normal procedure for keying descriptors to system resources is as follows:

1. Obtaining descriptors.

   First call the Open() function provided for each resource. The return value of the function is the descriptor of that resource.

2. Operation of resources.

   Use the descriptor returned by the Open() function to specify the resource and perform the operation required.

3. Closing descriptors.

   After use, close the descriptor with the appropriate Close() function.

Example 1: Thread Descriptor

```
unsigned long th, th_new;
th = OpenTh(0x1000,0x1ffff0,0x00);
th_new = OpenTh(0x2000,0x18fff0,0x00);
ChangeTh(th);
ChangeTh(th_new);
CloseTh(th);
```

Example 2: File Descriptor

```
unsigned long fd, ret;
char buf[2048];
fd = open("cdrom:PSX.EXE;1",O_RDONLY);
ret = read(fd, buf, 1048);
close(fd);
```

Example 3: Event Descriptor

```
unsigned long ev;
extern long (*handle)();
ev = OpenEvent(RCntCNT0, EvSpINT, EvMdINTR, handle);
EnableEvent(ev);
DisableEvent(ev);
CloseEvent(ev);
```

Callbacks

In libraries (such as libgpu or libsnd) that handle devices using DMA, there is a function for registering callback functions in the kernel. Callback functions are executed after an event has occurred.

Callback functions are executed in the Callback Context (last out), using their callback stack. This stack is declared in libetc and is included in the application data area.

The callback function is called automatically when a DMA transfer is completed. You can execute transfer completion processing by setting a flag in an external variable and issuing an event.

Inhibition of Interrupts

Any functions modifying data within the kernel must be executed in code where interrupts are inhibited. See the *Run-Time Library Reference* for information about specific functions.

A section of code in which interrupts are inhibited is called a *critical section*. Interrupts are inhibited at the following times:

- Immediately following system start. (They are enabled by calling the function ResetCallbacks()).
- By calling the function EnterCriticalSection().To re-enable interrupts, call ExitCriticalSection().
- Immediately following the start of an event handler. To re-enable interrupts, the handler can call ReturnFromException() to return to the original context. It can also call ExitCriticalSection(); however, if an interrupt occurs, control won't return to the main context but to original interrupt context.

Interrupt Context

We refer to the normal execution of a program as its Main Flow. When an interrupt or exception occurs:

- The system saves the contents of the registers in the Execution TCB as the Main Flow Context (see "Thread Management"). The status after saving is called the Interrupt Context.
- Processing begins at address 0x00000080, which contains the jump code to the kernel interrupt dispatcher, which in turn calls the appropriate routine to handle the interrupt.
- When interrupt processing is completed, the contents of the Main Flow Context registers are restored and execution of the Main Flow resumes.

Functions such as Event Handlers and Callbacks are executed in Interrupt Context (the former uses the interrupt stack, the latter uses the callback stack). When you write code that executes in Interrupt Context, keep the following cautions and prohibitions in mind.

Cautions

- Halting interrupts for a long time may adversely affect the system. You should design any routine to be executed in interrupt context so that it completes in the shortest time possible.
- Functions that generate internal exceptions (e.g., ExitCriticalSection()) cause destruction of the main flow context. This destruction may be prevented by using the thread management service to change the execution TCB.

- It is possible to return to the Main Flow by executing ReturnFromException() within an Event Handler. However, since this breaks off the action of the Interrupt Dispatcher and interrupt management returns to Main Flow as incomplete, device related malfunctions may occur. Use ReturnFromException() only for error management functions.

Prohibitions

Do not:

- Execute functions that use internal interrupts. If interrupts are not generated, the functions cannot complete.
- Execute non-re-entrant functions that may be called by the main flow. Most library functions, such as kernel services, are not re-entrant.
- Execute the function ReturnFromException() from within a callback function.

Kernel Reserved Memory Areas

The kernel uses the first 64K bytes of memory. The addresses that the user may use begin from 0x00010000.

Root Counter Control

The root counter control system provides functions such as time restrictions and timing adjustments-- indispensable features in game programs.

Since the root counter is a timer that automatically generates counter timing, the following three are provided:

- System clock
- System clock (8 cycles)
- Vertical synch.

A 16-bit target value may be set in each of these counters except vertical synch. Counters count up from zero and when they reach the target value, the following occurs:

1. An interrupt is generated (each counter can be masked).
2. The counter is cleared to zero (counter values capable of search are 0 to target value -1).
3. The counter starts counting again.

Since the target value of vertical synch is fixed at 1, an interrupt is generated at each vertical blank. Interrupts trigger counter generation and execute optional functions from the event management service (this is called an event handler). The value of each counter may be polled. Counter names are defined by macros, and the counters may be accessed using these macros.

Table 2-5: List of Root Counters

Macro	Root Counter	Notes
RCntCNT0		
RCntCNT1	System clock	Target value more than 2
RCntCNT2	System clock (8 cycles)	Same as above
RCntCNT3	Vertical synch*	Target value is fixed at 1

* halting count is invalid

Counter Timing

One tick is approximately equal to 0.03 microseconds when counting by the system clock. In the 8-cycle mode, 1 tick equals 8 times .03 microseconds (approximately .24 microseconds).

Table 2-6: Counter Timing

Counter Event	NTSC	PAL	Unit
Vertical Sync	1/60	1/50	Second
Horizontal Sync	63.56	64.00	Microsecond
Pixel Display	Nx0.0186243	Nx0.01879578	Microsecond

Table 2-7: Pixel Display Timing and Display Width

Display Width	N
256 pixels	10
320	8
384	7
512	5
640	4

The root counter uses the hardware counting function. For this reason, disabled interrupts and software operations calling functions unrelated to counting, such as StopRCnt(), will continue.

The function StopRCnt() will not stop counting. This function uses the RcntMdINTR macro for halting creation of interrupts for counters allowed interrupts. In the same way, the StartRCnt() function only allows interrupts; it does not affect counting.

Mode

For each counter the following modes may be set. Modes are defined by macros. The macros in Tables 2-9, 2-10, and 2-11 below can be set by logic.

Table 2-8: Root Counter Mode (1)

Macro	Contents
RCntMdINTR	Interrupt permitted
RCntMdNOINTR	Interrupt prohibited (polling only)

Table 2-9: Root Counter Mode (2)

Macro	Object root counter	Types of counter
RCnDtMdSP (Default)	RCntCNT0,1	(Use prohibited)
	RCntCNT2	System clock, 8 cycles
	RCntCNT3	Vertical blanking
RCntMdSC	RCntCNT0,1	System clock
	RCntCNT2,3	Not valid

Table 2-10: Root Counter Mode (3)

Macro	Contents
RCntMdFR (default)	Normal count
RCntMdGATE	Valid gate condition

Gate

Each counter will count up only when a condition called gate occurs.

Table 2-11: Root Counter Gate Condition

Root counter	Gate conditions
RCntCNT0	Not during horizontal blanking
RCntCNT1	Not during vertical blanking
RCntCNT2,3	None (usual time count)

Status Immediately After Kernel Starts

All counters are stopped immediately after activating the kernel. Immediately after the kernel starts all of the counters are stopped or free running. Thus, when they are used they must always be initialized. Also, depending on the service and the library, it may be that the user has to initialize the root counter before use.

Root Counter and Critical Section

A counter interrupt cannot occur within a critical section.

Use of the Root Counter by the Kernel

The kernel will use the root counter in the following circumstances. When using the pertinent service, reset the root counter to the state specified by the kernel.

Obtaining Controller Button Status

Use root counter 3 (vertical blanking) to obtain the status of the controller button. The state of the button cannot be read when root counter 3 is stopped or has not been initialized.

Events

An *event* is either a CPU exception or an interrupt from an external device. Since events can occur asynchronously with the execution of the main program, there are two main methods of dealing with events:

- *Polling* to determine whether an event has occurred and, if so, executing some code appropriately.
- Installing an *event handler* that the system executes automatically when the event occurs.

The system maintains a 4K interrupt stack (last out) within the memory area reserved for the kernel. Handlers execute in Interrupt Context (last out), using the interrupt stack.

GetSysSp() obtains the highest address of an interrupt stack area.

Cause Descriptor and Type of Event

An event is specified by two 32-bit integers called the *cause descriptor* and *event type.*

Table 2-12: Cause Descriptor (Kernel Library Related Only)

Cause descriptor	Contents	Event type
RCntCNT0	Root counter interrupt	EvSpINT
RCntCNT1	Root counter interrupt	EvSpINT
RCntCNT2	Root counter interrupt	EvSpINT
RCntCNT3	Root counter interrupt	EvSpINT
File descriptor	File input/output	EvSpEIO
Same as above	File close	EvSpCLOSE
HwCdRom	CD-ROM decoder interrupt	EvSpUNKNOWN*
HwSPU	SPU interrupt	EvSpTRAP
HwGPU	GPU interrupt	EvSpTRAP
HWPIO	Extension parallel port interrupt	EvSpTRAP
HwSIO	Extension serial port interrupt	EvSPTRAP
HwCPU	Exceptions	EvSpTRAP
DescUEV l m	User-defined event (m=0~0xffff)	Optional

*Other events are described in the individual libraries.

To install an event handler, you call OpenEvent(), passing in the following parameters:

- The cause descriptor (the cause of the event).
- The event type.
- The event mode.
- The address of the handler function.

If the call to OpenEvent() succeeds, it returns a 32-bit event descriptor that you use to identify the event to other functions such as EnableEvent(). The system keeps track of information about the event in an *event control block* structure:

```
struct EvCB {                /*event control block*/
    unsigned long desc;    /*cause descriptor*/
    long status;           /* status*/
    long spec;             /*event type*/
    long mode;             /*mode*/
    (long *FHandler)();    /*function format handler*/
    long system [2];       /*system reserved*/
};
```

Event Handler

An event handler is a function that is called when an event is triggered.

When the event occurs, the registers are saved and the handler begins executing. (Event handlers execute on an interrupt stack reserved in the kernel). When the handler completes its processing, it calls ReturnFromException(), which restores the registers and returns to the previous context.

Further, it is possible to permit an interrupt with the ExitCriticalSection() function, to avoid returning to the source of the interrupt and to make that routine the main flow as is. In this case, the user must provide their own stack, allocated before the interrupt. The stack can be changed with the SetSp() function.

Event Status

An event can have one of four possible statuses. Prior to opening an event, its status is EvStUNUSED. After opening an event with OpenEvent(), its status is EvStWAIT. After calling EnableEvent(), the status becomes EvStACTIVE; that is, the even may occur.

DisableEvent() switches EvStACTIVE and EvStREADY event states to an EvStWAIT state. Once in the EvStWAIT state, the next event activated by EnableEvent() must be in the EvStACTIVE state. The previous state is not saved.

Table 2-13: Event Conditions

Macro	Contents	Generation
EvStUNUSED	Not opened	Prohibited
EvStWAIT	Event generation prohibited	Prohibited
EvStACTIVE	Event not yet generated	Possible
EvStALREADY	Event already generated	Prohibited

Mode

Events can have two different modes, which you specify when opening the event. With EvMdINTR, you specify a handler function to be called when the event occurs. With EvMdNOINTR, you don't specify a handler and must test to see whether the event as occurred.

Table 2-14: Event Modes

Macro	Status after generation	Handler function
EvMdINTR	EvStACTIVE	Active
EvMdNOINTR	EvStALREADY	Not active

Event Creation

All applicable enabled events are switched over to the EvStALREADY state based on the source descriptors and event type specified when the DeliverEvent() function is executed. Events in EvMdINTR mode are handled by the event handler within the DeliverEvent() function.

Clearing an Event

Clearing an event means switching its state from EvStALREADY to EvStACTIVE. This may be done by calling UnDeliverEvent() or TestEvent().

UnDeliverEvent() takes a source descriptor and an event type, and clears all applicable events.

TestEvent() takes an event descriptor; if a corresponding event is in the EvStALREADY state, it is switched to EvStACTIVE. An event must be cleared with UnDeliverEvent() before it is reissued.

User-Defined Event

A user may define events using the DescUEV macro.

```
DeliverEvent(DescUEV|my_event_num, my_event_spec);
```

A user-defined event descriptor indicated by the number my_event_num and class my_event_spec may be called with this macro.

```
long ev;
ev = OpenEvent(DescUEV|my_event_num, my_event_spec, EvMdNOINTR, NULL);
```

is used by WaitEvent() and TestEvent(). The event handler is started when the third argument of OpenEvent() is EvMdINTR and the fourth argument is not NULL.

Threads

Threads allow an application to have multiple flows of control. They provide a form of multi-tasking in which contexts can be switched by calling a switching function. This feature may also be used for changing context at the time of an interrupt.

Context and TCB

The *thread context* consists of the complete contents of the registers. The context is stored in a data structure called a *task control block* (TCB). To switch threads, you store the current thread's context in a TCB and then assign the contents of another TCB to the registers.

The context at any given time will be stored in the *execute TCB* if triggered by the generation of an interrupt or an explicit function call. The execute TCB is pointed to by the task status queue (TCBH).

For registers, please refer to the section on *Register Specification Macros* on page 2-13 or to the *PlayStation™ Hardware Guide*.

Status Immediately After Kernel is Started

When the kernel starts, the task control block (TCB) array is allocated and the zero element is opened with OpenTh() and linked in the task status queue as the execution TCB. The default thread's descriptor is:

```
DescTH|0x0000=0xff000000
```

Thread Open and Switching Execution TCB

TCBs may be run using the ChangeTh() function, while allocating the second and later TCBs from the OpenTh() function.

```
unsigned long new_th;
new_th=OpenTh(0x80020000,0x1fffff0,0x00);
ChangeTh(new_th);
```

When the ChangeTh() function is called:

- A software interrupt is issued, which causes a jump into an internal kernel interrupt dispatch routine. Other interrupts are not allowed at the same time.
- Context of the ChangeTh() function being executed is shunted into the previously executed TCB.
- The specified TCB is linked to the task status queue
- The context read from the execution TCB is reopened when the interrupt dispatch routine finishes.

By changing the newly-executed V0 register value of the context saved in the previously executed TCB, the return value of ChangeTh() may change when execution is recommenced. From this point on, it is possible to transmit information from the thread space.

Interrupts and TCB

The context at the time of interrupt is stored in the TCB that is currently being executed by the interrupt handler. This content will be kept even during a return from the handler to the main flow and will be saved until the next interrupt.

TCB Status

The status of a TCB can be TcbStUNUSED or TcbStACTIVE. When a thread is opened with OpenTh(), its status becomes TcbStACTIVE and you may execute the TCB with ChangeTh().

Table 2-15: TCB status

Macro	Status
TcbStUNUSED	Not used
TcbStACTIVE	Execution possible

Register Specification Macros

This table shows the macros used to specify the R3000 registers (defined in `asm.h`).

Table 2-16: Register-Specified Macro

Macro (1)	Macro (2)	Contents
R_ZERO	R_R0	0 fixed
R_AT	R_R1	Assembler only
R_V0	R_R2	Return value
R_V1	R_R3	Return Value (for double type)
R_A0	R_R4	Argument #1
R_A1	R_R5	Argument #2
R_A2	R_R6	Argument #3
R_A3	R_R7	Argument #4
R_T0	R_R8	Function-internal work
R_T1	R_R9	Function-internal work
R_T2	R_R10	Function-internal work
R_T3	R_R11	Function-internal work
R_T4	R_R12	Function-internal work
R_T5	R_R13	Function-internal work
R_T6	R_R14	Function-internal work
R_T7	R_R15	Function-internal work
R_S0	R_R16	Function-internal save
R_S1	R_R17	Function-internal save
R_S2	R_R18	Function-internal save
R_S3	R_R19	Function-internal save

Macro (1)	Macro (2)	Contents
R_S4	R_R20	Function-internal save
R_S5	R_R21	Function-internal save
R_S6	R_R22	Function-internal save
R_S7	R_R23	Function-internal save
R_T8	R_R24	Function-internal save
R_T9	R_R25	Function-internal save
R_K0	R_R26	Kernel only #0
R_K1	R_R27	Kernel only #1
R_GP	R_R28	
R_SP	R_R29	Stack pointer
R_FP	R_R30	Frame pointer
R_RA	R_R31	Return previous address
R_EPC		Interrupt return address
R_MDHI		Multiplication/division Register (high)
R_MDLO		Multiplication/division Register (low)
R_SR		Status register
R_CAUSE		Cause register

I/O Management

The PlayStation supports low-level access to files and logical devices. Structures used by the system for input/output are defined by `sys/file.h`.

The following devices are supported:

Table 2-17: IO Devices

Device name	Contents	Example of file designation
cdrom	CD-ROM file system	`cd-rom:PSX.EXE;1`
bu	Memory Card file system	`bu00:ABCD12345`

Each device has a data access unit called its *block size*. All data access is done in multiples of the block size. If there is a fraction in the specified size, it is discarded.

CD-ROM File System

The PlayStation CD-ROM file system conforms to the level 1 format of ISO-9660. File system details are:

Table 2-18: CD-ROM File System (ISO 9660 Level 1)

Device name	cdrom
File format	\<basename\>.\<extension name\>;\<version number\> \<base name\>is 8 letters; \<extension\>up to 3 letters. Only English capital letters, numbers and "_" (underscore) may be used.
Directory name format	\<base name\> \<base name\> is 8 letters. Only English capital letters, numbers and "_"(underscore) may be used.
Directory hierarchy format	Maximum levels in the directory is 8. No root name
File arrangement	Physically arranges all file sectors so they are contiguous.
Block size	2048 bytes

The list of files and directories is only supported as far as it can be stored in one sector (2048 bytes). With standard names (8.3), this allows for up to 45 directories and 30 files per directory. When using short names, it's possible to store more information. **Note:** These limitations can be worked around by having the program keep track of its own files and not using CdSearchFile().

Memory Card File System

TheMemory Card file system manages the files on the removable Memory Card used for saving game data. (Mounting and initialization is performed by libcard BIOS calls). Details of the file system are as follows:

Table 2-19: Memory Card File System

Item	Description
Device name	buXY
	X: port (0: A port, 1 : B Port)
	Y: Extension connector number (1-) or 0
File format	\<base name\>
	\<base name\> ASCII character string to a maximum of 20 bytes.
	Extension cannot be used.
Directory structure	None
Block size	128 bytes

Standard I/O Stream

The standard I/O stream reserves File Descriptors 0 and 1.

On the game unit, the standard I/O stream is assigned to a NULL device. In the development environment, the standard input stream is assigned to a NULL device and the standard output stream is assigned to Debug Message Window #0.

Module Control

Functions are provided to allow you to load and execute application modules.

An execution file conforms to the PlayStation EXE format. It includes:

- Code and data linked to fixed addresses
- A starting address
- A gp register initial value
- Initial value data area starting address and size

Prior to executing the module, the stack area must be explicitly defined, or the current execution context will be used as is.

The execution file is divided into the following three sections:

Figure 2-1: Execution File Memory Map
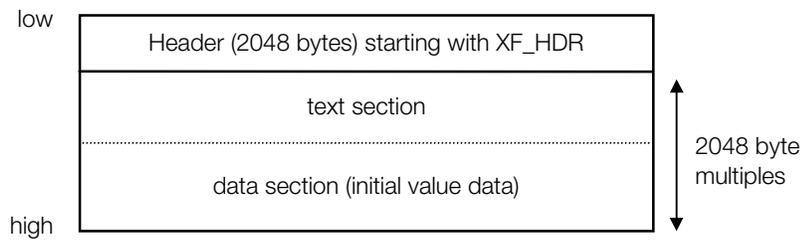
Execution File Data Structure

The execution file is structured as follows:

```
struct EXEC {                    /*execution file information*/
    unsigned long pc0;           /*execution start address*/
    unsigned long gp0;           /*gp register initial value*/
    unsigned long t_addr;        /*text and data section lead address with
                                     initial value*/
    unsigned long t_size;        /*text and data section size with initial
                                    value. */
    unsigned long d_addr;        /* system reserved */
    unsigned long d_size;        /* system reserved */
    unsigned long b_addr;        /*data section lead address with no initial
                                    value. Exec() clears this section; however,
                                    if b_addr is zero, the section doesn't
                                    exist, so it doesn't get cleared.*/
    unsigned long b_size;        /*data section size with no initial value*/
    unsigned long s_addr;        /*stack area lead address (user specified).
                                    Exec() substitutes s_addr+s_size for stack
                                    pointer, unless s_addr is zero. */
    unsigned long s_size;        /*stack area size (user specified)
    unsigned long sp, fp, gp, ret, base;       /*register shunt area for
                                    executing Exec()*/
    };
```

CONFIDENTIAL

The execution file header is structured as follows:

```
struct XF_HDR {                 /*execution file header */
    char key[8};                /*key code*/
    unsigned long text;         /*position of text section within file */
    unsigned long data;         /*position of data section within data file */
    struct EXEC exec;           /*execution file information*/
    char title[60];             /*license code */
};
```

Controller Features

Libapi provides a low-level interface to regulate certain controllers on the PlayStation's main input device. Applications may directly process received data; each type of controller may be identified dynamically.

Initialization

The normal procedure for initializing the controller is shown below:

```
InitPAD(buf0, len0, buf1, len1);
StartPAD();
```

InitPAD() sets up two buffers buf0 and buf1 to receive data from the controllers, and specifies their maximum input lengths, len0 and len1. StartPad() begins reading the controllers, triggered by the vertical blank interrupt.

The presence or absence of the device, as well as its state, may be determined by testing the input buffer's contents.

Buffer Data Format

Data stored in the receive buffer has the following format.

Table 2-20: Summary of Terminal Types

Terminal Type	Controller Name	Main Controller Model Number
1	Mouse	SCPH-1030
2	16-Button Analog	SLPH-00001 (Namco)
3	Gun Controller	SLPH-00014 (Konami)
4	16-Button	SCPH-1080, 1150
5	Analog Joystick	SCPH-1110
6	Gun Controller	SLPH-00034 (Namco)
7	Analog Controller	SCPH-1150
8	Multi Tap	SCPH-1070

Table 2-21: Mouse

Offset	Contents
0	Received result 0x00:Success, Other: Failure
1	Upper 4 bits: 0x1
	Lower 4 bits: Number of received data bytes/2
2,3	Button Status: 1: Release, 1: Push
4	Movement Value X Direction (-128~127)
5	Movement Value Y Direction (-128~127)

Table 2-22: 16-Button Analog

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 Bits: 0x2
	Lower 4 Bits: Number of received bytes/2
2,3	Button status 1: Release, 0: Push
4	Revolution area
0~128~255	
5	I Button 0~255
6	II Button 0~255
7	L Button 0~255

Table 2-23: Gun Controller (Konami)

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 bits: 0x3
	Lower 4 bits: Number of received data bytes/2
2,3	Button Status 1:Release, 0: Push

Table 2-24: 16-Button

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 bits: 0x4
	Lower 4 bits: Number of received data bytes/2
2.3	Button Status 1: Release, 2: Push

CONFIDENTIAL

Table 2-25: Analog Joystick

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 bits: 0x5
	Lower 4 bits: Number of received data bytes/2
2,3	Button status 1: Release, 0: Push
4	Position X Direction (Right stick) 0~128~255
5	Position Y Direction (Right stick) 0~128~255
6	Position X Direction (Left stick) 0~128~255
7	Position Y Direction (Left stick) 0~128~255

Table 2-26: Gun Controller (Namco)

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 bits: 0x6
	Lower 4 bits: Number of received data bytes/2
2,3	Button status 1: Release, 0: Push
4	Position X Direction Lower byte
5	Position X Direction Upper byte
6	Position Y Direction Lower byte
7	Position Y Direction Upper byte

Table 2-27: Analog Controller

Offset	Contents
0	Received result 0x00: Success, Other: Failure
1	Upper 4 bits: 0x7
	Lower 4 bits: Number of received data bytes/2
2,3	Button status 1: Release, 0: Push
4	Position X Direction (Right stick) 0~128~255
5	Position Y Direction (Right stick) 0~128~255
6	Position X Direction (Left stick) 0~128~255
7	Position Y Direction (Left stick) 0~128~255

Table 2-28: Multi Tap Received Data Configuration

Offset		Contents
0		Received result 0x00: Success, Other: Failure
1		0x80
2		Received result 0x00: Success, Other: Failure
3		Upper 4 bits: Terminal types
	Port A	Lower 4 bits: Number of received data bytes+2
4-9		Received data
10		Received result 0x00: Success, Other: Failure
11		Upper 4 bits: Terminal types
	Port B	Lower 4 bits: Number of received data bytes+2
12-17		Received data
18		Received result 0x00: Success, Other: Failure
19		Upper 4 bits: Terminal types
	Port C	Lower 4 bits: Number of data bytes+2
20-25		Received data
26		Received result 0x00: Success, Other: Failure
27		Upper 4 bits: Terminal types
	Port D	Lower 4 bits: Number of receive data bytes+2
28-33		Received data

Table 2-29: Button status bit assignments

Bit	D15	D14	D13	D12	D11	D10	D9	D8
16-button	←	↓	→	↑	ST			SEL
Analog Controller	←	↓	→	↑	ST	R3	L3	SEL
Analog joystick	←	↓	→	↑	ST			SEL
16-button analog	←	↓	→	↑	ST			
Mouse								
Gun controller (Konami)					ST			
Gun controller (Namco)					A			

Bit	D7	D6	D5	D4	D3	D2	D1	D0
16-button	□	X	○	△	R1	L1	R2	L2
Analog Controller	□	X	○	△	R1	L1	R2	L2
Analog joystick	□	X	○	△	R1	L1	R2	L2
16-button analog			A	B	R			
Mouse					Left	Right		
Gun controller (Konami)	TRG	X						
Gun controller (Namco)		B	TRG					

(All bits 1: released, 0: pressed)

The upper 4 bits of the first byte in the buffer are the terminal type, the lower 4 bits are half the value of the number of bytes received from the terminal (stored in or after the 3rd byte of the buffer.) See the terminal documentation for the physical arrangement and correspondence of each button and channel.

CONFIDENTIAL

Kanji Fonts

The PlayStation kernel ROM includes 16 dot x 16 dot 2-value bitmap kanji fonts. Font data must not be stored consecutively in memory to accommodate memory capacity. Use the service function to obtain the starting address of the data for each character.

Table 2-30: Kanji Fonts

Item	Description
Data Format	16 dot x 16 dot 2 value bitmap Character size is 15 dot x 15 dot
Contents	JIS 1st standard kanji and non-kanji; gothic type non-kanji have a top space (0x2121) Refer to the codeview samples in \psx\kanji\sjiscode for a list of usable fonts and the fonts themselves.
Code System	Shift-JIS
Access Method	The starting address in ROM of the applicable font pattern may be obtained from the shift-JIS code given to the service function. With that information, the font pattern may be accessed directly.

Data Format

In the figure below, the byte of the upper left of the pattern is first, followed by the byte on the upper right. The most significant bit (MSB) faces left.

Table 2-31: Font Data Format

#0	#1
#2	#3
:	:
:	:
:	:
#30	#31

Usage Example

In the following sample program, the function _get_font() returns a font pattern corresponding to the specified shift-JIS code. This pattern is in a format that can be transferred to VRAM as a 16-bit texture.

Example: Getting a Kanji Font

```
unsigned long _get (char *sjis)
{
        unsigned short sjiscode;
        sjiscode = *sjis << 8 | *(sjis+1);
        return Krom2RawAdd(sjiscode);  /* get kanji font pattern address */
}


#define COLOR 0x4210
#define BLACK 0x3000

void _get_font (char *s, unsigned short *data )
{
        unsigned short *p, *d, w;
        long i,j;
        if ((p=(unsigned short *)_get(s))!=-1)
        {
                d = data;
                for (i=0; i<15; i++)
                {
                        for(j=7; j>=0; j--)
                                *d++ = (((*p>>j)&0x01)==0x01)?COLOR:BLACK;
                        for(j=15; j>=8; j--)
                                *d++ = (((*p>>j)&0x01)==0x01)?COLOR:BLACK;
                        p++;
                }
        }
        else
        {
                for (d=data, i=0; i<2*16*16; i++)
                        *d++ = BLACK;
        }
}
```

CONFIDENTIAL

Memory Allocation

There are three systems of memory allocation: a ROM-based version (malloc), a RAM-based version (malloc2), and a high-speed RAM-based version (malloc3).

There is a bug in the malloc system in which the area allocated cannot be completely released in free().This bug was fixed in malloc2. (Since malloc is part of the ROM, it cannot be corrected and is left in order to maintain compatibility.) malloc3, which improved upon the speed of malloc2, was added in Library 4.0.

Note: all of these functions allocate memory blocks based on first fit rather than best fit. In some cases, developers may wish to write their own memory allocation routines.

Table 2-32: Memory Card allocation functions

Name	LIBAPI function	LIBC/C2 function
Version which calls ROM routine (malloc system)	InitHeap()	malloc()
		calloc()
		realloc()
		free()
RAM-based version (malloc2 system)	InitHeap2()	
	malloc2()	
	realloc2()	
	calloc2()	
	free2()	
High-speed RAM-based version (malloc3 system)	InitHeap3()	
	malloc3()	
	realloc3()	
	calloc3()	
	free3()	

The table below shows a performance comparison between the three memory allocation systems, in terms of speed of operation and size of code.

Table 2-33: Performance comparison between memory allocation functions

	Speed	
Code size	Slow	Fast
Large	malloc2 system	
		malloc3 system
Small	malloc system	

CONFIDENTIAL

Chapter 3:
Standard C Library

Table of Contents

Overview

The C standard libraries are a subset of the K & R-based C standard libraries, including functions such as character functions and memory operations .

There are two versions of the standard C libraries:

- libc accesses library routines in the kernel ROM. This provides a small size advantage, since no additional code needs to be linked with your application. However, libc routines are slower than libc2 routines, because ROM code is not cacheable.
- libc2 links with your application. It provides a speed advantage, because the code is cacheable.

Library and Header Files

The standard C library files are `libc.lib` and `libc2.lib`. To use standard C routines, you must link with one of these files.

The following header files declare the routines in the C standard library. The *Run-Time Library Reference* describes the functions in the standard C library and which header file must be included for each one.

Table 3–1: Header Files

  abs.h
  assert.h
  convert.h
  ctype.h
  malloc.h
  memory.h
  rand.h
  setjmp.h
  stdarg.h
  stddef.h
  stdlib.h
  strings.h
  qsort.h
  sys/types.h

Chapter 4:
Math Library

Table of Contents

Overview

The Math library provides a floating point operation package and K & R-based standard C library math functions.

Library and Header Files

To use the Math library file, your application must link with the file `libmath.lib`.

Your source code should include the header files `libmath.h` and `limits.h`.

Floating-Point Numbers

The math library supports IEEE754 standard single-precision floating-point numbers (float) and double-precision floating-point numbers (double). It also has an internal floating-point arithmetic operation package.

The PlayStation hardware doesn't support float and double operations directly, because the CPU has no floating-point coprocessor. By linking mathlib with your application, it is possible to use the float and double types.

Table 4–1: Float Format

Item	Specification
Size	4 bytes
Significant digit count	6 decimal digits
Overflow limit value	$2.0^{**}128 = 3.4e38$
Underflow limit value	$0.5^{**}126 = 2.2e-38$

Table 4–2: Double Format

Item	Specification
Size	8 bytes
Significant digit count	15 decimal digits
Overflow limit value	$20^{**}1024 = 1.8e308$
Underflow limit value	$0.5^{**}1022 = 2.2e-308$

Error Processing

Events are used to report errors in floating-point operations. Error status recording by C standard style external variables is also supported.

Error Types

Math library functions are used to test the range of arguments. These tests are performed on the functions whose specifications cover the range of argument values. If an inappropriate value is detected, the response "area error" (EDOM) is generated.

If the results exceed the area of expression in an application which uses internal functions and arithmetic operators, the response "range error" (ERANGE) is generated.

Internal Processing at the Time of an Error

For any area and range errors, notice is given by the assignment of an error code to an event and external variables.

The result of an operation is an unsigned infinite value, so that operation can be carried on wherever possible. The following are positive infinite bit patterns:

- Floating-point value: 0x7F800000
- Double-precision value: 0x7FF0000000000000

The following are negative infinite bit patterns:

- Floating-point value: 0xFF800000
- Double-precision value: 0xFFF0000000000000

The following are return values for division by zero:

- NaN
- Floating-point value: 0x7FFFFFFF
- Double-precision value: 0x7FFFFFFFFFFFFFFF

or

- -NaN
- Floating-point value: 0xFFFFFFFF
- Double-precision value: 0xFFFFFFFFFFFFFFFF

(NaN is not a numerical value, but a bit pattern reserved by the operation subroutine to report an error. A normal double-precision variable does not store the same bit pattern as NaN. Thus, subjecting NaN to floating-point operation cannot provide correct results.)

Error Event

An error in a math library function causes an event with cause descriptor SwMATH. Thus, an overflow and division by zero can be detected and a corresponding error generated.

Error Variable

The variable math_errno for storing error codes is defined in `libmath.lib` and declared as extern in the header file `libmath.h`; it is initialized to zero. When an error arises in the library, however, one of the constants EDOM or ERANGE (defined in sys/errno.h) is stored in math_errno. This variable is not automatically reset to zero; you must explicitly reset it after error processing.

Table 4–3: Error Notificaton

Error math_errno	Event value	Cause descriptor	Type
Area error	EDOM	SwMATH	EvSpEDOM
Range error	ERANGE	SwMATH	EvSpERANGE

Chapter 5:
Memory Card Library

Table of Contents

CONFIDENTIAL

Overview

The Memory Card library provides functions which make smooth access to the Memory Card in a realtime environment possible. It also performs data reading and writing and calls the Memory Card BIOS service.

Library and Header Files

Programs that use the Memory Card library must link with the file `libcard.lib`.

The Memory Card library has no unique header file. The header files `libapi.h` and `sys/file.h` must be included.

Memory Card

The Memory Card is a memory device that saves data after a reset or power-off. The Memory Card may be inserted or removed while the power is on.

Hardware

The basics of Memory Card hardware are as follows:

Table 5-1: Memory Card Specifications

Feature	Specification(s)
Capacity	120 Kbytes at format (accessed in 128-byte sectors)
Communication Configuration	Synchronous serial communication sharing controller port
Access Speed	1. Cannot access for 20 ms after reading 1 sector
	2. Maximum continuous reading speed is about 10 Kbyte/sec.
Other	May insert/remove without turning power off
	100,000 reads guaranteed

Events

The Memory Card library uses the following two source descriptors. Also, the Memory Card library does not use internal event descriptors.

Table 5-2: Events Associated with the Memory Card

Source descriptor	Event class	Meaning
HwCARD	EvSpIOE	Processing complete
	EvSpERROR	Card no good
	EvSpTIMOUT	No card

Source descriptor	Event class	Meaning
SwCARD	EvSpIOE	Processing complete
	EvSpERROR	Card no good
	EvSpTIMOUT	No card
	EvSpNEW	New card or uninitialized card

Note: SwCARD/EvSpNEW has one of two meanings, depending on the function that issued the input/output request.

Automatic clearing of events relating to HwCARD

Events related to the descriptor HwCARD are automatically cleared by every vertical sync interrupt.

Functions which wait for a vertical interrupt, such as libgpu VSync(), etc., interpose themselves to perform event generation tests, and so run the danger of not being able to detect event generation.

BIOS

Services such as checking the Memory Card connection, logical format testing, accessing in sector units (128 bytes), etc., are provided by the BIOS.

In order to support concurrent controller reading and the accessing of two AB connectors, the BIOS accesses the card at each of two vertical blanks. One sector, 128 bytes of data, may be read in 1 access. Access using BIOS is as follows:

Table 5-3: Memory Card BIOS

Feature	Description
Start Timing	After a vertical blanking interrupt, controller reading occurs, the card connection is checked and then the hardware is checked.
	Sending and receiving of data is driven by receiving interrupts in units of bytes.
Effective Speed	Effective speed 30 sectors/sec = 3.75 Kbyte/sec
CPU Load	2.5% when reading continuously from 2 cards
	3.2% when writing continuously to 2 cards

Testing for Card Presence and Testing Logical Formats

The procedure for testing in the BIOS for the presence of a Memory Card and for logical format is as follows:

1. Test for card presence using _card_info().

    If an IOE event has occurred, a card whose connection has already been confirmed remains connected. Go to (5).

    If a NEWCARD event has occurred, a card which was not confirmed by _card_clear() after connection is connected. Go to (2).

    If a TIMOUT event has occurred, no card is connected. No more operations are necessary. A communication error is possible, so perform a retry.

2. Perform a confirmation operation using _card_clear().

    Usually there is no failure. If a failure occurs, either the card was removed or a communication error occurred. In the case of failure, return to (1) and perform a retry.

3. Test logical format using _card_load().

   If an IOE event has occurred, formatting is completed. Go to (5).

   If a NEWCARD event has occurred, formatting has not been performed. Go to (4).

   In other cases, either the card was removed or a communication error occurred. In these cases, return to (1) and perform a retry.

4. Perform logical format using format().

   If formatting ends normally, go to (5). In other cases, either the card was removed or a communication error occurred. In such cases, return to (1) and perform a retry.

5. Perform input/output using the file system.

Unconfirm Flags

Inside the card there is a bit switch called the unconfirm flag. This bit is set if the card is inserted in its slot, and is cleared by _card_clear(). This flag provides a means for detecting card replacement. In order to prevent erroneous access, the default is that data cannot be read from or written to a card with this flag set. Any attempt to read or write causes an error. The flag may be accessed after explicitly clearing it with _card_clear().

If you want to create an error for testing, etc., the _new_card() function masks the default test parameters in order to ignore the unconfirmed flag and allow access. This is a function which does not require normal access through the filesystem, so it is different from other libcard functions.

Card Test

Here is a list of sample code for testing cards. See the following section "File System" for the events used.

```
unsigned long ev0,ev1,ev2,ev3;
unsigned long ev10,ev11,ev12,ev13;

main()
{
      ev0 = OpenEvent(SwCARD, EvSpIOE, EvMdNOINTR, NULL);
      ev1 = OpenEvent(SwCARD, EvSpERROR, EvMdNOINTR, NULL);
      ev2 = OpenEvent(SwCARD, EvSpTIMOUT, EvMdNOINTR, NULL);
      ev3 = OpenEvent(SwCARD, EvSpNEW, EvMdNOINTR, NULL);
      ev10 = OpenEvent(HwCARD, EvSpIOE, EvMdNOINTR, NULL);
      ev11 = OpenEvent(HwCARD, EvSpERROR, EvMdNOINTR, NULL);
      ev12 = OpenEvent(HwCARD, EvSpTIMOUT, EvMdNOINTR, NULL);
      ev13 = OpenEvent(HwCARD, EvSpNEW, EvMdNOINTR, NULL);

      PadInit(0);
      InitCARD(1);
      StartCARD();
      _bu_init();
      test_card();
}

test_card()
{
      long ret;

      _card_info(0x00);       /* deliver a TEST CARD request */
      ret = _card_event();  /* get the result */
      if(ret==1 || ret==2)) /* NO CARD or Communication error */
            goto skip;
      if(ret==3) {    /* if NEWCARD, call _card_clear() */
            _clear_event();
```

```
                  _card_clear(0x00);      /* clear NEW CARD FLAG */
                  ret = _card_event();    /* wait events */
            }
        _clear_event();
        _card_load(0x00);       /* deliver a TEST FORMAT request */
        if(ret==3) {   /* if NEWCARD, call format() */
                /* put a message to the operator */
                ret = format("bu00:");  /* synchronous function */
                if(ret==1)
                        FntPrint("\nDONE\n");
                else {    /* error happened in format() */
                        FntPrint("\nERROR IN FORMATTING\n");
                        goto skip;
                }
        }
        /* put i/o requests */
        return 1;
skip:
        return 0;
}
_card_event()
{
        while(1) {
                if(TestEvent(ev0)==1) { /* IOE */
                        return 0;
                }
                if(TestEvent(ev1)==1) { /* ERROR */
                        return 1;
                }
                if(TestEvent(ev2)==1) { /* TIMEOUT */
                        return 2;
                }
                if(TestEvent(ev3)==1) { /* NEW CARD */
                        return 3;
                }
        }
}
_clear_event()
{
        TestEvent(ev0);
        TestEvent(ev1);
        TestEvent(ev2);
        TestEvent(ev3);
}
_card_event_x()
{
          while(1) {
                  if(TestEvent(ev10)==1) {  /* IOE */
                      return 0;
                  }
                  if(TestEvent(ev11)==1) {  /* ERROR */
                      return 1;
                  }
                  if(TestEvent(ev12)==1) {  /* TIMEOUT */
                      return 2;
```

CONFIDENTIAL

```
            if(TestEvent(ev13)==1) {  /* NEW CARD */
                    return 3;
            }
        }
}

_clear_event_x()
{
        TestEvent(ev10);
        TestEvent(ev11);
        TestEvent(ev12);
        TestEvent(ev13);
}
```

Use with the Multi Tap

When switching between multiple Memory Cards connected to one Multi Tap, _card_load() must be executed each time a different Memory Card is accessed.

This is because each port of the PlayStation console has only one directory information buffer and libcard can only control one directory information for multiple Memory Cards connected to one Multi Tap.

File System

The file system as it relates to the Memory Card is as follows:

Table 5-4: Memory Card File System

Feature	Description
Device Name	buX0X: Connector number (0 or 1)
File Name	ASCII characters, up to 21 characters
Directory Structure	None
Control Unit: Slot	8 Kbyte (64 sectors) --> file size unit
Number of Slots	15/card (max. no. of files = 15)
Automatic Replacement Sector Function	20 replacement sectors/card

Kernel library services which request a file name as an argument may be applied to all bu devices.

File size is given as a parameter during file creation. Afterwards the file size cannot be changed. Size is in units of slots. During file creation, the file system must combine any fragmented memory regions left after deleting files and guarantee the needed capacity.

Example: File Deletion and Creation:

```
/* Driver initialization */
InitCARD(0);  /* Does not coexist with controller */
StartCARD();
_bu_init();

/* Delete file L01 on the card in Port A */
printf("delete\n");
delete("bu00:L01");
```

```
/* Create new file L01, 2 slots long, on card in Port A */
printf("create\n");
if((fd=open("bu00:L01",O_CREAT|(2<<16)))==-1)
        printf("error\n");
close(fd);
/* Always close once after creating */
```

Realtime Access

Device bu assumes operation under a realtime environment and supports non-blocking mode. If the macro 0_NOWAITin sys\file.h is used when open, read() and write() end as soon as an input/output request is registered in the driver. Completion of input/output is reported by posting an event.

A slot accepts only one input/output request for checking access speed.

Example: Asynchronous Access

_clear_event() and _card_event() have the same contents as the previous example

```
sample()
{
      long fd,i,ret;
      fd = open("bu00:L01",O_WRONLY|O_NOWAIT);
      printf("open=%d\n",fd);
      for(i=0;i<50;i++) {
            clear_event();
            while((ret = write(fd,data,384))!=0)
                  ;
            printf("write=%d\n",ret);
            ret = _card_event();
            printf("event=%d\n",ret);
            if(ret==1)
            break;
      }
      close(fd);
}
```

Rules for Use of Memory Card

The Memory Card is a resource shared by many applications, so use it according to the rules for sharing.

Abnormal Processing

No standard screen or message is set up to deal with cases of insufficient capacity or detection of an unformatted card while executing an application. Each application should have an abnormal processing screen or message designed for it.

Keep the following points in mind during this design process.

1. Always query the user (game-player) when performing logical initialization. Do not use the automatic initialization function.
2. When a card is not detected, and it is determined that this may limit future operation, always notify the user (game-player). If possible, ask the user whether it is okay to continue processing.

CONFIDENTIAL

Terminology

The unit for required memory capacity in the product catalog is block. This is equivalent to the previously-noted slot (8 Kbytes).

File Names

Use the following structure for file names:

Table 5-5: Memory Card File Names

Bytes	Contents	Notes
0	Magic Number	Always 'B'
1	Region	Japan: 'I' North America: 'A' Europe: 'E' (*1)
2-11	Title	SCE product number (*2)
12-20	User/Public	Use only non-0x00, 0x2a("*"), 0x3f("?") ASCII End with 0x00

*1: None are checked by the system

*2: The first disk for multi-disk titles

The SCE product number is decided by our Release Planning Committee (about three weeks before the master is released), and reported to the responsible parties in each company's sales department. Based on this, please decide the following.

Example: If the product code is SLPS-00001, the file name's first 12 characters are BISLPS-00001. Always add zeros to make the numerical portion 5 digits.

File Headers

Put the following headers at the start of each file.

Table 5-6: Memory Card File Header

Item	Size (bytes)
Header	128
Magic number	2 (always "SC")
Type (see table below)	1
No. of slots	1
Text name	64 (Shift JIS, *1)
Pad	28 (All packed in 0x00)
CLUT	32
Icon image (1)	128 (16 x 16 x 4 bits)
Icon image (2)	128 (Type:0x12, 0x13 only)
Icon image (3)	128 (Type:0x13 only)
Data	Varies (128Byte x N)

*1: Non-kanji and primary standard kanji only, full-size 32 characters. The end of the character string terminates at 0x00.

Table 5-7: Type Field

Type	Number of icon images (automatically replaced animation)
0x11	1
0x12	2
0x13	3

Written Data Contents Protection

Applications should take precautions to prevent damage to data in the event of a reset or card removal or power off during data writing.

For example, you can set things up so that data is written in duplicate. Writing is performed reciprocally and an individual checksum is added for the final byte of each sector. Test checksum when reading, and use the other data set if an error is detected.

Warning: the file system replacement sector function is only effective on card memory writing errors. The writing contents guarantee function is not supported by hardware or library.

Example: Sector Checksum

```
/*
 * test check sum for 128byte block
 * return     1:OK
 *     0:NG
 */
_test_csum(buf)
unsigned char *buf;
{
    long i;
    unsigned char c;
    c = 0x00;
    for(i=0;i<127;i++)
        c ^= *buf++;
    if(*buf==c)
        return 1;
    return 0;
}

/* set check sum to the last byte of 128byte block */
void _set_csum(unsigned char *buf)

{
    long i;
    unsigned char c;
    c = 0x00;
    for(i=0;i<127;i++)
        c ^= *buf++;
    *buf = c;
}

/* sample data strucure */
struct SDB {
    char name[8];
    long size, attr, sector, mode;
}

/* common load buffer */
unsigned char load_buf[1024];
```

CONFIDENTIAL

```
/* get data from Memory Card with checksum test */
int get(long num, struct SDB *data)
{
        long i,fd;

        if((fd=open("bu00:L01",O_WRONLY))<0)
                return 0;
        memcpy(&load_buf[0],data,sizeof(struct SDB));
        set_csum(&load_buf[0]);
        i = write(fd,&load_buf[0],128);
        close(fd);

        return (i==128)?1:0;
}

/* get data from Memory Card with checksum test */
int get()
{
        long i,fd;

        if((fd=open("bu00:L01",O_RDONLY))<0)
                return 0;
        if(read(fd,&load_buf[0],1024)!=1024) {
                close(fd);
                return 0;
        }

        for(i=0;i<8;i++)
                if(_test_csum(&load_buf[128*i])==1)
                        memcpy(&data[i],&load_buf[128*i],sizeof(struct SDB));
                else
                        memset(&data[i],0xff,sizeof(struct SDB));
        close(fd);
        return 1;
}
```

Handling Communications Errors

There are cases in which access fails due to static discharge or power source noise even though the card connection and access program are normal. Test for the presence or absence of a card, writing and reading with retry (at 1-2 second intervals).

Other

Coding Notes

Consider the following point when coding: Call _new_card() before _card_info() and suppress EvSpNEW events.

Known Bugs

- If read() or write() is issued immediately after open(), an error occurs. When creating a file using open(), make sure you call close() to close the file.
- In asynchronous access using read(), the file pointer is updated by 128 bytes too few. It must be corrected using lseek().

CONFIDENTIAL

Chapter 6:
Extended Memory Card Library

Table of Contents

CONFIDENTIAL

Overview

The high-level Memory Card library (libmcrd) provides a convenient interface for using Memory Cards installed in the PlayStation.

Library and Header Files

Programs that use extended Memory Card library services must link with the file `libmcrd.lib`. Internally, libmcrd uses `libcard.lib` and `libapi.lib`, so these libraries must also be linked.

Source files must include the header file `libmcrd.h`.

Features of the Library

- Check for presence of Memory Card, check to see if Memory Card is uninitialized, and check for Memory Card invalid state
- Write data to Memory Card
- Read data from Memory Card
- Logical initialization (formatting) of Memory Card
- File deletion
- File creation
- Get directory information

Checking Memory Card Status

The Memory Card can be inserted or removed when the PlayStation is on. Thus, the user application must be designed to take into account the fact that the Memory Card may be inserted or removed at any time.

Libmcrd provides the MemCardAccept() function to determine if a card has been inserted or removed. The status of the card can be obtained by this function when the card is installed.

Reading/Writing Data

When data is being written to or read from the Memory Card, communication errors may occur. Libmcrd internally performs retries when communication errors occur during data reads or writes. However, the written or read data must be checked by the user application using a verification process, checksum, or other method to verify that it is correct.

Detecting a New Card

A Memory Card that has just been inserted is treated as a new card. Because the new card may be unformatted or invalid, the library is designed so the card cannot be accessed until MemCardAccept() is executed.

If the new card was detected with MemCardAccept(), however, it is not necessary to execute MemCardAccept() again, as the various processes such as format checking were already performed.

If a new card is inserted, any function other than MemCardAccept() (such as MemCardExit() or MemCardReadFile()) will return "New card detected" as its result no matter how many times it is called.

Libcard and the Card BIOS

Libmcrd uses libcard and card BIOS functions and resources such as the HwCARD and SwCARD events. Consequently, user applications cannot directly use libcard or the card BIOS.

If a user application needs to perform an operation that cannot be implemented using libmcrd, the application must implement all Memory Card operations which use libcard and the card BIOS.

Use with Multi Tap

When switching access between multiple Memory Cards connected to one Multi Tap, call MemCardAccept() every time you access a different Memory Card. The reason for this is that in libmcrd, each port on the PlayStation unit has only one directory information buffer. When multiple Memory Cards are connected to one Multi Tap, only one directory information can be controlled

The Memory Card

The Memory Card is a storage device that retains data even after the PlayStation has been powered off or reset.

Memory Cards can be inserted or removed while the PlayStation is turned on.

Hardware

The specifications for the Memory Card hardware are shown below.

Table 6-1: Memory Card Specifications

Capacity	120 KBytes (formatted)
	(Accessed in 128 byte sectors)
Communication	Synchronous serial port also serving as a controller port
Access speed	(1) No access for 20 msec after writing one sector
	(2) Approximately 10 KBytes/sec maximum continuous read
Other	Can be inserted or removed without turning power off.
	Guaranteed for 100,000 writes

Rules for Using the Memory Card

The Memory Card is a resource that is shared by multiple applications. Therefore the Memory Card should be used according to a common set of rules.

Handling Irregularities

There is no required screen or message to be output when a card runs out of memory during application execution or when an unformatted card is detected (i.e., there is no requirement that the display be the same as OSD). Rather, these situations can be customized according to the application. Nevertheless, the following points should be taken into consideration.

- The user should be notified before formatting occurs. Formatting should not be performed automatically.

- The user should be notified when no card is detected but one is expected. If possible, the user should be prompted to insert a card.

CONFIDENTIAL

Terminology

In the product catalogs, the required memory capacity is expressed in terms of "blocks", where one block is 8192 bytes.

File Names

File names should be assigned as follows:

Table 6-2: Memory Card Filenames

Byte	Description	Notes
0	Magic number	Always 'B'
1	Location	'I' for Japan, 'A' for North America, 'E' for Europe (*1)
2-11	Title	SCE product number (*2)
12-20	User defined	Use ASCII characters excluding 0x00, 0x2a ("*"), 0x3f ("?"). End with 0x00

*1: Not checked by the system

*2: If multi-disc title, use the product number from the first disc.

The SCE product number will be determined at a preliminary sales meeting held by us (approximately three weeks before submission of master) and we will notify the business contact of the subject company. Please use the product number in the following manner.

For example, if the product code is "SLPS-00001",

The first 12 characters of the filename would be BASLPS-00001".

The numerical portion must be five digits padded with zeros.

File Header

Please use the following header at the beginning of each file:

Table 6-3: Memory Card File Header

Item	Size (bytes)
Magic number	2 (always 'SC')
Type (see table 6-4)	1
Number of blocks	1
Name	64 (Shift-JIS *1)
pad	28 (All packed at 0x00)
Clut	32
Icon image (1)	128 (16x16x4 bits)
Icon image (2)	128 (Type==0x12,0x13 only)
Icon image (3)	128 (Type==0x13 only)
Data	Variable (128 bytes x n)

*1: Non-kanji and Level 1 kanji only. Full-width, 32 characters.
However, 0x84bf to 0x889e cannot be used.
The end of the character string terminates at 0x00.

Table 6-4: Type Field

Type	Number of icon images (Animation through automatic replacement)
0x11	1
0x12	2
0x13	3

Saving Write Data

The application must handle cases where data is destroyed because the unit was reset, the card was removed, or the power was turned off during a data write operation.

To save data: Write data twice, writing to one data set and then to the other. At the end of each sector add a checksum for the sector. Do a checksum test when reading the sectors. If an error is detected, use the other data set.

Caution: The replacement sector feature of the file system is valid only for memory write errors in the Memory Card. There is no hardware or library support for saving write data contents.

CONFIDENTIAL

Chapter 7:
Data Compression Library

Table of Contents

Overview

The data compression library (libpress) is a low-level function library for compressing (encoding) and decompressing (decoding) image and sound data.

Image data that can be compressed and decompressed includes:

- Single images.
- Frames from a video sequence that have been compressed into the PlayStation MDEC format.

  The MDEC is a customized portion of the PlayStation hardware specializing in image decompression.

Three methods of compressing images are available:

- DCT (Discrete Cosine Transform) can be used to compress direct color images
- BVQ (Block Vector Quantization) likewise combines the number of colors in the direct color image together to create 256/16 colors
- Huffman Encoding (fixed codebook) reversibly compresses 4 bit index colors.

For compressing sound data, the library uses ADPCM to compress 16-bit straight PCM to about 1/4. The compressed sound data can be used as SPU sound source data.

Library and Header Files

The filename of the data compression library is `libpress.lib`; to use library services, you must link with this file. The library header is `libpress.h`; programs that call library routines must include this file.

Compressor and Decompressor Functions

Compressor functions compress image and sound data in main memory, and return the results to main memory. Compressor functions are used when data needs to be compressed dynamically inside an application, and when data is generated off-line by remote activation from the authoring environment. In fact, the local environment has a built-in DCT circuit which can be used to carry out high-speed compression of images by means of DCT.

Decompressor functions expand compressed data in real time. Note that in some cases, compressor functions produce data formats that are processed without conversion but rather via local environment hardware, like BVQ. Data in these formats cannot be handled by decompressor functions.

MDEC

The PlayStation provides a specialized data display engine, the MDEC (Motion DECoder), for high-speed image data expansion. MDEC expands compressed data in main memory and returns the result back to main memory. This result is transferred to the frame buffer display area, and displayed as an image.

Figure 7-1: Data Expansion and Display by MDEC
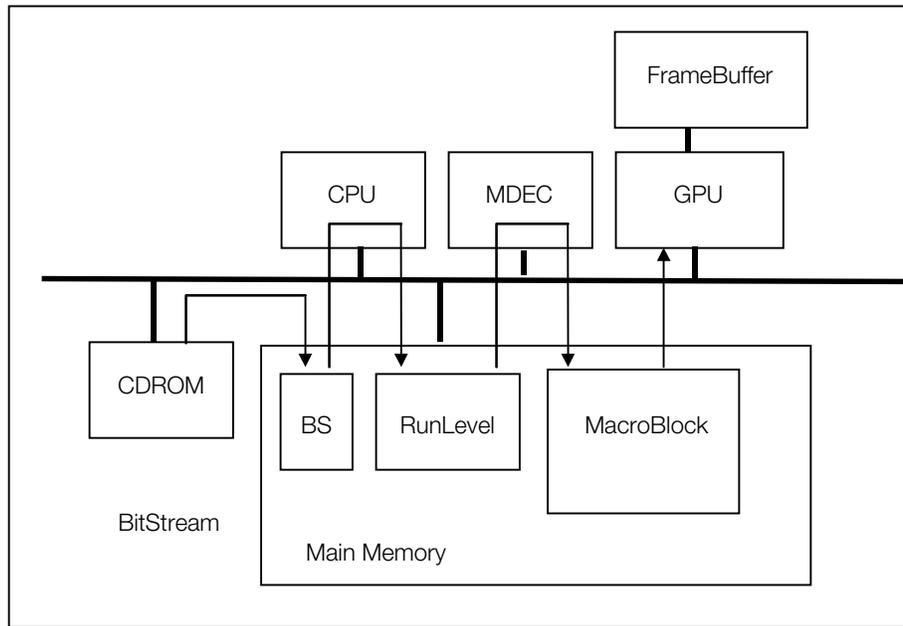
The main bus access which was saved to the main memory is carried out by time sharing with the CPU and other peripheral equipment and can perform expansion processing in parallel with the program and frame buffer transfer, etc.

Compression of Image Data

Algorithms used to compress image data vary according to the type and intended use of the data.

DCT (Discrete Cosine Transform)

DCT is the compression method used in JPEG/MPEG. It compresses direct-color (24-bit/16-bit) images with a high efficiency ratio. The compression is lossy, but the compression ratio can be controlled at will. The compression ratio specified is usually between 5% and 10%.

In DCT, the basic processing unit is a 16x16 24-bit direct-color image called a *macroblock*. All the images are broken down into macroblocks before being compressed into *bitstream format*. The output of decompression is also in macroblock units.

For example, when 320x240 image data is broken down into a large number of 16x16 macroblocks, as shown below, they are each compressed into bitstreams.

Figure 7-2: 320x240 Image Breakdown
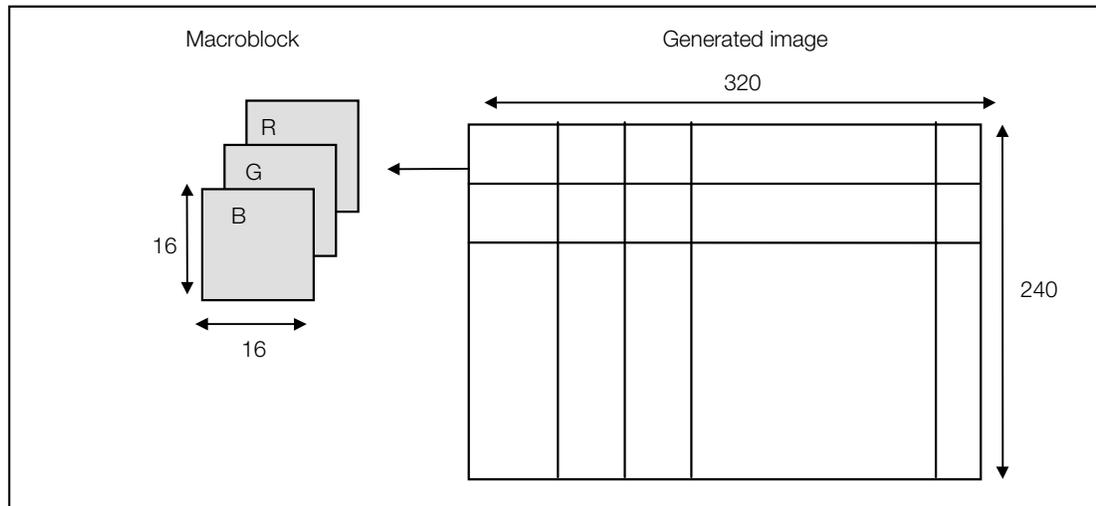
Figure 7-3: DCT Processing

Macroblock (16x16 RGB rectangular area)

↓ compression

Bitstream

↓ decompression

Macroblock (16x16 RGB rectangular area)
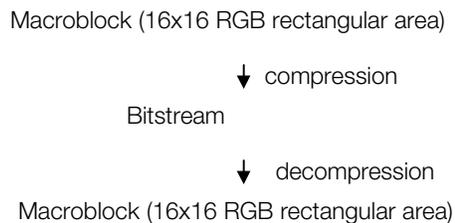
BVQ (Block Vector Quantization)

BVQ carries out vector quantization on direct-color images, combining colors to give a total of 256 or 16 colors, and generating 8-bit or 4-bit index-color images.

Index-color images are expressed as a two-dimensional array consisting of the CLUT (Color Look Up Table) which gives the actual brightness values, and the index to the CLUT.

Index-color images allow a slightly greater total reduction in data volume than the equivalent direct-color images. For example, if the brightness value of the individual pixels in a picture is 16 or below, the index only takes 4 bits. The volume of an index-color image can therefore be compressed to 25% of the volume of a 16-bit direct-color image.

4-bit/8-bit index-colors can be used as 4-bit/8-bit texture-patterns, doing away with the need for a special decompression filter.

In BVQ, the image is split up into several small areas when compression is carried out, and vector quantization is carried out on each small area, allowing the number of colors to be reduced by combination. At this stage, vector quantization is carried out again on the CLUT generated for each small area, so the number of CLUTs can also be reduced by combination. In this case, each pixel of the image data is indexed doubly: once by the CLUT number held by the small area to which the pixel belongs, and by the index value for that CLUT.

Vector quantization in which the index reference is carried out in stages in this way is called Block Vector Quantization.

Huffman Encoding

DCT and BVQ compression and decompression are lossy. Therefore, a Huffman encoding function is provided for reversible compression of 4-bit index colors. The Huffman encoding is the classical type in which the codebook is generated once at the beginning.

Huffman encoding compresses data by assigning codes with a short code length (Huffman codes) in order, starting with the pixel values (index values) which appear most frequently. The table showing the actual pixel values and their corresponding Huffman codes is called the codebook.

The compression ratio for Huffman code varies according to the nature of the source image. Generally, the greater the polarization of the pixel values appearing, the higher the compression ratio will be.

The following table summarizes the compression and decompression methods:

Table 7-1: Compression and Decompression Algorithms

	DCT	BVQ	Huffman
Type	Lossy	Lossy	Loss-less
Input format	24-bit/16-bit	24-bit/16-bit	4-bit
Output format	BitStream	4-bit/8-bit	BitStream
Compression ratio	From 10% to 5%	From 50% to 25%	

DCT (Discrete Cosine Transform)

Basic Principles

Compression

DCT belongs to the category of linear transforms generally termed direct transforms, and can be thought of as a kind of frequency transform.

When DCT conversion is carried out on an NxN rectangular image, the low-frequency constituents of that image are concentrated in one place. Compression of the data is achieved by Huffman-encoding the results. In short, DCT is a method for making data compression easier, and does not, in itself, reduce the data size. The actual data compression is done by the Huffman encoding.

When DCT conversion is carried out on an ordinary image, the frequency constituents are concentrated in the low region, so after conversion, most of the constituents are at 0. This means that a much higher compression ratio can be achieved than if the image had been Huffman-encoded directly. This type of Huffman-encoding is called VLC (Variable Length Coding).

The byte/word boundary of VLC-processed data is logically meaningless, and the data is expressed simply as a stream of bits. This is known as a bitstream.

The basic unit for all the processes in this sequence is a 16x16 rectangular area. This unit is known as a macroblock. Accordingly, in DCT compression, macroblocks can be input, compressed, and converted to bitstream format.

After the image has been subjected to DCT conversion, quantization is carried out all at once in given units. The compression ratio can be controlled by controlling the quantization step. Generally speaking, broadening the quantization step improves the compression ratio.

Decompression

DCT decompression is carried out in the reverse order to that of compression. That is to say, once VLC decoding has been carried out on the captured bitstream, the result is subjected to IDCT (Inverse Discrete Cosine Transform) to restore the original image.

CONFIDENTIAL

The decompression of the bitstream therefore consists of two passes:

1. VLC decoding
2. IDCT

Methods Supported

Compression

In the case of 24-bit color data, intermediate data is output in a format (run level) where the run-length is compressed once DCT conversion has been carried out. This data is subjected to VLC, and a bitstream is output. The compression ratio is controlled by specifying the quantization step in the process generating the run level.

When the actual compression is carried out, the run level (the intermediate data) is not output.

Figure 7-4: DCT Compression

24bit 16x16 macroblock

↓ ←   compression ratio setting:
            Lossy conversion

run-level (intermediate result: not output)

↓ ←   Huffman encoding:
            Lossy conversion

bitstream
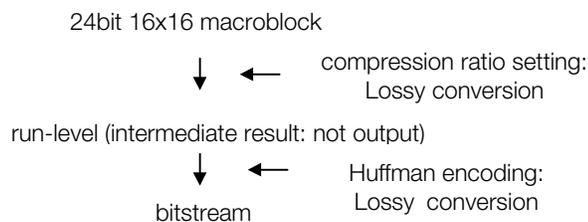
Macroblock encoding is performed in the following fashion:

*   Performs CSC (Color Space Conversion) on the RGB macroblock to convert the Y, Cb and Cr elements. Y is the brightness element and Cb, Cr are color difference elements.

*   Within the YCbCr macroblock, divides Y into four 8x8 blocks. Thins out Cb, Cr and arranges them as 8x8 macroblocks. As a result, the YCbCr macroblock is divided into six blocks (Y0, Y1, Y2, Y3, Cb, Cr).

*   Converts each block by DCT (Discrete Cosine Translation).

*   Quantizes (divides) each element of the block as a fixed value.

*   Lists each element of the block in zig-zag order.

*   Run length compresses each element of the block and converts to run level.

*   Performs VLC (Huffman encoding) on the run level and creates BS.

Decompression is carried out by operations which are the reverse of those used in compression.

The image data handled in DCT is 24-bit direct-color data, but the bitstream produced by compressing this data can be decompressed in either 16-bit or 24-bit mode. The mode can be selected when decompression is carried out.

In the case of a 16-bit pixel, the On/Off status of the first bit (the STP bit) can also be selected when the data is decompressed.

Figure 7-5: DCT Decompression

bitstream

VLC decoding

Function name: DecVLC()

run level

IDCT

Function name:
24bit 16x16 macroblock   DecDCTin()/DecDCTout()
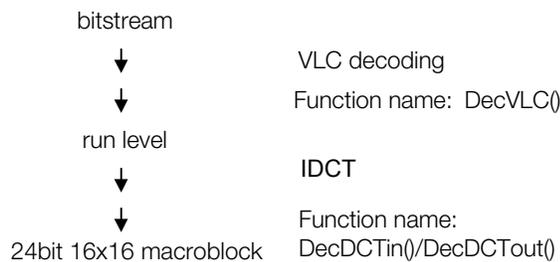
MDEC performs decompression from runlevel to macroblock.

The function DecDCTvlc() is used for VLC decoding.

Because IDCT processing takes time, a separate piece of hardware (the MDEC) performs the processing in parallel with the CPU. The function DecDCTin() is therefore provided for transferring the data to the MDEC, and the function DecDCTout() is provided for receiving the decompressed data.

Asynchronous Decoding

The MDEC and the CPU work in parallel, sharing the main memory.

The function DecDCTin() ìsews togetherî the intervals in which the CPU provides the image sections and transmits the run level to the MDEC in the background.

In the same way, the function DecDCTout() transfers decompressed macroblocks to the main memory in the background.

The data decompressed by the MDEC is always transmitted to the frame buffer, via the main memory. When this is done, the exchange between the MDEC and the main memory can be carried out asynchronously. Accordingly, one frame's worth of (640x240) images can be decompressed without creating a frame's worth of buffer in the main memory.

In the example below, the image is split up into long narrow 16x240 (15-macroblock) areas (slices), and the data for each slice is received and transmitted separately.

Example:

```
extern unsigned long *mdec_bs;     /*bitstream*/
extern unsigned long *mdec_rl;     /*run level (intermediate data)*/
  extern unsigned short mdec_image[15][16][16];
  /*decode macroblock*/
DecDCTvlc(mdec_bs, mdec_rl);       /*VLC decompression*/
DecDCTin(mdec_rl, 0); /*transmit run level*/
for (rect.x = 0; rect.x < width; rect.x += 16)
{
DecDCTout(mdec_image, slice);      /*receive*/
LoadImage(&rect, mdec_image);      /*transfer to frame buffer*/
}
```

The bitstream transmitted by one execution of the function DecDCTin() is thus received by several executions of the function DecDCTout(), allowing the size of the buffer in the main memory to be reduced.

However, in this case, there has to be a match between the bitstream transmitted and the number of macroblocks received.

Callback

DecDCTin() and DecDCTout() are both non-blocking functions that return without waiting for data transmission/reception to terminate.

To detect the termination of the transmission, you can either poll, using the functions DecDCToutSync() and DecDCTinSync(), or register a callback function to be called when the transfer terminates.

To register a callback function, use DecDCToutCallback() and DecDCTinCallback(). You can arrange for image decompression to be carried out asynchronously by designing the callback so that it activates the next data transmission/reception.

In the example below, the next DecDCTout() is activated within DecDCTout's callback function.

Example:

```
main()
{
      DecDCTout(mdec_image, slice);      /*transmission of first block*/
      DecDCToutCallback(callback);       /*define callback*/
      DecDCTvlc(mdec_bs, mdec_rl);       /*VLC decoding*/
      DecDCTin(mdec_rl, 0);              /*transmit run level*/
      :
      /*foreground processing described here*/
      :
}

callback()
{
      LoadImage(&rect, mdec_image);      /*transfer to frame buffer*/
      if((rect.x += 16) < width)

            DecDCTout(mdec_image, slice); /*receive next*/
      else
            DecDCToutCallback(0);         /*terminate*/
}
```

Playing Movies with the CD-ROM

Movies can be played by reading in and playing bitstreams continuously from the CD-ROM. The resolution and number of frames is determined by the decompression speed and the CD-ROM transmission speed.

The MDEC's maximum decompression speed is 9,000 macroblocks per second, or the equivalent of 30 320x240 images. The decompression speed has nothing to do with the compression ratio.

The image resolution and the number of frames played are, of course, inversely proportional. That is to say, with a 320x240 image, a speed of 30 frames a second can be achieved, and with a 640x240 image, speed of 15 frames a second can be achieved.

The process of continuously reading data from a CD-ROM is called *streaming*. Streaming functions are supplied separately in the libcd library.

Movies are played by placing the bitstream in the containers supplied by the streaming mechanism. Supplementary information such as movie size, etc., is not included in bit stream; therefore, the infomation needed to play a movie is defined separately in the data format (STR format) added to the header.

Table 7-2: Decompression Speed and Resolution

Resolution	Frames per second
320 x 240	30
640 x 240	15
640 x 480	7.5 ...

The CD-ROM transmission rate can be set to either 150KB/sec (standard speed) or 300KB/sec (double speed). When playing at double speed, if the bitstream forming one frame is compressed to 10KB (= 300KB/30) or less, and then recorded on the CD-ROM, 30 frames of data per second would be read off the CD-ROM.

Table 7-3: Transfer Speed and Data Size

Data size	Frames per second
10KB	30
20KB	15
30KB	7.5 ...

The moving picture play rate is determined by these two conditions. For example, when playing at double speed, the bitstream comprising one frame (320x240) would be compressed to 10KB (= 300KB/30) before being recorded on the CD-ROM.

Within the range satisfying these conditions, any number of frames, any image resolution, and any compression ratio can be selected.

Direct Transmission and Texture Transmission

Simple moving-picture playback is achieved by using VRAM as a double buffer, and transmitting the images decompressed in the drawing buffer, in succession. The movie transmission is used to clear the background and is also able to draw the object primitive.

The method whereby decompressed images are transferred directly to the drawing area of the frame buffer is called direct transmission.

Conversely, the method whereby texture transmission is carried out by temporarily transmitting decompressed images to the texture area is called texture transmission. When texture transmission is used, the textures used are limited to 16-bit mode.

Encoding by Means of the Local Environment

DCT compression is not normally carried out at run time.

However, if the images created on the drawing device are captured from the frame buffer and compressed there, it is assumed that when authoring is carried out, data compression will be performed using the CPU power of the local environment, so DCT compression functions are also provided in libpress.

The DCT calculations required for compression processing can also be carried out using the MDEC's IDCT calculation circuit, so if the local environment is used, faster encoding is possible.

BVQ (Block Vector Quantization)

BVQ reduces the number of colors in a 24-bit/16-bit direct-color image by vector quantization, and generates an image in 8-bit/4-bit index-color format. Vector quantization is a method in which quantities

(vectors) which cannot be ordered one-dimensionally are quantized adaptively, according to their frequency of occurrence.

The data compressed by DCT has already been recoded to 16 bits when it is transmitted to the frame buffer, so there is no saving in terms of the area in the frame buffer itself. However, vector-quantized images have the advantage that they can be transmitted, still in compressed format, to the frame buffer, and used, without conversion, as texture patterns.

To carry out block vector quantization, one image has to be divided up beforehand into several small areas. The division method used generally depends on the way in which the image is used as a texture pattern.

On the PlayStation, an individual CLUT can be assigned to each polygon to be texture-mapped. Accordingly, the areas are normally delineated according to the primitive values (u,v) of each polygon.

CLUT Vector Quantization

When vector quantization is carried out individually on small areas, the number of CLUTs generated is only as big as the number of areas produced by division. However, when the number of divisions is large, the area occupied by the CLUTs becomes too big to be negligible.

To avoid this situation, a function is provided for carrying out further vector quantization on the CLUT itself. For example, when a 320x240 image is divided into 300 16x16 4-bit cells, the 300 CLUTs generated for the cells can be quantized further and combined into 8 CLUTs, for example.

Huffman Encoding

The Huffman encoding supported by libpress is the classical type in which the codebook is fixed. Huffman encoding is only carried out on 4-bit index-color data.

In Huffman encoding, the content of the data is preserved by the process of compression or decompression. This compression method is called reversible compression (or loss-less compression). Generally speaking, in loss-less compression, the compression ratio cannot be controlled.

The Huffman encoder starts by generating a codebook from the frequency of occurrence of the input pixels. The size of the codebook is fixed, regardless of the number of pixels, so when there are not many pixels, the space occupied by the codebook is proportionally high, and compression efficiency is low.

When the codebook is generated, each pixel is compressed in accordance with it. As a result, the data generated is in bitstream format, as with DCT.

The compressed data is always decompressed as a set along with the codebook.

Compression of Sound Data

The PlayStation uses sound data that has been compressed from 16-bit straight PCM data to 4-bit ADPCM. The compressed sound data can be used, without conversion, as SPU sound-source data.

The SPU provides a function called *looping* so that periodic sound data can be recorded using a small number of samples. When compressing sound data, you can set a suitable loop point.

Chapter 8:
Basic Graphics Library

Table of Contents

CONFIDENTIAL

Overview

The Basic Graphics library (libgpu) is a low-level function library that allows you to work with primitives, such as triangles, rectangles, and sprites. It provides:

- **System** functions for controlling the entire graphics system (for example, graphics system reset).
- **Frame buffer access** functions for directly reading and writing the contents of the frame buffer.
- **Primitive** functions for initializing and manipulating primitive structures and setting the texture page.
- **Ordering table** functions for recording primitives in an ordering table, manipulating ordering tables, and drawing ordering table primitives.
- **Synchronization** functions for synchronizing your code with hardware events, such as the vertical blank period and the completion of drawing operations.

Library and Header Files

To use graphics library services, you must link with the file `libgpu.lib`. You must also link `libapi.lib` and `libetc.lib` when using `libgpu.lib`.

Your source files should include the header file `libgpu.h`. In addition, you must include `libgte.h` and `sys/types.h`. You include `sys/types.h` because it defines the following data types used by `libgpu.h`:

```
typedef unsigned char           u_char;
typedef unsigned short          u_short;
typedef unsigned int            u_int;
typedef unsigned long           u_long;
```

Graphics System

The PlayStation's graphics system consists of:

A specialized high-speed graphics rendering engine known as the *GPU* (Graphics Processing Unit).

A 1MB area of high-speed video memory called the *frame buffer*. It is used for storing graphics data, including the information used for the current video display, a drawing area, as well as textures and color tables.

A coprocessor (the *GTE*) for performing high-speed geometry operations. The GPU can use the results of GTE calculations in its commands. The GTE is discussed in Chapter 9 (Basic Geometry Library).

Figure 8–1: Graphics System
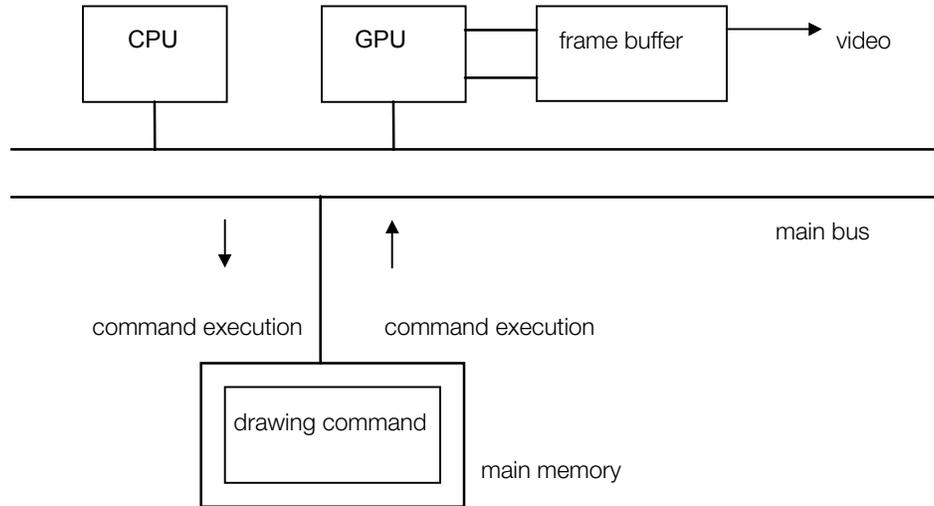
The GPU draws graphics into the frame buffer's drawing area by executing instruction strings (*primitives*) stored in main memory. Libgpu's data structures closely correspond to the primitives recognized by the GPU hardware itself.

Data from the frame buffer is continuously used to create the video signal displayed on your television monitor. By rewriting the frame buffer contents at speeds of up to 60 times per second, moving images are generated. **Note:** The graphics system contains no special background plane for displaying image data after it is drawn temporarily in the frame buffer.

Frame Buffer Addressing

The frame buffer is arranged as a bitmap that is 1024 pixels wide by 512 pixels tall, with 16 bits per pixel. The total size of the frame buffer is therefore one megabyte (1024 x 512 pixels x 2 bytes per pixel). It is used to store texture patterns and color lookup tables (CLUTs) as well containing drawing and display areas.

Figure 8–2: Frame Buffer

CONFIDENTIAL
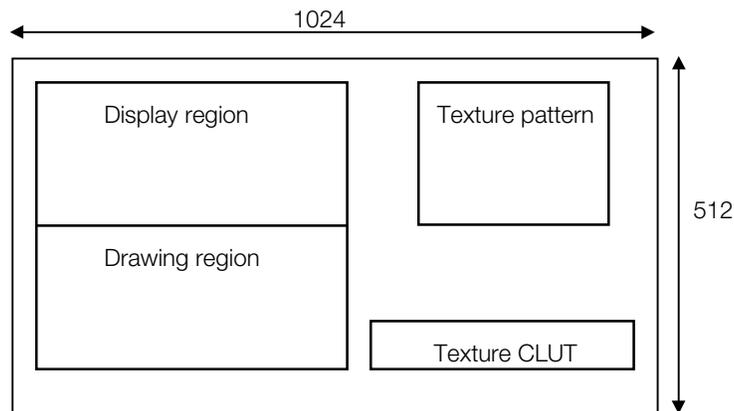
Pixels in the frame buffer are specified by 2-dimensional coordinates. X-coordinates range from 0 to 1023 and Y-coordinates from 0 to 511. Each pixel has a 16-bit depth: 5 bits for blue, 5 bits for green, and 5 bits for red; the high-order bit indicates semi-transparent mode status, as shown below:

Figure 8–3: Pixels
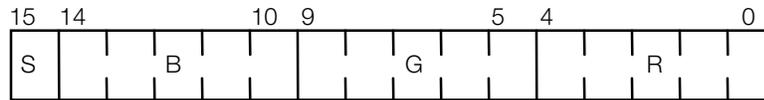
S: semi-transparent FLAG(STP)

Display Area and Drawing Area

The *display area* is a rectangular section of the frame buffer used to display the video image. Its size depends on the display mode, which ranges from 256 x 240 to 640 x 480 (709 x 488 during overscan). Any of the following combinations can be chosen:

Table 8-1: Display Modes

Width	256, 320, 360, 512, 640
Height	240 (interlace off), 480 (interlace on),
Pixel mode	24-bit, 16-bit
Interlace	On, off (must be off in 480-line mode)

Note: The screen heights assume an NTSC system. For information on working with PAL, see the section "Handling PAL Format".

The *drawing area* is a rectangular section of the frame buffer into which graphics data are drawn. Its size is not limited as long as it is fully contained within the frame buffer.

If any part of the drawing area overlaps the display area, its data is shown on the screen. To avoid this effect, a double buffering scheme is typically used. You prepare two separate areas of the same size in the frame buffer. One area is used for drawing while the other is being displayed. After drawing into the drawing area has completed, you switch the areas. Typically, the switching is done during the vertical blank period in order to avoid unsightly screen flashing or tearing.

Drawing Environment

The *drawing environment* contains general information related to two-dimensional primitive drawing, such as the position of the drawing area and the drawing offset. This information is held in the DRAWENV structure, defined as follows:

```
typedef struct DRAWENV
{
    RECT clip;                  /*clipping (drawing) area*/
    short ofs[2];               /*drawing offset*/
    RECT tw;                    /*texture window*/
    unsigned short tpage;        /*texture page*/
    unsigned char dtd;           /*dither flag (0:off, 1:on)*/
    unsigned char dfe;           /*display area drawing flag*
    unsigned char isbg;          /*enable to auto-clear)*/
    unsigned char r0, g0, b0;   /*initial background color*/
    DR_ENV dr_env;              /*reserved*/
}DRAWENV;
```

You can use the function SetDefDrawEnv() to set the fields of a DRAWENV structure. You use PutDrawEnv() to make it the current drawing environment. To get a pointer to the current drawing environment, call GetDrawEnv().

DRAWENV contains the following information:

- **Clipping:** The drawing (clipping) area is a rectangular area in the frame buffer defined by *(clip.x, clip.y) - (clip.x + clip.w, clip.y + clip.h)*.

- **Offset:** The offsets *ofs[0]* and *ofs[1]* are added to the X and Y values, respectively, of all primitives before drawing.

- **Texture Window:** *(tw.x, tw.y) - (tw.x + tw.w, tw.y + tw.h)* specifies a rectangle inside the texture page, to be used for drawing textures.

- **Texture Page:** *tpage* specifies the texture page to be used as the default texture pattern. One texture page has a size of 256 x 256 pixels.

- **Dither Processing Flag:** If *dtd* is set to 1, the drawing engine performs dithering when drawing pixels.

- **Display Area Drawing Flag:** When *dfe* is 1, drawing is permitted in the display area. (By default, drawing into the display area is blocked.)

- **Drawing Area Clear Flag:** If *isbg* is set to 1, the clipping area is cleared to the RGB color specified by the *r0, g0,* & *b0* fields when the drawing environment is set.

- **Background Color:** *r0, g0, b0* are the RGB color values used for clearing clipping area when *isbg* field is set to 1.

Display Environment

Information related to the frame buffer display, such as the position of the display region, is called the *display environment*. Display environment information is held in the DISPENV structure, defined as follows:

```
typedef struct DISPENV
{
    RECT disp;                  /*display area*/
    RECT screen;                /*display start point*/
    unsigned char isinter;      /*interlace 0: off 1: on*/
    unsigned char isrgb24;      /*RGB 24-bit mode */
    unsigned short pad0, pad1;  /*reserved */
}DISPENV;
```

You can use SetDefDispEnv() to set the fields of a DISPENV structure. To make it the current display environment, call PutDispEnv(). To get a pointer to the current display environment, call GetDispEnv().

DISPENV contains the following information:

- **Display Area:** The rectangular area within the frame buffer (*disp.x*, *disp.y*) - (*disp.x + disp.w*, *disp.y + disp.h*) is the display area. Its width (*disp.w*) can be 256, 320, 360, 512 or 640 pixels. Its height (*disp.h*) can be 240 or 480 pixels.

- **Screen Area** *(screen.x, screen.y, screen.w, screen.h)*: Specifies where on the actual physical screen the display area is shown. The standard monitor screen coordinates are (0, 0) - (256, 240). If you specify a smaller screen area, it is an underscan; if you specify a larger screen area, it is an overscan. For example, if *screen.w* is set to a value greater than 256, more pixels than 256 cannot be displayed, even if in 320 mode. The size of each pixel does not change.

- **Interlace:** If *isinter* is set to 1, the display will be in interlace mode. (If the height is 480, display is in interlace mode regardless of the setting of this flag.)

- **24-Bit Mode Flag:** If *isrgb24* is set to 1, frame buffer data is interpreted as being in 24-bit pixel format instead of the standard 16-bit.

Display Area and Screen Area

The following figure shows the relationship between the display area and the screen area:

Figure 8–4: Display Area and Screen Area
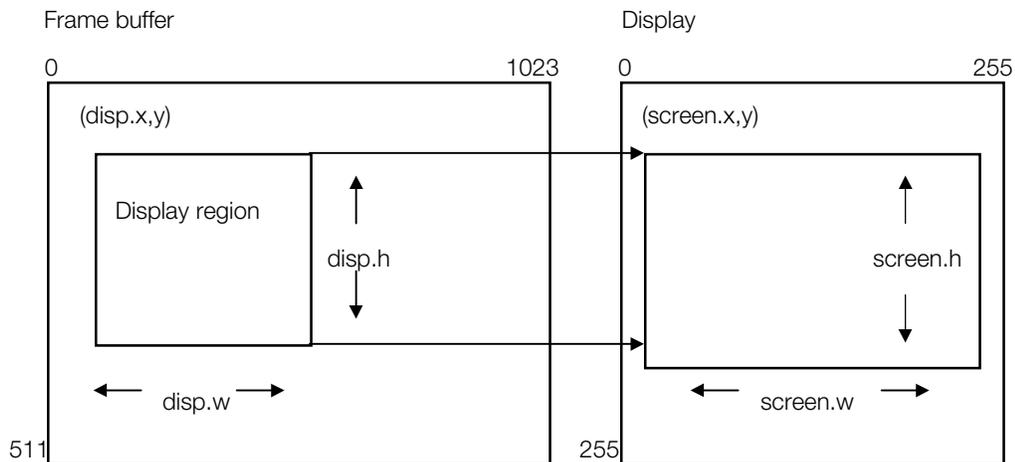
Switching Display and Drawing Environments (Double Buffer)

A double buffering system uses two areas in the frame buffer that switch between display and drawing environments. For example, when buffer 0 occupies the rectangular area (0,0)-(320,240) in the frame buffer and buffer 1 is at (0,240)-(320,480), the respective drawing and display environments are set as follows:

Table 8-2: Double Buffer

	Buffer 0	Buffer 1	Notes
Drawing environment (clip.x, clip.y) Clip start point	(0,0)		(0,240)
(ofs[0], ofs[1]) Display environment	(0,0)	(0,240)	Drawing offset
(disp.x, disp.y)	(0,240)	(0,0)	Display area origin

The fields of the DRAWENV and DISPENV structures may be set with the functions SetDefDrawEnv() and SetDefDispEnv(). To switch the drawing and display buffers, use PutDrawEnv() and PutDispEnv()to set the new drawing and display environments.

If you change the drawing environment using PutDrawEnv() while drawing is already taking place, there is no effect on the current primitive being executed or on the remainder of the current primitive list. The new drawing environment takes effect with the next drawing operation.

In addition to using PutDrawEnv(), you may also dynamically switch all or a portion of the drawing environment in the middle of drawing by registering a special primitive in the ordering table. See "Primitives" and "Ordering Tables" for more information.

On the other hand, settings made in the display environment become effective immediately. Therefore, the display location and display area can be changed even when drawing is being carried on in the background.

The following code shows the basic method of switching double buffers:

```
DRAWENV drawenv[2];                    /*drawing environments*/
DISPENV dispenv[2];                    /*display environments*/
int dispid = 0;                        /*display buffer ID*/

while (1) {
    VSync(0);                          /*wait for vertical blank*/
    dispid = (dispid + 1) %2;          /*toggle buffer ID between 0 and 1*/
    PutDrawEnv(&drawenv[dispid]);   /*switch drawing environment*/
    PutDispEnv(&dispenv[dispid]);   /*switch display environment*/
    }
```

If you use interlace mode with a height of 480 lines, it may not be possible or practical to set up a double buffer. (For example, in 640 x 480 mode there isn't room for two buffers in the frame buffer.) Therefore, a single buffer may be used for both drawing and display.

In interlace mode, in each frame (1/60 second), the display updates either the odd or even lines of the buffer, alternately. In effect, odd lines are re-displayed every 1/30 second, and the same for even lines.

If you set the *dfe* flage of your DRAWENV structure to zero, drawing is prohibited to the areas of the screen currently being displayed. This has the effect of allowing drawing only to the odd lines when even lines are being displayed, and even lines when odd lines are being displayed. This is the equivalent of the usual double-buffer switching. You don't need to do any explicit switching between display and drawing environments.

Note: for this scheme to be effective, drawing must complete within 1/60 second.

Blocking Functions and Non-Blocking Functions

Functions that complete their processing before returning are called *blocking functions*. That is, the program is blocked and the next instruction can't execute until the current one finishes.

Several drawing functions that typically take a long time are processed in the background and return without awaiting completion. These are called *non-blocking functions*.

The following functions, which directly access the contents of the frame buffer, are non-blocking:

- LoadImage() - Transfer from main memory to frame buffer
- StoreImage() - Transfer from frame buffer to main memory
- MoveImage() - Transfer from frame buffer to frame buffer

The following functions, which draw primitives, are also non-blocking. See the sections on "Primitives" and "Ordering Tables" for more information.

- DrawPrim() - Draw a primitive
- DrawOTag() - Execute a list of GPU primitves.

All functions other than those listed above are blocking functions.

To detect whether non-blocking functions have finished, or to wait for them to finish, you can call DrawSync(). For example:

```
LoadImage(&rect, pix);      /*A non-blocking function*/
DrawSync(0);                /*Waits for drawing to complete*/
```

See "Synchronization" for more information about DrawSync().

A maximum of 64 non-blocking functions may be queued. For example:

```
DrawOTag(ot0);      /*0*/
DrawOTag(ot1);      /*1*/
DrawOTag(ot2);      /*2*/
        :
```

CONFIDENTIAL

If DrawOTag(ot0) is not completed when DrawOTag(ot1) is invoked, the system simply registers the request to the queue and returns. DrawOTag(ot1) waits until DrawOTag(ot0) has finished, and then executes automatically.

The queue contains a maximum of 64 items, so if a 65th request reaches the queue, it is blocked until the queue is opened.

```
for (i = 0; i<100; i++)
    LoadImage(....);
```

In this example, the 65th LoadImage is blocked until the first LoadImage is completed and the waiting queue is available.

Primitives

The smallest command that the graphics system can handle is called a *primitive* (or a *packet*). Primitives are data structures that are created and stored in main memory, and the CPU and the GPU may both refer to them at the same time.

Primitives are classified as one of the following:

- *Drawing primitives* actually draw pixels in the frame buffer.
- *Special primitives* change certain parameters of the GPU, such as the clipping area and texture page, while drawing is being done. They do not directly change the contents of the frame buffer.

Drawing Primitives

The drawing primitives are listed below. There are four different types of drawing primitive: Polygon, Line, Sprite, and Tile.

Polygon Primitives

When drawing polygons, you can choose:

- Number of sides (3 or 4)
- Shading (Gouraud or flat)
- Texture mapping (on or off)

Therefore, the following polygon primitives can be used:

Table 8-3: Polygon Primitives

Primitive name	Contents
POLY_F3	3-sided polygon (triangle), flat shaded
POLY_FT3	3-sided polygon (triangle), flat shaded, textured
POLY_G3	3-sided polygon (triangle), Gouraud shaded
POLY_GT3	3-sided polygon (triangle), Gouraud shaded, textured
POLY_F4	4-sided polygon (quad), flat shaded
POLY_FT4	4-sided polygon (quad), flat shaded, textured
POLY_G4	4-sided polygon (quad), Gouraud shaded
POLY_GT4	4-sided polygon (quad), Gouraud shaded, textured

Line Primitives

Line primitives draw straight lines.

Table 8-4: Line Primitives

Primitive name	Contents
LINE_F2	A straight line between two points
LINE_G2	Same as LINE_F2, except with color gradation
LINE_F3	Two connected lines running from points A to B, then B to C
LINE_G3	Same as LINE_F3, except with color gradation
LINE_F4	Three connected lines running from points A to B, B to C, and C to D
LINE_G4	Same as LINE_F4, except with color gradation

Sprite and Tile Primitives

These primitives are used for drawing rectangular areas. Tiles are drawn with a solid color, while sprites are texture-mapped.

Table 8-5: Sprite Primitives

Primitive Name	Contents
SPRT	Texture-mapped Sprite (free any size)
SPRT_8	Texture-mapped Sprite (fixed size of 8 x 8 pixels)
SPRT_16	Texture-mapped Sprite (fixed size of 16 x 16 pixels)
TILE	Non-textured solid color tile (free any size)
TILE_1	Non-textured solid color tile (fixed size of 1 pixel by 1 pixel, i.e. a single dot)
TILE_8	Non-textured Solid color tile (fixed size of 8 x 8 pixels)
TILE_16	Non-textured Solid color tile (fixed size of 16 x 16 pixels)

Special Primitives

Special primitives change all or part of the drawing environment during drawing.

Table 8-6: Special Primitives

Primitive name	Parameter to be changed	Corresponding DRAWENV members
DR_ENV	Changes drawing environment	All members
DR_MODE	Drawing, texture mode	*tpage, dtd, dfe, tw*
DR_TWIN	Texture window	*tw*
DR_AREA	Drawing area	*clip*
DR_OFFSET	Drawing offset	*offset*

Primitive Expression Format

Primitives are defined as C structures. The first two words of all drawing primitives are the same:

```
typedef struct
{
     unsigned long *tag;
     unsigned char r0, g0, b0, code;
} P_TAG;
```

`tag` represents an internal pointer to the next primitive. It allows primitives to be grouped in a linked list structure so that multiple primitives can be executed together.

The following is an example of a complete primitive structure. POLY_FT4 is defined as a four-sided, flat, textured polygon:

```
typedef struct
{
     unsigned long *tag;
     unsigned char r0, g0, b0, code;
     short x0, y0;
     unsigned char u0, v0;
     unsigned short clut;
     short x1, y1;
     unsigned char u1, v1;
     unsigned short tpage;
     short x2, y2;
     unsigned char u2, v2
     unsigned short pad1;
     short x3, y3;
     unsigned char u3, v3;
     unsigned short pad2
} POLY_FT4;
```

`tag:`	Top 8-bits: Number of GPU words in packet Bottom 24-bits: pointer to next primitive
`code:`	primitive identifier (system reserved value)
`r0,g0,b0:`	display color (Red, Green, Blue, values 0-255)
`tpage:`	texture page ID
`clut:`	CLUT (Color Look-Up Table) ID
`x0,y0,...x3,y3:`	Screen coordinates of polygon vertices
`u0,v0,...u3,v3:`	Coordinates within texture page for texture
`pad1, pad2:`	Reserved, must be set to 0

Initializing Primitives and Setting Their Members

Primitives must be initialized before they can be executed. When initializing a primitive, call the initializing function for that particular type of primitive; these functions set the tag, code, and pad members appropriately. For example, before drawing a POLY_FT4 (rectangular, flat-shaded, textured polygon) primitive, initialize it as follows:

```
POLY_FT4 ft4
SetPolyFT4(&ft4);
```

Most of the members of each primitive may be freely written to by your application unless specified as reserved. There are numerous macros provided in `libgpu.h` for setting primitive members. For example, examples 1 and 2 below generate the same code. For details, refer to `libgpu.h`.

Example 1

```
POLY_F4 f4;

SetPolyF4(&f4);              /*initialize primitive*/
setRGB0(&f4, 0, 0, 255);    /*R,G,B = 0, 0, 255*/
setXY4(&f4, 0, 0, 100, 0, 0, 100, 100, 100);
DrawPrim(&f4);               /*execute primitive*/
```

Example 2

```
POLY_F4 f4;
SetPolyF4(&f4);      /*initialize primitive*/

f4.r = 0;            /*These 3 lines are*/
f4.g = 0;            /*the same as doing*/
f4.b = 255;          /*setRGB0(&f4,0,0,255)*/

f4.x0 = 0;           /*These 8 lines are*/
f4.y0 = 0;           /*the same as doing*/
f4.x1 = 100;         /*setXY4( &f4,0,0,100,0,*/
f4.y1 = 0;           /*0,100,100,100 );*/
f4.x2 = 0;
f4.y2 = 100;
f4.x3 = 100;
f4.y3 = 100;

DrawPrim(&f4);       /*execute primitive*/
```

Primitive Attributes

The following attributes may be set for primitives:

> SemiTrans - Semi-transparent mode
>
> ShadeTex - Inhibits simultaneous texture mapping and shading

You can use SetSemiTrans() and SetShadeTex() to set or clear these attributes for each primitive, as shown below. These functions may be called at any time between initialization and execution of the primitive.

```
POLY_F4 f4;
SetPolyF4(&f4);              /*initialization*/
SetSemiTrans(&f4, 1);        /*make into semi-transparent primitive*/
SetShadeTex(&f4, 1);         /*turn shading OFF*/
```

Combining Primitives

Many primitives may be used in combination with other primitives; two primitives may be brought together to form a single new primitive. This is done using the MargePrim() function.

```
typedef struct
{
        DR_MODE mode;        /*set mode primitive*/
        SPRT sprt;           /*Sprite primitive*/
} TSPRT;

setTSPRT (TSPRT *p, int dfe, int dtd, int tpage, RECT *tw)
{
        SetDrawMode(&p->mode, dfe, dtd, tpage, tw);
        SetSprt(&p->sprt);
        return(MargePrim(&p->mode, &p->sprt));
}
```

The setTSPRT() function initializes a new user-defined primitive called TSPRT. A primitive TSPRT initialized in this manner can be used with AddPrim() and DrawPrim() in the same manner as other primitives.

Note: A combined primitive may not be more than 16 long words in total size.

Executing Primitives

Primitives that have been initialized may be executed individually with the DrawPrim() function as in the following example.

```
POLY_F4 f4;
SetPolyF4(&f4);
setXY4(&fr, 0, 0, 100, 0, 0, 100, 100, 100);  /*(0,0)-(100,100)*/
setRGB0(&f4, 0xff, 0x00, 0x00);               /*RGB = (255, 0, 0)*/
DrawPrim(&f4);                                /*draw*/
```

When displaying multiple primitives, the order of execution determines the display priority, because when a primitive is executed it is drawn on top of previously drawn primitives.

In the following example, prim[0] is displayed furthest back and prim[99] is displayed furthest forward.

```
for (i = 0; i<100; i++)
    DrawPrim(&prim[i]);
```

However, multiple primitives are usually stored as a linked list in an ordering table and executed together using the DrawOTag() function. See "Ordering Tables" for more information.

Primitive Drawing Rules

The pixels drawn for a primitive are those where the center of each pixel lies within the boundary of the polygon vertices. When the center of a pixel is outside this area, the following rules are used:

- If the pixel to the right is inside the drawing area --> can be drawn
- If the pixel to the left is inside the drawing area --> cannot be drawn
- If the pixel above is inside the drawing area --> cannot be drawn
- If the pixel below is inside the drawing area --> 33 can be drawn

With POLY_* primitives, the extreme right and lowest points cannot be drawn. In the case of drawing a quadrilateral, the rules apply as follows:

Figure 8–5: Drawing a Quadrilateral
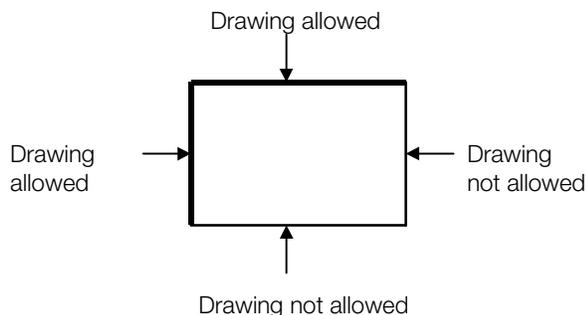
This ensures that the pixels along the polygon boundary are not drawn more than once when polygons are placed next to each other.

See "Texture Polygon Coordinate Specification" for more information on drawing rules involving texture mapping.

Ordering Tables

In order to more easily control the order of execution for large numbers of primitives, the graphics library uses a mechanism known as an *ordering table* (OT). The ordering table is a variation of a basic linked list, designed to allow easy insertion of drawing primitives which represent portions of a three-dimensional display.

Primitives can be registered in an ordering table with AddPrim() or AddPrims(). The registered primitives are then executed using DrawOTag(). Since DrawOTag()is a non-blocking function, the CPU can perform further processing without waiting for the completion of drawing by the GPU.

The OT consists of an array of pointers to primitives held in main memory. Its size is determined by the required resolution of the display priority. For example, the following example creates an ordering table with 256 levels of priority:

```
unsigned long ot[256];
ClearOTag(ot, 256);   /* initialize the OT */
```

ClearOTag() converts the basic array into a simple linked list, as shown below, where (`EndofPrim`) is a special value used to indicate the end of the list of primitives:

```
ot[0]-> ot[1] ->  ... -> ot[255] -> (EndofPrim)
```

Registering Primitives in the OT

Before drawing, primitives must be registered in the OT with AddPrim():

```
AddPrim (ot + i, &prim);   /* AddPrim(&ot[i], &prim);*/
```

The execution priority of each primitive is determined by its position in the OT. The primitives at the start of the OT will be executed first (and hence displayed furthest back), and the primitives at the end of the OT will be executed last (and hence displayed furthest forward).

In the following example, the primitives p1 and p2 are registered in the OT. Then DrawOTag() is called to execute the primitives in the table. p1 is executed first (displayed furthest back on the screen) and p2 is executed last(displayed furthest forward, i.e. it overwrites any primitives already drawn).

```
unsigned long ot[256];      /*OT (256 entries)*/
Clear0Tag(ot, 256);         /*OT initialization*/
AddPrim(&ot[0], p1);        /*register primitive p1 in ot[0]*/
AddPrim(&ot[255], p2);      /*register primitive p in ot[255]*/
DrawOTag(ot);               /*execute primitives in OT*/
```

Multiple primitives may be registered in the same OT entry. In this case, primitives will be executed after the primitives subsequently registered in the same entry. In the following example, primitives will be executed in the order p0, p3, p2, p1, p4.

```
AddPrim (&ot[2], p0);       /*register in ot[2]*/
AddPrim (&ot[3], p1);       /*register in ot[3]*/
AddPrim (&ot[3], p2);       /*register in ot[3]*/
AddPrim (&ot[3], p3);       /*register in ot[3]*/
AddPrim (&ot[4], p4);       /*register in ot[4]*/
```

Registering Special Primitives

Special primitives can be used to switch all or part of the drawing environment during the drawing process. These special primitives, like normal primitives, may be registered in the OT, then executed together with normal primitives using the DrawOTag() function.

CONFIDENTIAL

The scope of the special primitives depends on their location in the ordering table. In the following example the `env` primitive setting is valid for execution of primitives registered after `ot[128]`; therefore only `p2` receives the influence of the `env` primitive.

```
AddPrim(&ot[0], &p1);       /*register drawing primitive p1*/
AddPrim(&ot[128], &env);    /*register special primitive env*/
AddPrim(&ot[255], &p2);     /*register drawing primitive p2*/
DrawOTag(ot);
```

Linking Primitives Without an OT

You may set up your own linked list of primitives rather than using the ordering table structure. Such a list may still be executed using DrawOTag(). For example, the following provides the same operation as DrawPrim().

```
myDrawPrim(void *p)
{
        TermPrim(p);   /* terminate the primitive */
        DrawOTag(p);   /* list and execute it. */

}

drawSprites(SPRT *p, int n)
{
        int i;
        for (i = 0; i < n-1; i++, p++)
               CatPrim(p, p+1);  /* link primitive p to primitive p+1 */
        TermPrim(p);
        DrawOTag(p);
}
```

Note that when you link primitives directly to one another, you give up the flexibility of the ordering table structure.

Ordering Tables and Z Sorting

You can use an OT to implement Z sorting, which is a method of eliminating hidden surfaces by sorting a list of primitives by their depth (z-value) in 3D space. To do this, you calculate a primitive's position in the OT from its Z-value, as shown in this example:

```
unsigned long *ot[256];
        :
AddPrim(ot+256-z0,p0);
              :
```

In the basic geometry library (libgte), many of the functions calculate an *otz* value (to help create a Z-ordered OT) while performing 3-dimensional coordinate conversion.

```
SVECTOR x3, x2;
int flg, otz;

otz = RotTransPers(&x3, (long*)&x2, &flg);
```

In this case, the RotTransPers() function performs coordinate and transparent conversion of the 3-dimensional values pointed at by *x3*, using the current matrix, and stores the 2-dimensional coordinates obtained at *x2*. At the same time it returns an index to the OT called *otz*. The *otz* value is the Z coordinate divided by 4; therefore, it is sufficient to provide an OT with 1/4 of the dynamic range of the actual Z-depth. By making use of *otz*, a 3-dimensional Z sort can be performed at high speed.

CONFIDENTIAL

Reverse OT

The *otz* variable takes a large value for distant objects; as they get closer, the value approaches zero. Because of this, it is necessary to invert the value of *otz* before using it as an index into the OT array. To avoid this, the libraries make it possible to reverse the order of the entries in the OT. The ClearOTagR() function initializes the OT in reverse order. Then the order of OT execution will be reversed.

The ClearOTag() function will initialize the OT array as follows:

```
ClearOTag(ot, OTSIZE)
ot[0]-> ot[1] -> ot[2] -> ... -> ot[OTSIZE-1] -> (EndofPrim)
```

The ClearOTagR() function will initialize the OT array as follows:

```
ClearOTagR(ot, OTSIZE)
ot[OTSIZE-1]-> ot[OTSIZE-2] -> ... -> ot[0] -> (EndofPrim)
```

When using ClearOTagR(), the parameters you pass to other functions are changed accordingly, as shown in the table below:

Table 8-7: OT

Using ClearOTag()	Using ClearOTagR()
#define OTSIZE 1024	#define OTSIZE 1024
unsigned long *ot[OTSIZE];	unsigned long *ot[OTSIZE];
.....	.....
ClearOTag (ot,OTSIZE);	ClearOTagR (ot, OTSIZE);
.....	.....
AddPrim (ot+OTSIZE-otz, &prim); AddPrim (ot+otz, &prim);	AddPrim (ot+otz, &prim);
.....	.....
DrawOTag (ot);	DrawOTag (ot+OTSIZE-1);

Note how the pointers into the OT are done differently when using ClearOTagR(). In particular, the calculations required to calculate the index into the OT for the AddPrim function are simpler, and since this function is likely to be called very often, the result is a net savings.

The normal order OT is most often used for 2-dimensional graphics applications such as sprite-based games, where the position of each primitive is not necessarily based on a position in 3D space. The reverse order OT is used more often for 3-dimensional graphics applications where the Z-depth of the 3D calculations correspond more directly to positions within the OT.

The reverse order OT is initialized via a high speed hardware function, whereas the normal order OT is initialized via software. Because of this, large OT arrays are initialized much more quickly if they are reverse order.

Combining with Geometry Functions

To display three-dimensional objects, each object is broken up into combinations of triangles and quadrilaterals, and the coordinates of each polygon determine the position of the corresponding primitive which must be drawn. In other words, the (x,y) coordinates of the primitive in the frame buffer are obtained from the 3D coordinates of a polygon component of an object. This coordinate transformation is performed by the geometry library.

Object movement/rotation, and viewpoint movement/rotation may be described in a single rotation matrix and movement vector. The vertices of the polygons which make up the objects are described below.

Figure 8–6: Polygon Vertex Format

$$
\begin{bmatrix} Sx \\ Sy \\ Sx \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix} \times \begin{bmatrix} Wx \\ Wy \\ Wz \end{bmatrix} + \begin{bmatrix} Tx \\ Ty \\ Tz \end{bmatrix}
$$

(Wx, Wy, Wz): - Coordinate position in world coordinates
(Sx, Sy, Sz): - Coordinate position in screen coordinates
(m00,...,m22): - Rotation matrix

The primitive which is drawn is actually a projection onto a two-dimensional plane (the screen). The screen is an imaginary plane a certain distance *h* from the point of view. This process is known as perspective transformation.

Figure 8–7: Perspective Transformation

$$
\begin{bmatrix} x \\ y \\ otz \end{bmatrix} = \begin{bmatrix} h \times Sx / Sz \\ h \times Sy / Sz \\ Sz / 4 \end{bmatrix}
$$

Here the calculated (x, y) are the (x, y) members of the primitive and *otz* is an OT entry. See the libgte documentation for details. Following is an example of a function performing this operation.

```
void rotTransPersAddPrim(
        SVECTOR *pos;                /*position*/
        SPRT *sp;                    /*Sprite primitive*/
        unsigned long *ot;        /*OT*/
        int ot_size)              /*size of OT*/
{
        long otz, dmy, flg;

        otz = RotTransPers(&( pppos->x[0],,
                (long*)sp ->x0,&dmy,&flg);
        if (otz > 0 && otz < ot_size)
                AddPrim(ot+otz, sp);
}
```

Multiple OTs

An entire OT can be inserted into another OT if desired. This method is valid for using more than one hierarchical coordinate system at once.

The following example connects the child-OT *ot1* with a length of *n* to the parent-OT *ot0*.

```
AddOT(unsigned long *ot0, unsigned long *ot1, int n)
{
        AddPrims(ot0, ot1, ot1+n-1);
}
```

However, since the link destination for ot1[n-1] is replaced with the ot0 link destination in AddPrims(ot0, ot1+n-1), in certain cases, the primitive linked to ot1[n-1] is not rendered. As a result, the primitive must not be registered in the final ot1 entry.

Synchronization and Reset

Reset

To reset the graphics system, call ResetGraph(). This function takes one parameter, which determines the reset level. All levels immediately interrupt the drawing command in progress, cancel all the requests remaining in the queue, and enter wait status.

- **Level 0 (ResetGraph (0))** Completely resets the graphics system. It should be executed only once, when the program is activated. The drawing command and queue commands are cancelled and callbacks are initialized. The display mode is initialized at 256x240 and the display is masked (the screen goes black.)

- **Level 1 (ResetGraph (1))** Cancels the command currently being executed and the commands remaining in the queue. The drawing environment and display environment are preserved. This level is used frequently when switching the double buffer.

- **Level 3 (ResetGraph (3))** Equivalent to Level 0 complete reset, except that the display environment and the drawing environment are preserved. Also, the display is not masked. This level is used to initialize all child processes while saving the display screen status set by the parent processes. When shifting control from parent processes to child processes using Exec(), a complete reset is needed in order to switch the callback, but with a level 0 reset, the display is also initialized. Therefore, once the display synchronization misses, the screen becomes disturbed when shifting to child processes. In order to avoid this, child processes should be initialized using ResetGraph(3) at the start, rather than ResetGraph(0).

Below is a summary of the above points:

Table 8-8: Reset Levels

Reset Level	Callback	DISPENV	DRAWENV	command queue
0	Initialize	Initialize	Initialize	Initialize
1	Save	Save	Save	Initialize
3	InitializeSave	Save	Initialize	

Synchronization

In order to provide a smooth display, programs need a way to synchronize their graphics operations (and other processing) to the vertical blank period of the video display. In addition, programs need a method of detecting the end of drawing operations being performed in the background; that is, non-blocking functions such as DrawOTag().

There are two methods for detecting when asynchronous events have occurred:

- Polling: that is, checking to see whether the event has occurred.
- Callbacks: setting up functions that are automatically executed when the event occurs.

Polling

The DrawSync() function allows you to detect the end of drawing operations. It has the following options:

- DrawSync (0) - Blocks until all requests remaining in the queue are finished.
- DrawSync (1) - Returns the number of positions in the drawing queue.

The VSync() function allows you to detect the next vertical blank period, as well as providing other information. It can be used in several different ways:

- VSync (0) - Block until the next vertical blank period begins.
- VSync (1) - Return the number of horizontal sync units since the previous VSync(0) or VSync(n) call.
- VSync (*n*) - Where *n>0*, waits for the *nth* vertical blank period. (VSync(0) waits for the next VB period. VSync(2) waits for the 2nd VB period, etc.)
- VSync (-n) - Where *n<0*, returns the number of vertical blank periods since the program was started.

Callbacks

A callback is a function that is called when background processing has been completed. Libgpu provides two functions that let you register callbacks:

Table 8-9: libgpu callback registering functions

Function Name	Trigger
VSyncCallback()	Vertical Synchronization
DrawSyncCallback()	Drawing completion

DrawSyncCallback lets you define a function that is called at the completion of a non-blocking drawing operation such as DrawOTag().

VSyncCallback() lets you define a function that is called at the beginning of the vertical blank period. This function can be used to switch the display from one buffer to another and to perform other graphics operations which much be synchronized in this fashion.

```
int buffer = 0;                         /*Active buffer indicator*/
int new_frame_is_ready = 0;       /*"ready to switch buffers" flag*/


        :
        :


void main()
{                       /*initialization routine entered here*/
        :
        :
        VSyncCallback( vbcallback );     /*defines callback routine*/
        :
        :
}


vbcallback()
{
        if( new_frame_is_ready )                 /*This is set within our*/
        {                                         /*DrawSyncCallback function*/
                                                  /*(not shown here)
                buffer = 1 - buffer;       /*Switch buffers*/
                PutDispEnv(&db[buffer].disp);
                PutDrawEnv(&db[buffer].draw);
                new_frame_is_ready = 0;      /*Reset flag*/
        }
}
```

In the following code, the callback routine increments a counter. The routine MyVSync(), by looping until the counter changes, is functionally equivalent to Vsync(0).

```
main() {
/* Initialization routine entered here */
        VSyncCallback (callback);  /* Define callback */
        while (1) {
                /* Processing carried out within the frame entered here */
                myVSync();
        }
}
static volatile int Vsync_Count = 0;  /* Vertical Synchronization counter */
void myVSync(void) {                   /* Blocks until Vsync_Count variable
                                          is updated */
        int i = Vsync_Count;
        while  (i == Vsync_Count);
}
void callback() {                      /* Counter increases when vertical
                                          synchronization is started */
        Vsync_Count++;
}
```

Frame Synchronization

To avoid screen flicker, the drawing and display buffers should be switched at the same time as the vertical synchronization. DrawSync() and Vsync() are used to accomplish this.

```
/* (1) After drawing has concluded, waits until the next vertical
        synchronization and starts the next drawing */
DrawSync(0);
Vsync(0)
Draw0Tag(ot);

/* (2) Regardless of whether drawing has concluded or not, waits until
        the next vertical synchronization and starts the next drawing */
Vsync(0);
ResetGraph(1);
Draw0Tag(ot);

/* (3) Regardless of whether drawing has concluded or not, waits until
        the next 2 vertical synchronizations and starts the next drawing */
Vsync(2);
ResetGraph (1);
Draw0Tag (ot);
```

Please note that drawing at 60 frames/second in example (2) is not guaranteed. Drawing at 60 frames can be achieved only when the CPU processing terminates in 1/60 second. Also, note that in example (3), counting from the Vsync called immediately prior to the Vsync(2) blocks for two frameblockss.

Packet Double Buffer

The general term for the area in memory used for the OT and primitives is *packet buffer*.

Waiting for primitives to be drawn after they have been registered in the OT makes it impossible to operate the CPU and the graphics system in parallel. The primitives and OT cannot be accessed by the CPU until after the graphics system has finished processing them.

Figure 8–8: Drawing After Registering in OT

CPU	Registering in OT	—	Registering in OT

GPU	—	OT drawing	—
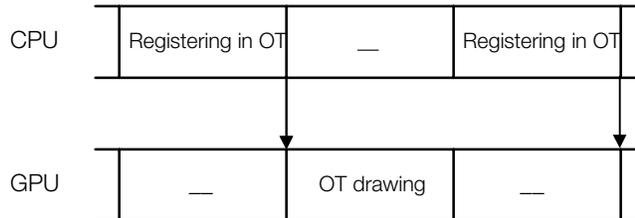
Operating the graphic system and the CPU in parallel requires two packet buffers, one is used to contain the OT and primitives currently being generated, the other is used for the OT and primitives which were previously generated and which are now being executed by the graphics system. The two packet buffers assume the tasks of drawing and execution alternately. This is referred to as a packet double buffer system.

Figure 8–9: Packet Double Buffer

CPU	Registering in OT #0	Registering in OT #1	Registering in OT #0

GPU	Drawing of OT #1	Drawing of OT #0	Drawing of OT#1
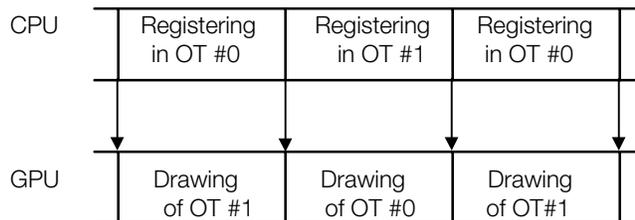
This is a packet double buffer. An example of a packet double buffer is given below. The OT and primitive must be combined together when using a packet double buffer.

```
typedef struct{
    unsigned long ot[256];          /*OT*/
    SPRT sprt[256];                 /*Sprite Primitive*/
} DB;

main(){
    int j;
    DB db[2], *cdb;
cdb = db[0];
    while (1) {
        cdb=(cdb==db)? db+1: db;    /*switch buffers*/
        ClearOTag(cdb->ot);         /*clear OT*/
        for(j=0; j<256; j++){       /*register Sprites in OT*/
                /*at this point, calculate the Sprite position*/
                AddPrim(cdb->ot, cdb->sprt[j];
        }
        DrawOTag(cdb->ot);          /*Draw*/
    }
}
```

Asynchronous Double Buffer

Normally, the packet double buffer is switched at the completion of drawing. When using interlace mode, however, drawing must be updated every 1/60 second, regardless of the calculation/rendering time. In such cases, callbacks can be used to forcibly carry out redrawing.

```
/*Asynchronous DrawOTag:
*The specified OT waits for the next VSync and is executed.
*/
main() {
        .....

        VSyncCallback (callback);
        .....

        while (1) {
                /*Create primitive list*/
                        DrawSync(0);
                        make_packet();

                        unsyncDrawOTag(ot);
        }
}

static void *completed_ot = 0;
unsyncDrawOTag (void *ot)
{
        completed_ot = ot;
}
void callback (void) {
        if (completed_ot) {
                ode_patch();
                ResetGraph (1);      /* stop drawing */
                DrawOTagR (completed_ot);  /*
        }
}
/*Patch for interlace double buffer.
*/
static void ode_patch (void)
{
        static int ode = 0;
        DRAWENV draw;
        GetDispEnv (&draw);
        if (draw.dfe) {
                while (GetODE() ==ode);
                ode = (ode+1) &0x01;
        }
}
```

In this example, DrawOTag is executed in each field regardless of the load on the CPU. However, when updating of the OT was not performed in time, the previous OT will be reused.

Note: The purpose of `ode_patch()` is to adjust for a problem in VSync timing when switching the odd/even fields in interlace mode; see "VSync Synchronization in Interlace Mode" for details.

CONFIDENTIAL

Texture Mapping

Texture mapping is a method of mapping a two-dimensional bitmapped image known as a *texture pattern* onto the surfaces of triangles and quadrilaterals.

Textures are stored in areas of the frame buffer (outside the display and drawing areas) called *texture pages.* A texture page consists of a 256 x 256 bitmap. Its upper left X coordinate in the frame buffer must be a multiple of 64 and the Y coordinate a multiple of 256. (Therefore, it's possible for texture pages to overlap horizontally.)

Texture Pattern Format

There are three pixel format modes used in texture patterns, as shown in the table below. Each primitive may have a different mode.

Table 8-10: Texture Pattern Modes

Mode	Type	Colors	Texture page width
4-bit	CLUT-based	16	64
8-bit	CLUT-based	256	128
16-bit	Direct RGB	32767	256

In 16-bit mode, the pixel value from the texture is used directly: 15 bits are used for RGB color information, allowing 32767 colors, plus 1 bit to specify semi-transparent status for that pixel.

The 4-bit and 8-bit texture modes use a color lookup table (CLUT), also known as a *palette*, to specify the actual color values. Each pixel value in these modes is used as an index into the appropriate CLUT. The CLUT itself a series of 16-bit pixel values arranged in a horizontal format within the frame buffer. Each 16-bit pixel value represents one of the colors to be used for the texture. A 4-bit texture requires a CLUT with 16 consecutive entries, and an 8-bit texture requires a CLUT with 256 consecutive entries.

16-bit textures are stored with one pixel per 16-bit word, while 8-bit textures store 2 pixels in each word, and 4-bit textures store 4 pixels in each word, as shown in the figure below. Since a 256 x 256 pixel texture pattern is placed in 1 texture page, the area actually occupied by a texture page in the frame buffer varies from 256 x 256 (16-bit mode) to 64 x 256 (4-bit mode).

Figure 8–10: Texture Pattern Format

(a) 4bit mode (pseudo color)

```
15            12  11          8  7          4  3          0
┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│     pix3     │     pix2     │     pix1     │     pix0     │
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
```
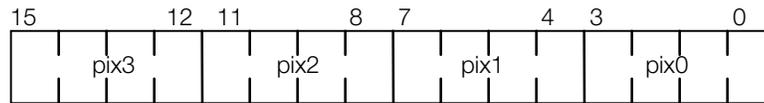
(b) 8bit mode (pseudo color)

```
15                          8  7                          0
┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│          pix1             │          pix0             │
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
```
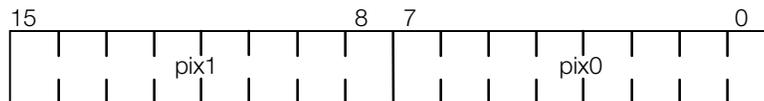
(c) 16bit mode (direct color)

```
15   14          10  9          5  4          0
┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│ S │     B      │      G      │      R      │
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
```
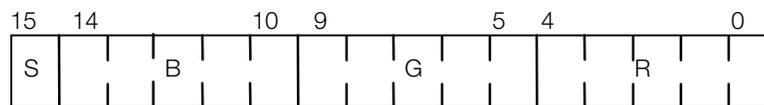
S: semi-transparent (STP) bit

When using 4-bit and 8-bit textures, the coordinates of the texture pattern (U,V) and the coordinates in the frame buffer will not directly map to each other. Care must be taken, when using LoadImage(), to load texture patterns into the frame buffer. The same applies to MoveImage() and StoreImage().

The rectangular area specified for these functions is based on standard frame buffer coordinates using 16-bit pixels. For 4-bit textures, the rectangle width must be divided by 4. For 8-bit textures it must be divided by 2. This means that 8-bit textures must be an even multiple of 2 pixels in width, and that 4-bit textures must be an even multiple of 4 pixels in width.

The following code sample illustrates texture mapping on a quadrilateral:

```
POLY_FT4 ft4;

SetPolyFT4(&ft4);                        /*initialize primitive */
ft4.tpage = GetTpage (0, 0, 640, 0);   /*texture page = (640,0)*/
ft4.clut = GetClut (0, 480);            /*texture CLUT = (0, 480)*/
/* texture pattern within the (x,y) = (0,0) – (256, 256) is
/* textured mapped to (u,v) = (0,0)-(128,128) within the */
/* texture page */

setXY4(&ft4, 0, 0, 256, 0, 0, 256, 256, 256);
setUV4(&ft4, 0, 0, 128, 0, 0, 128, 128, 128);

DrawPrim(&ft4);                          /*execute primitive*/
```

Note: GetTPage() and GetClut() require that LoadImage() be used to load the texture and texture CLUT in advance. LoadTPage() and LoadClut() load the texture page and texture CLUT and return the texture page ID and the texture CLUT ID respectively.

Texture CLUTs may be set independently for each primitive regardless of the texture to be used. Multiple textures may use the same CLUT. A 4-bit texture can use any 16 entries from a larger CLUT.

Setting the Current Texture Page

Unlike polygons, sprite primitives (SPRT) do not specify a texture page. Therefore, you must make sure the *current texture page* is set correctly when executing sprites.

You can specify the initial current texture page in the drawing environment. The special primitive DR_MODE can be used to explicitly change the current texture page. This switches the current texture page mode.

```
DR_MODE dr_mode;                /*mode primitive*/
SPRT16 sprt;                    /*16 x 16 Sprite primitive*/

SetDrawMode(*&dr_mode, 0, 0, GetTPage(2, 0, 640, 0), 0);
SetSprt16(&sprt);
setXYO(&sprt, 100, 100);

ClearOTag(ot, 2);
AddPrim(ot + 1, &sprt);       /*register SPRT16 in ot[1]*/
AddPrim(ot + 1, &dr_mode);   /*register DR_MODE in ot[1]*/
DrawOTag(ot);
```

Note that two primitives are registered in the same OT entry. The latest one registered (DR_MODE) is executed first.

Transparent Pixels and Semi-Transparent Pixels

You may select transparent, opaque or semi-transparent for each pixel when performing texture mapping. The high bit (bit 15) of each pixel value (or the corresponding CLUT entry in 4 and 8-bit mode) is the semi-transparent (STP) bit.

When the pixel value of the texture pattern is 0x0000 (STP, R, G and B are all zero), the pixels are transparent and therefore not drawn.

Pixels with the STP bit set to 1 will be displayed as semi-transparent, if the primitive they are mapped onto is set in semi-transparent mode with the SetSemiTrans() function. Pixels with the STP bit set to 0 but not with R, G and B all zero will always be opaque.

Table 8-11: Transparent/Semi-Transparent Pixels

STP, B, G, R	(0, 0, 0, 0)	(1, 0, 0, 0)	(0, n, n, n)	(1, n, n, n)
Non-transparent primitive	Transparent	Black	Non-transparent	Non-transparent
Semi-transparent primitive	Transparent	Semi-transparent	Non-transparent black	Semi transparent

Primitives that do not use texture mapping may also be set to semi-transparent mode using SetSemiTrans(). In these cases, the primitive's pixels will all be semi-transparent.

Note: The processing speed of semi-transparent polygons is greatly reduced, because the existing pixels in the frame buffer must be read, processed, and then written back.

The rates of semi-transparent primitives are specified in primitive units. Below is a list of semi-transparency rates which may be specified.

Table 8-12: Semi-Transparency Rates

Background Brightness Value	Primitive Brightness Value
0.5	0.5
1.0	1.0
1.0	-1.0
1.0	0.25

The brightness value is clipped when it exceeds the maximum value. Semi-transparency rates may be used specified by the texture page specified using the DR_MODE primitive. The same rate is applied to primitives that do not perform texture mapping.

See the section above on "Primitive Attributes" for more information.

Texture-Mapping Primitive Brightness Values

In the case of a texture-mapped primitive, the texture pattern brightness value of the pixels of a polygon is specified by the (*r, g, b*) members of the primitives. These values taken together comprise the actual brightness value.

The brightness value of a pixel being drawn is calculated from the corresponding texture pattern pixel value and the brightness value specified by the (r,g,b) members of the primitive, as shown below:

```
T =     Texture pattern pixel value

L =     Brightness value of the pixel as specified by the R,G,B fields
        of the primitive.

P =     (T*L)/128
```

In other words, if the (*r, g, b)* fields of the primitive are all set to 128, then all the pixels drawn will be the same brightness value as the source texture. If the resulting brightness value (P) exceeds 255, it will be clipped to a maximum value of 255.

Either the r, g, b members must be set, or this option must be prohibited using the SetShadeTex() function when a texture mapping primitive is initialized.

```
POLY_FT4 ft4;
SetPolyFT4(&f4);                    /*initializes the primitive*/

SetRGB0(&fT4, 0x80, 0x80, 0x80);   /*initializes the RGB values*/
/*or*/
SetShadeTex( &ft4, 1 );            /*inhibit shading*/
```

Repeating Texture Patterns

It is possible to set one portion of a texture page as a texture window and within that space wrap round (repeat) a texture pattern.

Setting a texture window can be done when setting the drawing environment through the *tw* field of the DRAWENV structure, or by using the DR_MODE primitive. Please refer to the following example.

Texture windows are normally set to (0,0) - (255, 255), which causes the texture not to be repeated. Setting the texture window to a smaller region will cause the texture to be repeated as necessary when drawing a primitive.

When specifying a texture window in order to repeat a texture, the texture coordinates (U,V) of the primitive should be within the texture window.

```
u_short tws[2], twe[2];
DR_MODE dr_mode;                    /*drawing mode primitive*/

tws[0] = tws[1] = 32;              /*texture window (32,32)-(64,64)*/
tws[0] = tws[1] = 64;

/*initialization drawing mode primitive*/
SetDrawMode(&dr_mode,0,0, GetTPage(0, 0, 640, 0), tws, twe );
        :
```

CONFIDENTIAL

```
AddPrim(ot+n, &dr_mode);
```

Texture Cache

When rendering a texture-mapped polygon, the texture pattern must be read from the frame buffer. To improve rendering speed, the PlayStation's GPU contains a 2K high-speed texture cache. When textures are used, they are read from the frame buffer into the cache. Subsequent uses of the same texture pixels (*texels*) are read directly from the cache, which is much faster than reading from the frame buffer.

Like the frame buffer, textures in the texture cache are referred to by two-dimensional addresses. These addresses depend on the pixel mode of the polygon being drawn. The following table shows the cache sizes for each pixel mode:

Table 8-13: Texture Cache Size

Pixel Mode	Size (width x length)
4 bit/pixel	64x64
8	64x32
16	32x32

Primitive Rendering Speed

To improve rendering speed on the PlayStation, it's necessary to determine which is slower, the rendering speed of the GPU or the computation speed of the CPU. This can be determined by calling DrawSync().

- If the speed bottleneck is in the CPU, DrawSync() returns immediately.
- If the speed bottleneck is in the GPU, DrawSync() is blocked; that is, it doesn't return immediately.

  The amount of time DrawSync() is forced to wait is a measure of the latency through the GPU.

When the bottleneck is in the GPU, program code optimization will not improve performance, so a means for improving rendering speed is necessary.

This section explains a few of the factors that determine the rendering performance of the GPU and general methods for improving rendering speed.

In the PlayStation, frames are first rendered in the frame buffer, then output to the display. Therefore, rendering performance can be determined essentially from the number of read and write accesses to VRAM (Video RAM).

The rendering (execution) speed of a specific primitive depends on its area and type. Primitive rendering consists of repeated reads and writes to the frame buffer VRAM. The larger the area of the primitive, the greater the amount of VRAM written and hence the greater the rendering time. Semi-transparent rendering is slower than opaque rendering of the same primitive, since semi-transparent rendering requires read access as well as write access.

The rendering speed depends on the primitive type. The execution speed of primitives with the same rendering area are in the following order:

Figure 8–11: Primitive Rendering Speed

```
┌─────────────────────────────────────────────────────────────────┐
│ High speed  ⟵                              ⟶  Low speed          │
│                                                                   │
│ TILE                                                              │
│ POLY_F*  POLY_G*                                                  │
│           POLY_FT*(4bit/OnC)          POLY_FT*(4bit/OffC)         │
│           POLY_FT*(8bit/OnC)              POLY_FT*(8bit/OffC)     │
│           POLY_FT*(16bit/OnC)                 POLY_FT*(16bit/OffC)│
│           POLY_GT*(4bit/OnC)          POLY_GT*(4bit/OffC)         │
│           POLY_GT*(8bit/OnC)              POLY_GT*(8bit/OffC)     │
│           POLY_GT*(16bit/OnC)                 POLY_GT*(16bit/OffC)│
│                                                                   │
│                                                                   │
│ SPRT(4bit/OnC)              SPRT(4bit/OffC)                       │
│ SPRT(8bit/OnC)                  SPRT(8bit/OffC)                   │
│ SPRT(16bit/OnC)                     SPRT(16bit/OffC)              │
└─────────────────────────────────────────────────────────────────┘
```

OnC indicates that the texture cache is in hit status, and OffC indicates that the texture cache is in miss status, while 4bit/8bit/16bit indicates the texture mode. Rendering speed is always faster in cache hit, while when the texture cache is in miss state, 4bit mode texture is faster than 8bit, which is faster than 16bit.

Access Rules

Primitive rendering speed can be calculated from the frame buffer access cycle.

Once all the primitives have been rendered to the frame buffer, they are displayed. Rendering performance is related to the frequency of read-write access to the frame buffer.

Basic Rules

A write access to the frame buffer corresponds directly to a rendering operation. Read accesses to the frame buffer take place when a texture pattern is being read and when in semi-transparent mode. The rules for access cycles are as follows.

Table 8-14: Access Cycles

Access direction	pixels/cycle	Notes
Write	2	SPRT,TILE,POLY_F3,POLY_F4
	1	Other
Read	1	Texture mapping semi-transparent rendering

CONFIDENTIAL

For example, the number of cycles required to render a 100x100 POLY_G4 and a 100x100 POLY_F4 are shown below.

Table 8-15: Number of Access Cycles

Primitive type	POLY_G4	POLY_F4
Total number of pixels	100x100=10000	100x100=10000
Total number of reads	0	0
Total number of writes	10000	5000
Total cycles	10000	5000

Texture mapping

Calculating the rendering speed of texture maps is extremely complex. However, we can first consider a simple texture miss/hit, where the mapping is 1:1. In the 4-bit texture pattern, four texels (texture pixels) are packed into one 16-bit word. Therefore, four texels can be read together with one access. SImilarly, with an 8-bit texture pattern, two texels are read simultaneously.

In the case of a 100x100 POLY_FT4:

Table 8-16: Number of Cycles in POLY_FT4

Mode	4-bit	8-bit	16-bit
Total number of reads	10000/4=2500	10000/2=5000	10000
Total number of writes	10000	10000	10000
Total	12500	15000	20000

In the case of a 100x100 SPRT:

Table 8-17: Number of Cycles in SPRT

Mode	4-bit	8-bit	16-bit
Total number of reads	10000/4=2500	10000/2=5000	10000
Total number of writes	5000	5000	5000
Total	7500	10000	15000

It can be seen from this presentation that 4-bit textures are the fastest.

Texture Enlargement Ratio Dependencies

In these examples, 1:1 texture mapping was performed. However, the calculations differ if the texture is enlarged or reduced. Below, the primitives from the previous examples are reduced horizontally by 2 (4-bit mode).

Table 8-18: Number of Cycles Used when Reduction Is Involved

Item	Cycles
Total number of reads	100x100/4=2500
Total number of writes	50x100=5000
Total	7500

Note that simply halving the area will not halve the rendering time.

Rendering speed improves when a texture is expanded, because the same texels can be used multiple times, and fewer texture reads are needed to render an area.

Texture Cache Dependencies

In the above examples, the texture cache is always missed. However, a texture for which there is a cache hit can be used directly without a read operation. Calculating using the previous example:

Table 8-19: Texture Cache Dependencies

In the case of a 100x100 SPRT

Mode	4-bit	8-bit	16-bit
Cache hit	5000	5000	5000
Cache miss	7500	10000	15000

Note that if the texture pattern is in cache, rendering speed is constant regardless of the mode.

Clipping

The number of rendering cycles also depends on how polygons are clipped. Rendered polygons are clipped within the rendering area. During clipping, the left and upper portions of the polygon generate empty cycles.

Figure 8–12: Clipping

Drawing polygon
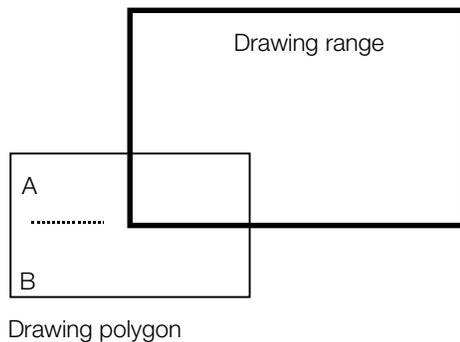
In this example, empty cycles are generated at A but are not generated at B. Empty cycles also include empty texture read cycles.

Structure of the Texture Cache

This section describes the texture cache for the benefit of programmers who are concerned with optimizing its use.

Cache blocks

A texture page is divided into rectangular regions based on cache size. Each of these regions is referred to as a *cache block*. Cache blocks are numbered in sequence (according to block number).

In 4-bit mode, the size of the cache is 64x64. The texture page is divided into 16 cache blocks as shown below.

Figure 8–13: Cache Blocks in Texture Page

0	64	128	192	255

64	0	1	2	3
128	4	5	6	7
192	8	9	10	11
255	12	13	14	15

Cache entries

Each cache block can be divided further into 16 x 1 regions known as *cache entries*. In 4-bit mode, there are 256 cache entries and they are arranged as shown below.

Figure 8–14: Cache Entries

	0	16	32	48	63
0	0	1	2	3	
1	4	5	6	7	

... ...

61	244	245	246	247
62	248	249	250	251
63	252	253	254	255

Each entry is structured as follows:

```
struct {
    u_char   block_id;    /*block number tag*/
    u_short  data[4];     /*texture pattern data*/
} Entry[256];
```

Since cache data consists of 4 short words, a 4-bit texture would store 16 texture pixels in a single entry.

Cache strategies

A block number is saved in each entry, and this is used to determine when there is a hit or miss in the cache. In texture mapping, the determination of whether texture pixel (u,v) is in cache or not is performed in the following manner.

The block number to which a texture pixel (u,v) belongs can be calculated as:

```
block_id = (v>>6)<<2 + (u>>6)
```

And, the entry number associated with (u,v) can be calculated as:

```
entry_id = (v&0x3f)<<2 + (u&0x3f)>>4
```

CONFIDENTIAL

Based on these calculations, a cache hit evaluation can be performed with the following code:

```
is_cache_hit_4bit(u_char u, u_char v)
{
        int block_id = (v>>6)<<2 + (u>>6);
        int entry_id = (v&0x3f)<<2 + (u&0x3f)>>4;

        if (Entry[entry_id].block_id == block_id)
                return(1);   /*cache hit*/
        else
                return(0);   /*cache miss*/
}
```

Since cache block numbers are saved independently in each cache entry, texture pixels having different block numbers can coexist in the cache as long as their entry numbers are different. For example, since the texture pixels in the rectangular area defined by

```
(u,v) = (0,0)-(63,63)
```

all belong to the same texture block, they will be saved in the cache together. The texture pixels in the rectangular region defined by

```
(u,v) = (16,16)-(79,79)
```

span multiple texture blocks, but they will also be saved in the cache together since there are no overlapping entries. However, the texture pixels in the rectangular region defined by

```
(u,v) = (8,8)-(71,71)
```

have some overlapping entry numbers (e.g. (u,v) = (8,8)-(15,8) and (u,v) = (64,8)-(71,8)). Therefore, these pixels will not be saved in the cache together even though the rectangular area itself fits in a 64x64 area.

Also, pixels that are not in contiguous regions and do not have overlapping entries, such as

```
(u,v) = ( 0, 0)-(15,15)
(u,v) = (80,64)-(95,79)
```

can be saved in the cache together.

Mode dependencies

The sizes of the cache blocks and cache entries vary according to mode. However, the number of entries is always 256.

Table 8-20: Size of Cache Blocks and Cache Entries

Mode	Block	Number of blocks	Entry	Number of entries
4	64x64	16	16x1	256
8	64x32	32	8x1	256
16	32x32	64	4x1	256

CONFIDENTIAL

Primitive Division

Problems can arise when drawing an area that occupies a large space in the display screen with a single texture primitive.

Texture Mapping Distortion

PlayStation uses affine transformation for texture mapping. When mapping a large primitive, the image can become distorted due to conversion errors. The texture coordinates (u ,v) attached to the points within the polygon (x, y) are calculated as:

```
u = a0*x + a1*y + a2
v = b0*x + b1*y + b2
```

This does not produce a correct image. On the other hand, the following equations contain correct perspective conversion:

```
u = (a0*x + a1*y + a2*z + a3) / (c0*x + c1*y + c2*z +c3)
v = (b0*x + b1*y + b2*z + b3) / (c0*x + c1*y + c2*z +c3)
```

If the *z* depth within the polygon is fixed or changes are few, a correct mapping image can be created.

As a result, when a quadrilateral primitive which occupies a comparatively large area of ground and has large depth changes is drawn, a mapping image with a bent diagonal line will result due to conversion errors.

Texture Cache Mistakes

Texture mapping a large primitive cannot get any value from using the texture cache. In 4-bit mode, one texture cache entry is maintained horizontally 16 texels (4 short words). When the necessary texels are not in the cache, GPU combines the cache entries containing those texels and speculatively reads them. If the surplus 15 texels which are read additionally are used before the entry is flashedflushed, this contributes to the drawing efficiency, but if they are not used, they are discarded as useless.

However, in 4-bit mode the texture cache size is restricted to a 64x64 size. This means that texels separated by 64 texels share the same cache entry. Therefore, when drawing something which has a maximum displacement (*d*u, *d*v) of the (u,v) within the primitive greater than 64, cache mistakes will occur without fail during drawing.

In the worst case (when the mapping of the u,v direction and x,y direction are exactly 90 degrees) the cache entry texel read speculatively is flashed before it is used next and then becomes useless. In such a situation, the drawing speed is reduced by half.

The difficult point in this problem is that the drawing speed can fluctuate greatly depending on the primitive drawing direction. The drawing direction fluctuates dynamically with the local screen matrix value and the matrix value is moved by the controller input. Undecided elements such as this which cause major fluctuations in the drawing efficiency can make the program more difficult.

Clip Overhead

Drawing a large primitive is disadvantageous from a clipping standpoint. If the entire primitive is outside the drawing area, that primitive is not drawn (that is, not recorded to the OT). However, if even one section falls within the drawing area, that section must be recorded to the OT. Although drawing outside the drawing area can be cancelled by the GPU clipping function, an empty cycle can sometimes be produced. As the primitive increases in size, the chance that its entire area will all be outside the drawing area decreases and the chance that an empty cycle will be produced increases.

Primitive Division

Almost all of these problems can be solved by dividing a large primitive in advance. For a primitive which has an area which has the possibility of becoming large, the area and distance from viewpoint before perspective conversion are evaluated during drawing and the decided frequency is recursively divided at the mid-point. Depending on the objective, there are several division algorithms and packaging methods. Following is a simple POLY_FT4 recurrent division example:

```
typedef struct {
        short         x, y, z
        u_char        u, v
        short         x2, y2
}VERTEX

/*Macro for performing mid-point division*/
#define half (v0, v1, v2) /
(v0)->x3 = ((v1)->x3+(v2)->x3)>>1, /
(v0)->y3 = ((v1)->y3+(v2)->y3)>>1, /
(v0)->z3 = ((v1)->z3+(v2)->z3)>>1, /
(v0)->u  = ((v1)->u+(v2)->v )>>1, /
(v0)->v  = ((v1)->u+(v2)->v)>>1, /
get_RotTransPers (&((v0)->x3), &((v0)->x2, &dmy, &dmy, &dmy);

extern POLY_FT4     *heap  /*Buffer which saves primitive after division*/
extern POLY_FT4 *sle;tpm;  /*Template*/
extern int min_x, max_y;    /*Drawing area*/
extern int max_x, max_y;
void divideFT4 (int ndiv, VERTEX *v0, VERTEX *v1, VERTEX *v2, VERTEX *v3)
{
        if (min4(v0-> x, v1->x, v2->x, v3->x) > max_x) return;
        if (min4(v0-> x, v1->x, v2->x, v3->x) > min_x) return;
        if (min4(v0-> x, v1->x, v2->x, v3->x) > max_y) return;
        if (min4(v0-> x, v1->x, v2->x, v3->x) > min y) return;
        if (ndiv)
        {
                u_long d;
                VERTEX v4, v5, v6, v7, v8
                half (&v4,  v0,  v1);
                half (&v5,  v2,  v3);
                half (&v6,  v0,  v2);
                half (&v7,  v1,  v3);
                half (&v8, &v5, &v6);
                divideFT4(ndiv-1,  v0, &v4, &v6, &v8);
                divideFT4(ndiv-1, &v4,  v1, &v8, &v7);
                divideFT4(ndiv-1, &v6, &v8,  v2, &v5);

                divideFT4(ndiv-1, &v8, &v7, &v5, v3);
                return;
        }
        else
        {
                *heap = *skelton;
                setXY4 (heap, v0->x, v0->y, v1->x, v1->y,
                            v2->x, v2->y, v2->x, v2->y);
                setUV4(heap, v0->u, v0->v, v1->u, v1->v;
                            v2->u, v2->v, v2->u, v2->v);
                heap++;
        }
}
```

CONFIDENTIAL

Perspective conversion is carried out correctly on the new vertex produced by the division (division vertex). With this method correct conversion is performed only on the division vertex and can be thought of as being interpolated via the primary method. Therefore, when the number of divisions increases (if the number of division vertices increases), the approximate precision increases and the quality of the texture images also increases.

Division also has an effect on the texture cache. The (du, dv) of the primitive after conversion are smaller than those before division. When division is repeated and the (du, dv) stay within the size of the texture cache, the drawing efficiency is greatly increased.

Division is also effective for clipping. By dividing a large primitive the probability that the entire primitive area will fall outside the drawing area increases. As a result, it is possible to reduce the useless empty cycles outside the drawing area.

Debug Environment

Debug Mode

When debug mode is set, each function checks the conformity of the data as far as possible. If there is any problem, it will print a return code and the contents as a debug string.

Debug String

When debug mode is set by the SetGraphDebug() function, or the contents of a structure is output by using the Dump...() function, the output character string is stored in the specified character string buffer. The Fnt...() function is used to display this on screen.

High-Level Library Interface

Libgpu is designed to avoid dependence on any particular data structure and paradigm. It contains no functions that can work directly with PlayStation graphics formats such as TIM (a two-dimensional image related data structure) or TMD (a three-dimensional object data structure). To handle these formats directly, you can use functions of the extended graphics library (libgs).

However, the OpenTMD()/ReadTMD() and OpenTIM()/ReadTIM() functions are available to analyze the contents of TMD data and TIM data only for the debugging of the data itself. There is also an interface between libgpu and libgs.

ReadTIM()interprets as much as possible the header information within TIM format image data of TIM data.

ReadTMD()interprets as much as possible the information of any polygon data inside any object with TMD data.

Cautionary Programming Notes

This section discusses some topics that you should be aware of when using libgpu.

- Texture polygon coordinate specification
- Handling PAL format
- Timing for updating the frame buffer
- VSync synchronization in interlace mode

Texture Polygon Coordinate Specification

The following problems have been reported when drawing textured polygons:

1. When attempting to display a 16x16 texture map on a 16x16 polygon, using the parameters (0, 0) to (15, 0) and (0, 15) to (15, 15) causes the lines at the bottom and right edges not to be displayed.

2. With the textured polygon POLY_FT4, enlarging the texture before displaying the polygon causes an extra dot to be displayed on the right and bottom edges.

    ```
    (x,y)=(0,0)-(16,16), (u,v)=(0,0)-(16,16)        Normal
    (x,y)=(0,0)-(17,17), (u,v)=(0,0)-(16,16)        Normal
    (x,y)=(0,0)-(31,31), (u,v)=(0,0)-(16,16)        Normal
    (x,y)=(0,0)-(32,32), (u,v)=(0,0)-(16,16)        Extra dot displayed
    ```

3. With the textured polygon POLY_FT4, texture patterns cannot be specified if they touch the right or bottom sides of the texture page.

These problems relate to the following drawing rules.

Drawing Rules

The drawing rules for PlayStation POLY_... primitives specify that drawing cannot be performed along the right and bottom edges. This rule prevents the polygon boundary lines from being written twice when polygons are used to cover an area.

An example of this is shown in the following diagram. As can be seen, without this rule, the center intersecting lines of polygons P0, P1, P2, and P3 would be written twice. This might be a problem in some cases, such as when using semi-transparent mode.

Figure 8–15: Drawing Rule
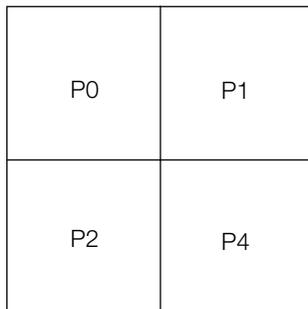
The above example assumes that a square specified by the coordinates (x, y) = (0, 0) to (8, 8) and (u, v) = (0, 0) to (8, 8) is being drawn as POLY_FT4. In other words, the following is assumed.

```
POLY_FT4 ft4;

SetXY4( &ft4, 0,0, 8,0, 0,8, 8,8 );
SetUV4( &ft4, 0,0, 8,0, 0,8, 8,8 );
```

CONFIDENTIAL

The texture pattern for the above is mapped as shown below.

The numbers in the map represent the texture pattern values (v, u) that are copied to the corresponding pixels. These values are entered in the order of (v, u), not (u, v), in accordance with frame buffer addressing.

Figure 8–16: Mapping

	0	1	2	3	4	5	6	7	8
0	00	01	02	03	04	05	06	07	08
1	10	11	12	13	14	15	16	17	18
2	20	21	22	23	24	25	26	27	28

....
....

	0	1	2	3	4	5	6	7	8
7	70	71	72	73	74	75	76	77	78
8	80	81	82	83	84	85	86	87	88

← The (u.v) values at this point are (8,8).
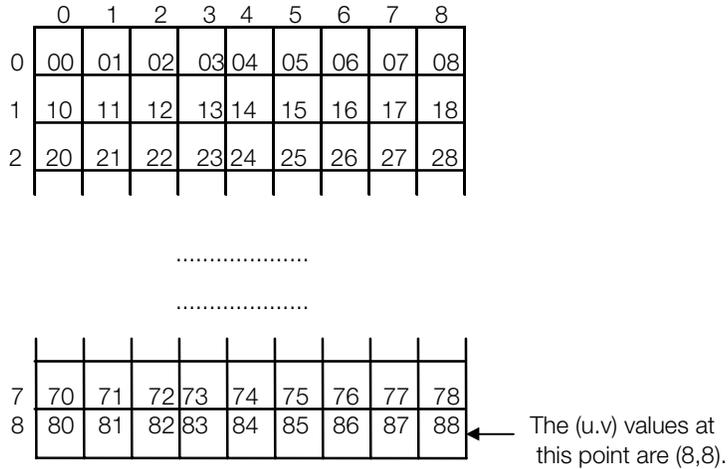
If the drawing rule described earlier is applied under this condition, the lines at the right and bottom edges are not displayed, so the actual display is as shown below.

Figure 8–17: Displayed contents

	0	1	2	3	4	5	6	7
0	00	01	02	03	04	05	06	07
1	10	11	12	13	14	15	16	17
2	20	21	22	23	24	25	26	27

....
....

	0	1	2	3	4	5	6	7
7	70	71	72	73	74	75	76	77
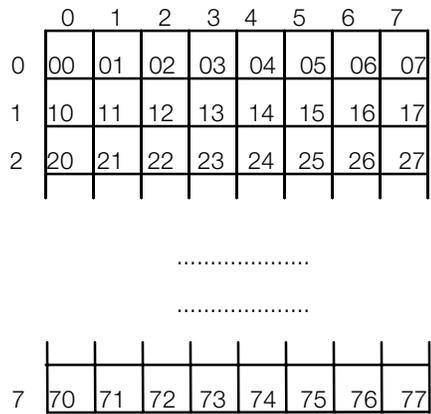
In the example above, texture mapping at (0, 0) to (7, 7) is accurate from pixels (0, 0) to (7, 7).

Next, if (u, v) = (0, 0) to (7, 7), the mapping is as shown below.

Figure 8–18: Mapping

	0	1	2	3	4	5	6	7	8
0	00	00	01	02	03	04	05	06	07
1	00	00	01	02	03	04	05	06	07

....

....

| 8 | 70 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

← The (u,v) values at this point are (7,7).
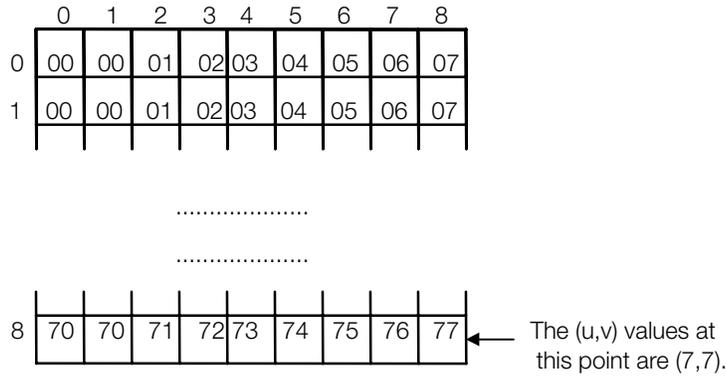
Applying the drawing rule described above, the lines at the right and bottom edges are deleted, so that the lines represented by texture values u = 7 and v = 7, i.e., the lines at the right and bottom edges, are not displayed.

Figure 8–19: Displayed Contents

	0	1	2	3	4	5	6	7
0	00	00	01	02	03	04	05	06
1	00	00	01	02	03	04	05	06

....

....

| 7 | 60 | 60 | 61 | 62 | 63 | 64 | 65 | 66 |
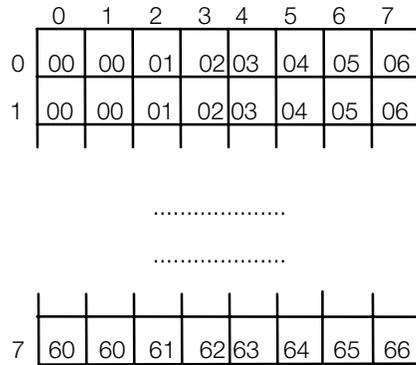
As shown above, correct results will be obtained if (x, y) = (0, 0) to (8, 8) and (u, v) = (0, 0) to (8, 8) are used.

In ordinary texture mapping, no problems should occur when the mapping is contiguous, i.e., when adjacent polygons have adjacent texture patterns applied to them.

However, in background displays, adjacent cells (POLY_FT4 cells) are not required to use adjacent textures by necessity.

In this case, using the normal specification described above would cause the following types of problems to occur.

Inverting or Rotating Textures

For example, suppose you want to rotate the texture described above 180 degrees in the XY direction and then display the rotated texture. Without changing the (u, v) values for POLY_FT4, you can specify the following for (x, y).

```
SetXY4( &ft4, 8,0, 0,0, 8,8, 0,8 );
```

The texture pattern for the above is mapped as shown below.

The numbers in the map represent the texture pattern values (u, v) that are copied to the corresponding pixels.

CONFIDENTIAL

Figure 8–20: Mapping

	0	1	2	3	4	5	6	7	8
0	08	07	06	05	04	03	02	01	00
1	18	17	16	15	14	13	12	11	10
2	28	27	26	25	24	23	22	21	20

....

....

	0	1	2	3	4	5	6	7	8
7	78	77	76	75	74	73	72	71	70
8	88	87	86	85	84	83	82	81	80

The (u,v) values at this point are (0,8).
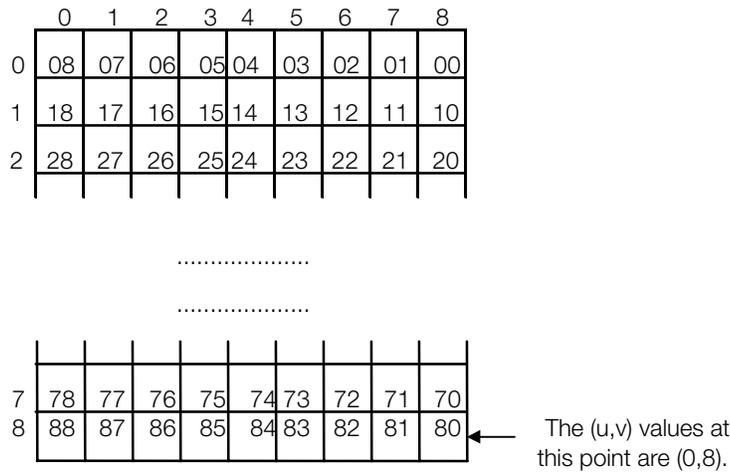
Applying the drawing rule described earlier, the lines at the right and bottom edges are not displayed, and so, as shown below, the line defined as (u, v) = (0, 0) to (0, 7) (the points at each pair of UV spatial coordinates from (0, 0) to (7, 0)) are not mapped. Instead, the points from (8, 0) to (8, 7) are shown as the left edge, meaning that the entire image is shifted one dot to the left when it is mapped.

Figure 8–21: Displayed Contents

	0	1	2	3	4	5	6	7
0	08	07	06	05	04	03	02	01
1	18	17	16	15	14	13	12	11
2	28	27	26	25	24	23	22	21

Here (u,v)=(1,0) is mapped instead of the expected (u,v) = (0,0).

....

....

	0	1	2	3	4	5	6	7
7	78	77	76	75	74	73	72	71
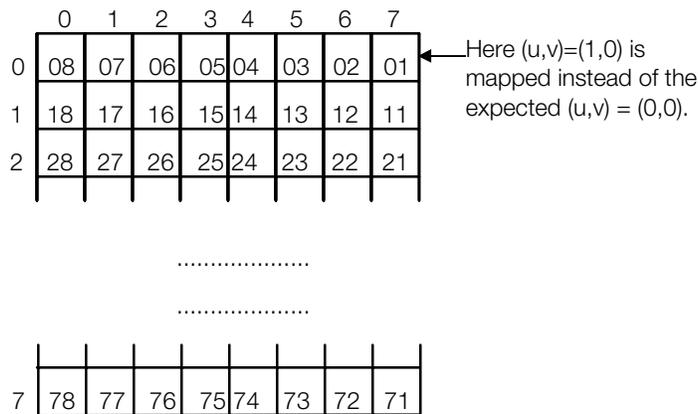
This effect may occur when the texture pattern is inverted vertically or when the polygon is rotated 90 or more degrees.

In particular, when the polygon is rotated, the mapped texture pixels change depending on the angle of rotation.

Enlarging Textures

Let us consider an example in which the same POLY_FT4 as above is enlarged to twice its size.

In this case, specifying (x, y) = (0, 0) to (16, 16) and (u, v) = (0, 0) to (8, 8) causes the texture pattern to be mapped as shown below (values are rounded to the nearest whole number).

CONFIDENTIAL

Figure 8–22: Mapping

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	00	01	01	02	02	03	03	04	04	05	05	06	06	07	07	08	08
1	00	01	01	02	02	03	03	04	04	05	05	06	06	07	07	08	08
2	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18	18
3	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18	18

....
....

14	70	71	71	72	72	73	73	74	74	75	75	76	76	77	77	78	78
15	80	81	81	82	82	83	83	84	84	85	85	86	86	87	87	88	88
16	80	81	81	82	82	83	83	84	84	85	85	86	86	87	87	88	88

Applying the same drawing rule, the lines at the right and bottom edges are not displayed, so (u, v) is displayed in the range (0, 0) to (8, 8) as shown below.

Figure 8–23: Displayed Contents

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	01	01	02	02	03	03	04	04	05	05	06	06	07	07	08
1	00	01	01	02	02	03	03	04	04	05	05	06	06	07	07	08
2	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18
3	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18

....
....

14	70	71	71	72	72	73	73	74	74	75	75	76	76	77	77	78
15	80	81	81	82	82	83	83	84	84	85	85	86	86	87	87	88

The "u=8" point remains here

When the mapping is gradually increased from the same scaling factor, this effect occurs precisely when the scaling factor becomes 2.

Specifying Pixels on the Left and Bottom Edges of the Texture Page

For the same reason as explained above, it is not possible to display the lines at the texture pattern right and bottom edges (the lines specified as u = 255 and v = 255, respectively).

In the example used earlier, you must specify the following to display the 8 x 8 section at the right bottom edge of the texture page.

```
SetUV4( &ft4, 248,248, 256,248 256,256, 248,256 );
```

But because the resolution of (u, v) is 8 bits, you cannot set a value of 256, because it will overflow to a value of zero. Therefore, these values must be rounded down to 255 as shown below.

```
SetUV4( &ft4, 248,248, 255,248 255,255, 248,255 );
```

Note that the line defined by u = 255 and v = 255 is not displayed. This problem occurs when polygons are enlarged by a factor of two or more.

Similarly, this problem occurs at the leftmost line (the line at u = 0) if the texture is mapped as horizontally flipped. If the texture is mapped as both horizontally and vertically flipped, this problem occurs at the top and leftmost lines (the lines at v = 0 and u = 0). In other words, neither of these lines is displayed.

Corrective Measures

Subtracting a 1 from the (u, v) values will avoid all of the problems described above, although other problems may occur. This will, however, prevent the right and bottom edges of the texture pattern from being displayed. In other words, if the texture pattern is a 16 x 16 pattern, it will be enlarged to 16:15, and if it is an 8 x 8 pattern, it will be enlarged to 8:7 when displayed. Be sure to note this when creating Sprite patterns.

Handling PAL Format

The information in this chapter assumes an NTSC display. A number of changes are necessary in order to output a signal for PAL-format TV receivers.

Differences between NTSC and PAL

The major differences between NTSC and PAL are shown in the following table:

Table 8-21: Differences between NTSC and PAL

Video format	NTSC	PAL
Field rate	60Hz	50Hz
Standard vertical resolution	240	256

Since the PAL field rate is 50Hz, the maximum display rate is 50 frames/second. Also, since vertical sync interrupts only occur 50 times a second, programs that use Vsync for timing will appear to slow down to 5/6 of the NTSC rate.

The vertical resolution that can be displayed on a standard TV is greater for the PAL format. Thus, a larger display area within the frame buffer is needed for full-screen displays on a PAL-format TV. If the NTSC-format display area is used on a PAL TV, an upper or lower section of the screen will appear dark.

Changes to handle PAL format

Programs designed with the NTSC format in mind must be changed in the following ways to handle PAL-format monitors.

- Enable PAL mode using the SetVideoMode() function
- Adjust the display starting position
- Adjust timing
- Adjust display area

Of these, changes in the first two are required. Changes in the last two are optional depending on the application.

Enabling PAL mode

For the DTL-H2000, PAL mode can be enabled by setting the DIP switch on the main unit and then using the SetDispMode() function.

```
#include <libetc.h>
main()
{
SetVideoMode(MODE_PAL);
.....
}
```

Programs operating in this mode will run on PAL PlayStations as well.

Adjusting the display starting position

With PAL format, although the vertical resolution prerequisite is 256 lines, the vertical resolution of the display is 240 lines in the standard setting. Thus, without any changes, the display area on a PAL TV will appear as if it were shifted to the top of the screen. However, the display area should be centered on the screen by modifying the default values of the screen structure in DISPENV.

Figure 8–24: Display Starting Position
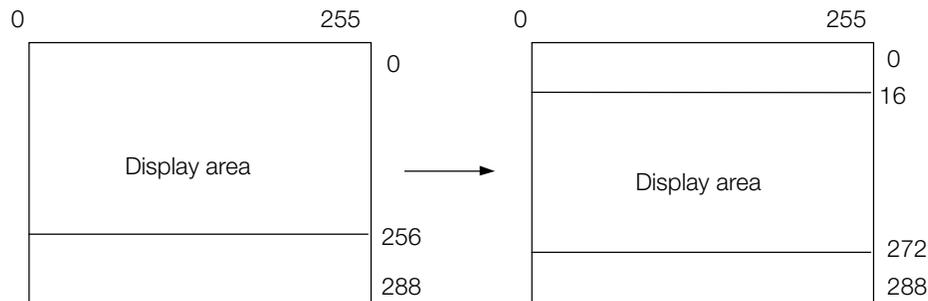

```
DISPENV        disp;

disp.screen.x = 0;          /*same as NTSC*/
disp.screen.y = 16;         /*(288-256)/2*/
disp.screen.w = 256;        /*same as NTSC*/
disp.screen.h = 256;        /*256*/

PutDispEnv(&disp);
```

Adjusting timing

In PAL mode, VSync interrupts are generated only 50 times a second. Many programs use the vertical sync interrupt (VSync()) to handle timing, so in these cases timing should be adjusted by 6/5.

Timing for Updating the Frame Buffer

The PlayStation can update the display area in the frame buffer (in operations such as swapping double buffers) at a rate different from the video frame rate (1/60 sec). However, if the display is updated at an irregular rate (that is, not a multiple of 1/60 second), flicker may result on the screen, which may be perceived by users as a bug in the application.

If the time spent in calculation and rendering operations exceeds the frame rate, motion on the screen may appear delayed for a moment. This is referred to as a skipped frame and is generally tolerated, but it may still be interpreted by some users as a defect.

The following points relate to managing frame rates when developing an application.

Timing for updating double-buffer switching

Switching double buffers is generally synchronized with the vertical sync.

In (A), buffer-switching depends on either rendering or displaying, whichever is slower. Thus, switching will become out of sync with vertical retrace and will take place during the display period. Therefore, unless a special effect is being performed intentionally, a VSync(0) should be executed when the buffers are switched so that synchronization is maintained.

```
(A)                             (B)
while (1) {                     while (1) {
....                            ....
DrawSync(0);                    DrawSync(0);
                                VSync(0);
swap_buffer();                  swap_buffer();
DrawOTag(ot);                   DrawOTag(ot);
}                               }
```

Keeping the frame rate constant

If the time required to create a frame is sometimes slightly less than 1/60 second and sometimes slightly more, the application's frame rate will jitter between 60fps and 30fps. This can cause objects on the screen to move in an unnatural manner, leading to complaints from users.

The same thing applies for other frame rates (such as 20, 15, etc.) whenever the time period for creating a frame is close to a 1/60 second boundary. (Or a 1/50 second boundary for PAL video systems.)

In these cases, the frame rate can be fixed (at the slower rate) by using the VSync counter to determine when to change buffers. The following example fixes the frame rate at 30 fps by calling VSync(2), which waits for the second vertical blank after the last VSync call:

```
while (1)
{
        DrawSync(0);

        /*Wait for 2nd vertical blank after last VSync call*/
        VSync(2);

        swap_buffer();
        . . .               /*build next frame*/
        DrawOTag(ot);       /*draw next frame*/
}
```

You can fix the frame rate at 20 frames/sec by using VSync(3) and at 15 frames/sec by using VSync(4).

Using absolute time

Sometimes it's not desirable to synchronize an application to the slowest frame rate, especially if frame rates drop only at specific and relatively rare instances. Also, forcing buffer switching to stay in sync with the fixed vertical sync generates idle periods where the CPU and GPU perform no real operations.

CONFIDENTIAL

If the frame rate varies, the internal clock of an application should not be based on the frame rate, because the speed of an object on screen will vary depending on the frame rate. Instead, you can use an absolute counter such as VSync(-1) or RCnt3 for such calculations.

```
(A)                              (B)
while (1) {                      while (1) {
....                             ....
        DrawSync(0);                     DrawSync(0);
        VSync(0);                        VSync(0);
        swap_buffer();                   swap_buffer();
        frame++;                         frame = VSync(-1);
        DrawOTag(ot);                    DrawOTag(ot);
}                                }
```

In these examples, *frame* is used to increment stages of motion or animation. The code in (A) will result in a movement slowdown when the frame rate drops, and a speedup when the frame rate increases. To avoid this, you can calculate *frame* as shown in (B). In this case, the overall motion will not be delayed or speeded up, even if frames are occasionally skipped due to overflow in calculation or rendering.

If the displacement or scrolling of an object is based on an absolute counter, there is no need to keep a fixed frame rate. Therefore, this approach can be considered a more thorough solution than forcing a fixed frame rate. However, there is an increased load on the program if updates are consistently made independently from double-buffer switching. Therefore, the choice of method should be determined based on the application's objectives.

Cancelling rendering operations

In interlaced mode, both calculation and rendering operations must be completed within 1/60 second. Therefore, when switching buffers, the vertical sync (VSync) must have a higher priority than drawing completion (DrawSync)

Therefore, rendering must be reset midway to synchronize with VSync(0). In general, rendering time varies more than calculation time so predicting rendering time is difficult. If a large figure is to be drawn and there is an overflow in rendering time, rendering can be reset midway so that screen flicker from interlace mode can be avoided.

```
(A)                              (B)
while (1) {                      while (1) {
....                             ....
DrawSync(0);                     VSync(0);
VSync(0);                        ResetGraph(1);
swap_buffer();                   swap_buffer();
DrawOTag(ot);                    DrawOTag(ot);
}                                }
```

In particular, performing MoveImage() on rectangles that are 16 dots wide or less and rendering of polygons that are narrow in width generate frequent page breaks and tend to have varying processing times. If these kinds of operations are to be performed often in interlace mode, buffer switching should not be dependent on rendering speed.

Return value from VSync(1)

If you call VSync(1), the return value is the time elapsed since the last VSync(0) or VSync(*n*) call, in horizontal sync units.

```
VSync(0);            /*wait for V-BLNK*/
t = VSync(1);        /*value of H from last VSync(0)*/
```

In this code, you'd expect *t* to be close to 0. However, this may not be the case, because functions such as sound callbacks and controller drivers are executed by the system during the vertical blank period.

VSync Synchronization in Interlace Mode

A problem arises when using the interlace single buffer (vertical 480 dot) mode. When switching the drawing in VSyncCallback() rather than VSync(), only the even fields of the first primitive rendering recorded to the OT are cached. As a result, the background may not be able to be cleared and an afterimage may remain.

Cause

In interlace mode, the even and odd fields are alternatively displayed at 1/60 second intervals. In other words, if the mode is 640x480, *y* displays the even line and the odd line alternatively. At such times, the GPU performs the following operations depending on whether even or odd is being displayed:

* When displaying an odd field, only the even line is rendered.
* When displaying an even field, only the odd line is rendered.

Though it is a single buffer, it becomes a mechanism whereby the screen being rendered is not displayed.

Therefore, the GPU must know whether the current video output is an odd field or an even field. However, during the vertical blank, the GPU always recognizes it as an even field.

Figure 8–25: Switching between even and odd fields

Video	even	V-BLNK	odd	V-BLNK	even	V-BLNK
GPU	even	even*	odd	even		even*

Note: even* refers to the fact that the GPU recognizes the current field as even when it should be odd.

Vsync is called at the V-BLNK start point (not the end point). Therefore, the items rendered during V-BLNK are only rendered in an even field. In Z-sorting, the section which is normally rendered first becomes the background section. As a result, the background of only half a field is not cleared, BG rendering is not performed, and other such problems arise.

Countermeasures

To avoid this, rendering must be started immediately after the vertical blank has *terminated*. Since VSyncCallback() cannot detect V-BLNK termination, you can use the following options:

* Add a callback using H-Sync callback (RCnt2)
* Increase the frequency of H-Sync during V-BLNK using VSync(1) is needed. However, a function GetODE() is being introduced as a more reliable method for distinguishing whether the current field is even or odd.

```
u_long GetODE(void); / *0...EVEN 1...ODD */
```

GetODE() is formally introduced in Library Ver. 3.7. When using prior libraries, incorporate the above-mentioned declaration into libgpu.h, etc.

Please note that the GetODE() value does not indicate the even or odd field of the current video output, but rather the value of the even or odd field recognized by the GPU. Since GetODE() returns only even frames during a V-BLNK immediately after a VSyncCallback(), an expedient is required. Refer to the sample program for details.

Supplement

In interlace mode, the odd and even fields are forcibly switched to 1/60[sec] with NTSC standards. Therefore, when in this mode, all processes must be completed at 1/60[sec]. When rendering time is the cause of processing mistakes (GPU trouble), these can be avoided by cancelling rendering (ResetGraph(1)), although of course a partial polygon will be produced. If the CPU is the cause (CPU trouble), screen disturbance can be avoided by using the previous OT until the CPU generates the next OT to perform re-rendering.

A rendering command (Draw0Tag) must be issued within the VSyncCallback() in order to implement this.

GPU timeout message

The following message may appear during program execution, and processing will stop:

```
GPU timeout:que=...,stat=...,chcr=...,madr=...
```

This GPU timeout is issued in cases where four seconds (240VSync) have elapsed, but the GPU non-blocking function has not yet terminated. An incorrect primitive link may be the cause.

For example, rendering will never terminate when primitives are listed in a loop, as below:

```
p0->p1->p2->p3....->p2
```

In this situation, a GPU timeout will be issued. The meaning of each value is as follows:

GPU timeout:que=%d,stat=%08x,chcr=%08x,madr=%08x

- **que**: Command queue remainder (0 when idle)
- **stat**: Only bit 26 has meaning

    1: Rendering in progress

    0: idle
- **chcr**: Only bit 24 has meaning

    1: Executing in the background

    0: idle
- **madr**: Address where DrawOTag() / LoadImage(), etc. are being executed

Chapter 9:
Basic Geometry Library

Table of Contents

CONFIDENTIAL

Overview

On the PlayStation, polygons are not drawn directly after calculation. Instead, the polygons on a given screen are sorted before drawing takes place:

Geometry arithmetic --> sorting --> drawing

Library and Header Files

Programs using the basic geometry library must link with the file `libgte.lib`.

Source code must include the header file `libgte.h`.

Theoretical Geometry Operations Using the Basic Geometry Library

When you use the functions provided by the basic geometry library, the GTE is activated to perform high-speed calculations. It performs two primary types of calculation:

Coordinate calculation, which takes the three-dimensional coordinates of polygon vertices and generates two-dimensional coordinates on the screen. These calculations can involve coordinate and/or perspective conversions.

Light source calculation, which finds the lighting of a polygon on the screen from the direction, color and intensity of a light source and the position of the polygon.

Coordinate Calculation

The basic geometry library assumes three types of fixed coordinate systems on the screen:

- Local coordinate system: the fixed coordinate system of the object.
- World coordinate system: the fixed coordinates of the world in which the object is placed.
- Screen coordinate system: the fixed coordinates of the screen.

An "object" consists of multiple polygons, and multiple objects compose one screen. Therefore, it is possible to have multiple local coordinate systems.

Normally, vertex data for each polygon are specified in the local coordinate system. To convert these into screen coordinates, the following conversions are necessary:
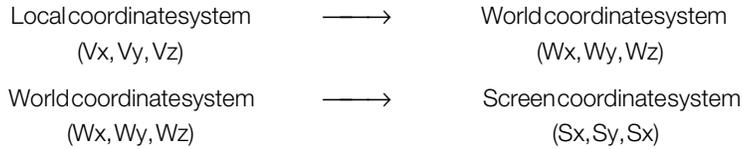
Local coordinate system $\longrightarrow$ World coordinate system

(Vx, Vy, Vz)   (Wx, Wy, Wz)

World coordinate system $\longrightarrow$ Screen coordinate system

(Wx, Wy, Wz)   (Sx, Sy, Sx)

$$\begin{bmatrix} Wx \\ Wy \\ Wz \end{bmatrix} = \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Vx \\ Vy \\ Vz \end{bmatrix} + \begin{bmatrix} WLx \\ WLy \\ WLz \end{bmatrix}$$

$$\begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix} = \begin{bmatrix} SWij \end{bmatrix} \begin{bmatrix} Wx \\ Wy \\ Wz \end{bmatrix} + \begin{bmatrix} SWx \\ SWy \\ SWz \end{bmatrix}$$

[WLij] is the world/local conversion matrix
[WLx, WLy, WLz] is the world/local translating vector
[SWij] is the screen/world conversion matrix
[SWx, SWy, SWz] are the screen/world translating vectors

Synthesizing this results in the following:

$$\begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix} = \begin{bmatrix} SWij \end{bmatrix} \left(\begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Vx \\ Vy \\ Vz \end{bmatrix} + \begin{bmatrix} WLx \\ WLy \\ WLz \end{bmatrix} \right) + \begin{bmatrix} SWx \\ SWy \\ SWz \end{bmatrix}$$

$$\begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix} = \begin{bmatrix} SWij \end{bmatrix} \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Vx \\ Vy \\ Vz \end{bmatrix} + \left(\begin{bmatrix} SWij \end{bmatrix} \begin{bmatrix} WLx \\ WLy \\ WLz \end{bmatrix} + \begin{bmatrix} SWx \\ SWy \\ SWz \end{bmatrix} \right)$$

The synthesized coordinate conversion matrices between coordinate systems and translating vectors are called:

Rotation matrix (RTM)
$$\begin{bmatrix} Rij \end{bmatrix} = \begin{bmatrix} SWij \end{bmatrix} \begin{bmatrix} WLij \end{bmatrix}$$

Translating vector (TRV)
$$\begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix} = \begin{bmatrix} SWij \end{bmatrix} \begin{bmatrix} WLx \\ WLy \\ WLz \end{bmatrix} + \begin{bmatrix} SWx \\ SWy \\ SWz \end{bmatrix}$$

The local coordinate values may be calculated with the rotation matrix by adding vectors to one matrix multiplied by the screen coordinate.

Local coordinate system $\longrightarrow$ Screen coordinate system

(Vx, Vy, Vz) (Sx, Sy, Sz)

$$\begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix} = [Rij] \begin{bmatrix} Vx \\ Vy \\ Vz \end{bmatrix} + \begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix}$$

You can use SetRotMatrix() and SetTransMatrix() to set the constant rotation matrix and the constant translating vector. They don't need to be changed if the coordinate system and position don't change. However, when a different local coordinate system is used for each object, each needs to be set separately.

Note: The local coordinate system setting method is up to the user.

You can use RotTrans() to find the screen coordinate value from the local coordinate system. Then coordinate conversion can be performed by the previously-set rotation matrix and translating vector.

Using a screen coordinate value found with RotTrans(), a parallel projected image of the object may be formed on the screen. In real vision, a distant object must be perspective-converted so that it appears small.

Screen coordinate system $\xrightarrow{\text{perspective conversion}}$ Screen coordinate system

(Sx, Sy, Sz) (SSx, SSy)

$$\begin{bmatrix} SSx \\ SSy \end{bmatrix} = (h/Sz) \begin{bmatrix} Sx \\ Sy \end{bmatrix}$$

h is the distance from the eye to the screen. Perspective conversion is done by multiplying the screen coordinate X and Y components by h/Sz.

You can use RotTransPers() to perform RotTrans() and perspective conversion together.

Note: In practice the following offset value is added in RotTransPers():

$$\begin{bmatrix} SSx \\ SSy \end{bmatrix} = (h/Sz) \begin{bmatrix} Sx \\ Sy \end{bmatrix} + \begin{bmatrix} OFX \\ OFY \end{bmatrix}$$

Also, the depth cueing interpolation coefficient p is calculated at the same time.

Light Source Calculation

The GTE uses a parallel light source complete diffusion reflection model for light source calculations. It doesn't rely on the position of the point of view, but determines the lighting on the basis of light source attributes and polygon attributes.

The following describes a light source calculation for one vertex of a polygon:

Each vertex of a polygon has two attributes:

- Normal line vector - (Nx, Ny, Nz)
- Vertex color - (R, G, B)

The normal line vector is usually given in the local coordinate system.

The light source has three attributes:

- Light source vector (direction and intensity) - (Lx, Ly, Lz)
- Color of the light source - (Lr, Lg, Lb)
- The ambient color - (BKr, BKg, BKb)

Since the light source is a parallel light source, there is no position information. Since it is the same for each object, it is given in the world coordinate system. Other than the influence of the light source, the background ambient color is present at all of the vertices.

The color (RR, GG, BB) in which the vertices are depicted on the screen is calculated as follows:

1. Convert normal line vector coordinates into the world coordinate system.

$$\text{Normal line vector (local)} \longrightarrow \text{Normal line vector (world)}$$

$$\begin{bmatrix} NWx \\ NWy \\ NWz \end{bmatrix} = \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

2. Calculate the "light source effect" by taking the inner product of the light source vector and the normal line vector (world).

$$\text{Normal line vector (world)} \bullet \text{Light source vector} \longrightarrow \text{Light source effect (L)}$$

$$L = \begin{bmatrix} Lx & Ly & Lz \end{bmatrix} \begin{bmatrix} NWx \\ NWy \\ NWz \end{bmatrix} = \begin{bmatrix} Lx & Ly & Lz \end{bmatrix} \bullet \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

Note: $\bullet$ means dot product.

3. Multiply the light source effect by the light source color for each item to get the color effect of the light source (local color) for vertices.

$$\text{Light source effect (L)} * \text{Light source color (Lr, Lg, Lb)} \longrightarrow \text{Light source color effect (Ll)}$$

$$\begin{bmatrix} Llr \\ Llg \\ Llb \end{bmatrix} = L \begin{bmatrix} Lr \\ Lg \\ Lb \end{bmatrix}$$

4. Add the light source color effects and the ambient colors to find the color effect of the whole environment.

$$\text{Light source color effect (Ll)} + \text{Ambient color (BK)} \longrightarrow \text{Color effect (LT)}$$

$$\begin{bmatrix} LTr \\ LTg \\ LTb \end{bmatrix} = L \begin{bmatrix} Lr \\ Lg \\ Lb \end{bmatrix} + \begin{bmatrix} BKr \\ BKg \\ BKb \end{bmatrix}$$

5. Multiply the vertex color by the color effect to find the vertex color for display.

$$RR = R * LTr$$
$$GG = G * LTg$$
$$BB = B * LTb$$

CONFIDENTIAL

For example, if there were three light sources in the above procedure, (1) and (2) would respectively be as follows:

$$\begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} = \begin{bmatrix} Lx1 & Ly1 & Lz1 \\ Lx2 & Ly2 & Lz2 \\ Lx3 & Ly3 & Lz3 \end{bmatrix} \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

Here, if the product of multiplying

$$\begin{bmatrix} Lx1 & Ly1 & Lz1 \\ Lx2 & Ly2 & Lz2 \\ Lx3 & Ly3 & Lz3 \end{bmatrix} \begin{bmatrix} WLij \end{bmatrix}$$

is [Lij], then(1) and (2) can result in the following one-off matrix calculation.

$$\begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} = \begin{bmatrix} Lij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

This matrix [Lij] is called the Local Light Matrix (LLM) in GTE.

Therefore, there is no need to convert the normal line vector for each polygon into the world coordinate system. It is sufficient to calculate just the local light matrix [Lij] for each object.

The local light matrix, like the rotation matrix, is a GTE constant matrix. The local light matrix [Lij] may be set by SetLightMatrix().

Further, if there are three light sources, then there are three light source colors so that (3) above is as follows:

$$\begin{bmatrix} Llr \\ Llg \\ Llb \end{bmatrix} = L1 \begin{bmatrix} L1r \\ L1g \\ L1b \end{bmatrix} + L2 \begin{bmatrix} L2r \\ L2g \\ L2b \end{bmatrix} + L3 \begin{bmatrix} L3r \\ L3g \\ L3b \end{bmatrix}$$

$$= \begin{bmatrix} L1r & L2r & L3r \\ L1g & L2g & L3g \\ L1b & L2b & L3b \end{bmatrix} \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix}$$

If this matrix,

$$\begin{bmatrix} LRij \end{bmatrix} = \begin{bmatrix} L1r & L2r & L3r \\ L1g & L2g & L3g \\ L1b & L2b & L3b \end{bmatrix}$$

is [LRij], then (3) and (4) above for 3 light sources will be as follows:

$$\begin{bmatrix} LTr \\ LTg \\ LTb \end{bmatrix} = \begin{bmatrix} LRij \end{bmatrix} \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} + \begin{bmatrix} BKr \\ BKg \\ BKb \end{bmatrix}$$

This matrix, [LRij] is called the Local Color Matrix (LCM) in GTE. The local color matrix and local light matrix, like the rotation matrix, are GTE constant matrices. Each may be set by SetLightMatrix() and SetColorMatrix().

Also, ambient color is called Back Color (BK) and may be designated by SetBackColor().

CONFIDENTIAL

The procedures (1) (2) (3) (4) and (5) explained above may be summarized as follows based on up to three light sources.

6. Normal line vector (local) --> Light source effect (local light matrix)

$$\begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} = \begin{bmatrix} Lij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

7. Light source effect --> Color effect (local color matrix, back color)

$$\begin{bmatrix} LTr \\ LTg \\ LTb \end{bmatrix} = \begin{bmatrix} LRij \end{bmatrix} \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} + \begin{bmatrix} BKr \\ BKg \\ BKb \end{bmatrix}$$

8. Color effect, vertex color --> Vertex screen color

$$RR = R * LTr$$
$$GG = G * LTg$$
$$BB = B * LTb$$

There is a function in the basic geometry library

```
NormalColorCol()
```

which performs this 6), 7) and 8) once. In GTE light source effect is called local color (LC).

Normal Line Vector, Light Source Vector Direction

The normal line vector given to each vertex of a polygon should be placed in a direction from the front to the back (pointing into the object). The light source vector is not the position of the light source, but should be the direction of the rays.

GPU Code

The GTE has a register that maintains the GPU packet code. Light source calculation functions output a GPU packet with the RGB value placed at the beginning of the packet. The GPU packet and the RGB code are a single word, so the RGBcd portion of the packet may be created with one memory write. If the GPU packet code register is not specified correctly, the GPU packet cannot be properly generated.

With functions that have no input primary color vector, you should call SetRGBcd() to set the primary color vector and GPU code. These functions are Intpl(), NormalColor(), and NormalColor3().

With functions that do have input color vectors, the GPU packet code is automatically copied from the upper 8 bits of the color vector to the GPU packet code register. These functions are: ColorCol(), ColorDpq(), ColorMatCol(),ColorMatDpq(), DpqColorLight(), DpqColor(), DpqColor3(), NormalColorDpq(), NormalColorDpq3(), NormalColorCol(), NormalColorCol3(), RotColorDpq(), RotColorDpq3(), RotAverageNclipColorDpq3(), RotAverageNclipColorCol3(), RotColorMatDpq().

Normal Line Clipping

Normal line clipping is a method of increasing drawing speed by not drawing polygons that are visible from the back. Whether something is visible from the front or from the back is determined by the sign of the Z component of the normal line screen coordinate system of the polygon.

Normal line clipping is effective when there is a closed curved surface such as that of a sphere. This is also effective in reducing the so-called Z sorting problem.

Normal Line Clipping Function

The Z component of a polygon normal line screen coordinate system is found by converting the coordinates of the normal line. It may also be found by the vector product of the two sides of the polygon.

A function calculating the 2-dimensional vector product for normal line clipping, Normal Clip() is provided in the basic geometry library. NormalClip() calculates a value to distinguish between the front and back of triangles from the screen coordinates of three vertices. Front and back can be judged by whether the return value is positive or negative, but the sign will change with the direction of the coordinate axis, and the order of the vertices. Here we hypothesize a coordinate system.

Figure 9-1: Coordinate Axes
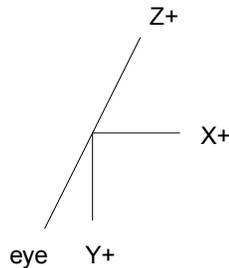
The viewpoint is in the negative direction of the Z axis. Looking from the view point, with the three vertices arranged clockwise, NormalClip() will return a positive value.

Figure 9-2: Vertex Order
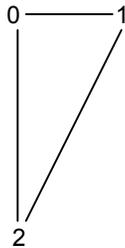
With the following performing the same calculation as NormalClip(), normal line clipping is performed, coordinate calculation is halted and an incorrect sx, sy value is returned when the vector product is negative or 0. When using these functions, the order of the vertices of a polygon must be modeled so that they will rotate clockwise when seen from the front.

    RotNclip()
    RotNclip3()
    RotNclip4()
    RotAverageNclip3()
    RotAverageNclip4()
    RotAverageNclipColorDpq3()
    RotAverageNclipColorCol3()

RotNclip4() and RotAverageNclip4() are functions which perform the same calculations as NormalClip(). Since these functions use the first three points and calculate a vector product value, you must use one of the vertex orderings from Figure 8-3.

Figure 9-3: Four Vertices
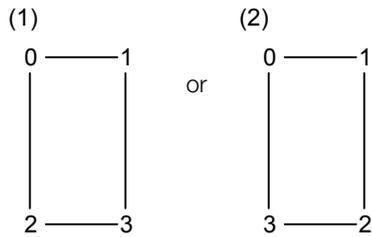
However, since GPU will not draw rectangles in the order indicated by (2), it is sensible to use (1).

Depth Cueing

Depth cueing is an effect that makes objects at a distance appear hazy. It is accomplished by blending the original polygon color with the far color, as a function of the Z-value of the screen coordinate system. If the far color is white, distant objects appear slightly obscured by fog. If the far color is black, distant objects appear dark.

The following depth-queuing terms are used in the PlayStation:

*   Back Color, BK - Ambient color set by SetBackColor()
*   Far Color, FC - Far color set by SetFarColor()
*   BG Color - The color applied in the background

To blend colors into the background with depth cueing, match the far color and the background color. Note that back color and BG color differ.

Depth cueing methods can be broadly divided into two categories:

*   Using vertex colors. This method is used with non-textured polygons. It can also be used with textured polygons, in cases where

    a) either the far color is black or very dark

    b) The texture is relatively bright and composed only of colors close to the far color without any dark points. In this case, the object may not completely blend with the far color.
*   Using texture colors. This method can be used with textured polygons in general.

Implementation of Depth Cueing (Common Operations)

Interpolation coefficients

The GTE can perform efficient depth cueing through non-linear interpolation of the far color. You set the depth for depth cueing by calling SetFogNear(), SetFogFar(), or SetFogNearFar().

Once the depth has been set, the non-linear interpolation coefficient p can be obtained by calling one of the RotTransPers() functions, with p having a value within the range 0 to 4096.

If Z is sufficiently small, the value of p will saturate at 0. If Z is sufficiently large, the value of p will saturate at 4096 (please refer to the descriptions for the SetFog…() functions for more information).

In general, the depth cueing interpolation calculation can be represented by

```
interpolation calculation (o,f,p) = ((o x p) + (f x (4096-p))) / 4096
```

where o is the original color, f is the far color, and p is the interpolation coefficient.

CONFIDENTIAL

If the far color is the same as the background color, rendering the polygon is unnecessary if p is 4096. For a given otz value, otz2p() can be used to obtain roughly the same value of p as the interpolation coefficient generated by GTE. Conversely, p2otz() can be used to determine otz from p. p2otz() and otz2p() are relatively expensive functions that make use of division. It is also possible for the user to specify an independent value for the interpolation coefficient p.

Method for preparing interpolated data (using CLUT or texture)

Data can be prepared for different values of p beforehand or generated for specific values of p at runtime. DpqColor() is useful for interpolating colors such as these. (Use SetFarColor() to set the far color before calling DpqColor().)

Depth Cueing Using Vertex Colors

The vertex color method of depth cueing intepolates the polygon vertex colors with the far color. To use it:

- Set depth using the SetFog…() functions
- Call SetFarColor() to set the far color.
- Use the RotTransPers…() functions to obtain the interpolation coefficient p for each vertex of each polygon.
- Pass p to a function such as NormalColorDpq(), which selects a vertex screen color that has been interpolated with the far color. Besides NormalColorDpq…(), DpqColor…(), …ColorDpq() and Intpl() functions can also be used for this calculation.

Depth Cueing Using Textures

The texture method of depth cueing interpolates the texture color and the far color. The implementation method is different for each case.

What color to interpolate:

- Interpolation using CLUT colors

  CLUT interpolation can be performed for textures that use a CLUT.
- Interpolation using the colors of the texture itself

For textures that do not use a CLUT, the only option is to interpolate with the colors of the texture itself. For textures that use a CLUT, depth cueing can be applied when textures are selected according to depth (e.g. by using the mip-map method). In these cases, interpolation based on texture color can be used in conjunction with interpolation based on the CLUT.

Methods of generating interpolation data

- CLUT or texture data can be generated in advance for different values of the interpolation coefficient p, then used according to the Z and p values of the polygon or object to be rendered. This method minimizes calculation time, but requires more memory to store the data, and the resolution of p cannot be very high.
- CLUT or texture data can be generated at runtime for specific values of p. This method requires more calculation time during execution, but relatively little memory is needed, and p can have a high resolution.

Changing the rendered texture

- Changing the coordinates referred to by the polygon (CLUT or texture)

  Data for different values of p are saved in free areas in the frame buffer. Depth cueing is implemented by changing the texture coordinates, the CLUT ID, the texture page ID, etc. referred to by each polygon.

- Changing the CLUT or texture in the frame buffer

   In this method, the rendered texture is changed by substituting the CLUT or texture in the frame buffer. Since polygon packets do not need to be changed, texture depth cueing can be easily rendered using the extended graphics library (libgs).

The following three methods can be used to modify data in the frame buffer:

1. Using DR_LOAD primitives

   Data is set up in DR_LOAD primitives and written to the frame buffer. Large amounts of data can be divided up into multiple DR_LOAD primitives. In these cases, it is faster to divide up the data so that it is arranged as wide as possible in the frame buffer.

   If one interpolation coefficient p can be used for a single CLUT or texture in a single frame, then this information is entered into the beginning of the ordering table and rendered. If multiple occurrences of p, corresponding to depth, are to be used for a single CLUT or texture in a single frame, the following operations should be performed. DR_LOAD primitives are entered into multiple places in the ordering table to transfer data for the values of p corresponding to the otz values.

   In the following examples, four values of p are used. Polygons are assumed to have been entered beforehand into the ordering table using DrawOTag() or libgs.

Figure 9-4: Writing data using DR_LOAD primitives
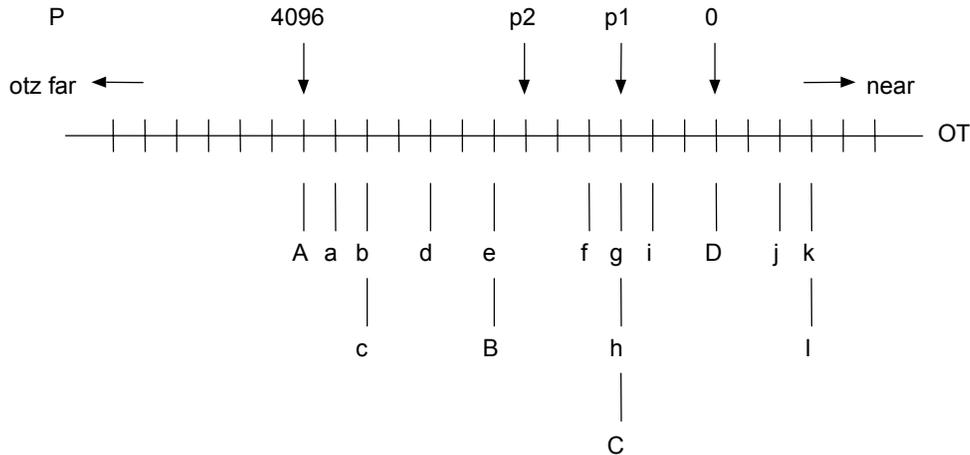
Since polygons deeper than P=4096 do not merge into the background, they are not drawn. The DR_LOAD primitive is registered to the otz head when drawing is necessary.

a-1 : polygon

A-D : DR_LOAD primitive

A : DR_LOAD transfers (p = 4096 + p2) / 2) data
B : DR_LOAD transfers (p = p2 + p1) / 2) data
C : DR_LOAD transfers (p = p1 + 0) / 2) data
D : DR_LOAD transfers (p = 0) / 2) data

For a value of p corresponding to a rendering range from p1 to p2, a DR_LOAD primitive (B in the Figure) for transferring data using p is entered into the otz position corresponding to p2. A DR_LOAD (C in the figure) for transferring data using the next p is entered into the otz position corresponding to p1.

For example, the rendering sequence for the case shown in the figure would be

A - a - b - c - d - e - B - f - g - h - C - i - D - j - k - l

Polygons a - e would be rendered with the data transferred using DR_LOAD A, polygons f - h are rendered with the data transferred using DR_LOAD B, and so on.

2. Writing data using DR_MOVE primitives

   In this method, data for different values of p is written into free areas of the frame buffer. The data is transferred to the actual locations used for rendering before the polygons are rendered. As in the case with the DR_LOAD primitives above, data can be saved in the ordering table so that depth cueing can be achieved on multiple polygons using a single CLUT or texture with values of p corresponding to the depths of each of the polygons.

3. Using LoadImage() to transfer data from main memory

   In this method, LoadImage()is used to transfer data from main memory to the area to be used in the frame buffer. Note that using LoadImage() too often causes a heavy load on the CPU. Thus, this method is not appropriate if multiple values of p need to be used for a single CLUT or texture within a single frame.

Material Light Source Calculation with Material Quality

Light source calculation in PlayStation (without depth cueing) is summarized as follows:

1.

$$\begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} = \begin{bmatrix} Lij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

2.

$$\begin{bmatrix} LTr \\ LTg \\ LTb \end{bmatrix} = \begin{bmatrix} LRij \end{bmatrix} \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} + \begin{bmatrix} BKr \\ BKg \\ BKb \end{bmatrix}$$

3.

$$RR = R \times LTr$$
$$GG = G \times LTg$$
$$BB = B \times LTb$$

Notes:

(Nz, Ny, Nz): Normal vectors
[Lij]: Local light matrix (LLM)
(L1,L2,L3): Local light vector (LLV)
[Lri]: Local color matrix (LCM)
(BKr, BKg, BKb): Back color (BK)
(RLT,GLT,BLT): Local color (LC)
(R, G, B): Original color vectors
(RR, GG, BB): Output color vectors

In this manual the calculation above is abbreviated in the following way:

1. LLV = LLM x v0
2. LC = BK + LCM x LLV
3. v2 = v1 x LC

Following the calculation of (1) you may also obtain LLV again with each item of LLV squared in the following manner:

1. LLV = LLM x v0
2. LLV = LLV^2 = (L1^2, L2^2, L3^2)
3. LC = BK + LCM x LLV
4. v2 = v1 x LC

If this is done, the lighted portion on screen will become narrower and the material quality of the object will appear to have changed. The basic geometry library provides

- RotColorMatDpq
- ColorMatDpq
- ColorMatCol

as functions with material quality.

Functions with Three or Four Vertices

There are functions in the basic geometry library which perform one-off coordinate conversion of polygons with three or four vertices, and light source calculation.

For example, RotTransPers3() and RotTransPers4() functions do one-off coordinate conversion of three and four vertex polygons respectively. Also NormalColorCol3() and NormalColorDpq3() convert the 3 vertex light source calculation once.

By using these functions, triangles and rectangles with independent vertices may be drawn at high speed.

libgte Argument Format

In the GTE, all numbers are expressed in fixed-point notation. For example, each component of a rotational matrix is a (1,3,12) fixed-point number, which means:

- Sign : 1 bit
- Integer value: 3 bits
- Fractional value: 12 bits

Because of this, RotTrans (&v0, &v1, &flag) is calculated in the following manner.

$$\begin{bmatrix} v1.vx \\ v1.vy \\ v1.vz \end{bmatrix} = \begin{bmatrix} Rij \end{bmatrix} \begin{bmatrix} v0.vx \\ v0.vy \\ v0.vz \end{bmatrix} + \begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix}$$

$$= \begin{bmatrix} (R00 \times v0.vx + R01 \times v0.vy + R02 \times v0.vz) >> 12 \\ (R10 \times v0.vx + R11 \times v0.vy + R12 \times v0.vz) >> 12 \\ (R20 \times v0.vx + R21 \times v0.vy + R22 \times v0.vz) >> 12 \end{bmatrix} + \begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix}$$

$$v1.vx = TRX + (R00 \times v0.vx + R01 \times v0.vy + R02 \times v0.vz) >> 12$$

$$v1.vy = TRY + (R10 \times v0.vx + R11 \times v0.vy + R12 \times v0.vz) >> 12$$

$$v1.vz = TRZ + (R20 \times v0.vx + R21 \times v0.vy + R22 \times v0.vz) >> 12$$

v0, v1 are of type SVECTOR.

```
typedef struct{
        short vx, vy;
        short vz, pad;
} SVECTOR;
```

[Rij] is a rotation matrix, (TRX, TRY, TRZ) are translating vectors. Therefore the formats of V0 and V1 less than the decimal point are the same as (TRX, TRY, TRZ).

The format of (TRX, TRY, TRZ) is (1, 31, 0), so that v0 is (1, 15, 0) and v1 is (1, 31, 0).

Recommended Format

The recommended format for GTE constants is shown below. Though formats other than this may be calculated, it becomes difficult and it must be taken into account that a 12-bit shift is built into the GTE. Please refer to the libgte reference manual for the argument format of each function.

Table 9-1: Recommended Format for GTE Constants

Item	Bit Format
Rotational matrix [Rij]	(1, 3, 12)
Translating vector (TRX, TRY, TRZ)	(1, 31, 0)
Local light matrix [Lij]	(1, 3, 12)
Local color matrix [L (R, G, B) i]	(1, 3, 12)
Back color (RBK, GBK, BBK)	(0,32,0)(0...255)
Far color (RFC, GFC, BFC)	(0,32,0)(0...255)

Libgte Function Flag Variables

Flag variables are returned by the following coordinate calculation functions for performing clipping:

RotTransPers(), RotTransPers3(), RotTrans(), RotTransPers4(), RotAverage3(), RotAverage4(), RotNclip(), RotNclip3(), RotNclip4(), RotAverageNclip3(), RotAverageNclip4(), RotColorDpq(), RotColorDpq3(), RotAverageNclipColorDpq3(), RotAverageNclipColorCol3(), RotColorMatDpq()

The flags return to their original state immediately after the functions have finished their coordinate transformations.

Functions doing coordinate transformations on 3 or 4 vertices, such as RotTransPers3() or RotTransPers4(), return an OR of the coordinate transformation result for each vertex. When RotNclip4() or RotAverageNclip4() return a value of -1 (that is, when a vertex cannot be calculated due to a normal clip) it is treated as if it were an OR of the result from a 3-vertex coordinate transformation.

The flag bits are as follows:

Table 9-2: Flag Bit Settings

Bit	Contents
31	(30) | (29) | (28) | (27) | (26) | (25) | (24) | (23) | (18) | (17) | (16) | (15) | (14) | (13) | (11)
30	Calculation overflow (>=2^43)
29	Calculation overflow (>=2^43)

Bit	Contents
28	Calculation overflow (>=2^43)
27	Calculation overflow (<-2^43)
26	Calculation overflow (<-2^43)
25	Calculation overflow (<-2^43)
24	The output value exceeds (-2^15, 2^15)
23	The output value exceeds (-2^15, 2^15)
22	The output value exceeds (-2^15, 2^15)
21	Output value exceeds(0, 2^8)
20	Output value exceeds (0, 2^8)
19	Output value exceeds (0, 2^8)
18	The value of Z in the screen coordinate system exceeds (0, 2^16)
17	The Z coordinate is smaller than h/2 after perspective transformation*
16	Calculation overflow (>=2^32)
15	Calculation overflow (<-2^32)
14	The X coordinate exceeds (-2^10, 2^10) after perspective transformation
13	The Y coordinate exceeds (-2^10, 2^10) after perspective transformation
12	The value of p exceeds (0, 2^12)
11~0	Not used

* `h` is the distance between the viewpoint and the screen.

The following functions return 16-bit flags: RotTransPersN(), RotTransPers3N()

The 16-bit flag bits are as follows:

Table 9-3: 16-Bit Flag Bit Settings

Bit	Contents
15	Calculation overflow (>=2^43)
14	Calculation overflow (>=2^43)
13	Calculation overflow (>=2^43)
12	The value of X in the screen coordinate system before perspective transformation exceeds (-2^15, 2^15)
11	The value of Y in the screen coordinate system before perspective transformation exceeds (-2^15, 2^15)
10	The value of Z in the screen coordinate system exceeds (-2^15, 2^15)
9	Output value exceeds (0, 2^8)
8	Output value exceeds (0, 2^8)
7	Output value exceeds (0, 2^8)
6	The value of Z on the screen coordinate system exceeds (0, 2^16)
5	The Z coordinate is smaller than h/2 after perspective transformation*
4	Calculation overflow (>=2^32)
3	Calculation overflow (>=2^32)
2	The X coordinate exceeds (-2^10, 2^10) after perspective transformation
1	The Y coordinate exceeds (-2^10, 2^10) after perspective transformation
0	The value of p exceeds (0, 2^12)

* `h` is the distance between the viewpoint and the screen.

CONFIDENTIAL

About libgte Mesh Functions

The basic geometry library supports two types of triangular mesh data. By using mesh data, the number of vertex calculations and the volume of data can be reduced.

One is called Strip Mesh and the vertices are arranged in zig-zags as shown below:

Figure 9-5: Strip Mesh

```
     1 ——— 3
    /| |\
   / | | \
  /  | |  \
 0 ——— 2 ——— 4
```
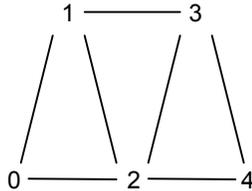
The other is called Round Mesh and the vertices surround vertex 0 as shown below:

Figure 9-6: Round Mesh

```
     2 ——— 3
    /| |\
   / | | \
  /  | |  \
 1 ——— 0 ——— 4
```
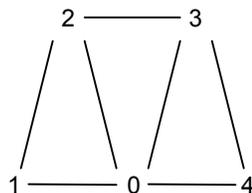
In either case, when the first triangle 012 is clockwise in this order, 012 is displayed and the fronts and backs of the other three triangles will be determined by this triangle.

However, when performing light source calculation (shading and depth cueing) with this type of data, normal line clipping cannot be performed so the calculation is not always speeded up. Mesh data is effective in improving calculations with "no shading and depth cueing" and "flat shading."

Changing Screen Offsets

There are two methods for altering the PlayStation screen offset. One is to use the libgpu function SetDefDrawEnv(). The other is to use the SetGeomOffset() function provided in the basic geometry library.

PMD Functions

Libgte has PMD functions that link GPU packets to the created OT, after they perform coordinate transformations, when reading the data formats shown below. GPU packet data is preset for constants, color variables, texture variables, and the like, so drawings can be done at high speed if just the coordinate variables are set.

PRIMITIVE Group

In PMD data, when polygons having the same attributes are grouped together and the PRIMITIVE Gp object components (primitives) drawing packet is drawn up, one packet represents one primitive.

A primitive defined in PMD is different from a libgpu primitive. Together with the processing of the perspective conversion by libgs it is also converted to the drawing primitive.

CONFIDENTIAL

One primitive group is shown below.

Figure 9-7: PACKET Gp Configuration

bit31(MSB)   bit0(LSB)

TYPE	NPACKET
Packet Data #0	
Packet Data #1	
Packet Data #2	
:	

NPACKET:Number of packets
TYPE : Packet type
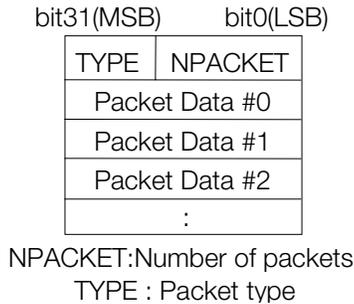
Table 9-4: 4-Type Bit Configuration

bit No.	When 0	When 1
16	Triangle	Quadrilateral
17	Flat	Gouraud
18	Texture-On	Texture-Off
19	Independent Vertex	Public Vertex
20	Light Source Calculation OFF	Light Source Calculation ON
21	With Back clip	No Back clip
22-31	(Reserved)	

The Packet Data configuration changes with the TYPE value. The separate TYPE Packet Data configuration is as follows:

Note 1: In order to make the Primitive section (POLY_***) in the configuration correspond to the double buffer, two sets are provided.

Both contents must be initialized beforehand.

Note 2: Bit 20,21 have no effect on the Packet Data configuration.

TYPE Packet Data Configurations

TYPE=00 (Triangle/Flat/Texture-On/Independent Vertex)

```
struct _poly_ft3 {
POLY_FT3 pkt[2];
SVECTOR v1, v2, v3;
}
```

TYPE=01 (Quadrilateral/Flat/Texture-On/Independent Vertex)

```
struct _poly_ft4 {
POLY_FT4 pkt[2];
SVECTOR v1, v2, v3, v4;
}
```

TYPE=02 (Triangle/Gouraud/Texture-On/Independent Vertex)

```
struct _poly_gt3 {
POLY_GT3 pkt[2];
SVECTOR v1, v2, v3;
}
```

CONFIDENTIAL

TYPE=03 (Quadrilateral/Gouraud/Texture-On/Independent Vertex)

```
struct _poly_gt4 {
POLY_GT4 pkt[2];
SVECTOR v1, v2, v3, v4;
}
```

TYPE=04 (Triangle/Flat/Texture-Off/Independent Vertex)

```
struct _poly_f3 {
POLY_F3 pkt[2];
SVECTOR v1, v2, v3;
}
```

TYPE=05 (Quadrilateral/Flat/Texture-Off/Independent Vertex)

```
struct _poly_f4 {
POLY_F4 pkt[2];
SVECTOR v1, v2, v3, v4;
}
```

TYPE=06 (Triangle/Gouraud/Texture-Off/Independent Vertex)

```
struct _poly_g3 {
POLY_G3 pkt[2];
SVECTOR v1, v2, v3;
}
```

TYPE=07 (Quadrilateral/Gouraud/Texture-Off/Independent Vertex)

```
struct _poly_g4 {
POLY_G4 pkt[2];
SVECTOR v1, v2, v3, v4;
}
```

TYPE=08 (Triangle/Flat/Texture-On/Shared Vertex)

```
struct _poly_ft3c {
POLY_FT3 pkt[2];
long vp1, vp2, vp3;
}
```

TYPE=09 (Quadrilateral/Flat/Texture-On/Shared Vertex)

```
struct _poly_ft4c {
POLY_FT4 pkt[2];
long vp1, vp2, vp3, vp4;
}
```

TYPE=0a (Triangle/Gouraud/Texture-On/Shared Vertex)

```
struct _poly_gt3c {
POLY_GT3 pkt[2];
long vp1, vp2, vp3;
}
```

TYPE=0b (Quadrilateral/Gouraud/Texture-On/Shared Vertex)

```
struct _poly_gt4c {
POLY_GT4 pkt[2];
long vp1, vp2, vp3, vp4;
}
```

TYPE=0c (Triangle/Flat/Texture-Off/Shared Vertex)

```
struct _poly_f3c {
POLY_F3 pkt[2];
long vp1, vp2, vp3;
}
```

TYPE=0d (Quadrilateral/Flat/Texture-Off/Shared Vertex)

```
struct _poly_f4c {
POLY_F4 pkt[2];
long vp1, vp2, vp3, vp4;
}
```

TYPE=0e (Triangle/Gouraud/Texture-Off/Shared Vertex)

```
struct _poly_g3c {
POLY_G3 pkt[2];
long vp1, vp2, vp3;
}
```

TYPE=0f (Quadrilateral/Gouraud/Texture-Off/Shared Vertex)

```
struct _poly_g4c {
POLY_G4 pkt[2];
long vp1, vp2, vp3, vp4;
}
```

The pkt[] is the corresponding drawing primitive packet, the v1~v4 values are the vertex coordinate values, and the vp1~vp4 values are the offsets from the head of the shared coordinates string.

VERTEX

The VERTEX section is the SVECTOR structure array which displays the shared vertex. One structure format is shown below:

Figure 9-8: VERTEX

VX,VY, VZ: x,y,z values of vertex coordinates (16 bit integers)

SMD, RMD Functions

The SMD and RMD functions are high-speed versions of the PMD function. They both process the same data format as the PMD function. The SMD function usually performs normal clipping, while the RMD function usually does not.

Polygon Division

The PlayStation is designed to form many small polygons efficiently. When larger polygons are broken down into smaller ones using division, clipping is performed more efficiently, and texture distortion is reduced.

The polygon division process uses the automatic division attribute of libgs that is applied to objects, or it can be invoked by directly calling one of the functions listed below.

Table 9-5: Polygon Division Functions

Function name	Corresponding primitive
DivideF3	Flat Triangle
DivideF4	Flat Quadrilateral
DivideFT3	Flat Texture Triangle
DivideFT4	Flat Texture Quadrilateral
DivideG3	Gouraud Triangle
DivideG4	Gouraud Quadrilateral
DivideGT3	Gouraud Texture Triangle
DivideGT4	Gouraud Texture Quadrilateral

Chapter 10:
Extended Graphics Library

Table of Contents

CONFIDENTIAL

Overview

The extended graphics library (libgs) integrates the 2D and 3D graphics systems used in libgpu and libgte. It is designed to work well with the standard graphics file formats that can be created by PlayStation authoring tools:

- The TIM format stores image resolution, color numbers and color look-up table information.
- The TMD format stores multiple objects, scale information and texture address information.
- HMD is a new format that was added in version 4.0 of the libraries. See Chapter 18, "HMD Library", for more information about this format.

In contrast with the libgpu and libgte libraries which process polygon-level data, libgs processes data by object units (groups of polygons), allowing 3D programs to be prototyped easily. By adding attributes to objects, it's easy to create special effects.

Using libgs involves extra overhead compared to using libgpu and libgte. However, libgs is an open architecture. Therefore, once you are ready to produce your game, you can optimize it by adding user-defined functions (via a jump table) that use libgpu and libgte services.

Library and Header Files

To use the extended graphics library, you must link with the library file `libgs.lib`.

Source code must include the header file `libgs.h`.

Libgs features

The main features of libgs are:

- Hierarchical coordinate systems

  Any object's coordinate system can be designated as a parent or a subordinate of another. Changes to the parent coordinate system are automatically applied to the subordinates.
- Light source calculation (3 light sources, depth queuing, ambient)

  Libgs performs automatic lighting calculations using parameters set by the user.
- Automatic division of polygons

  Libgs can automatically sub-divide large polygons to avoid problems associated with clipping.
- Semi-transparent processing

  Objects and/or their textures can be drawn as semi-transparent/translucent by setting the appropriate attributes
- Perspective texture mapping of objects.
- Viewpoint control

  You can easily manipulate the viewing angle using the view structures defined within libgs.
- Z-sort processing

  You can sort and draw objects according to their Z-depth values by using the GsOT structure and the GsSort functions.
- OT initialization hierarchic compression

  Objects with greatly differing Z values may be sorted into separate OTs (Z-sort ordering tables) and then linked into one OT prior to drawing.

- Frame double buffer

  Libgs implements a graphics double-buffer system , in order to avoid the problems associated with drawing into memory being displayed. Initialization and switching of the buffers are easily performed.

- Automatic adjustment of aspect ratio

  When the view aspect ratio is not normal dot, the display of an object's vertical is automatically adjusted to appear as a normal dot aspect ratio.

- 2D clipping offset processing

  Libgs performs 2D clipping according to values set by the user. In addition, you can define a 2D offset, which will be added to the screen coordinates of all objects prior to display.

- Sprite/BG/Line

  Libgs provides structures and routines for easily displaying 2D sprites, lines, and cell-based scrolling backgrounds.

- libgpu/libgte coexistence

  Libgs combines Libgs(2D) and Libgte(3D) into a complete, easy to manage, 3D graphics pipeline.

Coordinate Systems

GsCOORDINATE2 is a structure describing a libgs coordinate system. The coordinate system is a hierarchical structure which takes the world coordinate system as the most significant, and it is integrated from a lower level to a higher level.

GsCOORDINATE2's members describe the coordinate system and a work area for speeding up coordinate calculations. The MATRIX parameters describe the coordinate system relative to its parent coordinate system. For the size of the coordinate system space, X, Y and Z are all 32 bits.

The definition of the GsCORDINATE2 is as follows:

```
struct GsCOORDINATE2{
        unsigned long flg;          /*0: coord has been rewritten 1: workm
                                    values are still valid*/
        MATRIX coord;               /*A 3 x 3 matrix containing coordinate
                                    rotation, translation, and scale info*/
        MATRIX workm;               /*Result of multiplication of coord with
                                    the WORLD coordinate system*/
        GsCOORD2PARAM *param;        /*rotation, scale, and translation
                                    parameters*/
        GsCOORDINATE2 *super;        /*pointer to superior coordinate system*/
        GsCOORDINATE2 *sub;          /*pointer to subordinate coordinate
                                    system*/
};
```

GsInitCoordinate2() initializes the members of GsCOORDINATE2. You can also set the members directly.

Order of Rotation/Translation

Rotation is executed first, followed by translation.

The order of rotation,when the rotation matrix is created by the function RotMatrix() and set in GsCOORDINATE2, is [Z --> Y --> X].

Clearing Flags

When requesting local-to-world matrix from the hierarchical coordinate system, optimization is accomplished by setting the *flg* member of a previously calculated coordinate system to 1 and preserving the results stored in the member *workm*.

If the parameters of a GsCOORDINATE2 have been rewritten, always remember to set *flg* to zero, indicating that the contents of *workm* have already been used. Recalculation will not be performed unless *flg* is zero.

If parent coordinates are modified, this is automatically reflected in all of the child coordinates, so there is no need to clear the child coordinate's flag.

Examples of Coordinate System Setting

Examples of translation and rotation are presented below.

Example 1: Translation

```
GsCOORDINATE2 sample_coord;    /*coordinate system which sets translation */
int x,y,z;                     /*amount of parallel shift*/

sample_coord.coord.t[0] = x;
sample_coord.coord.t[1] = y;
sample_coord.coord.t[2] = z;
```

Example 2: Rotation

```
GsCOORDINATE2 sample_coord;    /*coordinate system in which rotation is
                                 set */
SVECTOR rot;                   /*rotation angle set (x,y,z)*/
MATRIX tmp1;                   /*rotation matrix requested*/

RotMatrix(&rot, &tmp1);        /*when RotMatrix() is used the order of
                                 rotation is zyx*/
sample_coord.coord = tmp1;
```

Objects

You manipulate objects by means of 3D object *handlers*. This section explains the basic object handler used by libgs, GsDOBJ2. The other types of object handlers are GsDOBJ3 and GsDOBJ5.

GsDOBJ2 is defined as follows:

```
struct GsDOBJ2{
        unsigned long attribute;
        GsCOORDINATE2 *coord2;
        unsigned long *tmd;
        unsigned long id;
}
```

coord2 is a pointer to the coordinate system. An object's location may be controlled by setting the members of the corresponding GsCOORDINATE2 structure.

attribute is for setting object attributes: general attributes such as display/non-display or special effects such as switching of the light source calculation method. See the explanation of GsDOBJ2 in the *Run-Time Library Reference* for details.

Object Initialization

Handling an object with GsDOBJ2 requires linking the handler with the read-in TMD data. To do this, you can use GsLinkObject4(), which sets the GsDOBJ2's *tmd* member to the address of the TMD object to be linked to it.

Object Movement (Hierarchical Structuring)

Different objects may be linked by defining a hierarchy in the coordinate system in which an object's member *coord2* is specified as subordinate to another object's GsCOORDINATE2.

In the following figure, Object 1 is defined as the *super* coordinate system of Object 2:

Table 10-1: Hierarchical Structuring

GSDOBJ2		COORDINATE2
Object1	$\longrightarrow$	Coordinate1
	$\longrightarrow$	↑
Object2		Coordinate2

When Object 1 moves, the location of object 2 is affected as well. When Object 2 moves, Object 1's location is not affected.

Object Attribute Control

The bits of the GsDOBJ2 *attribute* member control several properties of objects, such as material attenuation, lighting mode, near clipping, back clipping, and automatic division. See the description of GsDOBJ2 in the *Run-Time Library Reference* for more detailed information.

Viewing System

In 3D graphics, the image displayed on the two-dimensional screen is a projection of 3D space onto a "window" (viewing-plane), at a specified distance in front of the viewpoint. The viewpoint and "window" value must be set in order to project an image.

Viewpoint Setting

The viewpoint is set by substituting values in the GsRVIEW2 or GsVIEW2 structure members and calling GsSetRefView2() or GsSetView2().

The difference between GsRVIEW2 and GsVIEW2 is in the viewpoint setting method.

GsRVIEW2 sets the viewpoint by setting the coordinates of the viewpoint and a reference point. GsVIEW2 sets the viewpoint by directly setting a transformation matrix to the viewpoint coordinate system.

Both GsVIEW2 and GsRVIEW2 can set a coordinate system which becomes the standard in *super*. For example, if the standard coordinate system is treated as a world coordinate system, it becomes an objective viewing camera, and if the coordinate systems of each object are taken as local coordinate systems, it becomes a subjective viewing camera for that object.

Screen Setting

The distance between the viewpoint and the screen is called projection (h). Projection is set by the GsSetProjection() function.

The vertical and horizontal of the screen are equal to the current display resolution. For example, if the display resolution is 640/480, the horizontal of the screen is 640 and the vertical is 480.

When the display resolution is not normal dot (4 to 3 aspect ratio) the vertical is adjusted. For example, in the case of 640/240 dot, the vertical of an object is displayed by reducing it by 1/2. In appearance, this is the same as a normal dot aspect ratio.

To use the libgs three-dimensional service, it is necessary to execute the GsInit3D() function and initialize the screen coordinates. In this way, the center of the screen is the origin of the screen coordinates.

Figure 10-1: Viewpoint and Screen
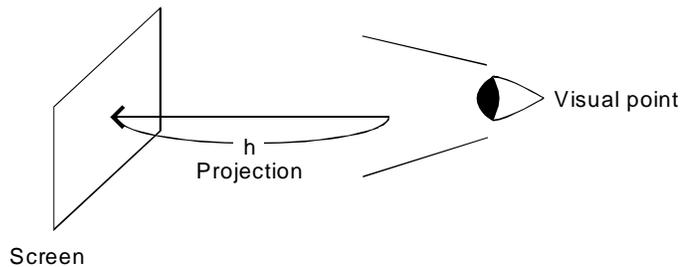
Projection adjusts the angle of an image. If projection is large, the image angle is narrow and is close to parallel projection. If projection is small, the image angle widens, and it becomes a picture in which the impression of perspective is emphasized.

Light Sources

Parallel Light

Libgs allows a maximum of 3 parallel light sources. With a parallel light source, the brightness of a polygon is determined only by the light source and the angle of the polygon. A light source is set by the direction of the light source and its color.

A light source is specified in the system by setting the members of a GsF_LIGHT structure and calling GsSetFlatLight().

Ambient Light

Ambient light is the surrounding light. Even though light does not directly strike it, the shape of an object may be seen with the surrounding light. Ambient light is created in order to achieve this type of phenomena.

For example, if the spot where the light strikes is 1 and spots where the light does not strike are 0.5, GsSetAmbient() is executed as follows (ONE expresses the fixed-point 1):

```
GsSetAmbient(ONE/2,ONE/2,ONE/2);
```

In general, the image becomes warm when the ambient light values are increased and cool when they are decreased.

Depth Cueing

When varying the brightness of an object according to the distance from the viewpoint, distant objects may be dimmed. This is called depth cueing or fog.

Depth cueing may be executed normally for all non-textured polygons. It is possible for texture-mapped polygons only when the color is black.

To use depth cueing, call GsSetLightMode(1) or GsSetLightMode(3) to set cueing on. Then call GsSetFogParam(), passing a GsFOGPARAM structure specifying background colors.

When the background colors are made whitish, the result is the fog effect. When they are made black, there is a "night-time" effect. This is effective in making something like a dungeon difficult to see by darkening the distance.

Depth cueing can be performed on texture-mapped polygons at any time by switching the length of the CLUT. This method is not currently supported in libgs.

Moreover, please be aware that background color and ambient color are generally quite different.

Material Lighting

The intensity of light is determined by the angles of the polygon and the light source. However, the feel of a material can be changed to metallic by making the light attenuation curve steeper. This is called material lighting.

Call GsSetLightMode(2) or GsSetLightMode(3) to execute material lighting.

Attenuation is controlled by setting the material attenuation bit of the GsDOBJ2 member *attribute*, object by object. The higher the value, the steeper the attenuation and the more the metallic feel increases.

However, this is not possible with the current version.

Drawing Priority Order (Ordering Table)

The PlayStation uses Z-sorting as a method of hidden-surface removal. To speed up the performance of Z-sorting, the concept of an ordering table (OT) has been introduced. Hereafter, Z refers to a coordinate value on the axis perpendicular to the view plane; in other words, the distance between the screen and the polygon.

An ordering table is a kind of Z ruler applied in memory. Each graduation of the ruler may hold any number of polygons.

Polygons are sorted by placing them at the graduation equivalent to their Z value. This means that if polygons are placed all the way up to the end, all the polygons will hang on the ruler according to their Z values. Hidden-surface removal is achieved by transmitting this to a rendering chip and drawing the polygons at the end of the OT (with the largest Z value) first.

GsOT and GsOT_TAG

Ordering tables are handled in libgs by the GsOT structure, which stores a pointer (member *org*) to an actual OT and parameters that indicate the attributes of that OT. The member *length* represents the Z graduation resolution as a power of 2 (from 1 to 14). For example, if *length* is 4, the OT has 2^4 graduations. Each graduation is represented by a GsOT_TAG structure.

OT Initialization

An OT is initialized by the function GsClearOt(). GsClearOt() takes 3 arguments, *offset*, *point* and *otp*. otp is a pointer to the OT handler. *offset* and *point* are explained below.

When an OT is initialized, the polygons are unlinked, and only then is a re-sort possible. Therefore, it is always necessary to initialize an OT prior to executing a sort.

Multiple OTs

libgs allows multiple Ots, which may be sorted by the GsSortOt() function. The GsOT member *point* refers to the representative value Z of an OT.

It is possible to control the sorting order by using multiple OTs. For example, if local OTs are prepared object by object, and finally collated by sorting the local OTs, sorting by object units is possible.

This is effective when the relationship between before and after is already known, in such cases as when a helicopter is looking down from above at cars which are being driven on a road.

Also, multiple OTs can also be used to achieve a "split-screen" effect. For example, by drawing one OT to the top half of the drawing area, switching the drawing environment and viewpoint settings, and drawing the second OT to the bottom half of the drawing area, two different views can be shown onscreen simultaneously.

OT Compression

Sort speed will be increased by using OTs. However, OTs consume a considerable amount of memory.

Memory consumption can be reduced by decreasing OT resolution. However, this can cause a polygon flicker phenomenon (Z-sort problem) due to errors in Z relationships.

Therefore, there is a method of using an offset to reduce OT memory usage without reducing resolution. If it is known that the Z values of the polygons to be sorted are greater than a certain value, this value can be passed to GsClearOt() as *offset*. The OT will not store the part up to *offset* in memory; therefore, memory consumption will be reduced.

Z-Sort Problem

When using Z-sort to perform hidden-surface removal, polygons can flicker because of errors in priority ordering. This phenomena is likely to occur with a polygon of particularly long depth, because its Z value varies greatly across the polygon, but it is sorted with only one average Z value (the center of gravity).

The Z-sort problem can be resolved by dividing polygons into smaller polygons; however, this has the drawbacks associated with an increased number of polygons.

Another countermeasure is to sort by object units. When the Z relationship is clear for every object, if this condition is reflected when sorting, sorting may be achieved without mutual interference of objects.

OT Double Buffer

An OT in which polygons are linked cannot be accessed while it is being drawn. For this reason, you must use a double buffer technique of preparing 2 OTs when drawing in the background. You sort the OT that is not being used for drawing.

Frame Double Buffer

The PlayStation has a two-dimensional frame buffer, and the image displayed in the window can be reproduced in video memory as is.

The screen can be switched without being disturbed during vertical synchronization (V Blank). If the switched screen is accessed during the time when the television screen is being displayed, the screen will become disturbed.

Due to this both the screen being displayed and the switched screen are necessary. This is called the display double buffer.

In libgs ,the double buffer is defined by GsDefDispBuff().GsSwapDispBuff() switches the buffers. GsGetActiveBuff() can be used to determine which double buffer is currently being drawn.

Double Buffer Expression

Double buffering may be achieved by altering the location of the display area in the frame buffer. The upper left point of the display area (starting point) does not necessarily have to be in the upper left point of the frame buffer.

Drawing that goes to the frame buffer must have an offset attached. You may choose from two methods of offsetting with libgs, determined by the third argument of GsInitGraph():

- Put the offset at the libgte level. The double buffer offset is added at the stage where the packet calculation is being made.
- Place the offset at the libgpu level. The offset is added at the stage where a frame buffer not attached to the packet is drawn.

If you are planning on using this in combination with libgpu functions, using the latter method, placing the offset at the libgpu level, is recommended. Using the former method, compatibility with other than previous versions cannot be assured.

Frame Double Buffer During Interlace

In interlace mode, you can specify a vertical resolution of 480. In this case, double-buffering is automatic between even- and odd-numbered scan lines. Therefore, you designate the same buffers as the GsDefDispBuff() arguments.

When vertical resolution is specified as 240 during interlace mode, it is necessary to set different buffers, as you do in non-interlace mode.

Clipping

Libgs supports the following kinds of clipping:

- Two-dimensional clipping is clipping after transforming the screen coordinate system.
- Three-dimensional clipping is clipping according to the distance from the viewpoint.

Two-dimensional Clipping

The GPU can designate any rectangle in the frame buffer as a clipping area.

The clipping area is registered in the libgs internal variable set by the GsSetClip2D() function. GsSetDrawBuffClip() sets the internal contents of the variable and makes them effective.

Also, when switching double buffers, switch the clipping area so that an overflowing polygon does not destroy another buffer.

Three-dimensional Clipping

Libgs supports default values for three-dimensional clipping. Any other clipping must be performed at the application level. The three-dimensional clipping default values are as follows:

- **FAR CLIP** - When the screen coordinate system Z value is greater than 65536, the Z value can be clipped (because the Z value is uncoded 16-bit).
- **NEAR CLIP** - When the screen coordinate system Z value is less than h/2, the Z value can be clipped (h is projection).

Near Clipping Problem

The near clipping problem occurs when polygons approach the viewpoint, such as in the road surface of a racing game, and become extremely large due to their nearness. When clipped by polygon units, large holes appear in the road surface close to the viewpoint, and this makes viewing difficult.

As a solution to this problem, libgs supports automatic division of polygons. When an approaching object reaches the near clipping plane, near clipping can be performed smoothly by the setting of the automatic division attribute. However, since the load from automatic division is heavy, use it only when absolutely necessary.

Packet Preparation Function

libgs has three kinds of packet creation functions, GsSortObject3(), GsSortObject4(), and GsSortObject5(). Each of these functions is an appropriate choice under different conditions.

Packet Buffer

There are two types of packet buffer:

- Preset packet buffer
- Run-time packet buffer

The Preset packet buffer (1) is essential when using the Preset packet buffer object. The object type which uses the preset packet is the GsSortObject5() function which uses GsDOBJ5. The size of the present packet is fixed by the model.

Using the GsPresetObject() return value it is possible to find out how far the buffer has been preset. Initially one preset is necessary.

Since the preset packet creates the packet in the preset buffer area, it does not use up the run-time packet buffer.

However, when automatic division is set to ON in the GsDOBJ5 attribute, the packet created does consume the run-time packet buffer.

The Run-time packet buffer (2) is the buffer used when a packet is created during execution.

GsSortObject4(), GsSortSprite, etc. use this buffer.

The head of the buffer is specified by GsSetWorkBase() and when GsSortObject4() is called, the packet is created in that area and the current packet area pointer is taken by GsGetWorkBase().

The amount of the buffer used per frame will increase or decrease depending on the number of polygons calculated.

Preset Packets

Preset packets are packets that have been made ahead of time. If preset packets are used, it is not necessary to rewrite every frame. Speed is improved by not having to perform tasks like writing U, V texture values to memory.

PMD format is an exclusive preset packet modeling format which incorporates both modeling data and preset packets. GsSortObject5() is the packet creation function for preset packets.

The packet is a collection of structures (primitives) such as libgpu POLY_FT4. The primitive class can be determined by looking at its type.

To set tpage, set the tpage of the packet structure tpage (if the packet is POLY_FT4, set the tpage of the structure POLY_FT4).

Figure 10-2: Preset Packet Format

header0: 32 bits. The upper eight bits indicate the preset packet type. The lower 24 bits are a pointer to the next packet group.

header1: Indicates the number of preset packets in the packet group. The packet group is composed of packets of the same type.

preset packets: The number of preset packets uses double the number of packets to provide a packet double buffer.

terminator: The address section is all zeros.

TMD Sort

TMD format modeling data allows the setting-up of random polygons. In realtime, when random polygon types appear that create packets from TMD data and which are subsequently converted, the decode routine is swapped out of the I cache and the processor is unable to keep up.

This is the reason for a TMD data high speed technique for ordering polygons. This technique is the TMD sort.

The GsSortObject4() or GsSortObject5() packet-creating functions are faster if they use sorted TMD data.

TMD sort is carried out at the authoring level. TMDSORT.EXE is the conversion command. See *Data Conversion Utilities* for details on using this command.

Packet Creating Functions

GsSortObject3()

GsSortObject3() creates PMD format packets. It uses the object handler GsDOBJ3. For GsDOBJ3 to handle PMD data, GsLinkObject3() must be called first to link the PMD data and the handler.

The PMD format combines the modeling data and preset packet.

The conversion of TMD to PMD takes place at the authoring level. TMD2PMD.EXE is the conversion tool. See *Data Conversion Utilities* for details.

GsSortObject4()

GsSortObject4() is the most generic object calculation routine. It uses sorted TMD format data for greater speed. The TMD data sort is carried out by the tmdsort.exe command. The object handler GsDOBJ2 is used.

For GsDOBJ4 to handle the TMD data, GsLinkObject4() must be called first to link the TMD data and the handler.

GsSortObject4() uses the preset local/screen matrix and the local/screen light matrix as a reference. The object is local screen converted, sorted and allocated to the OT.

The local/screen matrix is set by GsSetLsMatrix(). Local/screen light matrix setting is performed by GsSetLightMatrix().

The polygons allocated to OT are drawn by GsDrawOt(). This drawing function can return quickly, and drawing may be done in the background.

GsSortObject5()

GsSortObject5() is a packet creation function that uses preset packets. It uses sorted TMD format data to increase speed. GsSortObject5() uses the object handler GsDOBJ5. TMD data sort is carried out in the tmdsort.exe command. For GsDOBJ5 to handle the TMD data, GsLinkObject5() must be called first to link the TMD data and the handler.

GsSortObject5() uses GsPresetObject() to create preset packets. For GsSortObject5() to create a packet, GsPresetObject() must be initialized once and a preset packet created.

Packet Creation Function

The functionality of each packet creation function is shown below.

Table 10-2: Packet Creation Function Comparison Chart 1

A	B	C	D	E	F	G	H	I
GsSortObject3	GsDOBJ3	X	X	X	X	X	X	X	250K
GsSortObject4	GsDOBJ2	X	O	O	X	X	O	O	?
GsSortObject5	GsDOBJ5	X	O	O	X	X	X	O	220K

A. OBJTYPE - Object handler used
B. Material attenuation - (See attribute)
C. FOG - (See attribute)
D. Light source calculation off - (See attribute)
E. NearZ CLIP - (See attribute)
F. Back CLIP - (See attribute)
G. Semi-transparency rate - (See attribute)
H. Automatic division - (See attribute)
I. Efficiency - 10x10 (Real measurement value of a flat triangle)

GsSortObject4 is more efficient than GsSortObject3 and less efficient than GsSortObject5

Table 10-3: Packet Creation Function Comparison Chart 2

	Presort	Preset	Preshade	WorkBase	Tools
GsSortObject3	OK	OK	OK	NG	Tmd2pdm
GsSortObject4	OK	NG	OK/NG	OK	rsdlink, TMDSORT
GsSortObject5	OK	OK	OK/NG	NG(normal) OK(autodivision)	rsdlink, TMDSORT

Packet Area

GsSortObject4() creates the packet and allocates it to the ordering table.

The packet creation area is set by the GsSetWorkBase() function.

Packets increase and decrease depending on the type and number of polygons (flat/gouraud, with/without texture). Only a rough estimate can be made of how much area should be maintained. If the area of an actual packet is smaller than the packet created, it will destroy the area behind the packet area.

GsGetWorkBase() is a function to return the area currently available for use by a packet. A program may use this function to estimate the danger of overflow.

It is not necessary to use GsSetWorkBase() to maintain a new packet area when using GsSortObject5(), because the packet area for the preset packet area may be reserved.

You must define a packet area with GsSetWorkBase() when using automatic division, because a packet that has been divided and increased in size may use the packet area set aside by GsSetWorkBase().

Packet Double Buffer

Drawing is executed in the background, so the packets in a drawing cannot be destroyed. Consequently, it is necessary to prepare two packet areas to make a double buffer.

Drawing

Call the GsDrawOt() function to begin drawing. The drawing area is swapped each time GsSwapDispBuff() is called. Drawing occurs in the background so sufficient time must be allowed to complete the operation.

During the drawing process images from the previous two frames that remain in the drawing area are cleared. Call GsSortClear() to register the "screen-clear" Special Primitive to the OT before clearing the screen. The cleared screen color may be specified as an RGB value in the arguments to the function.

Processing Flowchart

A typical flowchart of 3D processing required for each frame is shown below. See the sample program for details.

Figure 10-3: Three-dimensional Processing Flowchart


```
┌─────────────────────────────────────┐
│      Select the buffer to use        │
└─────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────┐
│              Clear OT                │
└─────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────┐
│   Perspective transformation/Z sort  │
│       (record command to OT)         │
└─────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────┐
│           V-blanking sync            │
└─────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────┐
│          Start OT drawing            │
└─────────────────────────────────────┘
```
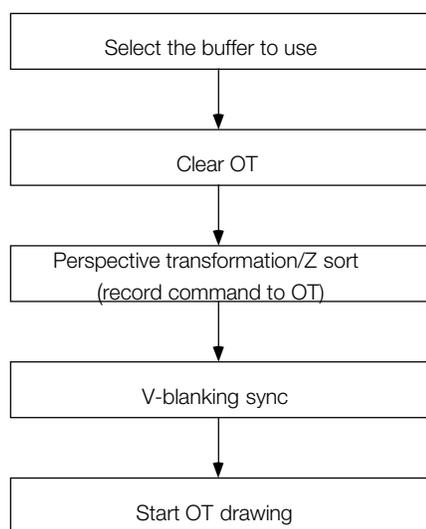
Jump Tables

Purpose

GsSortObject5(),GsSortObject4() dispatches attributes, pre-set data, etc. and calls low-level functions.

There are 64 low-level functions, and a single application is unlikely to use all of them.

You don't need to link GsSortObject5J() and GsSortObject4J() with unnecessary low-level functions, thereby making the code more compact.

In addition to decreasing code size, the GsSortObject…J functions allow the user to customize Libgs. Support for non-standard actions, such as material attenuation, reflection-mapping, etc can be added to the Libgs, by linking user defined functions in place of the library function.

These functions are compatible with GsSortObject5() and GsSortObject4(), which organize low-levell functions as tables.

GsFCALL is the structure in which the function table is defined. The function table is organized according to polygon type, whether or not division is performed, and the light-source calculation mode.

Usage

The relevant functions are linked by entering the pointers of the appropriate low-level functions in each of the elements. It is possible to avoid linking by not including the pointers and not making extern declarations.

However, if a function that does not have a pointer is called, a BUS ERROR will be generated. To avoid this, Libgte provides dummy (dmy…) functions. With these linked, if a call is made with an unanticipated type, the appropriate dummy function will print its name to standard out.

The abbreviated example below, shows the use of GsSortObject5() with appropriate functions in all the elements. In this example, GsSortObject5J() functions the same as GsSortObject5(). This example is included in the comments

In the file libgs.h.

```
/* extern and hook only necessary functions */

extern _GsFCALL GsFCALL5;          /* GsSortObject5J Func Table */
jt_init()                          /* Gs SortObject5J Hook Func */
{
PACKET *GsPrstF3NL(),*GsPrstF3LFG(),*GsPrstF3L(),*GsPrstNF3();
PACKET *GsTMDdivF3NL(),*GsTMDdivF3LFG(),*GsTMDdivF3L(),*GsTMDdivNF3();
PACKET *GsPrstG3NL(),*GsPrstG3LFG(),*GsPrstG3L(),*GsPrstNG3();
PACKET *GsTMDdivG3NL(),*GsTMDdivG3LFG(),*GsTMDdivG3L(),*GsTMDdivNG3();
PACKET *GsPrstTF3NL(),*GsPrstTF3LFG(),*GsPrstTF3L(),*GsPrstTNF3();
PACKET *GsTMDdivTF3NL(),*GsTMDdivTF3LFG(),*GsTMDdivTF3L(),*GsTMDdivTNF3();
PACKET *GsPrstTG3NL(),*GsPrstTG3LFG(),*GsPrstTG3L(),*GsPrstTNG3();
PACKET *GsTMDdivTG3NL(),*GsTMDdivTG3LFG(),*GsTMDdivTG3L(),*GsTMDdivTNG3();
PACKET *GsPrstF4NL(),*GsPrstF4LFG(),*GsPrstF4L(),*GsPrstNF4();
PACKET *GsTMDdivF4NL(),*GsTMDdivF4LFG(),*GsTMDdivF4L(),*GsTMDdivNF4();
PACKET *GsPrstG4NL(),*GsPrstG4LFG(),*GsPrstG4L(),*GsPrstNG4();
PACKET *GsTMDdivG4NL(),*GsTMDdivG4LFG(),*GsTMDdivG4L(),*GsTMDdivNG4();
PACKET *GsPrstTF4NL(),*GsPrstTF4LFG(),*GsPrstTF4L(),*GsPrstTNF4();
PACKET *GsTMDdivTF4NL(),*GsTMDdivTF4LFG(),*GsTMDdivTF4L(),*GsTMDdivTNF4();
PACKET *GsPrstTG4NL(),*GsPrstTG4LFG(),*GsPrstTG4L(),*GsPrstTNG4();
PACKET *GsTMDdivTG4NL(),*GsTMDdivTG4LFG(),*GsTMDdivTG4L(),*GsTMDdivTNG4();
PACKET *GsPrstF3GNL(),*GsPrstF3GLFG(),*GsPrstF3GL();
PACKET *GsPrst3GNL(),*GsPrstF3GLFG(),*GsPrstF3GL();
/* flat triangle */
. . .
. . .
}
```

Scratch Pad Usage Volume

In the Libgs the Scratch Pad address can be passed by argument to GsSortObject4, GsSortObject4J, GsSortObject5 and GsSortObject5J. The scratch pad, a feature of the CPU, allows "high speed access" to as much as 1k of data. It is used in polygon division to improve speed.

Scratch Pad Consumption Status

The scratch pad consumption condition uses the following functions and attributes:

Table 10-4: State of Scratch Pad Consumption

Item	Description
Function name	GsSortObject4()
	GsSortObject4J()
	GsSortObject5()
	GsSortObject5J()
attribute	GsDIV1, GsDIV2, GsDIV3,
	GsDIV4, GsDIV5

The scratch pad area is not used when automatic division is not carried out.

Scratch Pad Consumption Volume

The scratch pad consumption volume is as follows: (unit: byte)

Table 10-5: Scratch pad usage volume

	GsDIV1	GsDIV2	GsDIV3	GsDIV4	GsDIV5
Triangular Polygon	184	272	360	448	536
Rectangular Polygon	260	400	540	680	820

Method for Common Use of Scratch Pad by the User Program and Library

The scratch pad base address given by the GsSortObject...() argument is shifted lower and the higher is used in the user program. The scratch pad area used by the library is extended down in relation to the address.

Mip-map Library

Libgs supports mip-mapping, which means switching the texture of a textured rectangular polygon according to the polygon's size. Using mip-mapping, it is easier to hit the texture cache, and drawing time is shortened.

Usage Method

The GsSortObject4J() low-level functions which support mip-mapping are as follows:

Table 10-6: mip-map Low-level Function Group

Function Name	Polygon	Options
GsTMDfastTF4LM	Flat textured quadrangle	(light source calculation)
GsTMDfastTF4LFGM	Flat textured quadrangle	(light source calculation+FOG)
GsTMDfastTF4NLM	Flat textured quadrangle	(no light source calculation)
GsTMDfastTNF4M	Flat textured quadrangle	(no light source calculation)
GsTMDfastTG4LM	Gouraud textured quadrangle	(light source calculation)

Function Name	Polygon	Options
GsTMDfastTG4LFGM	Gouraud textured quadrangle	(light source calculation+FOG)
GsTMDfastTG4NLM	Gouraud textured quadrangle	(no light source calculation)
GsTMDfastTNG4M	Gouraud textured quadrangle	(no light source calculation)
GsTMDdivTF4LM	Flat textured quadrangle	(fixed division+ light source calculation)
GsTMDdivTF4LFGM	Flat textured quadrangle	(fixed division+light source calculation+FOG)
GsTMDdivTF4NLM	Flat textured quadrangle	(fixed division+no light source calculation)
GsTMDdivTNF4M	Gouraud textured quadrangle	(fixed division+no light source calculation)
GsTMDdivTG4LM	Gouraud textured quadrangle	(fixed division+light source calculation+FOG)
GsTMDdivTG4LFGM	Gouraud textured quadrangle	(fixed division+light source calculation+FOG)
GsTMDdivTG4NLM	Gouraud textured quadrangle	(fixed division+no light source calculation)
GsTMDdivTNG4M	Gouraud textured quadrangle	(fixed division+no light source calculation)
GsA4divTF4LM	Flat textured quadrangle	(automatic division+light source calculation)
GsA4divTF4LFGM	Flat textured quadrangle	(automatic division+light source calculation+FOG)
GsA4divTF4NLM	Flat textured quadrangle	(automatic division+no light source calculation)
GsA4divTNF4M	Flat textured quadrangle	(automatic division+no light source calculation)
GsA4divTG4LM	Gouraud textured quadrangle	(automatic division+light source calculation)
GsA4divTG4LFGM	Gouraud textured quadrangle	(automatic division+light source calculation+FOG)
GsA4divTNG4M	Gouraud textured quadrangle	(automatic division+no light source calculation)
GsA4divTNG4M	Gouraud textured quadrangle	(automatic division+no light source calculation)

Texture Location

When using mip-mapping, textures should be positioned in the frame buffer as follows:

Figure 10-4: Texture Location
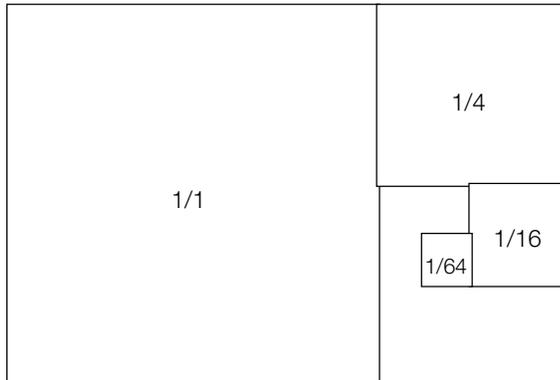
The texture size is in four stages: 1, 1/4. 1/16 and 1/64. The texture being used can be calculated by using the external product value. The above four textures must be within the same texture page.

Polygon Vertex

The polygon vertex order must be as follows:

Figure 10-5: Polygon Vertex Order

CONFIDENTIAL
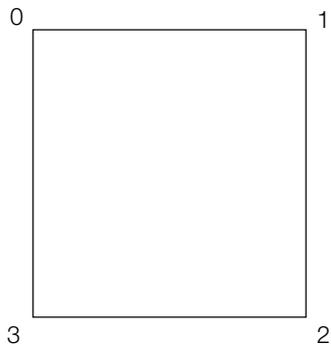
Chapter 11:
CD/Streaming Library

Table of Contents

Overview

The CD/Streaming Library (libcd) consists of two separate libraries:

- The CD-ROM Library, which provides functions for controlling the PlayStation built-in CD-ROM drive. It provides CD sound control and other services.
- The Streaming Library, which is a group of functions for continuous reading of realtime data such as movies, sounds or vertex data stored on high-capacity media. For an overview of the Streaming library, see the *Streaming Library Overview*, page 11-33.

Library and Header Files

Every program accessing CD-ROM and streaming services must link with the file `libcd.lib`.

Source code must include the header file `libcd.h`. When using the streaming library, `libds.h` may be included instead.

CD-ROM Sectors

Digital data is recorded on a CD-ROM in a spiral, the same as with a CD audio disk. This digital data is controlled by a processing unit called a sector. A digital data region lasting one second is divided into 75 sectors. Each sector is classified in one of the following sector types according to what it is used for.

Table 11-1: Sector Types

Sector type	Stored data
Audio sector	CD-DA audio data
ADPCM sector	ADPCM compressed audio sector
Data sector	User data sector

Audio Sectors

An audio sector records fs = 44.1 kHz digital stereo audio data (ordinary CD audio data). An audio sector may be played by the CdlPlay command and cannot be read as user data.

Data Sectors

User data is recorded on a data sector. A data sector's effective user area varies somewhat according to mode, but the standard is to use 2048 bytes (mode-1 format).

ADPCM Sectors

Strictly speaking, this indicates a sector called a realtime sector or mode-2 form-2 sector. ADPCM compressed audio data is stored here, and can be played as audio in the same way as an audio sector.

Interleave

On an ADPCM sector, ordinary audio data is recorded after being compressed by 1/4, relative to data on an audio sector. ADPCM sectors need to be arranged on a disk every four sectors in order that the CD-ROM may play ADPCM without having to seek each sector. This is known as interleaving. Interleaving ADPCM sectors makes it possible to record other data on the remaining sectors, and makes it possible to play audio while reading data.

When the disk is played at twice normal speed (double speed) the interleave separation must be every 8 sectors.

Addressing (Location Specification)

CD-ROM addressing (position setting) is done using track number, index number, minute, second, and sector for compatibility with CD audio. That is, the position of CD-ROM data can be established as a track number and index number when seen as audio data, or as a point which is x minutes x seconds x sectors from the header of the disk.

There are 75 sectors in one second and 60 seconds in one minute. The starting sector begins at 00 minutes 02 seconds 00 sector.

Tracks

On a disk, a track signal is recorded at the header of each track, and a position table for track signals is recorded at the header of the disk as the TOC (Table of Contents). The location for starting to play an audio sector is detected using the TOC and track signals.

Absolute Sectors

A data sector is addressed by minute/second/sector, but to make position calculation easy, there is also a method which sets it by counting the total number of sectors from the header (00 minutes 02 seconds 00 sector). This is called absolute sector setting. The absolute sector can easily be calculated from minute/second/sector by using the CdIntToPos() and CdPosToInt() functions.

File System

This is a method for getting the absolute value of a disk through the 9660 file system, besides specifying through low-level addressing. This method can only be used when the disk is recorded using the ISO-9660 file system format.

A CD-ROM is read-only, so the files on a disk can be arranged so that they all have continuous sector regions. Therefore a file can be read simply by specifying that file's start location, and something equivalent to an ordinary FAT(File Allocation Table) is not necessary. In the library, the function CdSearchFile(), which searches for a file's starting location, is used as an index of file names.

CONFIDENTIAL

Transfer Rate

A CD-ROM can rotate the disk at either normal speed or double speed. Normal speed has the same RPMs as an ordinary CD player, and double speed is twice as fast. The faster the disk rotation the faster the disk transfer speed.

CD-ROM transfer modes correspond to normal speed and double speed, and are 150KB/sec and 300KB/sec respectively. This means that in one second 75 sectors of data are read at normal speed and 150 sectors of data are read at double speed.

Sector Buffer

A CD-ROM's transfer speed is very slow compared to the host system's bus speed (132MB/sec), so the CD-ROM system has an internal local memory for one sector of data, called the sector buffer, and data from the CD-ROM is temporarily stored in the sector buffer before being collected and transferred. Data transfer from the CD-ROM follows the procedure shown below.

Figure 11-1: Process of CD-ROM Transfer

```
        ┌─────────────────┐
        │     CD-ROM      │
        └─────────────────┘
                 │  CdlReadN()
                 ▼
        ┌─────────────────┐
        │  Sector buffer  │
        └─────────────────┘
                 │  CdGetSector()
                 ▼
        ┌─────────────────┐
        │   Main memory   │
        └─────────────────┘
                 │
                 ▼
        ┌─────────────────┐
        │ Each device name│
        └─────────────────┘
```
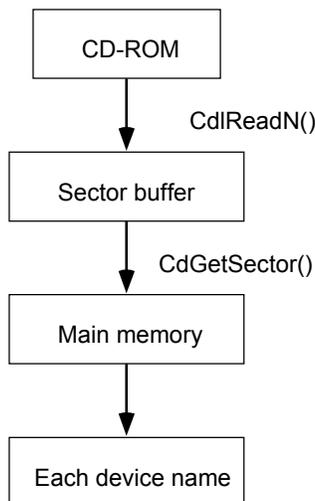
However, this is an example of a low-level interface. A high-level interface, such as CdRead() that can read data more easily is also provided.

Sound Control

The CD-ROM subsystem outputs two channels of audio signal: right (R) and left (L). Both CD audio and ADPCM audio are handled this way. Audio signals are sent to the SPU, then added to and synthesized with signals from an audio source inside the SPU and finally output as the composite sound. Four attenuators control the CD-ROM's audio output. Attenuator control is set through the CdMix() function using the CdlATV structure.

```
cd (L) --> ATV0 --> SPU (L)
cd (L) --> ATV1 --> SPU (R)
cd (R) --> ATV2 --> SPU (R)
cd (R) --> ATV3 --> SPU (L)
```

Primitive Commands (Low Level Interface)

The lowest level of operation for the CD-ROM is done by issuing direct commands to the CD-ROM subsystem.

The CdControl() function is used to issue each command and takes the following arguments.

```
CdControl(
        u_char  com,      /* command code */
        u_char  *param,  /* command argument set address */
        u_char  *result) /* command return value storage address */
```

For example, when playing a CD from 1 minute 00 seconds using CdControl(), a CdlPlay primitive command (code 0x03)is issued as follows:

```
#include <libcd.h>

CdlLOC  pos;
u_char result[8];

pos.minute = 0x01;  /* 1 min */
pos.second = 0x00;  /* 0 sec */
pos.sector = 0x00;  /* 0 sector (void) */
pos.track  = 0x00;  /* void */

CdControl(CdlPlay, &pos, result);
```

The details of `param` and `result` and the respective bit assignments are different for each command. Low level commands defined by CdControl() functions are called primitives. Primitive commands and their corresponding command codes are assigned as follows:

Table 11-2: Primitive Commands and Corresponding Codes

Symbol	Code	Type	Details
CdlNop	0x01	B	NOP (No Operation)
CdlSetloc	0x02	B	Set seek packet location
CdlPlay	0x03	B	CD-DA start play
CdlForward	0x04	B	Fast forward
CdlBackward	0x05	B	Rewind
CdlReadN	0x06	B	Data read start (with retry)
CdlStanby	0x07	N	Wait with disk rotating
CdlStop	0x08	N	Stop disk rotation
CdlPause	0x09	N	Temporarily stop at current location
CdlMute	0x0b	B	CD-DA mute
CdlDemute	0x0c	B	Release mute
CdlSetfilter	0x0d	B	Select play ADPCM sector
CdlSetmode	0x0e	B	Set basic mode
CdlGetlocL	0x10	B	Get logical location (data sector)
CdlGetlocP	0x11	B	Get physical location (audio sector)
CdlGetparam	0x0f	B	Get CD subsystem current mode
CdlSeekL	0x15	N	Logical seek (data sector seek)
CdlSeekP	0x16	N	Physical seek (audio sector seek)
CdlReadS	0x1b	B	Start data read (no retry)

B: Blocking; N: Non-Blocking operation

There are two types of primitive commands: blocking, which waits for processing to complete before return, and non-blocking, which returns without waiting for completion. When the commands are not queued , the next command is not issued, and after confirming that the previously issued command is complete, issuance will be blocked.

Command Arguments (Parameters)

A primitive command needs a list of arguments called parameters, as shown below. Command arguments are as follows:

Table 11-3: Primitive Command Arguments

Symbol	Parameter Type	Details
CdlSetloc	CdlLOC *	Start sector location
CdlReadN	CdlLOC *	Start sector location
CdlReadS	CdlLOC *	Start sector location
CdlPlay	CdlLOC *	Start sector location
CdlSetfilter	CdlFILTER *	Set play ADPCM sector
CdlSetmode	u_char *	Set basic mode
CdlGetTD	u_char *	Track no (BCD)

Commands other than these do not need arguments. NULL (0) is set in the argument pointer in commands that don't need arguments.

CdlLOC specifies the disk location, and has the following structure.

```
struct {
u_char minute;        /* sector location(min)*/
u_char second;        /* sector location(sec)*/
u_char sector;        /* sector location(sector)*/
u_char track; /* reserved */
} CDlLOC;
```

Minute/second/sector are given in BCD format. In BCD, each digit of a decimal number is assigned a 4-bit field. For example, decimal 60 is specified by a hexadecimal 0x60 notation.

The CdlFILTER structure is used to specify the multi-channel ADPCM play channel, and has the following structure.

```
struct {
u_char file;  /* play file ID */
u_char chan;  /* play channel ID */
unsigned short pad;
} CdlFILTER;
```

Command Return Value (Result)

After a primitive command is executed, an 8-byte value is always returned. The meaning of the return value varies according to the command, as shown below.

Table 11-4: Primitive Command Return Values

	Symbol Return Value and Stored Byte Position							
	0	1	2	3	4	5	6	7
CdlNop	Status							
CdlSetloc	Status							
CdlPlay	Status							
CdlForward	Status							
CdlBackward	Status							
CdlReadN	Status							
CdlStanby	Status							
CdlStop	Status							
CdlPause	Status							
CdlMute	Status							
CdlDemute	Status							
CdlSetfilter	Status							
CdlSetmode	Status							
CdlGetparam	Status	Mode						
CdlGetlocL	Min	Sec	Sector	Mode	File	Chan		
CdlGetlocP	Track	Index	Min	Sec	Frame	Amin	Asec	Aframe
CdlSeekL	Status	Btrack	Etrack					
CdlSeekP	Status	Min	Sec					
CdlReadS	Status							

The buffer region that stores the return value needs 8 bytes even when the command's return value status is only one byte.

Also, setting a the result parameter to NULL (0) suppresses the return value. In the following example, the function returns without setting CdlSeekL's return value.

```
CdlLOC  pos;
CdControl(CdlSeekL, &pos, 0);
```

Status Bit Assignments

The first byte of the result of almost all commands indicates CD-ROM status. The bit assignments of the status byte are as shown below. Use the command CdlNop if you wish to obtain the CD-ROM status only.

Table 11-5: Bit Assignments of Status Byte

Symbol	Code	Details
CdlStatPlay	0x80	1: CD-DA playing
CdlStatSeek	0x40	1: seeking
CdlStatRead	0x20	1: reading data sector
CdlStatShellOpen	0x10	1: shell open*
CdlStatSeekError	0x04	1: error during seeking/reading
CdlStatStandby	0x02	1: motor rotating
CdlStatError	0x01	1: command issue error

*This flag is cleared by the CdlNOP command. Therefore, in order to decide if the cover is currently open or not and before checking this flag, the CdlNOP command must be issued at least once.

Command Overview

This section gives a brief description of each command and explains the command.

CdlNop

Does nothing. Used for obtaining status.

CdlSetloc

Sets target position. This only sets the position; the actual operation is not performed. The target position set by this function is used prior to executing CdlPlay, CdlReadN, CdlReadS, CdlSeekP, or CdlSeekL.

CdlPlay

After the CD-ROM head seeks the target position, CD-DA play begins. Target position is set by argument. If the argument is set as NULL, the value set by the immediately preceding CdlSetloc or CdlSeekP is used.

CdlReadS/CdlReadN

After the CD-ROM head seeks the target position, the data sector contents are read and transferred to the local buffer. Target position is set by argument. If the argument is set as NULL, the value set by the immediately preceding CdlSetloc or CdlSeekL is used.

CdlReadS does not retry if an error occurs. This is used mainly for realtime reading such as streaming play, etc. CdlReadN can retry (max. 8 seconds) if a read error occurs. However, there is still the possibility of failure even with the retry.

CdlSeekL/CdlSeekP

After the CD-ROM head seeks the target position, it waits in pause status. Unlike a hard disk, a CD-ROM has a long seek time, so if the target address is known in advance, access can be sped up by moving the head to the target position in advance.

CdlSeekL does a logical seek of the data sector. The sector address has been recorded in the header of the data sector, so it is possible to perform an accurate seek. This operation is called a logical seek. A logical seek is only effective on a data sector.

On the other hand, a seek which uses a subcode (physical seek) is performed on an audio sector which does not have a sector header. A physical seek is imprecise but is effective on every type of sector.

The relationship between the two types of seek is shown in the table below.

Table 11-6: The Operation of CdlSeek/CdlSeekP

Command	Seek	Method precision	Sector used on
CdlSeekL	Logical	High	Anything but audio sectors
CdlSeekP	Physical	Low	All sectors

CdlForward/CdlBackward

Starts fast forwarding or rewinding an audio sector during play.

CdlStandby/CdlStop/CdlPause

CdlStandby waits with the spindle motor rotating.

CdlStop halts the spindle motor and returns the head to the home position. The next transition to seek or read or play can be done faster in standby status than in stop status.

CdlPause temporarily halts read or play, and waits at the head's position in standby status.

CdlMute/CdlDemute

This mutes (no sound) or releases the mute in CD-DA or ADPCM play.

CdlSetfilter

Sets play channel in multichannel ADPCM play. The channels which can be played are indexed by file number and channel number. The default file number and channel number is (1,1).

CdlSetmode

Sets the CD-ROM's basic operation mode.

Mode setting is done by taking the logical OR of the following bits and setting the result byte using the CdlSetmode command. The current mode can be obtained using CdlGetParam.

Table 11-7: Mode Settings of CdlSetmode

Symbol	Code	Details		
CdlModeStream	0x100	Normal streaming		
CdlModeStream2	0x120	SUB HEADER information includes		
CdlModeSpeed	0x80	Transfer speed	0: Normal speed	1: Double speed
CdlModeRT	0x40	ADPCM play	0: ADPCM OFF	1: ADPCM ON
CdlModeSize1	0x20	Sector size	0: 2048 byte	1: 2340byte
CdlModeSize0	0x10	Sector size	0: —	1: 2328byte
CdlModeSF	0x08	Subheader filter	0: Off	1: On
CdlModeRept	0x04	Report mode	0: Off	1: On
CdlModeAP	0x02	Autopause	0: Off	1: On
CdlModeDA	0x01	CD-DA play	0: CD-DA off	1: CD-DA on

CdlGetparam

Gets the CD subsystem current mode.

CdlGetlocL

Gets current position of the data sector being read or the ADPCM being played. The table below shows the meaning of the result code. CdlGetlocL does not work when an audio sector is playing.

Table 11-8: CdlGetlocL Parameters

Result byte no.	Details
0	Minute (BCD)
1	Second (BCD)
2	Sector (BCD)
3	Status
4	File number (see CdlSetfilter command)
5	Channel number (see CdlSetfilter command)

CONFIDENTIAL

CdlGetLocP

Gets the physical address of the sector being read or played. The table below shows the obtainable parameters. CdlGetlocP gets the subcode address, so it is effective on all sector types, including audio sectors.

Table 11-9: CdlGetlocP

Result byte no.	Details
0	Track number (BCD)
1	Index number (BCD)
2	Track relative minute (BCD)
3	Track relative second (BCD)
4	Track relative sector (BCD)
5	Absolute minute (BCD)
6	Absolute second (BCD)
7	Absolute sector (BCD)

Track relative minute/second/sector indicates an offset value from that track's header location. Absolute minute/second/sector specifies the location from the initial track.

CdlGetTN

Obtains number of TOC entries.

Table 11-10: CdlGetTN

Result	Contents
0	Status
1	Initial track No. (BCD)
2	Final track No. (BCD)

CdlGetTD

Obtains the TOC entries information (min, sec) corresponding to the track number specified in the parameters Please set the track No. in the BCD parameters.

Table 11-11: CdlGetTD

Result	Contents
0	Status
1	TOC min
2	TOC sec

However, when the track No. is set at 0:

min: total performance time (minutes)
sec: total performance time (seconds)

Command Synchronization

Primitive commands which take some time to process return without waiting for the actual completion of processing. These commands are called non-blocking (asynchronous) commands.

Commands which wait for the completion of processing before returning are called blocking (synchronous) commands.

Non-blocking commands continue processing in the background even after CdControl() returns. During this time, the program can continue processing in parallel.

The actual completion of non-blocking command processing uses CdSync() or the callback function (described later).

The return value (result) of CdControl() when a non-blocking command is actually issued is temporary, so it must be determined by the return value of the last status, the function CdSync(), or by an argument passed by the argument of a callback function.

To block all commands until complete, use CdControlB() instead of CdControl().

Command Execution Status

Primitive commands have the following processing status.

Table 11-12: Primitive Command Processing Status

Processing Status	Details
CdlNoIntr	Command being executed
CdlComplete	Execution complete, waiting
CdlDiskError	Error occurred

When a command is issued, the execution status changes from CdlComplete to CdlNoIntr. When a command ends normally and the next command can be received, the status shifts to CdlComplete. If an error is detected during execution, the status becomes CdlDiskError.

Blocking commands and non-blocking commands can be defined based on the processing status when the function returns.

A blocking command waits for CdlComplete/CdlDiskError status after a command is issued and then returns, but a non-blocking command returns CdlNoIntr as-is.

Getting Command Execution Status

The execution status of non-blocking commands are obtained from the return value of the function CdSync(). The format of CdSync() is as follows:

```
CdSync(
        u_char mode,          /* mode 0: blocking;  1:non-blocking  */
        u_char *result)       /* command's return value storage address */
```

It is possible to set blocking and non-blocking commands with CdSync() according to mode arguments. Accordingly, 1 and 2 below give the same result.

Table 11-13: CdSync() Mode Argument Values and Contents

Mode	Details
0	Do not return until execution status shifts to something other than CdlNoIntr
1	Return immediately regardless of the execution status

Example A:

```
CdControl(CdlSeekL, (u_char *)pos, 0);
CdSync(0, result);
```

Example B:

```
CdControl(CdlSeekL, (u_char *)pos, 0);
while (CdSync(1, result) == CdlNoIntr);
```

Furthermore, at the point when the execution status of the CdSync() return value (recall) is CdlComplete/CdlDiskError, it is fixed for the first time.

If the processing status is CdlNoIntr, the next command cannot be received. Command execution is not queued, so a new command waits until the previous command completes and the execution status becomes something other than CdlNoIntr. Therefore the following code produces the same result.

Example A:

```
CdControl(CdlSeekP, (u_char *)pos, 0);
CdControl(CdlPlay, 0, result);
```

Example B:

```
CdControl(CdlSeekP, (u_char *)pos, 0);
CdSync(0, 0);
CdControl(CdlPlay, 0, result);
```

In both examples a and b, the processing is blocked while seeking. This can be avoided by setting the direct location and issuing CdlPlay or by starting CdlPlay within a callback function.

```
/* Blocked During Seek */
CdControl(CdlSeekP, (u_char *)pos, 0);
CdControl(CdlPlay, 0, result);

/* Not Blocked During Seek */
CdControl(CdlPlay, (u_char *)pos, result);
```

Command Synchronization Callbacks

A callback function is a function that may be called when the command execution status shifts from CdlNoIntr to CdlComplete/CdlDiskError. Callback registration uses the CdSyncCallback() function. The following types of arguments are transferred in the callback function.

```
void callback(
        u_char intr,         /* execution status at that point in time */
        u_char *result)      /* newest return value at that point in time */
```

An example of using a callback is provided below.

Example: Execute CdlPlay if CdlSeek terminates

```
main() {
        void    callback();
        CdlLOC  pos;
        ....

        /* register callback function callback()  */
        CdSyncCallback(callback);
        ....

        /* issue command */
        CdControl(CdlSeekP, (u_char *)&pos, 0);
}
```

```
/* the following function is called when the command ends */
void callback(u_char intr, u_char *result) {
        if (intr == CdlComplete)
                CdControl(CdlPlay, 0, 0);
}
```

CdControlF Interface

CdControl() is blocked until a report that the command has been issued is sent to the subsystem. Since this blocked time is short when compared with the command execution time, it can usually be ignored. However, depending on the application, it is possible that you may want to run the program without having this time blocked. CdControlF() does not wait for command notification, it returns immediately after the command has been issued. For this reason, it cannot be easily determined if the command has been received or not. CdSync() must be issued and error processing must be done in polling.

Data Read

A CD-ROM is very slow compared to the transfer speed of the main bus. This is true even in double speed mode when data the transfer rate is 300KB/sec. Consequently, the CD-ROM has an internal sector data buffer, which merges and buffers the data from each sector.

When a data sector read command (CdlReadN/CdlReadS) is issued, the CD-ROM subsystem reads the sector data and temporarily places the data in the sector buffer. The contents of the data in the sector buffer are valid until overwritten by the next sector's data. Once data is valid in the sector buffer, it can be transferred to main memory at high speed using the CdGetSector() function.

Retry Read and No-Retry Read

There are two types of data reading. One type retries at the sector unit if an error occurs during reading (CdlReadN), and one type merely reports the error and does not retry (CdlReadS).

Reading data using CdlReadN ensures that the read data is correct, because it retries when an error occurs. Retrying means that the sector is read again, so this operation cannot be used at the same time when playing ADPCM. Nor is it appropriate when you want to maintain a fixed transfer rate for data quality, as in streaming. In this case, CdReadS is used; it does not retry, even if errors occur.

Table 11-14: Retry Read/No-Retry Read

Read command	Retry	Error Detection
CdlReadN	Yes	Yes
CdlReadS	No	Yes

Sector Ready Synchronization

The CdReady() function detects whether or not data is ready in the sector buffer. CdReady() function format is as follows:

```
CdReady(
        u_char mode,        /* Mode 0: blocking; 1:non-blocking  */
        u_char *result)     /* Most recent command return value */
```

The CdReady() function returns the following sector buffer status.

CONFIDENTIAL

Table 11-15: Sector Buffer Status

Processing Status	Details
CdlNoIntr	Being prepared
CdlDataReady	Data preparation complete
CdlDiskError	Error occurred

When data in the sector buffer is valid, the status shifts from CdlNoIntr to CdlDataReady/CdlDiskError. If 0 is set in the CdReady() mode argument, processing is blocked until the status shifts from CdlNoIntr. Also, when the CdReady() function returns CdDataReady/CdDiskError, the status returns to CdNoIntr.

Note that the CdReady() function reports the sector buffer status, so please be aware that it uses a lower-level interface than the CdReadSync() function. CdReadSync() reports completion of CdRead(), and is described later.

Data Ready Synchronous Callback

As with CdSyncCallback(), you may register a call back function when the sector buffer status shifts from CdlNoIntr to CdlDataReady/CdlDiskError. Callback registration uses the CdReadyCallback() function.

The callback function registered with CdReadyCallback() starts when 1 sector of data is ready. Please note that the specifications for this differ from CdReadCallback(). CdReadCallback() is described later.

Sector Buffer Transfer

A sector buffer is constantly overwritten with new sector data. Therefore sector data needs to be transferred to main memory before being overwritten. The CdGetSector() function is used to transfer sector buffer data to main memory. In the case when sector buffer data is transferred to a direct frame buffer or sound buffer, it is transferred to main memory once before it is re-transferred to each device.

The size of the sector buffer is 1 sector. Sector size varies according to CD-ROM mode, but 2KB is usually used. In this case, the upper limit of the size of data size which can be transferred to main memory by one CdGetSector() function is 2KB. Data can be transferred to different locations a number of times, but in these cases, the total size of the transferred data must equal the sector size as well.

An example of reading n sectors of data from a CD-ROM follows. This example performs the transfer in the foreground, but it is possible to do the transfer in the background using CdReadyCallback().

```
cd_read(
        CdlLOC *loc,        /* target position */
        unsigned long *buf, /* read buffer  */
        int nsec)           /* number of sectors */
{
        u_char  param[4];
        /* set double speed mode */
        param[0] = CdlModeSpeed;
        CdControl(CdlSetmode, param, 0);
        /* issue retry command */
        CdControl(CdlReadN, (u_char *)loc, 0);
        /* transfer to main memory as soon data is ready */
        while (nsec--) {
                if (CdReady(0, 0) != CdlDataReady)
                        return(-1);
                CdGetSector(buf, 2048/4);
                buf += 2048/4;
        }
}
```

Sector Transfer Synchronization

Data transfer from sector buffer to main memory is done in CdGetSector.

Since CdGetSector is a blocking function, the transfer of data is complete when it returns from the function. Therefore, there is no need to monitor the completion of the data transfer asynchronously.

High-Level Interface

Data Read

Data on a CD-ROM can be read by combining the CdlReadN primitive command and the CdGetSector() function, but the library also has the function CdRead(), which combines these and expands multiple sectors in main memory.

```
CdRead(
        int sectors,        /* number of sectors read */
        u_long *buf,        /* main memory address */
        u_char mode)        /* read mode */
```

CdRead() uses CdReadyCallback() internally. So this callback cannot be used when using the CdRead() function.

Data Read Synchronization

The CdRead() function works as a non-blocking function. The actual completion of CdRead() uses the CdReadSync() function. When the CdReadSync() function operates in non-blocking mode, it returns the number of unread sectors remaining.

The following example is a block-type CD-ROM read function.

```
int CdReadB(
        CdlLOC *loc, /* target position */
        u_long *buf, /* memory address */
        int nsector) /* number of sectors read */
{
        int cnt;
        u_char param[4];
        /* set double speed mode */
        param[0] = CdlModeSpeed;
        CdControl(CdlSetmode, param, 0);
        /* set target position */
        CdControl(CdlSetloc, (u_char *)&loc, 0);
        /* start read */
        CdRead(nsector, buf, mode);
        /* monitor number of sectors remaining until read ends */
        while ((cnt = CdReadSync(1, 0)) > 0 );

        return(cnt);
}
```

ADPCM

ADPCM (Adaptive Differential PCM) compresses audio data encoded as 16-bit straight PCM by 1/4. A sector storing ADPCM data is called an ADPCM sector. In order to play an audio series, ADPCM sectors are recorded on the disk at every fourth sector for normal speed playing and at every eighth sector for double speed playing. (This kind of processing is called interleave)

Double speed ADPCM sector interleave is as shown below.

Figure 11-2: ADPCM Sector Interleave

```
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  .  .
A  D  D  D  D  D  D  D  A  D  D  D  D  D  D  D  .  .
```
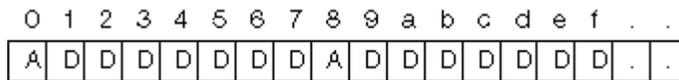
A: ADPCM sector
D: Data sector

Interleaving makes it possible to read data while playing ADPCM.

Multichannel

ADPCM sectors for another ADPCM channel can be interleaved with ADPCM data sectors. The figure below shows an example of an array.

Figure 11-3: Example Multichannel Interleave

```
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  .  .
A  A  A  A  A  A  A  A  A  A  A  A  A  A  A  A  .  .
0  1  2  3  4  5  6  7  0  1  2  3  4  5  6  7
```

An: n-channel ADPCM data

This example shows 8 channels of ADPCM sectors (A0-A7) interleaved and recorded on a disk. In this case, it is possible to switch between 8 channels of audio play without having to seek on the disk.

When playing this sort of multi-channel ADPCM tracks, the CdlSetFilter command is used to decide which channel to play. ADPCM tracks are confirmed by the CdlFILTER structure file members and channel members.

In order to make the CdSetFilter command effective, CdlModeSF must be set by the mode setting command.

Position-Confirmation Utility

Direct addressing of a CD-ROM is done by setting the minute, second, and sector in the CdlLOC structure and issuing the corresponding primitive command. The absolute position of each track and file on the CD-ROM was determined in advance before the disk was created, so basically it isn't necessary to dynamically search for a track or file's header position within the application.

However, for program development and debugging, a libcd utility is provided to dynamically search for the target track or files header position when executing.

TOC Read

As a CD player function, a CD-ROM is given a track index at the head of audio sectors and data sectors when the disk is created. The track index is recorded in the disk's TOC region, and is obtained using the CdGetToc() function.

TOC addressing is required basically to confirm an audio track play location. Therefore it has only second resolution and is not precise.

Directory Read

If a disk is recorded in the ISO-9660 file system format, the disk's absolute value can be obtained using the ISO-9660 format. Addressing using the ISO-9660 format provides more accurate locations than TOC addressing, but the ISO-9660 file system needs to be installed and cannot be used in audio sectors.

The CdSearchFile() function is used in searching for file header locations using the ISO-9660 format. CdSearchFile() searches for the file header location using the file's absolute path. The search result is stored in the structure CdlFILE.

An example of reading a 9660 file from a disk is shown below.

```
CdlFILE fp;

CdSearchFile(&fp0, "\\PSX\\SAMPLE\\RCUBE.TIM) == 0)
CdControl(CdlSetloc, (u_char *)&fp.pos, 0);
CdRead((fp.size+2047)/2048, sectbuf, CdlModeSpee);
```

CdSearchFile () returns the following CdlFILE structure members.

```
typedef struct {
        CdlLOC pos;          /* file position */
        u_long size;         /* file size */
        char name[16];       /* file name(body) */
} CdlFILE;
```

Report Mode

This function periodically reports the play position when an audio sector is being played. This is called report mode. If the CdlModeRept bit is set in this mode, the status shifts to CdDataReady status 10 times during each second of CD audio play, and the report result is returned as the return value (result). The following information is stored in the return value.

Table 11-16: Information Obtained in Report Mode

0	1	2	3	4	5	6	7
Status	Track	Index	Amin	Asec	Aframe	LevelH	LevelL

Obtaining a report is done by reporting with the CdReady() function or by using CdReadyCallback() in the background.

CONFIDENTIAL

Event Services

At initialization, a default callback function is registered for each callback. These distribute the events shown below.

Table 11-17: Event Services

Cause	Descriptor details	Event type
HwCdRom	Processing complete	EvSpCOMP
HwCdRom	Data ready	EvSpDR
HwCdRom	Data end	EvSpDE
HwCdRom	Error occurred	EvSpERROR

Therefore command completion or data read completion can be detected via the event handler. However, at the moment that a new callback is set, the default callback is released, and event transmission halts. Restoring the released default callback is left to the application. Here is an example:

Example: Callback Setting and Restoration

```
void   (*old_callback)();
        :
/* recover previous callback pointer when setting callback */
old_callback = CdSyncCallback(local_callback);
        :
/* restore callback  */
CdSyncCallback(old_callback);
```

Callback, Synchronous Function Overview

Table 11-18: Callback, Synchronous Functions

Called function	Sync detect	Callback	Details
CdControl	CdSync	CdSyncCallback	Issue command
CdControl	CdReady	CdReadyCallback	Sector read
CdRead	CdReadSync	CdReadCallback	Multiple sector read

Special CD-ROM Notes

Notes on Disc Access

A CD-ROM has to meet the CD-ROM XA specifications for playback to occur. Specifically, the CD-ROM's data tracks must be positioned before the DA tracks. (The DA tracks are optional.) For example, it would be incorrect if CD track 1 were a data track, tracks 2 and 3 were CD-DA tracks, and track 4 were a data track. The tracks should be arranged so that tracks 1 and 4 are located together at the beginning as track 1, then track 2 and the following tracks should be used for CD-DA data.

The auto pause function may not work properly if a disc has no gap between tracks or if the gaps are very short. In this case, the disc may continue playing to the end. In order to prevent this from happening, the gap between tracks must be at least two seconds long. As an example, to repeat one track as background music for a game, there must either be a gap of two seconds or more with auto pause on, or the current

position must be continuously polled so that when the end of the track is reached, the track will be replayed from the beginning.

If there is a track jump within three minutes from the outer edge of the disc, it is possible for the head to fly off the disc. In order to prevent this from happening, the tracks within three minutes from the outer edge should not be accessed. Generally speaking, the outer three minutes of the disc should be burned with NULLs. However, NULLs do not have to be recorded as long as the outer three minutes of the disc are not accessed. For example, an ending movie of three minutes or more could be recorded in place of NULLs. As long as the ending movie is always played from the beginning, there will not be any access to the outer three minutes. The mute off function will not work when a CD-DA track is played back immediately after a data track. If this type of operation is desired, a mute off should be performed when the CD-DA track is reached.

If report mode is left on during a data read, the pick-up position interrupt and the interrupt for starting data transfers will be indistinguishable. Report mode should be turned on only when a CD-DA track is being played. The following rules apply to the playing time sent when report mode is on. The absolute time from the start of the disc and the relative time indicating the time elapsed within the track are sent one after the other. In order to indicate whether the transmitted data is for absolute time or relative time, a '1' is set in the highest bit of sector data. In report mode, the timing for sending reports is as follows.

The data read during ff, fr is limited, so everything that has been checked is sent. If the tens' column for the absolute time is an even number, the absolute time is sent. If the tens' column for the absolute time is an odd number, the relative time is sent. In this case the highest bit of the frame byte is set to '1'. Since frames only run from 0 to 74, this bit can be set without any difficulty. Generally speaking, position data can be read during normal playback. However, this data is also sent when the tens' column changes. The relationship between the absolute time and the relative time is as described above. Levels are also sent, which make up 15 bits out of the two bytes of data. The remaining one bit is used to indicate the L/R channel.

The audio output may be different between cases when the CD-DA is accessed continuously and when TOC data is retrieved and the data is accessed in absolute time. This is due to the fact that there is an allowance for a lag between the data written in the TOC and the actual position. When data is accessed continuously, the access destination is automatically calculated to the header where the index is 1. Thus, the gap isn't played back.

The reset command performs the operations described below when the mode is set from the host and the CD is paused at the beginning. The reset command can be used as often as necessary, but after a reset is issued, the speed will be set to the standard setting.

Thus, if data were read at double speed, the disc speed would take some time to become stable since there would be repeated transitions between standard and double speed settings. This can be avoided by setting the desired mode (either by overwriting the mode or by looking at the current mode and correcting it). This will allow faster data reads, as it will eliminate the time spent waiting for the disc to reach a stable speed.

- Mode after resetting
- Drive is in standard speed setting
- Real time
- AD-PCM: off
- Number of bytes in data transfer: 2340
- Subhead filter: off
- Report mode: off
- Auto-pause: off
- CD-DA playback in CD-ROM mode: disable

CONFIDENTIAL

- Clear position set by setloc command.
- Clear previous error status.

An error will be returned after a prescribed time if the disc is in bad condition and cannot be accessed. Please note that there is a tendency to forget about error handling for this event since this problem generally does not occur.

The following types of problems may also occur. It is possible for a user to be waiting for multiple sector reads when the data happens to be difficult to read. In this case, some data would be read and an error would occur. Then some more data would be read and another error would occur. Because of the large number of retries, the time spent reading data would be much longer than expected and it would appear as though the system was hung up.

Generally, FF and FR commands cannot be performed when data is being read. If these commands are used in an environment such as a movie, some sort of workaround is needed for the user interface.

The "setloc, seekL, read" sequence can be used to read data, but it is also possible to use "setloc, read" as well.

If the following commands come after a setloc, the location data that had been saved will be overwritten.

```
play(playN),readN,readS,seekL,seekP,ff,fr,stop,reset,allreset
```

Also, the operation will be cancelled when the cover is opened. The following cannot be used: performing a double read by specifying a position (with setloc), reading (readN or readS), then issuing another readN or readS again. In this case, the operation of specifying a position (with setloc) and then reading (readN or readS) must be repeated twice.

The CD-ROM decoder is equipped with 32Kbytes of local memory, but the user cannot use all areas of this memory. Since the control software for the decoder does not support read-ahead in local memory, a data read should start within 6.6 msec for double speed and 13.3 msec for standard speed after a data ready interrupt. Otherwise, the data sent to the host may be updated and some data might be skipped. Ideally two FIFO blocks should be used, with each block having a length of 2340 bytes. When one block is filled, a switch will be made to the other FIFO.

There is some variation in access time even when the same interval is measured, and there is some variation among individual machines. This should be taken into consideration so that read-ahead is performed to absorb the variations.

If, while playing background music, multiple accesses need to be performed and switching time is required, it may be efficient to use CD-ROM XA's multi-channel AD-PCM. Quick switching is not possible for CD-DA since access is needed. Depending on the settings, it would also be possible to read data while playing music.

The Outer Three Minutes Problem

In the current CD-ROM subsystem, seeking within three minutes of the outer edge of the CD-ROM may not produce the correct results depending on the starting position of the seek. The problem may be prevented in the following manner.

- Record dummy data on the outer three minutes (the last three minutes of data). Do not use the dummy data.
- When using CD-DA for background music, make sure that the last track is three minutes or longer. Then there would be no seeks to the outer three minutes as long as the track is not played from the middle and the track is not repeated midway. This will allow the CD-ROM subsystem to operate properly.
- If the outer three minutes have to be used as a data area, access the outer three minutes or more as a single continuous file (e.g., use the area for an opening or ending movie).

CONFIDENTIAL

Notes on Using Low Level Function Groups

Error handling and callbacks are needed when performing read accesses on a CD-ROM using a combination of the low-level functions for CdControl(). In these cases, please take note of the following points:

Skipped Sectors

In double-speed mode, data is read from a CD-ROM at 150 sectors/sec. Therefore, one sector will be skipped if the host system does not finish processing the read operation for the previous sector within 1/150 sec. This problem tends to occur especially when callbacks are used as they take a long time to process. Therefore, for places where sector skipping is a possibility, CdlModeSize1 should be called from the application to read the sector header so that continuity of the sectors can be confirmed. The CdlSetmode command should also be used beforehand to set CdlModeSize1 (the mode for reading the sector header).

```
param[0] = CdlModeSpeed|CdlModeSize1;
CdControlB(CdlSetmode, param, 0);
```

Then, when using CdGetSector() to read data, the first 12 bytes (3 words) should be read. This contains the sector address in CdlLOC format. Skipped sectors can be avoided by checking to see if there is continuity with the previously read sector address.

```
......
CdGetSector(buf, 3);
if (CdPosToInt((CdlLOC *)buf) != prev_pos+1)
        return(-1);
else
        prev_pos++;

CdGetSector(bufp, 512);
bufp += 512;
......
```

Analysis of Callbacks

Whether or not sector data is ready can generally be determined by the callbacks in the CdReady() or CdReadyCallback() functions. Please note that unlike other callbacks, the libcd callback uses two parameters.

```
CdReadyCallback(callback);
....

void callback(u_char intr, u_char *result)
{
....
}
```

Note that in this example, a call is made even if the read operation fails. The intr parameter can be used to determine if the callback operation was successful or not. Read errors will not be properly caught if this parameter is not checked. Please refer to the cd/tuto sample programs for details on how to do this. In the result buffer, the return value of the last command issued is saved in an 8-byte array and the actual result array (8 bytes) is saved. The data saved in the result buffer depends on the command that was issued.

Deleting Callbacks

When a callback completes it should be cleared quickly.

```
CdReadyCallback(callback);
/* Operation corresponding to CdRead() */
CdReadyCallback(0);
```

CONFIDENTIAL

In this example, if the clearing of the final callback is omitted, a CdlDataReady event could be generated later due to other factors. This can result in a function callback() being activated at an unexpected time. In cases where the function callback() rewrites main memory, data could be destroyed unpredictably resulting in a bug.

Caution should also be exercised when a CdControl() is issued from a callback which has been set up by CdSyncCallback().

```
CdSyncCallback(callback);
...
void callback(u_char intr, u_char *result) {
        ....
        CdControl(CdlSeekL, ....);
        ...
}
```

In this example, a callback is activated after the completion of the CdlSeekL issued from within the callback(). Depending on the way the code is written, this could result in a recursive call to CdlSeekL, leading to an endless loop.

Watch Dog

At the same time that error handling is included to handle individual errors locally, time-out procedures and monitoring procedures should be included that periodically check (i.e. every few Vsyncs) the state of the CD-ROM subsystem to handle unavoidable errors. This kind of "watch dog" operation allows the system to return to normal operating mode after a fixed interval regardless of the cause of the error.

Playing Back CD-DA/CD-XA

Playback of CD-DA/CD-XA can be halted by a seek error or by inappropriately opening the cover. The status of the CD-ROM can be polled by periodically issuing the CdlNop command. The status of the subsystem is stored in the first byte of the result buffer for CdlNop. If the CdlStatPlay bit in this byte is not on, the appropriate track should be played back again.

Since logical accesses with CdlSeekL and CdlGetlocL retrieve the position by reading the CD-ROM sector header, these commands cannot be used for CD-DA tracks. Logical access can be performed for CD-XA tracks, but this operation will fail if a seek is being performed. In particular, if a CdlGetlocL is issued, it is necessary to check to see if a read (playback) is being performed.

```
VSyncCallback(vcallback);
...

static CdlLOC pos;
vcallback(void)
{
        int ret;

        /* if normal, polling */
        if ((ret = CdReady(1, result)) == CdlDataReady) {
                if (CdLastCom() == CdlGetlocL)
                    pos = *(CdlLOC *)result;
                CdControlF(CdlGetlocL, 0);
        }

        /* if error, retry */
        else if (ret == CdlDiskError)
                CdControlF(CdlReadS, (u_char *)&pos);
}
```

In this example, the "watch dog" function may not operate properly. This is because CdlGetlocL may be performed while a seek is taking place, resulting in a CdlDiskError. Thus, CdlSeekL and CdlGetlocL would be repeated indefinitely. The first three bytes of the result buffer for CdlGetlocL provide the sector position in CdlLOC format.

When a Data Read is in Progress

It is possible for a CdlDataReady event to be interrupted in the middle of a CD-ROM read for the same reason as when an audio track is being played. This condition can be reliably detected by saving the time stamp for when CdlDataReady was issued last and restarting all read operations if the time stamp has not been updated for a fixed period of time (on the order of a few seconds).

```
void callback(u_char intr, u_char *result)
{
        .....
        called_time = VSync(-1);
        ....
}

main()
{
        .....
        CdReadyCallback(callback);
        ....
        while (1) {
        ....
                if (VSync(-1) > called_time + TIME_OUT)
                        break;
        }
}
```

For sections where an endless loop waiting for a CdlDataReady may occur, there should be a way to exit the loop after a fixed time period has elapsed.

Return Value for CdReadSync

When CdReadSync() is issued in non-block mode, the number of remaining unread sectors is returned. Note that CdRead() performs a retry internally if a read error occurs, so the return value may not always decrease consistently.

Error Correction in CdRead

Starting with ver 3.5, CdRead() internally checks the continuity of sector headers to prevent skipping sectors during reads. Thus, a sector will not be read if the sector header information is incorrectly recorded. If there are an extremely large number of errors in CdRead(), the recording format of the disc should be checked.

High-Level Functions

High-level functions which perform a number of operations together are provided for some specific functions. High-level functions should be used if speed is not an issue. Please refer to the "Function Reference" for details.

- CdReadFile

    Reads a file from the CD-ROM

    — Format

        int CdReadFile(char *file, u_long *addr, int nbyte)

— Parameters

file file name
addr destination main memory address
nbyte size of data to be read

- CdReadExec

    Load executable file from CD-ROM

    — Format

    struct EXEC *CdReadExec(char *file)

    — Parameter

    file executable file name

- CdPlay

    Plays back CD-DA track

    — Format

    int CdPlay(int mode, int *tracks, int offset)

    — Parameters

    mode playback mode
    tracks array indicating the tracks to be played back
    offset index of tracks to begin playing

Operations Required for Swapping CDs

For titles that require swapping CDs without resetting the main unit during the game, the following operations should always be performed to prevent problems when the program reaches the market.

Operations to be Performed Before Swapping CDs

Required: Before swapping CDs (before outputting the "Replace CD" message), the CD subsystem should be set to standard speed mode.

Optional: After setting standard speed mode, use CdlStop to stop rotation of the CD.

Sample code for setting standard speed mode is shown below.

```
com = 0;
CdControlB( CdlSetmode, &com, result );
```

Detecting a Swapped CD

To see whether the CD has been replaced, the following two tests should be performed: (1) determine whether the cover has been opened; and (2) determine the spindle rotation. Either test can be performed using the CdlNop command.

```
CdControlB( CdlNop, 0, result );  /* char result[ 8 ]; */
```

1. The opening and closing of the cover is reflected in the CdlStatShellOpen bit of result[0]. The CdlStatShellOpen bit detects an open cover, and has the following settings:

    Cover is open: always 1

    Cover is closed: 1, the first time this condition is detected, 0 for subsequent times

    Thus, if this bit makes a transition from 1 to 0, it can be assumed that the CD has been swapped.

2. Use the CdlNop command and wait for bit 1 of result [0] (0x02) to change to 1.

Operations to be Performed Immediately after Swapping a CD

When the CD has been replaced and the cover has been closed, the CD subsystem begins reading the TOC data. While this operation is being performed, commands other than CdlNop and CdlGetTN should

CONFIDENTIAL

not be issued. The CdlGetTN command is used to determine when the TOC read operation has completed. If this command executes successfully, the reading of TOC data will be finished and commands can execute normally. The CdlGetTN command should be issued repeatedly until it is successful.

```
CdControlB( CdlGetTN, 0, result );   /* char result[ 8 ]; */
```

Checking for PlayStation Disc

The logical access command CdlReadS/N should be issued to check to see that the mounted CD is a PlayStation disc (black disc).

A command error is generated when a logical access is performed on a CD not recognized as a PlayStation disc. Unlike the standard CdlDiskError, the command error generates a CdlDiskError while also setting

> bit 0 of result [0] (0x01)
>
> bit 6 of result [1] (0x40)

to 1.

If a command error has been detected, it will not be possible to perform a logical access. This can occur if the wrong CD is mounted (such as a CD-DA) or if the CD has not been properly mounted. The only way to recover from a command error is to open the cover and remount the CD, so a message indicating this should be output, and the operation should be reissued.

When a game involves a logical access, e.g. loading data, immediately after a CD swap, the command can also check to see that the mounted disc is a PlayStation disc. If there is no logical access command (such as when a DA track is to be played back), there should always be a dummy read to check the disc.

If the mounted disc is a standard CD-ROM such as a CD-DA disc, the operations up to and including step (3) will execute normally. Therefore, discs should always be checked to see that they are PlayStation discs. The debugging station will recognize CD-Rs as well as standard CD-ROMs as PlayStation discs, but the PlayStation will only recognize black discs as PlayStation discs. When using CdGetDiskType() to confirm that the disk is a PlayStation disk, be aware that the operation mode set by CdlSetmode becomes CdlModeSpeed after CdGetDiskType() is executed.

Other

- Steps (1) - (3) must always be performed in standard speed mode.
- The commands in steps (1) - (3) must always be issued using CdControlB to check that the command has successfully completed. The example above has been simplified for the purpose of explanation, but the results from each command should be checked with certainty.
- Relevant messages should be output during CD detection as needed.

Warnings Regarding Changing the Motor Speed in the CD Subsystem

In the PlayStation CD subsystem, it is necessary to maintain a fixed interval between switching speeds and issuing certain commands. If this is not handled properly, the problems which are described below will occur. This could result in a slew of complaints from customers, so programs should deal with these possibilities very thoroughly.

Problem

When a command to move the CD head (CdlSeekL/P, CdlReadS/N) is issued immediately after the CD transfer speed is changed, the system will lose control of the head, resulting in strange sounds coming from the CD.

This problem occurs because timing problems in the CD subsystem prevent proper control of the head immediately after the transfer speed has changed. In the worst case scenario, a command to move the head issued immediately after a speed change will result in the head running amok and then stopping when

CONFIDENTIAL

it hits the mechanical stopper. When this happens, the CD subsystem will recover control of the head so the program will not crash. Furthermore, when the head runs amok and hits the stopper, the safety mechanism will operate so there is no danger of damage to the mechanism. However, the operation of the safety mechanism will result in a strange sound, which could lead to complaints from customers.

The functions/commands relating to head movement are as follows:

```
CdRead(int sectors, u_long *buf, int mode)
CdRead2(long mode)
CdSearchFile(CdlFILE *fp, char *name)
CdReadFile(char *file, u_long *addr, int nbyte)
CdReadExec(char *file)
CdPlay(int mode, int *tracks, int offset)

CdlSeekP
CdlSeekL
CdlReadS
CdlReadN
CdlPlay
```

The following measures should be taken if any of the above functions or commands are to be issued after a change in transfer speed.

Countermeasure

If a command to move the CD head is to be issued after a change in CD transfer speed, always leave an interval of at least three vsyncs.

Example:

```
       :
com = CdlModeSpeed;
CdControl( CdlSetmode, &com, 0 );
       :
       :
/* Perform an operation that takes up at least three vsyncs */
/* For example, VSync( 3 ); */
       :
       :
ret1 = CdControl( CdlSeekL, &pos, result );
ret2 = CdControl( CdlReadN, &pos, result );
       :
```

This will prevent situations where the head cannot be properly controlled. The same problem will occur if a parameter to the functions below results in a change in transfer speed. Therefore, transfer speed should not be changed using parameters for these functions. Instead, transfer speed should be changed manually (with an interval of three vsyncs or more).

```
CdRead(int sectors, u_long *buf, int mode)
CdRead2(long mode)
```

Please note that the CD subsystem transfer speed will be set to standard speed after the following functions are executed.

```
CdInit(void)
CdReset(int mode)
```

Noise during CD-DA/XA playback

When noise occurs during CD-DA/XA playback, check the following points:

Is the converted data correct?

The sound tool assumes that data is 16-bit straight PCM data. Note that it is not compatible with AIFF. When converting AIFF, since the header and footer information which appears at the beginning and end is converted into sound, noise will be produced. The SoundDesignerII 2.5 sampling data format is 16-bit straight PCM, so it can be used as is.

Does the volume decrease when playback is paused or a seek is performed?

Pausing a CD or performing a seek while sound is playing can cause clip noise to be produced. When pausing a game where the CD also pauses, issue the CD command after performing a fade out.

Does the XA data contain a large number of high pass components?

With XA data, sound is compressed to 1/4, so noise is sometimes produced. The noise can become particularly evident when there are a large number of high pass components. Perform a pre-process such as installing a filter in advance to avoid this.

Libcd Message Reference

The error messages from libcd are described below. The levels here correspond to the modes in CdSetDebug().

Table 11-19: Error levels

Level	Output Conditions
0	Always output
1	Output if debug level is 1
2	Output if debug level is 2
3	Output if debug level is 3

CD timeout

Format:
CD timeout: [pos] ([status]) Sync=[sync], Ready=[ready]

Level:
0

Parameters:
[pos] - the position where the timeout occurred
[command] - the command that was issued last
[sync] - last CdSync status
[ready] - last CdReady status

Example:
CD timeout; CD_sync: (CdlNop) Sync=NoIntr, Ready=NoIntr

Reason:
A callback was not generated from the CD-ROM subsystem within the expected time period.

CDROM:
unknown intr   Unknown Interrupt from Subsystem

Format:
CDROM unknown intr ([num])

CONFIDENTIAL

Level:

0

Parameters:

[num] - susystem status

Reason:

An undefined subsystem status was obtained.
Normal subsystem status is as follows: CdlDataReady   0x01
CdlComplete   0x02
CdlAcknowledge   0x03
CdlDataEnd   0x04
CdlDiskError   0x05

CD_init   Initialization Data for Subsystem

Format:

CD_init: addr=[addr]

Level:

0

Parameters:

[addr] - start address of bios function table

Reason:

Occurs when the start address of the bios function is set by CdInit()/CdReset().

CdInit:
Init failed   Initialization Failed

Format:

CdInit: Init failed

Level:

0

Parameters:

None

Reason:

Occurs in many cases when the CdlStatShellOpen flag is set. In these cases, subsequent attempts will be successful.

DiskError

Format:

DiskError

Level:

0

Parameters:

None

Reason:

A fatal error was generated.

DiskError   A Fatal Error was Generated

Format:

DiskError

Level:

0

Parameters:

None

Reason:

The command could not be executed or data could not be properly read.

CdRead:
sector error   Sector Addresses were not in Sequence

Format:

CdRead: sector error

Level:

0

Parameters:

None

Reason:

For some reason, the addresses in the sector data were not in sequence. In this case, assume that there was a skipped sector during CdRead(), and retry from the first sector.

CdRead:
Shell open   The Cover (Shell) was Opened During a Read.

Format:

CdRead: Shell open

Level:

0

Parameters:

None

Reason:

The cover was opened during execution of CdRead(). In this case, CdRead() will return to the first sector and retry.

CdRead:
retry   A CdRead Retry was Generated

Format:

CdRead: retry

CONFIDENTIAL

Level:

0

Parameters:

None

Reason:

CdRead() returned to the first sector and a retry was performed.

No TOC found: An Audio Track was not Found.

Format:

No TOC found: please use CD-DA disc

Level:

0

Parameters:

None

Reason:

The CdPlay() function could not be executed since no audio track exists. This error is also generated when no disc is mounted.

cbdataready: CdlDataEnd   Automatic Repeat Generated

Format:

cbdataready: CdlDataEnd (track=[track],time=[time])

Level:

0

Parameters:

[track] - number of track for which playback was completed
[time] - absolute time since the last ResetCallback() was called

Reason:

An automatic repeat was generated in the background during the execution of CdPlay().

track overflow

Format:

[track]: track overflow

Level:

0

Parameters:

[track] - the number of the track that was to be played next

Reason:

CdPlay() cannot begin playing track number [track]. The corresponding track does notexist on the disc

com=   An Error was Detected in the Issued Command

> **Format:**
>
> com=[command],code=([result0]:[result1])
>
> **Level:**
>
> 1
>
> **Parameters:**
>
> [command] - the last command issued
> [result0] - the first byte in the result buffer from CdSync
> [result1] - the second byte in the result buffer from CdSync

no param   Parameters of Primitive Command were not Set.

> **Format:**
>
> [command]: no param
>
> **Level:**
>
> 1
>
> **Parameters:**
>
> [command] - the last command issued

CdSearchFile: Detailed Information on CdSearchFile

> **Format:**
>
> CdSearchFile: disc error
> [name]: path level ([num]) error
> [name]: dir was not found
>
> **Level:**
>
> 1
>
> **Parameters:**
>
> [name] - filename to be searched
> [num] - depth of path
>
> **Reason:**
>
> The root directory could not be read. The disc is not an ISO-9660 format disc.

CD_newmedia: Detailed Information Regarding Retrieval of Root Directory for CdSearchFile

> **Format:**
>
> CD_newmedia: Read error in cd_read(PVD)
> CD_newmedia: Disc format error in cd_read(PVD)
> CD_newmedia: Read error (PT:[pos]
> CD_newmedia: searching dir..\n"));
>   [min0]:[sec0]:[sector0]
>   [min1]:[sec1]:[sector1]
>   ....
>
> **Level:**
>
> 2

Parameters:

[pos] - position of root directory
min(n)] - position of directory (in minutes)
[sec(n)] - position of directory (in seconds)
[sector(n)] - position of directory (sector)

Reasons:

PVD sector cannot be read.
Format of PVD sector is not correct.
Format of sector is not correct.
The root directory cannot be read.
If the root directory can be read, its contents are output.

CD_cachefile: Display Contents of Current Directory of CdSearchFile

Format:

CD_cachefile: searching...
([min0]:[sec0]:[sector0])
([min1]:[sec1]:[sector1])
....
CD_cachefile: [num] files found

Level:

2

Parameters:

[min(n)] - position of files in current directory (in minutes)
[sec(n)] - position of files in current directory (in seconds)
[sector(n)] - position of files in current directory (sector)
[num] - number of files in current directory number

Streaming Library Overview

The streaming library is a group of functions for getting realtime data such as movies, sounds or vertex data stored on high-capacity media in units called frames. A frame consists of one or more sectors, the smallest unit of data on a CD-ROM.

High-capacity media at the present is assumed to be CD-ROM, semiconductor memory, or a hard disk; the current version supports CD-ROM.

A single frame of data obtained using the streaming library is guaranteed to be complete, have no omissions, and be contiguous.

The library has the following functions.

- Synchronous processing of CD-ROM and video
- CD-ROM data error processing
- Continuous data reading
- Suspend processing
- Complete processing

The streaming library is responsible for accessing the CD-ROM and putting the data needed, in units of time, into memory. The user program handles displaying this data on the screen and outputing it as sound and so forth.

CONFIDENTIAL

Streaming

Streaming is the process of continuously reading data from CD-ROM and transferring it to main memory. It is used for realtime processing of data, such as playing video or 3D vertex animation. The process of continuously reading CD-ROM sectors makes full use of the CD-ROM transfer rate.

Streaming combines data processing units (1 frame of compressed image data, etc.) consisting of multiple sectors in main memory, and transfers the header pointer to the application.

Synchronization Control

When continuously reading and processing sector data, one frame must be processed in less than the time it takes to read one frame from the CD-ROM. If this does not happen, the processing cannot keep up, CD-ROM data accumulates, and the buffer overflows.

However, frame processing of is not synchronized with CD-ROM reading, so processing must complete in less time it takes to read the frame. This makes synchronization difficult.

The streaming library solves the problem of synchronization. If processing of one frame exceeds the time it takes to read one frame, the read data is discarded in increments of frames. This mechanism ensures that data read from the CD-ROM has integrity at the frame unit level, and that data is always read, processed and synchronized at high speed. This function is implemented by using a ring buffer to store CD-ROM data.

However, depending on the application there will be times where you will definitely not want to discard the frame. At such times, a means for making time adjustments by returning the head is provided. Since synchronization is accompanied by head access in this method, XA audio and streaming cannot be used at the same time. Refer to StGetBackLoc and StRingStatus.

Ring Buffer

The streaming library has a ring buffer that is used to store and lock data.

The ring buffer size is optional in units of sectors, requiring that the main program ensure the integrity of this area. This is reported by StSetRing(). When the programmer has finished processing that data, he or she needs to release the lock. Releasing the lock is done with StFreeRing().

When the ring buffer fills up with locked data, the library discards data in units of frames. When the lock is released, data is read.

The library automatically adjusts the end of the ring buffer address so that it does not hit in the middle of one frame of data.

Ring Buffer Format

The ring buffer region is broadly divided into two regions, each of which is a ring buffer. The upper part is a header region for addresses, and the lower part is the data region.

The header region is a ring buffer with 8 words (32 bytes) in 1 sector. The data region is a ring buffer with 504 words (2016 bytes) in 1 sector.

For example, if the ring buffer size is 4, the following data reading occurs.

CONFIDENTIAL

Figure 11-4: Ring Buffer Size 4 Example

```
┌─────────────────────┐  ▲
│  SECTOR 0 HEADER     │  │
├─────────────────────┤  │
│  SECTOR 1 HEADER     │  │  32-byte sectors
├─────────────────────┤  │
│  SECTOR 2 HEADER     │  │
├─────────────────────┤  ▼
│  SECTOR 3 HEADER     │  ▲
├─────────────────────┤
│                     │
│  SECTOR 0 USER DATA  │
│                     │
├─────────────────────┤
│                     │
│  SECTOR 1 USER DATA  │
│                     │
├─────────────────────┤  2016-byte sectors
│                     │
│  SECTOR 2 USER DATA  │
│                     │
├─────────────────────┤
│                     │
│  SECTOR 3 USER DATA  │
│                     │
└─────────────────────┘  ▼
```
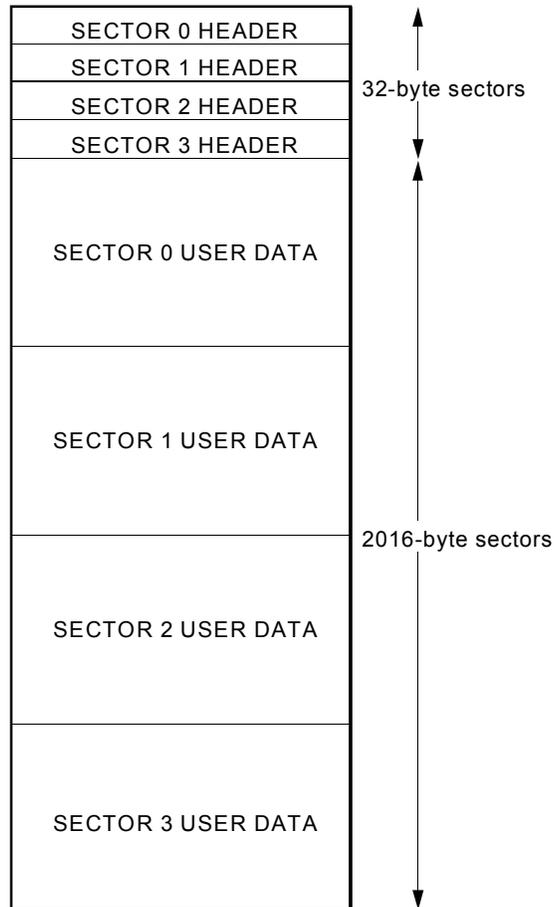
When data is read from a CD-ROM, that sector is locked.

When StGetNext() is called, the frame starting address is returned when a frame's worth of data is available. When the programmer finishes processing this frame of data, the frame region is released using StFreeRing(). New data may be read from the CD-ROM to the released region.

Memory Streaming

If one sequence is rather large going into the ring buffer region and reading stops before the ring buffer overflows, the sequence may be repeated not from the CD-ROM but by streaming from memory. (There is a limit to the number of times a sequence may be repeated.)

If the end_frame argument in StSetStream() and StSetEmulate() is set as 0, reading from the CD-ROM may be automatically halted at the ring buffer cutoff.

The processing described above makes it possible to implement memory streaming without ring buffer looping.

Interrupt Control of 24-Bit Movie Playback Time

The function StCdInterrupt() performs interrupt control during streaming. It is called automatically by interrupts from the CD-ROM, and usually does not need to be executed.

CONFIDENTIAL

However, StCdInterrupt() does relatively large 2K-byte DMA transfer from CD-ROM to main memory, so it occupies the bus for a relatively long time. A method for controlling the calling of this function is provided. This function is used when playing RGB 24-bit movies.

If bit 1 of 24-bit mode is set ON in the loc mode arguments in StSetStream() and StSetEmulate(), StCdInterrupt() is not called automatically. Instead, a flag called StCdIntrFlag is set. Timing can be controlled by the programmer by watching for this flag and calling this function at an appropriate time. StCdIntrFlag is defined in the library as an unsigned long global variable.

```
/* STCdIntrFlag usage example*/
extern unsigned long    StCdIntrFlag;

if (StCdIntrFlag == 1) {
        StCdInterrupt();
        StCdIntrFlag = 0;
}
```

Interrupt Functions Used

The streaming library uses the following interrupt functions.

Table 11-20: Interrupt functions

Libcd function name	Libds function name	Details
CdDataCallback	DsDataCallback	Sector data transfer completion callback
CdReadyCallback	DsReadyCallback	Sector data ready callback

Chapter 12:
Extended CD-ROM Library

Table of Contents

Overview

The extended CD-ROM library (libds) provides a new interface while using the kernel from the existing CD-ROM library (libcd). Libds implements a command queue which accommodates speed differences between the main CPU and the CD subsystem. libds also performs PlayStation-specific processing, such as operations involving the opening or closing of the CD cover.

This chapter assumes familiarity with libcd and mainly presents differences from libcd.

Library and Header Files

The library file for libds is `libds.lib`; programs that use services from libds must link with this library. Since libds uses libcd to control the CD subsystem, `libcd.lib` (version 4.0 or higher) must be linked as well. You must also link version 4.0 or higher of `libetc.lib`.

Source code must include the header file `libds.h`.

Description of libds

Description

Libds is a new interface implemented on top of the libcd kernel system. An independent kernel system is installed over libcd's control routines so that the CD subsystem can perform operations such as command queue control and operations when the CD cover is opened. Features equivalent to those provided by libcd are provided, programs can be updated easily.

Figure 12-1: CD libraries
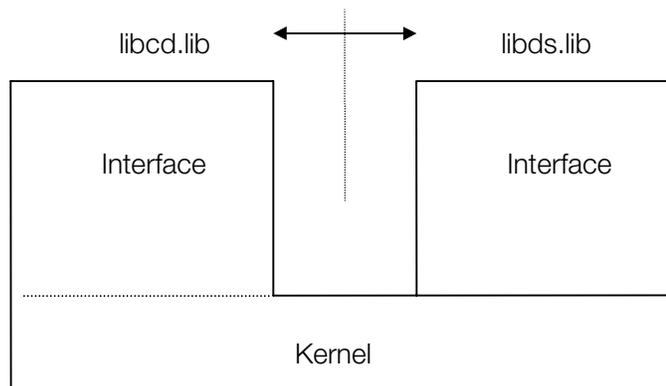
Relationship with libcd

The kernel system and the command queue used by libds operate exclusively from the libcd functions. Consequently, calling a libcd function while libds is being used will destroy the kernel system and the command queue. Thus, when libds is being used, libcd functions should not be used (functions beginning with "Cd", including CdInit()).

CONFIDENTIAL

Streaming Functions

Libds cannot be used simultaneously with libcd, but streaming should be performed normally using the St*() functions. The St*() functions are part of libcd.lib, but they do not affect libds operations since they do not control the CD subsystem. When initiating streaming, the functions and commands from libds should be used (such as DsRead2() and the DslReadS command).

libapi Functions

The functions in libapi used for CD control (such as 96_init, LoadExec, Load) should not be used when libds is running. If these functions need to be used, they should be used after libds is finished.

Differences from libcd

Primitive Commands

The primitive commands perform the same operations as libcd. The command codes are redefined in libds.h, with the initial "Cdl" in the symbols being replaced by "Dsl".

Table 12-1: Primitive Commands

Symbol	Code	Type	Details
DslNop	0x01	B	NOP (No Operation)
DslSetloc	0x02	B	Set target location for seek
DslPlay	0x03	B	Begin playing CD-DA
DslForward	0x04	B	Fast-forward
DslBackward	0x05	B	Rewind
DslReadN	0x06	B	Start reading data (with retry)
DslStandby	0x07	N	Wait while disk continues spinning
DslStop	0x08	N	Stop disk rotation
DslPause	0x09	N	Pause at current location
DslMute	0x0b	B	CD-DA mute
DslDemute	0x0c	B	Release mute
DslSetfilter	0x0d	B	Select ADPCM sector to play
DslSetmode	0x0e	B	Set basic mode
DslGetparam	0x0f	B	Get final status, operation mode
DslGetlocL	0x10	B	Get logical location (data sector)
DslGetlocP	0x11	B	Get physical location (audio sector)
DslGetTN	0x13	B	Get number of TOC entries
DslGetTD	0x14	B	Get TOC
DslSeekL	0x15	N	Logical seek (Data sector seek)
DslSeekP	0x16	N	Physical seek (Audio sector seek)
DslReadS	0x1b	B	Start reading data (no retries)

CONFIDENTIAL

Structures

For each structure used in libcd, libds contains an equivalent structure. The initial "Cdl" in the symbols for the structures are replaced with "Dsl".

Table 12-2: Structures

Symbol	Symbol under libcd	Details
DslATV	CdlATV	Audio attenuator
DslFILE	CdlFILE	9660 file descriptor
DslFILTER	CdlFILTER	ADPCM channel
DslLOC	CdlLOC	CD-ROM location

Functions

Libds contains functions equivalent to those in libcd. The initial "Cd" in the symbols are replaced with "Ds". However, some functions use different arguments or involve different timings. Please refer to the reference material for details on specific functions.

Processing Speed

Libds uses a command queue to manage primitive commands. Thus, precise processing speeds (timing) will vary from those in libcd.

In libcd, if a primitive command is issued while a previous command is still executing, the function that issued the new command (such as CdControl()) blocks and waits for the previous command to finish. Once the command has completed, the function issues the new command.

With libds, however, if a command is issued when another command is executing, the new command is entered into a command queue and the function that issued that command will exit at that point. When the previous command completes, and at every VSync, an evaluation is made whether a queued command can be executed. If the command can be executed, it is sent to the CD subsystem.

The advantage of this method is that CPU processing is not blocked when a command is issued, regardless of the state of the CD subsystem. Also, if multiple commands are issued simultaneously, commands can be issued (entered into the queue) without waiting for the other commands to finish.

Compatible Functions

The execution of the primitive commands in libds are all performed as non-blocking operations. However, libds also provides functions that correspond to CdControl (CdControlB) from libcd.

Initialization and Exit

System Initialization

When libds is used, DsInit() must be executed at the start of the program.

```
int DsInit( void );
```

Once DsInit() has executed, it will not be possible to control the CD subsystem through non-libds environments (such as libcd or libapi). DsInit() internally initializes libcd, so CdInit() does not need to be called even if the streaming library (the St*() functions) will be used.

Resetting after Initialization

After DsInit() is used to initialize the system, it should not be called again as results may be unpredictable. If the system needs to be reset during normal operations, DsFlush() should be used instead.

```
void DsFlush( void );
```

DsFlush() flushes the CD subsystem and clears the command queue (to be described later). If for some reason the system needs to be restored to its original state, DsReset() should be used.

```
int DsReset( void );
```

Using DsReset() will clear the callback functions set by the program, so these functions should be reinstated after resetting.

Exiting the System

When activating a child process (.EXE), the libds system should be exited. Use DsClose() to exit the system.

```
void DsClose( void );
```

After the child process is finished, DsInit() can be called if the system needs to be used again.

Caution

DsFlush(), DsReset() and DsClose() will not stop data read (playback)operations. Data reads (playback) must be explicitly halted from the program by issuing a DslPause. DslPause should be used with DsFlush(), DsReset() and DsClose().

For example,

```
        :
while( DsControlB( DslPause, 0, 0 ) == 0 );
DsClose();
        :
```

Incorrect operation may result if after exiting the system, LoadExec or a similar operation is performed during a data read (playback).

The Command Queue

The command queue is a facility that monitors the state of the CD subsystem and controls the issuing and completion of primitive commands.

When a command is issued it is added to the queue. The command is sent to the CD subsystem when the subsystem is ready to receive the command.

Another function of the command queue is to automatically perform those processes necessary for the operation of the CD subsystem. For example, operations that are performed when the cover is opened are handled automatically by the system. While these operations are being performed, commands cannot be sent to the CD subsystem, but they can be entered into the queue and executed once the operations have completed.

Issuing Commands

The DsCommand() function is used to send primitive commands to the command queue.

```
int DsCommand(
        u_char com,          /* command code */
        u_char* param,       /* command parameter (4 bytes) */
        DslCB func,          /* pointer to callback function */
        int count )          /* retry count (-1: unlimited retries) */
```

The third argument is a pointer to the callback function which will be invoked when the command has completed. Callback functions can be set individually for each command, and they will be called only when the corresponding command has completed.

When a command is successfully issued (entered into the queue), a command ID (>0) is set as the return value of DsCommand(). This command ID can subsequently be used to get the execution status or result of command execution.

A 0 will be returned if the command queue is full.

Confirming Completion of Command

In order to see if a primitive command from the command queue has finished executing, a callback function can be specified when the command is issued, or the DsSync() function can be used.

```
int DsSync(
        int id,              /* command id */
        u_char* result )     /* return value of command (8 bytes) */
```

DsSync() returns the execution status of the specified command at the point when it is called.

Table 12-3: Confirming Completion of Command

Symbol	Meaning
DslComplete	Command exited normally
DslDiskError	Command returned an error
DslNoIntr	Command has not yet been executed
DslNoResult	Command has exited but no results are available. When execution has completed, the return value is stored in 'result' for (DslComplete, DslDiskError).

The system can hold multiple execution results and the results from two previous commands can be retrieved. However, older execution results are overwritten, so a DsSync() for a command that is too old will return a DslNoResult. The number of execution results saved by the system is defined in the macro constant DslMaxRESULTS.

Checking Command Queue Status

The DsQueueLen() function can be used to retrieve the number of commands currently stored in the command queue.

```
int DsQueueLen( void );
```

DsQueueLen() returns the number of commands stored in the current queue. The command count includes commands that are currently being executed. The maximum number of commands that can be entered in the queue is defined by macro constant DslMaxCOMMANDS. The maximum number of commands may be changed with version upgrades, so please use references to the macro constant.

DsSystemStatus() is used to retrieve the status of the system.

```
int DsSystemStatus( void );
```

DsSystemStatus() returns the current status of the system. The return values are as follows.

```
DslReady              Ready to execute command
DslBusy               Command being executed or command cannot be executed
DslNoCD               CD is not set
```

DslReady is returned when the CD subsystem is in the normal state and no command is being executed. If a command is entered in the queue, the operation is begun immediately. In cases where timing is important, the program should double-check to confirm that the command count in the queue is 0.

DslBusy is returned when a command is currently being executed or when a command cannot be executed for some reason. Examples of cases when commands cannot be executed include when operations performed in response to the opening or closing of the cover are taking place, or when operations cannot be performed for a fixed time due to a change in CD speed. During this time, commands are added to the command queue and will be sent to the CD subsystem once the status changes.

DslNoCD is returned when there is no CD set in the drive. After operations are performed in response to the opening or closing of the cover, the status changes to DslNoCD if no CD is detected.

Timing

If execution is possible, a primitive command issued by DsCommand() is sent immediately to the CD subsystem. When execution is not possible, the command is added to the queue such as when a previous command has not completed. When the previous command is done, the new command will be issued (from a sync callback--sync chain).

When operations are blocked, such as when the cover has been opened or closed, the sync chain is broken. In such cases, the queue is polled with the VSync interrupt.

Error Operations

If an error is generated during execution of a command, the command is re-issued (retry). The number of times the instruction is retried is specified by the fourth argument of DsCommand() (count). Retries will be performed count times. The command will not be retried if count is equal to 0.

If the command is not successful after the specified number of retries, the command returns an error and is removed from the queue.

When count is equal to -1, retries will be performed until the command is successful (unlimited retries).

When the CD cover is opened, all commands entered in the queue are cancelled. If callback functions are specified for the command, the callback functions are called in the order in which they were queued.

Callbacks

Callback functions are invoked when a primitive command has completed. Callback functions can be specified individually for each command or one function can be specified as a common callback function.

When a callback function is specified for a command, the function is only called when that particular command has completed. Once the callback function has been called, the callback setting for that command is automatically removed.

When a common callback is set, the function will be called when all of the commands have completed or when an error is generated due to non-synchronization (such as when the CD cover is opened).

The callback function called from each command is described in the following format:

```
void function( u_char intr, u_char* result );
```

intr and result refer to the same interrupt information as normal data ready callback functions. Usually this is Intr ==DslComplete (Command success). Refer to the section titled "Simple Callbacks" for a description on cases when a callback function is called in intr==DslDiskError.

Multiple Operations

Multiple commands can be entered together in the command queue, but this does not mean that multiple operations (data reads or playback operations) can be performed simultaneously. The data read commands (DslReadN, DslReadS) and playback command (DslPlay) are considered complete the moment these commands are accepted by the CD subsystem. Once accepted, these commands are removed from the queue. If another command is available, it will be issued to the CD subsystem.

Depending on this newly issued command, a data read (playback) operation that is in progress may be halted and the data read operation may not be able to obtain its requested data.

Consequently, multiple data read (playback) operations cannot be entered in the command queue simultaneously.

When multiple data read (playback) operations need to be performed, the program should issue a single command corresponding to the first operation, obtain the desired data (i.e., perform playback over an appropriate interval), then issue the next command to the queue.

Command Packets

Primitive commands can be issued to the command queue by means of a *command packet*: a series of commands that are issued together. For performing a CD data read (CD-DA playback), the command packet consists of four primitive commands issued in the following sequence:

- Pause (to end the previous operation)
- Set the operating mode
- Specify the start position
- Perform the read

These four commands can be entered into the queue with a single function call.

Stable CD access can be achieved by issuing commands in the form of a command packet. The operating mode and the start position can be specified with each command packet, so the operation will not be affected by the previous state of the CD subsystem. Also, if an error occurs, the operation is retried from the start of the command packet. This makes it more likely that the retry will be successful.

Issuing Command Packets

DsPacket() is used to issue a command packet.

```
int DsPacket(
        u_char mode,          /* Operating mode */
        DslLOC* pos,          /* Start position */
        u_char com,           /* Read (playback, seek) command */
        DslCB cbsync,         /* pointer to callback function to be called
                                 when command completes */
        int count )           /* retry count (-1: unlimited retries) */
```

When this function is executed, the following four commands are entered into the queue.

- DslPause - 0
- DslSetmode - mode
- DslSetloc - pos
- The command specified by com - 0

The commands DslPlay, DslReadN, and DslReadS can be specified for com. If a seek command (DslSeekL, DslSeekP) is specified for com, everything up to the completion of the seek operation can be considered part of the packet. This allows pre-seeking.

Checking for Completion

To check to see if a command packet has finished executing, a callback function can be specified when the packet is issued. Alternatively, the DsSync() function can be used to test for command completion.

The command packet terminates when the execution of all its primitive commands has completed. When using DsSync(), the packet should be referenced using a command ID just as if it were a primitive command. The result from the execution of the final primitive command in the packet is saved in the result parameter of DsSync().

Timing

When a command packet is issued, the individual primitive commands contained in the packet are processed by the command queue. Therefore, timing is based on the operation of the command queue.

Error Operations

An error in one of the commands in a packet will result in the operation being retried. Unlike regular commands (commands issued through DsCommand()), command packet retries are performed starting with the first command in the packet. This is done to make it more likely that the retry will succeed.

For data read (playback) operations, it is recommended that commands be issued as packets rather than as individual commands.

The number of retries to be performed is specified by the count parameter. As in regular commands, no retries are performed when count is set to 0, and unlimited retries are performed when count is set to -1. If the retry count is exceeded when an error is generated, the packet is removed from the queue.

The Simple Callback

When data is to be read from the CD, a data read command is issued and data is transferred from the CD sector buffer to main memory after each data ready interrupt. The library provides a simple callback feature to allow easy handling of data ready interrupts.

Features of the Simple Callback

The simple callback is triggered from the data ready interrupt. It is triggered only when data is read normally. If an error occurs during the data read, recovery is performed automatically by the system.

To use the simple callback feature, call DsStartReadySystem().

```
int DsStartReadySystem(
        DslRCB func,          /* pointer to callback function called for
                                 successful data read. */
        int count )           /* retry count (-1: unlimited retries) */
```

The *count* parameter specifies the number of retries to perform; -1 specifies unlimited retries. If the retry count is exceeded and an error is generated, the callback function is triggered with DslDiskError.

DsStartReadySystem() should be called after checking within the callback for the corresponding read command (packet) to see if the command was successful. (If DsStartReadySystem() is called earlier, error recovery operations may not function properly.) The current CD-ROM system does not distinguish between errors that correspond to the command and other errors, so the system for the simple callback may respond to an error from a command. Also, the lead sector may be missed if the system is started too late.

Recovery Behavior

If an error is generated during a read, the simple callback system performs a recovery operation. During recovery, the command is reissued based on the last state saved by the system (the last command issued, the last operating mode, the last seek position, the current position, etc.).

The seek position for a recovery operation is determined by the system. The restarted read operation starts from the sector before the one where the error occurred. However, the callback function specified by *func* will not be triggered until the sector following the previously successful sector is read.

For example, if an error occurred in the first read operation at the fourth sector, three sectors will have already been read. Recovery processing is performed, and the callback function will trigger after the data from the fourth sector is read.

```
First read: 1, 2, 3, (4)
            Error occurs here (callback function is not triggered)
        Head is moved to the preceding sector by the recovery
        operation
Second read: ... 4, 5, 6
            Callback function triggered from this sector
```

Description of Callback Function

The callback function that is triggered for data reads is specified according to the following format.

```
void function( u_char intr, u_char* result, u_long* subhead);
```

As in the standard data ready callback function, *intr* and *result* refer to interrupt data. *intr* almost always has the value DslDataReady or DslDataEnd (only for DA playback). However, *intr* has the value DslDiskError when:

- The retry count is exceeded and an error occurs
- The CD cover was opened during reading

Recovery processing for these cases must be handled by the application. When a data read is successful and the callback is triggered, the sub-header of the data has already been transferred, because the system looks at the subhead to check the data. The sector buffer pointer is moved to the start of the data, so the data body can be transferred immediately. The size of the data body is 2048 bytes.

Exiting the System

When the desired data has been read, the simple callback should be exited. To end the simple callback, use DsEndReadySystem().

```
void DsEndReadySystem( void );
```

Exiting from the system must be performed immediately after the last sector has been read. If exiting is delayed, more read operations can take place. This may generate extra callbacks that can overwrite memory. Thus, the callback function should exit after the final sector has been transferred.

System Operation when Opening and Closing the CD Cover

When the CD cover is opened and closed during system operation, the simple callback in initial status is terminated at that point and the callback function set by the application is called by intr==DslDiskError. Although the application must perform the following recovery, this can be set to be performed with the simple callback system. Perform the setting with the DsReadySystemMode() function.

```
int DsReadySystemMode ( int mode );
        /* mode      0: Simple callback is terminated when cover is opened
                     1: Recovery from opening/closing is performed
                        automatically */
```

Calling a function when the mode is 1, will cause the system to not terminate if the CD cover is opened during the operation of a simple callback. The simple callback waits until the cover is closed to reissue the command and performs recovery in the same way as with normal errors. In such cases, application callback functions regarding opening the cover are not called. When the cover is closed and the disk is not set, the simple callback will terminate and the application callback function will be called by intr==DslDiskError. Confirmation of whether or not the disk has been set can be obtained using the result[0] DslStatStandby bit (if the disk is not set, the spindle will not move and this bit becomes 0) or with DsSystemStatus(). Furthermore, the initial status mode is 0.

Caution

- When using the simple callback to perform data reads, the operating mode should be set so that the sector size is 2340 bytes (DslModeSize1bit ON, DslModeSize0 bit OFF).

- The present library cannot recognize if the disk was changed when the cover was opened and closed. Therefore, when automatic opening/closing cover recovery is being carried out the player has intentionally replaced the disk when, this causes incorrect data to be read and there is the possibility that the game will be unable to continue.

- Simple callback is also used by the following high-level library functions:

  DsGetDiskType()
  DsPlay()
  DsRead()

Therefore, the DsReadySystemMode() change also uses these functions. Particularly since DsGetDiskType() is used when performing disk exchange, if it is used when recovery has been automized, problems such as the application being unable to recognize if the player has opened the cover again during processing may occur. Automatic recovery should not be performed in cases such as when disks are being swapped.

Other

Opening and Closing the CD Cover

The PlayStation CD-ROM drive requires special operations to be performed if the CD cover is opened or closed in the middle of an access. libds handles these operations within the system.

When the CD cover is opened, the system changes to the busy state (DslBusy), and commands from the user are blocked. When the cover is closed, the system performs operations to re-check the disk, then returns to the ready state (DslReady).

The operation that was being performed when the CD cover was opened will return an error. Also, all the commands entered in the command queue will be deleted. Thus, it will be necessary to wait for the system to return to the ready state at which time the operation will have to be repeated.

Immediately after the CD cover is opened or closed, the operating mode and the head position are initialized, so subsequent operations must take this into account.

Notes Regarding Swapping of CDs

If CDs are swapped during execution, the system performs cover opening/closing operations. However, the system does not determine the type of CD that is set, so this must be done by the application.

In the libds library, the operations up to CdDiskReady() provided by libcd are performed automatically. Therefore, the type of CD should be checked once the system status becomes DsSystemStatus() == DslReady. The library calls the function DsGetDiskType(), which is equivalent to the function provided in libcd to determine the CD type. Please refer to the reference material regarding this function.

The following steps are recommended for swapping CDs under libds:

1. Stop rotation of CD.
2. Output swapping message. If possible, have the user confirm that a new disk has been set by pressing a button.
3. Poll the current status with DsStatus() to confirm that the cover has been opened (DslStatShellOpen bit ON).
4. Wait for DsSystemStatus() == DslReady.
5. Determine the type of the CD using DsGetDiskType() and confirm that the CD is a PlayStation disk.
6. Confirm that the disk is the desired disk (check that it isn't a disk from another game, that it is the proper disk from a series, etc.)

If a DslNoCD is returned at step 4, this means that the library was not able to confirm that a disk was set (there was no rotation of the CD spindle within a predetermined period). In this case, a message such as "CD not detected" should be displayed, then processing should return to step 2. Similarly, if a PlayStation disk is not detected at step 5, a similar message should be output and processing should return to step 2.

Step 3 can be omitted if, in step 2, the user confirms that a disk has been set by pressing a button. Also, the confirmation of the disk in step 6 should be performed by the application using a method such as reading expected data from a specific position on the disk.

Transfer Speed Change

When the transfer speed of the PlayStation CD drive is changed, it will be impossible to execute commands for approximately three frames (1 frame =1/60 second). In libds, the system recognizes when the transfer speed is changed, sets the status of the 3VSync after the change to DslBusy, and blocks the execution of all commands. The commands issued during this period are stored in the command queue and since the commands are automatically executed 3VSync after the transfer speed change have passed, there is no need to wait 3VSync in the application program to issue a command.

Pre-seeking

Data reads can appear to execute more quickly by having the program seek to the start of the next data file beforehand if the next file to be read is known. Using command packets for seeking is recommended as retry processing can be automated. This minimizes the load on the main program flow when an error occurs.

Depending on the situation, the time required for data reads can be shortened by approximately 0.4 seconds when pre-seeking is performed. Also, it is easier to adjust timing when pre-seeking is used if XA audio is being played back during a game.

Performing a Continuous Read to Access Multiple Files

Reducing seeks is the most effective way to shorten loading time when multiple data files are read. Seeks can be reduced by laying out the CD so that data files are continuous, permitting a single read to access multiple files. For each seek eliminated, approximately 0.4 seconds are saved, so five fewer seeks will result in a loading time that is two seconds shorter.

If the files have different transfer destinations to main memory, the transfer address needs to be changed during the read operation. This is easy to implement using the simple data callback. The libds sample code gives an example of how the simple data callback can be used in this manner.

The Outer Three Minutes Problem

With the current CD-ROM subsystem, a seek to the outer three minute range of the CD-ROM can, depending on the starting point of the seek, result in incorrect seeks. One of the following measures must be taken to prevent this from occurring.

- Fill the outer three minutes (the final three minutes of data) with dummy data (the dummy data will not be used).
- If CD-DA is to be used for background music, make the final track three minutes or longer. In this case, no seeks will be generated for the outer three minutes as long as playback is not performed from the middle of the track and no repeats are performed within the track. This will allow the CD-ROM subsystem to operate properly.
- If the outer three minutes must be used as a data area, access the outer three minutes as a single, continuous file (such as for an opening or closing movie).

The outer three minutes here refers not to the outer area of the physical disc but rather to the outer area of the region in which data (DA) is recorded.

Notes Regarding DslPlay, DslReadN, DslReadS

With the current CD-ROM system, the commands DslPlay, DslReadN, DslReadS are considered complete and return a DslComplete once the command has been received by the CD subsystem. However, processing actually continues past this point, and errors may be generated.

In the following flow,

```
Command issued ... Success ... Seek completed ... (data read)...
                                an error may take place at this point
```

This type of error will be posted via an interrupt, but it will not be possible to associate the error with a particular command (the processing of the command is considered complete with the initial Complete). Thus, the command will not be retried in the command queue. Instead, error processing needs to be handled by the application.

Furthermore, these errors will not be reflected in the callback functions set for individual commands.

The callbacks that provide notification of these errors are the data ready callback (set with DsReadyCallback()) and the sync callback (set with DsSyncCallback()), which is independent of a specific command.

In order to recover from errors, an error recovery routine must be provided in these callbacks, or the status must be polled until the data read begins.

By setting the id argument for DsSync() to -1, it is possible to retrieve the execution results of the error for which the corresponding command cannot be determined.

DsStatus() can be used to check to see if a data read has begun.

The following is an example of error handling using polling. In this example, the operation is blocked until reading is begun. In order to avoid this, the routine that waits for the start of a read needs to be called once per frame.

```
        :
        :
/* issue read command as a packet (unlimited retries)
system will keep on retrying until success */
DsPacket( DslModeSpeed | DslModeSize1, &pos, DslReadN, 0, -1);

/* wait for start of read */
while( ( DsStatus() & DslStatRead ) == 0 ) {
        /* errors found with id == -1 are not handled by the system */
        if( DsSync( -1, result ) == DslDiskError ) {
                /* perform retry
                in this example, the packet is issued again */
        DsPacket( DslModeSpeed | DslModeSize1, &pos, DslReadN, 0, -1 );
          }
      }
        :
        :
```

Below is an example of the use of the data ready callback.

```
              :
              :
       /* set sync callback function when issuing packet */
       DsPacket( DslModeSpeed | DslModeSize1, pos, DslReadN, cbsync, -1 );
              :
              :


              :
       /* if sync callback function is successful, hook data ready callback */
       void cbsync( u_char intr, u_char* result )
       {
              if( intr == DslComplete ) {
                     /* by hooking the ready callback here, confusion errors
                        corresponding to commands is avoided */
              DsReadyCallback( cbready );
              }
       }
              :
              :
       /* Ready callback function
       use for data transfer but have it handle recovery on error*/
       void cbready( u_char intr, u_char* result )
       {
              if( intr == DslDiskError ) {
                     /* clear callback ... */
                     DsReadyCallback( 0 );
                     /* ... and then issue packet again */
                     DsPacket( DslModeSpeed | DslModeSize1, pos, DslReadN,
                            cbsync, -1 );
                     return;
       }
       if( intr == DslDataReady ) {
              /* data transfer routine (omitted) */
              }
       }
```

The point to be noted here is that the data ready callback will receive notification of all errors. Since errors corresponding to commands will be posted as well, the timing for hooking the callback function to the interrupt must be determined carefully. In this example, the callback function is hooked when the packet succeeds. If this method is used, the data ready callback must be removed as soon as the read is finished. Otherwise, read packets might be issued unpredictably. The simple callback handles the error, so the application does not need to perform any error handling.

Completion of Data Reads

For cases where the application sets up a system where a command is issued to read data from a CD and data is transferred using the data ready callback, there is a trick to handling the end of the read.

When a read operation is to be ended, issuing the DslPause command will halt operations, but one or two sectors may be read before the CD subsystem receives the command and halts the operation. Depending on the callback function hooked to the data ready interrupt, this excess data may be transferred to main memory, resulting in data loss. In order to avoid this, the callback function must be removed from the interrupt as soon as the desired number of sectors has been read. Callback functions are unhooked by calling DsReadyCallback() with an argument of 0. By unhooking the callback function, excess data will not be transferred to main memory even if extra sectors are read. DslPause should then be issued to halt the read operation.

Noise during CD-DA/XA playback

When noise occurs during CD-DA/XA playback, check the following points:

Is the converted data correct?

The sound tool assumes that data is 16-bit straight PCM data. Note that it is not compatible with AIFF. When converting AIFF, since the header and footer information which appears at the beginning and end is converted into sound, noise will be produced. The SoundDesignerII 2.5 sampling data format is 16-bit straight PCM, so it can be used as is.

Does the volume decrease when playback is paused or a seek is performed?

Pausing a CD or performing a seek while sound is playing can cause clip noise to be produced. When pausing a game where the CD also pauses, issue the CD command after performing a fade out.

Does the XA data contain a large number of high pass components?

With XA data, sound is compressed to 1/4, so noise is sometimes produced. The noise can become particularly evident when there are a large number of high pass components. Perform a pre-process such as installing a filter in advance to avoid this.

Chapter 13:
Controller/Peripherals Library

Table of Contents

CONFIDENTIAL

The Controller/Peripherals libraries are:

- ETC library (libetc): controls callbacks for performing low-level interrupt processing and controller-related functions. It also includes controller-related functions.
- Gun library (libgun): detects the position of the gun connected to the PlayStation and pointed towards the television screen.
- Multi Tap library (libtap): provides communication services for multiple controllers and Memory Cards when a Multi Tap is connected to the PlayStation.
- Controller library (libpad): provides services for managing ordinary controllers and DUAL SHOCK controllers connected to the PlayStation.

Some Controller/Peripheral functions are also provided in the Kernel library (libapi)

For a complete description of all Controller/Peripherals functions, refer to the *Run-Time Library Reference*.

ETC Library Overview

The ETC library (libetc) controls callbacks. All callback functions used in each library are managed by this library. At present, functions relating to the controller are also included in this chapter. The details relating to callbacks and corresponding non-blocking functions are described in the *Run-Time Library Reference*.

Library and Header Files

To use the ETC library, your application must link with the file `libetc.lib`.

Source code must include the header file `libetc.h`.

Callbacks

Many functions such as graphics drawing, transferring data to the sound buffer, and loading data from the CD-ROM, may execute in parallel (asynchronously) in the background. These functions are called non-blocking functions, because they don't block the CPU from performing other tasks.

You can define *callback functions* that execute when the non-blocking function actually terminates. What actually happens is that when the non-blocking function completes, it generates an interrupt and the program jumps to the address registered as the callback. When the callback returns, the program returns to the point where the callback began, and normal processing resumes.

A dedicated local stack is used for a callback function so that control can return to the original state after the callback returns. All interrupts are prohibited within callback functions. (Areas in which interrupts are prohibited are called *critical sections*.)

Figure 13–1: Callback Context
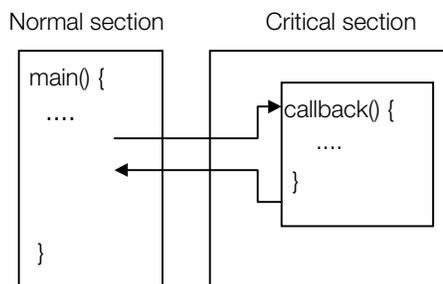
Callback Types

The following are some of the currently supported callbacks:

Table 13-1: Callback Types

Function Name	Corresponding non-blocking functions
VSyncCalllback	
DrawSyncCallback	DrawOTag()/LoadImage()/StoreImage()
DecDCTinCallback	DecDCTin
DecDCToutCallback	DecDCTout
CdSyncCallback	CdControl
CdReadyCallback	CdControl
CdDataCallback	

See the documentation for individual libraries for more information about each callback.

Callback Initialization

When using callbacks, the local stack must be created in advance, which is done with the initialization function ResetCallback(). The initialization functions of most libraries already include a call to ResetCallback(), so it is not usually necessary for an application to call this function explicitly.

The following table shows the initialization functions that call ResetCallback() automatically:

Table 13-2: Initialization Functions that Call ResetCallback()

Function Name	Contents
ResetGraph(0)	Drawing device initialization
DecDCTReset(0)	Decompression device initialization
CdInit(0)	CD-ROM initialization
SsInit()	Sound source device initialization
PadInit(0)	Controller initialization

After calling ResetCallback(), all callback pointers are initialized to NULL(0).

Callback Termination

Callbacks may be temporarily halted by calling StopCallback(). A callback halted by StopCallback can be restarted by calling ResetCallback() again. Callback function pointers recorded before StopCallback() is called are not saved when the callback is restarted.

Callback Pointers

A single callback is limited to recording one function pointer at a time. Setting a new callback discards the previous pointer. Therefore, it is an application's responsibility to create multiple pointers in a single callback if desired.

```
main(void){   /*Main Program*/
    void callback(void);
    ...
    VSyncCallback(callback);
    ..

}
void (*func)()={    /*callback table*/
    func0,   /*1st callback to be called*/
    func1,   /*2nd callback to be called*/
    func2,   /*3rd callback to be called*/
    0,
};
void callback(void) /*parent callback program*/
{
    int i;
    for (i=0; func[i]; i++)
       (*func[i])();
}
```

Previous callback pointers are discarded when a new callback is created. For this reason, if you are creating a temporary callback, you must return to the state that existed at the time the pointers were released.

```
void(*old)();
void addVSyncCallback(void(*func)())
{
        old=VSyncCallback(func);
}
void delVSyncCallback(void)
{
        VSyncCallback(old);
}
```

Multiple Callbacks

The callback context uses one local stack referenced by all current callbacks. Therefore, a callback cannot be launched from within another callback. When a callback request is generated from within a callback function, the requested function is held and its process is made to wait until the callback currently running has terminated.

The example below shows that if event1 and event2 are generated within a callback, execution of the corresponding callback waits until the callback at the top of the queue finishes processing. Note that time is required for processing within a callback. However, a callback with a timer used with a root-counter (RCnt) interface is given preference over normal callback processing.

Figure 13–2: Callback Context
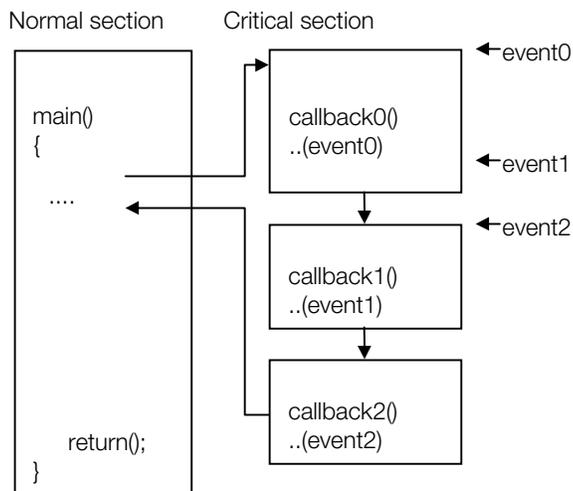
Default Callbacks and Events

When a library is initialized, the default callback functions are registered.

For example, when the CD library is initialized, it registers the CdSyncCallback() function, which is called when a CD-ROM primitive command terminates. A callback like the one below is created.

```
static void def_cbsync(unsigned char intr, unsigned char *result)
{
        DeliverEvent(HwCdRom, EvSpCOMP);
}
```

Termination of commands and data reading can be detected through the event handler when the default callback is used. Take care when installing a new callback, because the default callback is cancelled, and event transmission is halted.

Controller

Libetc provides a method of communicating with the standard controller:

- Call PadInit() to initialize the controller
- Call PadRead() to begin reading the controller
- Call Pad Stop() to end reading the controller

The content of the initialized controller is scanned once at the time of vertical blanking, and the most recent condition can be obtained at any time by the PadRead() function.

PadRead() returns a 32-bit integer value. The upper 16 bits are for controller A, and the lower 16 bits for controller B.

See libetc.h for a description of controller button assignments.

The PadInit/PadRead interface can be used only with the standard controller.

Video Mode

SetVideoMode() function is provided in the library for declaring the present video signal mode. Although the NTSC mode video signal environment is designed to be the default in the present library and due to the fact that the SetVideoMode() function mentioned above is called before all other library functions, the related library determines the mode. It will then be possible to perform operations which conform to the set video signal mode environment.

Please refer to the related libgpu and libsnd documents.

Programming Notes

This section describes the following programming issues:

- VSync callbacks
- The stack pointer and operations related to Exec processing
- Switching callbacks between processes

VSync Callbacks

Although there is only one callback entry internally, the RCnt interface maintains an internal linked list of callback function pointers. There is no predetermined sequence in which linked callbacks are called from the system.

If a single callback needs to call multiple functions in a specific sequence, VSyncCallback() should be used. See "Callback Pointers" for a code sample.

Timing of VSync Interrupts

VSync interrupts are generated at the beginning of a V-BLNK. Thus, rendering starts at the beginning of the callback function.

Rendering can be performed with a single buffer if it can be finished within a single V-BLNK interval. However, the start of rendering will be delayed if a sound driver is activated by the VSync interrupt and called (with SsSeqCalledTbyT) before the rendering function (DrawOTag). This prevents rendering from completing before the end of the V-BLNK interval.

```
callback()
{
        SsSeqCalledTbyT();
        DrawOTag(ot);
        DrawSync();
}
```

Figure 13–3: Timing with VSync Interrupts (1)
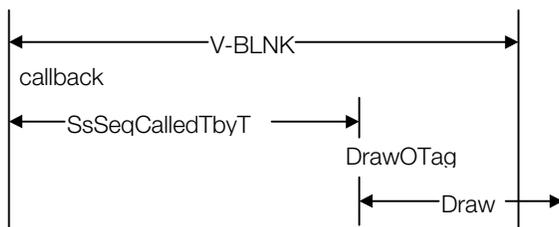
In the following example, rendering can be performed in parallel with sound driver execution.

```
callback()
{
        DrawOTag(ot);
        SsSeqCalledTbyT();
        DrawSync();
}
```

Note that since DrawOTag() is a non-blocking function, it should return immediately.

Figure 13–4: Timing with VSync Interrupts (2)
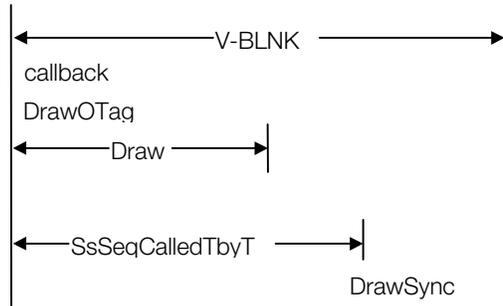
The Stack Pointer and Operations Related to Exec Processing

Libsn.lib contains useful functions related to the PC file system. It also contains routines that clear data areas and set the stack pointer, which are performed before invoking main(). Files created with a series of operations beginning with ccpsx always contain the section main. libsn.lib reads the DIP switch settings on the H2000 and sets the stack as far back as possible. 2MByte.OBJ and 8MByte.OBJ are provided for 2MByte and 8MByte settings.

A problem may occur when an executable file is called and control returns to the calling process after the completion of execution. Without creating a linker file, the value of the stack pointer of the executable program which was called cannot be determined. Even if a value were entered in the Exec structure, it would be ineffective because of processing prior to main().

In other words, the contents of the calling process's stack is destroyed by the called program. When the called program completes and tries to return, the return address is missing, resulting in a hang.

Thus, the called program needs to have processing prior to main() that is independent of either 2MBytes.OBJ, 8MBytes.OBJ, or libsn.lib, to ensure that no stack settings are made. This can be accomplished by linking the called program (which is expected to return) to NONE.OBJ. This is done in exactly the same way as if 2MByte.OBJ were used.

Switching Callbacks between Processes

In applications that use many events and callbacks, there have been reports of crashes when interrupts are issued while the system is switching between processes. Many of these crashes are due to callbacks that take place during process switching.

Problems caused by callbacks are difficult to trace and require a considerable amount of time to debug. Since these problems are not easily reproducible, it is possible for problems to surface after a program has already hit the market.

The following is a brief description of the callback initialization sequence during process switching. Please use this as a reference when writing applications.

"Process switching" means transferring control (changing the program counter) to a different program that is not linked to the same module. Process switching takes place when a child process is activated by a

resident parent process. The initial activation of an application (when control is transferred to PSX.EXE) is also considered a process switch from the OS to the application.

Child Processes and Callbacks

Data, together with a section of code for the callback, are linked in memory with the application.

When a parent process transfers control to a child process, the callback environment must be recreated.

For the parent process:

1.   Close all events (CloseEvent).
2.   Temporarily suspend callbacks (StopCallback).
3.   Jump to the child process (Exec).

For the child process:

1.   Initialize callbacks (ResetCallback).
2.   Re-initialize library (CdInit, ResetGraph etc).
3.   Reset application callbacks.
4.   Reopen events (OpenEvent).

These operations are also necessary when control returns from the child process back to the parent process.

Consider the following example.

```
/* Switching callbacks */

/* Parent: */
main() {
        .....
        .....
        StopCallback();     /* suspend callbacks*/
        Exec(&child_program);     /*activate child process*/
        ResetCallback();    /* reset callbacks*/
        .....
}

/* Child: */
main() {
        ResetCallback();    /* reset callbacks*/
        .....

        StopCallback();     /* stop callbacks*/
        return;
}
```

In this example, the child process is activated without switching callbacks, and the function pointers for the parent callbacks are kept in the callback table (the interrupt jump table).

This means that once the child process starts, interrupts that are generated will invoke the callback functions linked to the parent program, and control will not be transferred to the callbacks of the child program. This may lead to unexpected results.

The same analysis applies when the child process completes and control returns back to the parent.

Even if the called process (the child process) executes a ResetCallback() at the beginning, operations will be unstable if the calling process (the parent process) does not execute a StopCallback() at the end.

Interrupts generated in the interval between the activation of the child process and the reinitialization of the callback table by ResetCallback() will also produce callbacks from the parent process.

When the system is booted and an application is first executed, the OS sees the application as the first child process. Thus, for the same reasons as those described above, it is necessary to issue a ResetCallback() at the start of the program so that all the existing callbacks can be quickly replaced with callbacks linked to the application.

### Shared Libraries and Callbacks

When shared libraries are used, each process can share a single resident callback. In this case, there is no need to switch callbacks between processes. Even if shared libraries are used, however, it is crucial that ResetCallback() be executed immediately after an application is launched.

Callback Context

Callbacks are normally executed in the callback context; however, they do not always do so. Many callbacks are activated by a hardware interrupt, and when detected in the library function, the callback may be executed in the foreground, in order to reduce meaningless context switching.

For example, LoadImage() is normally executed in the background. However, when its transfer area is very small, the transfer is completed before returning from LoadImage(). Because main memory is shared by the drawing subsystem and the CPU, a reversal of processing terminations such as this by means of bus access timing can occur.

Figure 13–5:
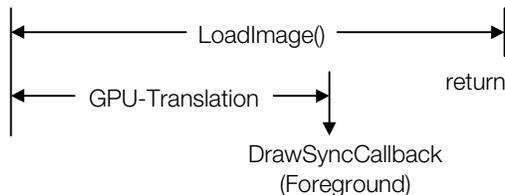
Since callbacks are prohibited within LoadImage(), the callback is held even if the transfer is completed within the function. At such times, LoadImage() will confirm the transfer status before termination, and even if the transfer has completed, the callback is activated at that location without the context being switched. As a result, functions which are registered in DrawSyncCallback() will be activated in the foreground.

Conversely, at LoadImage() termination, if the actual termination has not been completed or the command remains in the command queue, it is returned as is. Normally, since the termination of the actual transfer is slower than the LoadImage() function return, this strategy is selected.

Because of this feature, maximum coherency to memory access can be maintained even when extremely small areas of image (around 8x8) are transferred. On the other hand, it is possible for a callback to be activated in both the foreground and background (callback) contexts. Since the decision as to which context it is executed in depends on the space situation of the main bus, the CPU, and the subprocessor at that time, it is impossible to predict.

CheckCallback() can be used to determine whether a function is executing in a callback context (non-zero return value) or a normal context (zero return value). Normally, it's not necessary to know which context a callback will be activated in. However, there are cases, such as when switching threads within a callback, where the current context situation is known. In such cases it is necessary to use CheckCallback() to confirm the current context.

Controller Library

The controller library (libpad) provides services that allow applications to interact with the controllers, the input devices of the PlayStation. Applications can directly process data from the controllers, and dynamically identify each controller. The library also supports the additional features of DUAL SHOCK controllers.

Note: This library cannot be used in conjunction with the Multi Tap library (libtap) or the gun library (libgun). PadInitMtap() and PadInitGun() are provided in libpad for the Multi Tap and gun.

Library and Header Files

To use the controller library, your application must include the file `libpad.lib`.

Source code must include the header file `libpad.h`.

Additional Features Available for DUAL SHOCK Controllers

- Query the number of actuators (vibrators) available in the controller.
- Query actuator features.
- Query the current drain of the actuator.
- Set up the data list for controlling the actuator.
- Detect actuator combinations that can be used simultaneously.
- Query the type of terminal supported by the controller.
- Select the terminal type from the program.
- Select the lock/unlock setting of the terminal type selection switch.

Receive Buffer Data Format

The format used to store data in the receive buffer is described below.

Offset 0: 0x00 = successful; other values = failure

Offset 1:

- The upper four bits indicate the terminal type (see Table 13-3).
- The lower four bits represent half the byte count of the data received from the terminal (stored in the receive buffer starting at offset 3).

The remaining tables show the data layout for each type of controller.

Please refer to the documentation corresponding to the terminal type for information regarding the physical layout of the buttons and channels.

Table 13-3: Terminal Types

Terminal Type	Controller Name	Model Number
1	Mouse	SCPH-1030
2	16-button analog	SLPH-00001 (Namco Ltd)
3	Gun controller	SLPH-00014 (Konami Ltd)
4	16-button	SCPH-1080,1150,1200
5	Analog joystick	SCPH-1110

Terminal Type	Controller Name	Model Number
6	Gun controller	SLPH-00034 (Namco Ltd)
7	Analog Controller	SCPH-1150,1200
8	Multi Tap	SCPH-1070

Table 13-4: Mouse

Offset	Contents
1	Upper four bits: 0x1
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed
4	Displacement along the X axis (-128 to 127)
5	Displacement along the Y axis (-128 to 127)

Table 13-5: 16-button Analog

Offset	Contents
1	Upper four bits: 0x2
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed
4	Rotation 0 to 128 to 255
5	I button 0 to 255
6	II button 0 to 255
7	L button 0 to 255

Table 13-6: Gun Controller (Konami Ltd.)

Offset	Contents
1	Upper four bits: 0x3
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed

Table 13-7: Analog Joystick

Offset	Contents
1	Upper four bits: 0x5
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed
4	Position along the X axis (right stick) 0 to 128 to 255
5	Position along the Y axis (right stick) 0 to 128 to 255
6	Position along the X axis (left stick) 0 to 128 to 255
7	Position along the Y axis (left stick) 0 to 128 to 255

Table 13-8: Gun Controller (Namco Ltd.)

Offset	Contents
1	Upper four bits: 0x6
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed
4	Position along the X axis: Low-order byte
5	Position along the X axis: High-order byte
6	Position along the Y axis: Low-order byte
7	Position along the Y axis: High-order byte

Table 13-9: Analog Controller

Offset	Contents
1	Upper four bits: 0x7
	Lower four bits: (Byte count of received data) / 2
2,3	Button state 1: released, 0: pressed
4	Position along the X axis (right stick) 0 to 128 to 255
5	Position along the Y axis (right stick) 0 to 128 to 255
6	Position along the X axis (left stick) 0 to 128 to 255
7	Position along the Y axis (left stick) 0 to 128 to 255

Table 13-10: Receive Data Structure For Multi Tap Controller

Offset		Contents
1		Upper four bits: 0x8
2	Port A	Receive result 0x00: successful,
		other values: failed
3		Upper four bits: Terminal type
		Lower four bits: (Byte count of received data) / 2
4 - 9		Received data
10	Port B	Receive result 0x00: successful,
		other values: failed
11		Upper four bits: Terminal type
		Lower four bits: (Byte count of received data) / 2
12 - 17		Received data
18	Port C	Receive result 0x00: successful,
		other values: failed
19		Upper four bits: Terminal type
		Lower four bits: (Byte count of received data) / 2
20 - 25		Received data
26	Port D	Receive result 0x00: successful,
		other values: failed
27		Upper four bits: Terminal type
		Lower four bits: (Byte count of received data) / 2
28 - 33		Received data

Table 13-11: Button State Bit Assignments (1)

Bit	D15	D14	D13	D12	D11	D10	D9	D8
16-button	←	↓	→	↑	ST			SEL
Analog Controller	←	↓	→	↑	ST	R3	L3	SEL
Analog joystick	←	↓	→	↑	ST			SEL
16-button analog	←	↓	→	↑	ST			
Mouse								
Gun controller (Konami)			ST					
Gun controller (Namco)			A					

Table 13-12: Button State Bit Assignments (2)

Bit	D7	D6	D5	D4	D3	D2	D1	D0
16-button	□	X	◯	△	R1	L1	R2	L2
Analog Controller	□	X	◯	△	R1	L1	R2	L2
Analog joystick	□	X	◯	△	R1	L1	R2	L2
16-button analog			A	B	R			
Mouse						Left	Right	
Gun controller (Konami)	TRG	◯						
Gun controller (Namco)		B	TRG					

(All bits 1: released, 0: pressed)

Obtaining the Horizontal and Vertical Position with the Gun Interrupt (Terminal Type=3)

The horizontal and vertical positions of the gun can be obtained by using PadInitGun(). (For terminal type=6 guns, coordinate information is sent back to the receive buffer during normal communication with the controller, so this operation is not required.) PadInitGun() simply initializes the interrupt handler for obtaining horizontal and vertical position information. Therefore, apart from PadInitGun(), the controller communication environment must be initialized with either PadInitDirect() or PadInitMtap().

Before PadEnableGun() is called, interrupt requests for obtaining the coordinates are sent during each frame to the port at which the type 3 controller is connected. If coordinate information does not need to be retrieved from each port, the unneeded interrupts can be suppressed by masking them with PadEnableGun().

Adjustment of horizontal position data

Currently, the horizontal position data returns the system clock value, which is cleared to zero at each H blank. This value needs to be adjusted to compute the pixel value as a function of screen mode and horizontal resolution. The relationship between the system clock and the pixel clock is shown below.

Table 13-13: System Clock-Pixel Clock Conversion Table

Mode	Horizontal resolution	Coefficient
NTSC:	256	0.158532
	320	0.198166
	384	0.226475
	512	0.317065
	640	0.396332

CONFIDENTIAL

Mode	Horizontal resolution	Coefficient
PAL:	256	0.157086
	320	0.196358
	384	0.224409
	512	0.314173
	640	0.392717

[Pixel value] = [Coefficient] x [System clock value] + [Offset]

Initialization

Initialization flow

1. Initialize the controller environment with PadInitDirect(), PadInitMtap().
2. Set up the receive buffer with PadSetAct(...).
3. Begin communication with the controller with PadStartCom().
4. **DUAL SHOCK only:** When connection with the controller has been established, use PadSetActAlign() to set up the sequence to transmit actuator control data.

Identifying the connected controller and Obtaining actuator (vibrator) information

1. **Identify the controller:** Call PadGetState() to determine the connection state of the controller. When it is known that the controller has a vibration function, use PadInfoMode(port, InfoModeCurExID,0) to determine the controller ID.
2. **Obtain actuator information:** When the Controller ID (value obtained from PadInfoMode(port, InfoModeCurExID, 0)) is different, the number of actuators and the type are also different. When calling PadSetActAlign(), the actuator functions from PadActInfo() and the controller ID should be confirmed.
3. **Handle controller swapping, controller mode switching**: The return value from PadGetState() changes when controllers have been swapped or the controller mode has been switched (see below). Controller swapping and controller mode switching can be continuously monitored using this function.
4. **Monitor the controller connection state**: The return value from PadGetState() changes only during the vertical retrace interval, so the value need be polled only once per frame.

Changes in the return value from the controller connection state function (PadGetState())

1) DUAL SHOCK controllers

1. Controller not connected: PadStateDiscon

   ↓

2. Controller connection detected: PadStateFindPad

   ↓

3. Request for actuator information received: PadStateReqInfo

   ↓

4. Retrieval of actuator information completed: PadStateStable

If the controller changes mode in states 3 - 4, a transition is made to state 2.

2) Other controllers

1.   Controller not connected: PadStateDiscon

   ↓

2.   Controller connection detected: PadStateFindPad

   ↓

3.   Identify controller type: PadStateFindCTP1

If the controller changes mode in state 3, a transition is made to state 2.

Using controllers with a Memory Card

When using a Memory Card, the controller should be initialized after the Memory Card is initialized. The initialization parameter should be set to '0', as in InitCARD(0). Functions PadInitDirect(), PadInitMtap() should be called before calling PadStartCom().

```
InitCARD(0);
StartCARD();
_bu_init();
PadInitDirect();
PadSetAct(...);
PadStartCom();
```

The parameter of MemCardInit() should also be set to 0 when using the simple Memory Card library (libmcrd). MemCardInit(0) and MemCardStart() should be called before the controller is initialized.

Using the Multi Tap with Memory Cards

When more than one Memory Card is connected to a single Multi Tap, and a Memory Card access is performed after any of the Memory Cards are switched, MemCardAccept() should be called for each Memory Card that is accessed. This is because libmcrd keeps a single directory information buffer for each port on the main PlayStation unit, so only one directory information set can be controlled when more than one Memory Card is connected to a single Multi Tap.

Precautions

Limitations of the Analog Controller

This section discusses some limitations of the Analog Controller.

1) Margin of error for the stick center position

When the Analog Controller stick is released, it tries to return to the center position. However, depending on the position where the stick is released, it might actually return to an off-center position. In applications where stick release is determined from stick position information, it is also necessary to take into account the center position error. In the SCPH-1200, the guaranteed value for the range within which the stick will return to the center when released is 80 h +/- 25 h.

2) Number of actuators that can be used simultaneously

The Analog Controllers have a vibration feature, but the number of actuators that can be vibrated simultaneously is limited. This limit is determined by the maximum available current drain from the PlayStation. When using the vibration feature, the following rules should be followed.

The actuators should be operated so that the total current drain does not exceed 60 units. For actuator 2, any value other than "0x00" is interpreted as "ON".

The current drain for each of the actuator types is given below.

Table 13-14: Actuator Current Drain

Model	Actuator type	Current drain
SCPH-1150	Actuator 1	10 units
SCPH-1200	Actuator 1	10 units
SCPH-1200	Actuator 2	20 units

If the total current drain of an actuator exceeds 60 units, a limiter in the library is activated to prevent that actuator from operating.

Actuators are prioritized first by port number, then by actuator number. The following examples illustrate this.

Example 1: Assume that controllers are connected to ports 00, 01, 02, 03, and each actuator has a current drain of 20 units. A request is made to activate actuator 1 in all controllers.

The total current drain is obtained by summing the individual actuator current drains in sequence, by port number. Since the total current drain exceeds 60 units, the actuator request for the exceeding controller is denied. The actuators for ports 00, 01, 02 are activated, but the actuator on port 03 is not activated.

Example 2: Requests are received for:

Actuators 1, 2 for port 00	(10, 20 units)
Actuator 2 for port 01	(20 units)
Actuator 2 for port 02	(20 units)
Actuator 1 for port 03	(10 units)

The total current drain for ports 00, 01 is 50 units. The request for port 02 would exceed 60 units, so it is denied. However, if port 03 is included, the total current drain doesn't exceed 60 units and consequently, the request from port 03 is granted.

Precautions when transmitting data to the controller during specific frames

Theoretically, communication with the controller should take place in each Vsync. However, this may not happen when there are frequent, intensive interrupts, such as during streaming.

If PadChkVsync() is called during a frame when communication with the controller has occurred, a value of 1 is returned. If PadChkVsync() is called more than once and no communication has taken place with the controller, 0 is returned. The data stored in the transmit buffer will generally be sent during the next Vsync. A return value of 0 from PadChkVsync() indicates that the data from the previous frame was not sent to the controller. When data is transmitted to the controller only during particular frames, the return value of PadChkVsync() should be checked to ensure that the data was actually sent.

Calling PadInitDirect(), PadInitMtap(), and PadInitGun()

In libpad, controller connection state is maintained by the library. If the controller connection state is invalid, the controller will not be recognized. Therefore, when a controller is used by both parent and child processes, each process must call PadInitDirect() or PadInitMtap().

Gun connection state is also maintained by the library. If the gun connection state is invalid, gun position information cannot be obtained. Therefore, when both parent and child processes use an ID=3 gun, for example, each process must call PadInitGun().

Multi Tap Library

It is possible to use up to 4 controllers and Memory Cards cards for 1 port with a Multi Tap.

Library and Header Files

To use the Multi Tap library, you must link with the file `libtap.lib`.

Source code must include the header file `libtap.h`.

Overview

The communication can be performed if at least one controller is connected to the Multi Tap. If no controller is connected, a communication error will occur.

The communication is available only with the port A of the Multi Tap in the software which doesn't use libtap.lib. In this case, the communication data will be just passed through the Multi Tap in the same way the controller is connected directly to PlayStation.

The insertion and extraction of the Multi Tap and the controller connected to the Multi Tap are permitted during the operation.

Table 13-15: Receiving Packet Format

Byte	Content
0	Result of receiving
1	ID (0x80)
2	Controller_A Result
3	Controller_A ID
4-9	Controller_A Data
10	Controller_B Result
11	Controller_B ID
12-17	Controller_B Data
18	Controller_C Result
19	Controller_C ID
20-25	Controller_C Data
26	Controller_D Result
27	Controller_D ID
28-33	Controller_D Data

The access to the Memory Card is performed in the same way as the usual operation. The channel is specified with the "port number x16 + card number", and by setting from 0 to 3 for the card number, the access to each slot is available.

Table 13-16: Memory Card

	Port 1	Port 2
A	0x00	0x10
B	0x01	0x11
C	0x02	0x12
D	0x03	0x13

Caution

When using the Memory Card, the InitCARD argument must be set at '0'.

```
InitTAP(bufA, lenA, bufB, lenB);
InitCARD(0);
StartCARD();
_bu_init();
StartTAP();
ChangeClearPAD(0);
```

When the Multi Tap is not inserted into port A, it is sometimes not recognized. Also, please follow the instructions described in the usage manual.

Gun Library

Library and Header Files

To use the gun library, link with the file `libgun.lib`.

Source code must include the header file `libgun.h`.

Button Data

The lbgun initialization routine is

```
InitGUN(char *bufA, char*lenA, char*bufB, long lenB, char *buf0, char *buf1,
long len)
```

Table 13-17 shows the format of the bufA, bufB structures. Table 13-19 shows the format of the buf0, buf1 structures.

Table 13-17: Button Data (bufA, bufB)

Byte	Contents
0	Receive result
1	ID:
	Higher 4 bits - Controller type
	Lower 4 bits - Number of bytes of receive data / 2
	(When the lower 4 bits are all 0, it means the number of bytes of receive data is 32 bytes.)
2,3	Button data

1. ID

    The gun ID is 0x31 (Controller type: 3, Number of bytes of receive data: 2)

2. Button data

Table 13-18

	7	6	5	4	3	2	1	0
First byte	-	-	-	-	S	-	-	-
Second byte	□	x						

S: Start button
□: Trigger of gun
x: Button

Location Data in the Horizontal/Vertical Direction on the Screen

Table 13-19 shows the format of the buf0, buf1 structures.

The maximum number of receive data is 20.

In order to increase the accuracy of the gun, DMA and interrupt processing are all blocked within the gun interrupt processing from library 4.0 onwards. Since the overhead increases in Hsync units when the number of interrupts rises, it is recommended that it be set to a small number.

Table 13-19: buf0, buf1 structures defined in InitGUN

Byte	Contents
0	Not used
1	Number of available horizontal/vertical direction counter value
2,3	Vertical direction counter value 0
4,5	Horizontal direction counter value 0
6,7	Vertical direction counter value 1
8,9	Horizontal direction counter value 1
.	.
.	.
.	.
78,79	Vertical direction counter value 19
80,81	Horizontal direction counter value 19

(The counter value is given as a half word.)

Correction to Location Data in the Horizontal Direction on the screen

The horizontal direction location data currently returns the system clock value which zero clears every H blank. Therefore, it is necessary to make adjustments in accordance with the screen mode and horizontal resolution. The following table displays system clock and pixel clock compatibility:

Table 13-20: System Clock/Pixel Clock Conversion

Mode	Horizontal direction resolution	Coefficient
NTSC	256	0.158532
	320	0.198166
	384	0.226475
	512	0.317065
	640	0.396332

Mode	Horizontal direction resolution	Coefficient
PAL:	256	0.157086
	320	0.196358
	384	0.224409
	512	0.314173
	640	0.392717

[Pixel value] = [Coefficient] x [System clock value] + [Offset]

Memory Card

When using the Memory Card, pass 0 to InitCARD():

```
InitGun (bufA, lenA, bufB, lenB, buf1, buf2, len);
InitCARD(0);
StartCARD();
bu_init();
StartGUN();
ChangeClearPAD(0);
```

Chapter 14:
Link Cable Library

Table of Contents

CONFIDENTIAL

Overview

The Link Cable library (libcomb) provides services for connecting PlayStations with a link cable, which is used by many games, especially combat games.

Communication is performed by the read() and write() functions. Both functions support asynchronous mode, in which events occur when processing is complete, and synchronous mode, in which the functions are terminated when communication is complete. The maximum communication rate is 2M (2073600) bps. The communication method is asynchronous serial communication. Use _comb_control() to change the communication rate.

Library and Header Files

To use the link cable library, you must link with the library file `libcomb.lib`.

Source files must include the header file `libcomb.h`.

Driver and BIOS

The link cable library consists of the link cable driver and the link cable BIOS.

Link Cable Driver

The link cable driver allows you to use Standard C input/output functions.

To install the link cable driver, call AddCOMB(). To remove it, call DelCOMB(). Opening the sio device without installing the driver will cause an error.

Table 14-1: Link Cable Driver

Item	Contents
Device name	sio
Block size	1,2,4,8 bytes Asynchronous write is a 1 byte unit
Asynchronous mode	Specified in the O_NOWAIT macro on opening

Events

The following events occur with the input/output of the driver.

EvSpIOEW and EvSpIOER occur after sending asynchronous input/output requirements. EvSpERROR occurs in both synchronous reading and asynchronous reading.

Table 14-2: Events

Cause descriptor	Event type	Contents
HwSIO	EvSpIOEW	Completion of asynchronous writing
	EvSpIOER	Completion of asynchronous reading
	EvSpERROR	Read error
	EvSpTIMOUT	Timeout in synchronous reading/writing

Wait Callback

During the processing of synchronous reading/writing of the preceding data a test software loop of the serial controller status is executed. The Wait callback function is called during this loop. The callback function is not registered in the default state.

_comb_control (4,0,func) sets func() as a callback function. It must meet the following specifications:

```
Syntax          long func (long spec, unsigned long count)
Argument        spec  1:during sychronous read  2:during synchronous write
                count     current location of internal counter
Return Value    Returns 0 when the wait loop is timed out and
                returns 1 when the wait continues
```

The callback function registration can be cancelled by _comb_control(4,0,NULL).

Termination Conditions for Synchronous Input/Output

A synchronous read terminates when the specified number of characters are received. It can also terminate when a parity overrun frame receiving error is detected or when the wait callback function returns a prescribed value. In either case a unit-number of receiving characters is returned.

A synchronous write terminates when the specified number of characters are transmitted. It can also terminate when the wait callback function returns a prescribed value.

Interrupt and Read/Write Functions

Most libcomb functions are designed never to enter a critical section, and it is possible for them to be called within an event handler. **Note:** since single-threaded operation is assumed, perfect operation in a multi-thread environment is not guaranteed.

Although there is no problem when the existing PlayStation library is operated by a single thread, the library driver's operation cannot be guaranteed when the original thread control is being carried out.

Number of Characters for Receiving

The PlayStation is equipped with an 8-byte receive buffer, and an interrupt can be set to occur when 1,2,4 or 8 bytes are received when using in the asynchronous read package.

It is also used as the number of characters for DSR/DTR handshaking in synchronous communication.

Note: When performing an asynchronous write, this value must be set to 1, since an interrupt is expected for each byte transmitted.

Error Processing

When detecting each overrun parity frame receiving error during asynchronous input/output, _comb_control(2,3,0) will cancel the asynchronous read will then issue the next EvSpERROR event.

During an event handler, only short processes such as flag setting should be done. Error processing itself must take place in the main process.

CONFIDENTIAL

BIOS

The link cable BIOS provides precise driver control beyond that provided by standard C language functions. The interface function is _comb_control(). The BIOS will work without installing the driver.

Following is the explanation of the _comb_control() function.

```
Syntax:
        long _comb_control(unsigned long cmd, unsigned long arg,
                unsigned long param)

Arguments:
                cmd     command
                arg     subcommand
                param   argument
```

Table 14-3: Command Summary

cmd	arg	Function
0	0	Returns the serial controller status (see Table 14-4)
0	1	Returns the control line status (see Table 14-5)
0	2	Returns the communication mode (see Table 14-6)
0	3	Returns the communication rate in bps
0	4	Returns the "unit-number of characters for receiving"
0	5	Returns the amount of remaining data (bytes) from asynchronous input/output during processing. If the param is 0 it is asynchronous write, if 1 it is asynchronous read.
0	6	Returns an asynchronous input/output request whether it registered or not. If it has been registered, it will return 1. Others will return 0. If the param is 0, it is asynchronous write, if 1 it is asynchronous read
1	0	System reserved
1	1	Sets the value of param as the control line status (*2)
1	2	(Reserved)
1	3	Sets the value of param as the communication rate by bps
1	4	Sets the value of param as the "unit-number of characters for receiving"
2	0	Resets the serial controller. Controller status, communication mode and communication speed are saved.
2	1	Clears the bits related to the driver status error. Includes a function which indicates the completion of the interrupt processing to the driver.
2	2	Cancels the asynchronous writing
2	3	Cancels the asynchronous reading
3	0	When param is 1 RTS is made 1. When param is 0, RTS is made 0.
3	1	If (CTS==1) 1 is returned, the others return 0
4	0	The param value is considered to be the pointer to the function and is registered as the pointer to the wait callback function. The callback function pointer values up to that point are returned

Table 14-4: Driver Status

bit	Contents
31-10	Undefined
9	1: Interrupt is ON
8	1: CTS is ON
7	1: DSR is ON
6	Undefined
5	1: Frame error occurrence
4	1: Overrun error occurrence
3	1: Parity error occurrence
2	1: No sending data
1	1: Possible to read the receiving data
0	1: Possible to write the sending data

Table 14-5: Control Line Status

bit	Contents
31-2	Undefined
1	1: RTS is ON
0	1: DTR is ON

Table 14-6: Communication Mode

bit	Contents
31-8	Undefined
7,6	Stop bit length
	01:1
	10:1.5
	11:2
5	Parity check(2) 1: odd number 0: even number
4	Parity check(1) 1: enabled
3,2	Character length
	00:5 bits
	01:6
	10:7
	11:8
1	1 at all times
0	0 at all times

Serial Controller

The device that drives the link cable connector is a serial controller that supports asynchronous communication. It has a 1-byte transmission buffer and an 8-byte receiving buffer. It has two sets of control lines: DTR/DSR and RTS/CTS. Both are used for synchronous read()/write().

Table 14-7: Control Line

Transmission Name	Receiving Name	Receiving Function	Transmission Interrupt
DTR	DSR	Unusable during communication	None
RTS	CTS	Receiving functions automatically halt when OFF	None

Communication Specifications

Communication specifications can be selected from the following settings:

Table 14-8: Communication Specifications

Item	Range of Values	Default settings
Character Length	8 bits	8 bits
Stop Bit	1-2 bits	2 bits
Parity Check	None	Disabled
Communication Rate	1~2073600bps (2073600 divisor only)	9600 bps

Control Line Transition

The driver operates DTR (DSR for receiving) and RTS (CTS for receiving) as follows:

Table 14-9: Control Line Transition

Driver Operation	DTR	RTS
Power On (No other PlayStation or other PlayStation		
power supply off)	1	1
(Other PlayStation present, driver not initialized)	0	0
Driver Initialization		
AddCOMB():	0	0
Synchronous Write		
open (:sio", O_WRONLY);	-	-
write(...);	-	-
write completion	-	-
close();	-	-
Synchronous Read		
open ("sio", O_RDONLY);	-	-
read (...)	-	1
read completion	-	0
close();	-	-

Driver Operation	DTR	RTS
Asynchronous Write		
open("sio", O_WRONLY/NOWAIT)	-	-
write (...);	-	-
transmission interrupt occurrence	-	-
transmission interrupt completion	-	-
write completion	-	-
close();	-	-
Asynchronous Read		
open("sio", O_RDONLY/O_NOWAIT);	-	-
read(...);	-	1
Received interrupt occurrence	-	1
Received interrupt completion	-	1
Read completion	-	0
close();	-	-

Programming Hints

Detecting the Other PlayStation's Connection (1)

When the link cable is not connected, or when it is connected but the other PlayStation has no power, both the DSR and CTS become 1. However, if the link cable is connected *and* the other PlayStation has power, both DSR/CTS become 0; this does not change even after the driver has been installed according to AddCOMB(). Since no internal operation is carried out as long as the DSR/CTS signal does not issue a read, there is a method which by provisionally making either DTR or RTS 1 in the initialization process immediately after AddCOMB() execution provides notification that the other PlayStation has completed communication preparations. By enabling the other PlayStation to perform the same process, both sides can confirm the completion of each other's communication preparations. However, in order to prevent influencing future read/write functions when using RTS, it must be returned to 0.

Detecting the Other PlayStation's Connection (2)

When the link cable is disconnected during communication, or when the power to the other PlayStation is turned off, it's necessary to use a detection algorithm that establishes a time restriction to detect the occurrence of an unusual state in the other PlayStation.

A simpler method would use a wait callback to constantly monitor the other PlayStation's responses and infer the status of the other PlayStation from the existence of a response or its speed. However, this information alone is insufficient to determine whether abnormal status of the other PlayStation results from a lack of connection or from a communication error. If the status has not improved, even after exhausting all countermeasures such as performing the transfer again, it's necessary to determine whether the problem is a lack of connection. Either way, since there is no definitive way to detect the connection with the other PlayStation, it is necessary to use a time restriction-based detection algorithm.

Background Receiving by the Ring Buffer

The read function can be executed in a critical section. Therefore, by calling the read function in the EvSpIOER event handler, the next asynchronous receiving request can be registered to the driver. The ring buffer can easily receive in the background by operating the receive buffer pointer provided in the read function.

Since the write function can also be carried out in a critical section, it is possible to package processing such as a retransmitted request with the ring buffer operation code by testing the received data contents.

Slow Speed of Asynchronous Write

Asynchronous transmission carries the highest load in this library and driver operation. If receiving is not given priority over transmission, a receive error such as an overrun can occur, and this is generally attributable to a decrease in efficiency.

In principle it is unavoidable that the efficiency of asynchronous communication is lower than synchronous communication. Of all the combinations (synchronous read/write and asynchronous read/write), asynchronous write puts the heaviest load on the CPU. Because asynchronous write exhibits performance at a lower baud rate, normal operation cannot be expected for transmissions which exceed 57600bps in actual practice.

Lightest Overhead Transmission

To produce the lightest load on the CPU, divide data to be transmitted into 8-character packets, scattered suitably within the code and transmitted by synchronous write. Since each write should conclude within a definite time period, the timeout callback function sets the maximum waiting time for all write functions. When the time limit is reached, the transmission is interrupted. Since the transmission end character number for the point at which the interrupt occurred is obtained as the return value, the pointer which shows the transmission data provided to the next write function revises the value to the original.

Unit-Number of Characters for Receiving with the Exception of One Character

When an asynchronous read request is issued as the value of the unit-number of characters for receiving except for one character, data which does not satisfy the number of characters cannot detect the received status driver. Since this status connects to deadlock depending on the transmission-side activity, it is necessary to package the time out processing at the application level.

CONFIDENTIAL

Chapter 15:
Extended Sound Library

Table of Contents

CONFIDENTIAL

Overview

The Extended Sound library (libsnd) provides services that convert sound data so that it can be used by the PlayStation. It can work with files created by the dedicated PlayStation Sound Artist Tool.

Libsnd provides functions for:

- Accessing VAB (sound source) data.
- Activating and terminating the sound system.
- Handling music score (SEQ) data.
- Producing single sound sound effects, rather than musical sound effects.
- Setting the common attributes of each SPU voice.
- Changing the attribute table in VAB data at run-time and applying effects to the allocated voice after KeyOn.

Note: Libsnd is designed to use MIDI data. For sound effects and music which do not use MIDI data, use of libspu is recommended since it is smaller and uses less overhead.

Library and Header Files

To use the extended sound library, your application must link with the file `libsnd.lib`.

Your source code must include the header file `libsnd.h`.

Score Data

In libsnd, music data is defined in the SEQ and SEP data formats.

SEQ Data Format

SEQ is a format 1 Standard Midi File (SMF) converted for use with the PlayStation. In the SEQ format, the MIDI data structure track/chunk data is merged with the time order.

A single sound expression is the same as the SMF standard, that is:

- status (1 byte)
- data (number of bytes fixed according to status)
- delta time (variable length expression, max 4 Bytes)

Figure 15–1: SEQ data format

Sound ID (4 bytes)
Version number (4 bytes)
Quarter-note resolution (2 bytes)
Tempo (3 bytes)
Rhythm (2 bytes)
Data
File end (3 bytes)

In SEQ, use running status and all note off messages should be note on messages with velocity 0. SEQ format also supports the following status data used by MIDI.

- Note on
- Note off
- Program change
- Pitch bend

The list below is for control change:

- Bank Change (0)
- Data entry (6)
- Main volume (7)
- Panpot (10)
- Expression (11)
- Damper pedal (64)
- External effect depth (91)
- Nrpn data (98, 99)
- Rpn data (100, 101)
- Reset all controllers (121)

Note: Control numbers are printed inside parentheses ()

SEP (Sequence Package) Data Format

A SEP is a package containing multiple SEQ data files. SEPs enable multiple SEQ data files to be managed as one file. A maximum of 16 SEQ data files can be linked.

SEPs can be accessed by specifying the ID number returned when the SEP is opened, along with the SEQ number of the SEQ data to be accessed. See the *Run-Time Library 4.0/Reference* for details of access-related functions.

The SEP data format is illustrated below.

CONFIDENTIAL

Figure 15–2: SEP data format
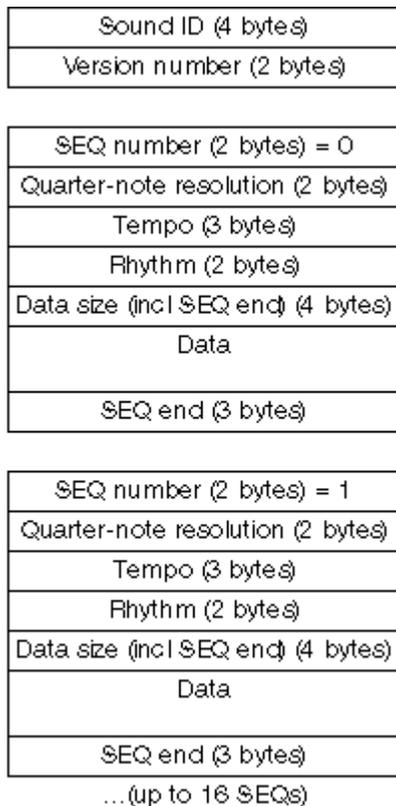
MIDI Support

Setting VAB Attribute Data Using Control Change

NRPN data that enables the setting of VAB attribute data is defined using the MIDI standard Control Change message for the NRPN.

When using a sequencer to create an SMF file for defining VAB attributes, the following values should be sent.

```
bnH        99        data1        (NRPN MSB)
bnH        98        data2        (NRPN LSB)
bnH        06        data3        (Data Entry)
```

The contents of data1, data2, and data3 are described below.

- Tone numbers range from 0 to 15.

- A value of 16 sets the attributes of all tones. Some values, such as reverb depth and feedback amount, must be set for the entire SPU; they can't be set for each tone or each MIDI channel.

- Reverb can be set only as on or off for each voice (i.e., each waveform). To make these settings, check the reverb switches shown on the SoundDelicatessen ADSR setting screen. You can also use the NRPN Mode setting to change from MIDI sequence in real-time.

Table 15-1: Data1-Data3 Contents

ATTRIBUTE	Data1 (CC99)	Data2 (CC98)	Data3 (CC06)
Priority	Tone Number	0	0~127
Mode	"	1	0~4 (*)
Limit low	"	2	0~127
Limit high	"	3	"
ADSR (AR-L)	"	4	"
ADSR (AR-E)	"	5	"
ADSR (DR)	"	6	"
ADSR (SL)	"	7	"
ADSR (SR-L)	"	8	"
ADSR (SR-E)	"	9	"
ADSR (RR-L)	"	10	"
ADSR (RR-E)	"	11	"
ADSR (SR-±)	"	12	0~64: + 65~127:–
Vibrate time	"	13	0~255(**)
Portamento depth	"	14	0~127(**)
Reverb type	16	15	0~9 (***)
Reverb depth	16	16	0~127
Echo feedback	16	17	"
Echo delay time	16	18	"
Delay delay time	16	19	"
Vibrate depth	Tone Number	21	0~127(**)
Portamento time	"	22	0~255(**)

(*) Mode Type

(**) Not currently supported

(***) Reverb Type (Refer to Sound Delicatessen DSP)

Table 15-2: Data3 Mode Type

Number	Mode
0	Off
1	Vibrate
2	Portamento
3	1&2 (Portamento and Vibrate on)
4	Reverb

CONFIDENTIAL

Table 15-3: Data3 Reverb Type (See Also Sound Delicatessen DSP)

Number	Reverb Type
0	Off
1	Room
2	Studio A
3	Studio B
4	Studio C
5	Hall
6	Space
7	Echo
8	Delay
9	Pipe

Using Control Changes to Set Repeating Loops within Music

NRPN data may be used to implement a repeat function for sections within music.

The symbol "‖:" identifies Loop1 and ":‖" identifies Loop2. Although the repeat function can be used any number of times within one piece of music, it is not possible to embed a loop within a loop, such as (Loop1 ... (Loop1' ... Loop2') ... Loop2).

Table 15-4: Looping Using Control Changes

ATTRIBUTE	Data1 (CC99)	Data2 (CC06)
Loop1(start)	20	0~127 (***)
Loop2(end)	30	

(***) For continuous looping, set 127(0x7f).

Note: Don't set repeat loops to the same Delta Time, because the data may be invalid. Depending on the sequence, the order can shift when it is modified to SMF even if it was input in regular sequence.

Also, values become valid from the KeyOn immediately after the Data Entry is read in VAB attribute data settings.

Marking Function Using Control Changes

NRPN data can be used to mark places in a song. When a library function detects one of these marks, it calls the function registered for the mark. The marking format is shown below.

Table 15-5: Marking via Control Changes

Attribute	Data1 (CC99)	Data2 (CC06)
Mark	40	Any value from 0~127
		(Passed to callback function)

Note: Please set the reverb and repeat at only one point in the music score data. There is no need to set them in each channel (track).

VAB Switching Using Control Change

Using bank change (CC 0) makes switching to any VAB possible.

The CC 0 must be able to use a VAB ID. Entering an incorrect VAB ID can cause malfunctions such as no sound production.

Although a bank change generally sets CC 0 and CC 32, CC 32 can be omitted since it is ignored in libsnd.

Figure 15-3: VAB Switching Using Control Changes

Attribute	data1 (CC 0)	data2 (CC 32)
VAB Change	Any VAB ID from 0-15	Ignored

When not performing VAB switching, CC 0 and CC 32 should be deleted from the MIDI data in order to avoid possible malfunctions. Setting the OutputMode -> Remove BankChange(#0) check to "ON" when converting to SEQ with SMF2SEQ will allow bank change to be deleted from the MIDI data and converted.

Sound Data

Two data formats are used to define sound data, VAG format and VAB format.

VAG Format

This is a waveform data format for ADPCM-encoded sampled sounds, such as piano sounds and explosions.

VAB Format

The VAB file format is designed to manage multiple VAG files as a single group. A VAB file contains all of the sounds, sound effects, and other sound-related data actually used in a scene. Hierarchical management is used to support multi-timbral (multi-sampling) functions.

Each VAB file is equivalent to a MIDI bank, and may contain up to 128 programs, which are equivalent to MIDI patch changes. Each program can contain up to 16 tone lists. A tone list is a set of attributes for a specific VAG. Also, each VAB file can contain up to 254 VAG files.

Since it is possible for multiple tone lists to reference the same waveform, users are able to set different playback parameters for the same waveform.

A VAB format file is organized as follows:

Figure 15–4: VAB format and VAB header

VAB File

Program Attribute Table
(128 Programs)

Tone Attribute Table (128
x 16 VAG Header List)

VAG Data Body
(254 VAG Data)

CONFIDENTIAL
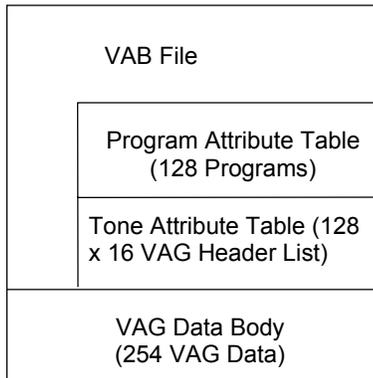
The structure of a VAB header is as follows. It is possible to set each attribute dynamically using this structure at the time of execution. Also, the VAB (Bank) editor can edit all values included in the VAB data format header and can confirm the local memory usage by using the bank sound source at execution.

VAB Header

```
struct VabHdr {
      long            form;       /*always "VABp"*/
      long            ver;        /*format version number*/
      long            id;         /*bank ID*/
      unsigned long   fsize;      /*file size*/
      unsigned short  reserved0;  /*system reserved*/
      unsigned short  ps;         /*total number of programs in this bank*/
      unsigned short  ts;         /*total number of effective tones*/
      unsigned short  vs;         /*number of waveforms (VAG)*/
      unsigned char   mvol;       /*master volume*/
      unsigned char   pan;        /*master pan*/
      unsigned char   attr1;      /*bank attribute*/
      unsigned char   attr2;      /*bank attribute*/
      unsigned long   reserved1;  /*system reserved*/
      };
```

Program Attributes

```
struct ProgAtr {
      unsigned char   tones;      /*number of effective tones which
                                    compose the program*/
      unsigned char   mvol;       /*program volume*/
      unsigned char   prior;      /*program priority*/
      unsigned char   mode;       /*program mode*/
      unsigned char   mpan;       /*program pan*/
      char            reserved0;  /*system reserved*/
      short           attr;       /*program attribute*/
      unsigned long   reserved1;  /*system reserved*/
      unsigned long   reserved2;  /*system reserved*/
      };
```

Tone Attributes

```
struct VagAtr {
      unsigned char  prior;       /*tone priority (0 – 127); used for
                                    controlling allocation when more voices
than
                                    can be keyed on are requested*/
      unsigned char  mode;        /*tone mode (0 = normal; 4 = reverb
                                    applied */
      unsigned char  vol;         /*tone volume*/
      unsigned char  pan;         /*tone pan*/
      unsigned char  center;      /*center note (0~127*/
      unsigned char  shift;       /*pitch correction (0~127,cent units)*/
      unsigned char  min;         /*minimum note limit (0~127)*/
      unsigned char  max;         /*maximum note limit (0~127,
                                    provided min < max)*/
      unsigned char  vibW;        /*vibrato width (1/128 rate,0~127)*/
      unsigned char  vibT;        /*1 cycle time of vibrato (tick units)*/
      unsigned char  porW;        /*portamento width (1/128 rate, 0~127)*/
      unsigned char  porT;        /*portamento holding time (tick units)*/
      unsigned char  pbmin;       /*pitch bend (-0~127, 127 = 1 octave)*/
      unsigned char  pbmax;       /*pitch bend (+0~127, 127 = 1 octave)*/
      unsigned char  reserved1;   /*system reserved*/
      unsigned char  reserved2;   /*system reserved*
```

```
unsigned short adsr1;          /*ADSR1*/
unsigned short adsr2;          /*ADSR2*/
short          prog;           /*parent program*/
short          vag;            /*waveform (VAG) used*/
short          reserved[4];    /*system reserved*/
};
```

Function Execution Sequence

When using libsnd, execute the functions in the following order. (See sample programs for more details.)

1) Initialization

Initialize the system with SsInit(). Use SsSetTableSize() to maintain the SEQ attribute data area.

2) Tick Mode Setting

Set the tick mode with SsSetTickMode().

3) Opening Data

- VAB data: SsVabOpenHead()-->SsVabTransBody(),SsVabTransCompleted()
- SEQ data: SsSeqOpen()
- SEP data: SsSepOpen()

4) Starting the Sound System

Call SsStart() to start the sound system. Opening Data above may be executed after SsStart().

Playback of SEQ data is dependent on proper callback timing. Other callback processing (such as CD reading) can cause the tempo of the music to become uneven unless the following methods are followed:

Case A: not using VSyncCallback()

1. Define TICK mode as SS_TICK60
    ```
    SsSetTickMode (SS_TICK60);
    ```

2. Use SsStart2() instead of SsStart()
    ```
    /*SsStart(); /* tempo changes */
    SsStart2();
    ```

Case B: using VSyncCallback()

1. Define TICK mode as SS_NOTICK
    ```
    SsSetTickMode (SS_NOTICK);
    ```

2. Call SsSeqCalledTByT() within the callback function set by VSyncCallback().
    ```
    int foo (void)
    {
            ...
            SsSeqCalledTByT();
            ...
    }
    ```

    Set the processing load corresponding to the position of SsSeqCalledTByT() in the function.
    ```
    main()
    {
            ...
            VSyncCallback (foo);
            ...
    }
    ```

Either solution will currently work for this problem, but from this point on, it would be better to use a TICK mode less than SS_TICK120.

5) Required Processing

Set main volume. Execute required processing. Use libcd function CdMix() to make CD(DA/XA) stereo/monoaural settings. Use SsSetMono() and SsSetStereo() to make SPU (SEQ, SEP, VAB, VAG) stereo/monaural settings.

6) Closing Data

*   VAB data: SsVabClose()
*   SEQ data: SsSeqClose()
*   SEP data: SsSepClose()

7) Halting the Sound System

Halt the sound system with SsEnd().

8) Terminating the Sound System

End the sound system with SsQuit().

CONFIDENTIAL

Chapter 16:
Basic Sound Library

Table of Contents

CONFIDENTIAL

Overview

The basic sound library (libspu) directly controls the PlayStation sound play processor (SPU). It controls the lower levels of the extended sound library (libsnd), and provides individual functions for operations such as transferring non-music data (texture data, etc.) to the sound buffer.

Libspu does not have time control functions; they are provided by libapi.

It is necessary to insert the SPU processing unit of at least 1/44100 seconds of space in order to perform function calling for the setting of identical functions.

Library and Header Files

To use the basic sound library, your application must link with the file `libspu.lib`.

Your source code must include the header file `libspu.h`.

VAG Format

VAGs are compressed audio data arranged in 16-byte blocks.

Note: AIFF2VAG for the PC creates files in an Intel or little-endian format, while AIFF2VAG for the Mac creates files in a Motorola or big-endian format.

Header

All VAGs have a 48 byte header, which must be removed for playback. This header should not be removed before converting VAGs to VABs on the Mac or PC; otherwise, improper conversion will occur.

- ID - 4 bytes. 'VAGp', identifies the file as a VAG.
- Version - 4 bytes. Identifies which version of AIFF2VAG created the file.
    — Mac converters
        v1.3   '00000002'
        v1.6+   '00000003'
    — PC converters
        -v1.8   '00000000'
        v2.0+   '00000020'
- System reserved - 4 bytes.
- Data size - 4 bytes. The data size of the file in bytes.
- Sampling frequency - 4 bytes. The sampling frequency of the AIFF. Can be used to determine the pitch at which to play the VAG. pitch = (sampling frequency <<12)/44100L Ex: 44.1kHz=0x1000 22.05kHz=0x800 etc.
- System reserved - 12 bytes.
- Name - 16 bytes. File name, used by Sound Delicatessen.

Intro

All VAGs must have a lead-in of 16 bytes of zero data. This data initializes the SPU in order to prevent clipping noises.

VAG Body

The VAG format and compression method is Sony proprietary information. The body of the VAG will be compressed approximately 3.5-1 by AIFF2VAG.

SPU IRQ Clear Block

One-shot VAGs will be created with an additional 16-byte block attached to the end. The block is used to prevent unnecessary SPU interrupts or SPU free-run. The block reads as follows: "00077777 77777777 77777777 77777777" or "00070000 00000000 00000000 00000000." Looping VAGs do not contain this block.

If the SPU IRQ is not being used, this block can be removed. Currently, the functions which use the SPU IRQ are: SpuGetIRQ(), SpuGetIRQAddr(), SpuSetIRQ, SpuSetIRQAddr(), and SpuSetIRQCallback(). The SpuStreaming library [all calls beginning with SpuSt…()] also uses the SPU IRQ.

If none of these functions will be used in the code, the SPU IRQ clear byte block at the end of one-shot VAGs can be removed. This frees up SPU RAM (up to 4K in a single VAB), slightly reduces CD load time (up to 1/75 sec at double speed for a single VAB), and very slightly reduces SPU load time (while the SPU DMA is slow, it is much faster than the CD).

Be sure to change the data size in the VAG header to reflect the fact that these bytes have been removed BEFORE building VABs with truncated VAGs.

Also, keep in mind that your SPU IRQ safety net has been removed with the removal of the clear bytes. Use this information wisely.

Voice Audio Source

Libgpu can control the following attributes, which may be set individually for 24 ADPCM audio sources (*voices*):

- Sound volume (can set L/R independently)
- Pitch
- Address of waveform data in sound buffer
- Envelope (ADSR)
- Loop point

Key on/key off can also be set independently for each of the 24 voices.

These attributes may be changed while key on is in effect and sound is being generated. Therefore, it is possible to continuously vary the sound interval during sound generation and to repeatedly generate sound while changing the loop point of waveform data having a loop point.

Noise Audio Source

The SPU has one noise generator, which may be set and used for each voice instead of sound buffer waveform data. Use SpuSetNoiseVoice() to determine which voices will playback the noise generator. It has effects such as envelope, and it can produce a noise sound effect by varying the auditory sound interval (noise clock) while sound is being generated. SpuSetNoiseClock() can be used to change the sound interval. SpuGetNoiseClock() returns the value of the interval and SpuGetNoiseVoice() returns the voices currently using the noise generator.

LFO in Intervals

By using adjoining voices, the SPU can produce a Low Frequency Oscillator (LFO) effect in an interval. Use SpuSetPitchLFOVoice() to create this effect, which can be expressed by the equation below. Be aware that two voices are used to generate one tone.

```
NewPitch(n) = (1 + V(n-1)) * Pitch(n)
```

Table 16-1: LFO Control Expression Format

NewPitch(n)	Voice (n) final pitch
V(n-1)	Voice (n-1) volume (changed according to time)
Pitch(n)	Pitch originally set for voice (n)

Reverb

Reverb is provided using various types of templates. These templates have many parameters that can be adjusted to vary the effects.

Reverb uses the sound buffer as its work area, with the starting address varying according to each parameter, which can be set inSpuSetReverbModeParam(). Since this is also prepared for use as a template, the area before the offset address may be used as a waveform data area.

Only one type of reverb can be active at a time. Reverb for individual voices may be turned on or off using SpuSetReverbVoice(); it also returns the voices that currently have reverb set. Reverb may also be applied to CD input and external digital input by using SpuSetCommonAttr() and setting the members SpuCommonAttr.*cd.reverb* or SpuCommon.*ext.reverb*.

Do not set the reverb depth with SpuSetReverbModeParam() or SpuSetReverbDepth() until reverb is actually required, or it will be necessary to clear the reverb work area with either SpuSetReverbModeParam() or SpuClearReverbWorkArea() to avoid noise being generated.

If you intend to use reverb, set the mode well in advance, not just before use. When you set the mode, the reverb depth goes to 0.

The order in which you perform reverb setup should be:

SpuSetReverb() followed by either

1. SpuSetReverbModeParam() (specifying Mode/Feedback/Delay/Depth)

or

2. SpuSetReverbModeParam() (specifying Mode/Feedback/Delay)

    SpuSetReverbDepth() (specifying Depth)

In order for SPU memory management to work properly with reverb, the following relationships should be applied:

1. Cases in which reverb work area has been reserved with SpuReserveReverbWorkArea (SPU_ON)

    SpuMalloc()/SpuMallocWithStartAddr(). This method should be used to save an area of SPU RAM for future reverb use.

    Depending on the mode, you can allocate an area of size (0x7ffff - work area size), starting from address 0x1010.

2. Cases in which a work area has not been reserved with SpuReserveReverbWorkArea (SPU_OFF)

    SpuMalloc()/SpuMallocWithStartAddr()

    Area can be allocated in the entire sound buffer area, addresses 0x1010 to 0x7ffff, unless SpuSetReverb(SPU_ON) has been called. In this case, even if reverb mode is SPU_REV_MODE_OFF, 128 bytes will be used as a reverb work area.

    SpuSetReverb()

    If an area with a size corresponding to the mode being used has been allocated as the reverb work area in another area with SpuMalloc()/SpuMallocWithStartAddr(), then SpuSetReverb (SPU_ON) will be invalid.

3. Regardless of the current reverb work area allocation, when a change is to be made to reverb mode, SpuSetReverbModeParam() analyzes whether or not it can allocate the area required as a work area, based on information from the sound buffer memory management mechanisms, and if possible reserves the area at that time. If the area cannot be allocated, SpuSetReverbModeParam() returns without reserving the area.

4. If you execute SpuMalloc()/SpuMallocWithStartAddr() when there is no reverb work area reserved by SpuReserveReverbWorkArea(), and afterward attempt to reserve the reverb work area again with SpuReserveReverbWorkArea(), it analyzes whether or not it can acquire a reverb work area of the size needed by the current reverb mode, based on information from the sound buffer memory management mechanisms, and reserves that region at that time if that area can be allocated. If that area cannot be allocated, it returns without reserving any work area.

5. The size of the reverb work area depends on the reverb mode. The only time that the reverb work area size changes is when you set the mode with SpuSetReverbModeParam().

    The behavior of SpuMalloc()/SpuMallocWithStartAddr(), SpuReserveReverbWorkArea(), and SpuSetReverb() change when the mode setting changes.

    When exiting a program that uses reverb, you must do the following:

    Basic Sound Library

    ```
    #include <libspu.h>

    SpuReverbAttr r_attr;
    r_attr.mask = (SPU_REV_MODE);
    r_attr.mode = SPU_REV_MODE_OFF;

    SpuSetReverbModeParam (&r_attr);
    SpuSetReverb (SPU_OFF); /*reverb off*/
    ```

    Otherwise, noise may sometimes occur the next time the program is executed.

Data Transfer Between Memory and Sound Buffer

You can transfer waveform data from main memory to the sound buffer with SpuWrite() and from the sound buffer to the main memory with SpuRead().

There are two transfer modes:

- DMA transfer (Write/Read). Transfers asynchronously using the DMA controller, so the CPU is able to do other processing during the transfer. The function SpuIsTransferCompleted() must be called after DMA transfer.
- I/O transfer (Write-only). Uses the CPU, so other processing cannot be performed during the transfer. You must select DMA transfer if you are transferring data while continuing playback. Since I/O transfer blocks CPU processing, SpuIsTransferCompleted() always returns 1, and need not be called.

The default mode is DMA transfer. Use SpuSetTransferMode() to change modes; use SpuGetTransferMode() to check transfer mode. SpuSetTransferStartAddr() must be called before writing or reading SPU RAM. SpuGetTransferStartAddr() returns the transfer address.

DMA transfer is always used when transferring from the sound buffer to main memory, so it is not necessary to set the transfer mode explicitly. However, the main memory address which stores transferred data or receives data must be the address of a variable allocated for a large area, or the address of a variable allocated to a heap area by malloc() or a similar function. It cannot be the address of a stack region or an auto variable declared within a function.

Note: If CDInit() is not called before SpuInit(), waveform transfer completion may fail, especially when SpuIsTransferCompleted(SPU_TRANSFER_WAIT).

To clear an area in SPU RAM, use SpuWrite0(). To transfer in steps, use SpuWritePartly().

Active memory management is not performed in the sound buffer. So, data transfer should avoid the areas listed below. Data transferred to these areas cannot be used as waveform data.

- 0x00000 ~ 0x00fff -- SPU decoded data transfer region
- 0x01000 ~ 0x0100f -- System reserved region
- After the reverb work area offset (starting) address

Figure 16-1: Sound Buffer Memory Layout
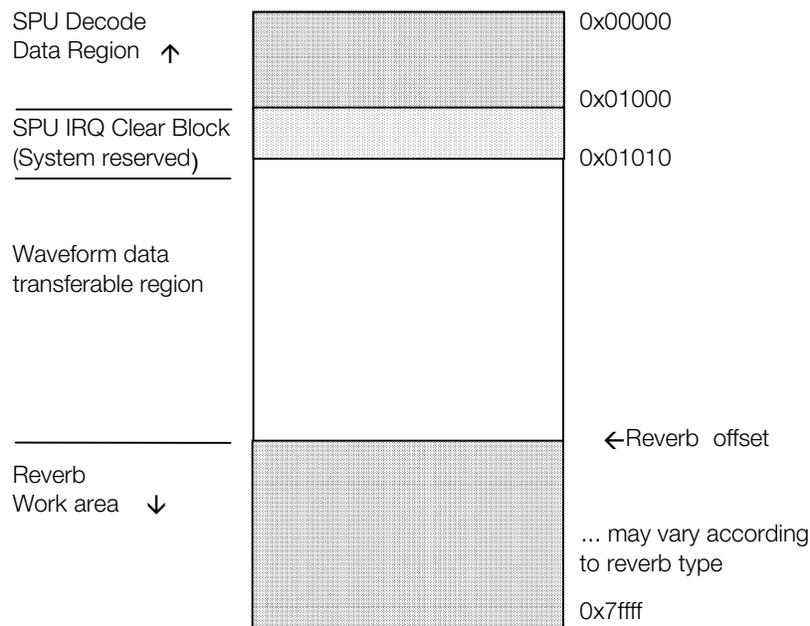
Interrupt Request for Sound Buffer Access

The sound buffer may be accessed for operations besides data transfer. The SPU is also able to access the sound buffer at any time while decoding in order to output the transferred waveform data as sound.

This optional access to the sound buffer is performed by generating a hardware interrupt (interrupt request) when access is made to a specific address. The specified address is set in SpuSetIRQAddr(). It is also possible to specify a function to be called in response to this interrupt request by calling the function SpuSetIRQCallback(). SpuSetIRQ() must also be called in order to enable or disable the IRQ. SpuGetIRQ() will return whether or not the interrupt has been enabled and SpuGetIRQAddr() will return the interrupt request address.

Sound Buffer Memory Management

Sound buffer memory management is limited. It manages a table of occupied memory and reports only that information. Simple sound buffer memory management is possible using this information. The following functions manage SPU memory:

- SpuInitMalloc() sets up the memory management table.
- SpuMalloc() allocates an area in SPU RAM.
- SpuMallocWithStartAddr() allocates an area in SPU RAM with a specific starting address.
- SpuFree() deletes the information from the designated area in the memory management table.

Mixing CD and External Digital Input

The SPU has the following two systems for external input:

- CD input
- External digital input

The sampling frequency of both is 44.1 kHz. Sound from these inputs and SPU output may be mixed digitally. The input may also be assigned to reverb.

This example sets up the CD volume:

```
#include <libspu.h>

SpuCommonAttr attr;

attr.mask = (SPU_COMMON_MVOLL |    /* master volume (left) */
        SPU_COMMON_MVOLR |         /* master volume (right) */
        SPU_COMMON_CDVOLL |        /* CD input volume (left) */
        SPU_COMMON_CDVOLR |        /* CD input volume (right) */
        SPU_COMMON_CDMIX);         /* CD input on /off */

/* set master volume to mid-range */
attr.mvol.left = 0x1fff;
attr.mvol.right = 0x1fff;

/* set CD input volume to mid-range */
attr.cd.volume.left = 0x1fff;
attr.cd.volume.right = 0x1fff;

/* CD input ON */
```

CONFIDENTIAL

```
attr.cd.mix = SPU_ON;

/* set attributes */
SpuSetCommonAttr (&attr);
```

Please note that calling SpuInit() resets the CD volume to zero. The proper order of initialization is CDInit(), then SpuInit(), then use the above example to reset the CD volume.

Transferring Data Decoded by SPU to Main Memory

The SPU writes to the sound buffer's first 0x1000 bytes, 16 bits at a time at each clock (44.1 kHz) pulse. Data is written after CD input volume processing and after Voice 1 and Voice 3 envelope processing. The individual sound buffers are each 0x400 bytes, divided into two halves. By deciding which buffer region to write to, it is possible to write up to 100 samples (100 / 44100 = 0.0022 ... seconds) of data at one time. SpuReadDecodeData() will perform this transfer.

Initializing, Starting and Stopping SPU Processing

- Call SpuInit() before any other libspu functions.
- In no particular order: Transfer data to SPU RAM; Set up reverb; Set main volume, CD input, and external input with SpuSetCommonAttr().
- Set voice attributes using SpuSetVoiceAttr().
- Key On voices using SpuSetKey() or SpuSetKeyOnWithAttr(); all main processing.
- Call SpuQuit() to stop all SPU processing.
- SPU Streaming Library

The PlayStation SPU originally played back only waveform data that could fit in SPU RAM (maximum 512 K). The SPU streaming library provides for playback of waveforms larger than the sound buffer, by transferring sections of data to designated areas in SPU RAM continuously during playback.

Note: The functions explained in this section are included in the basic sound library (libspu), and are separate from the libcd streaming library

Basic Operations

Waveform data is loaded into main memory and transferred to the SPU sound buffer for playback. With the SPU streaming library, only part of the data needs to be loaded at once. The data is a VB file containing only one VAG data, i.e. a VAG file that does not include the 48-byte header.

A *stream buffer* is allocated in SPU RAM for each voice used by the library. When the library uses more than one voice (up to 24 can be used), the size and pitch of the stream buffers must be the same for all voices.

The SPU plays back the waveform data via continuous transfer from main RAM to the stream buffer.

Waveform Data Processing

The library can handle waveform (VB) data larger than SPU RAM, and all parts of the waveform data need not reside in main RAM at the start of streaming.

At any point in the process, RAM must contain waveform data for each stream at least half as large as the stream buffer. When the transfer of the processed waveform data is requested, by specifying the start

address and the attributes of the necessary part of the waveform data, the library is informed of the continuation of the stream processing.

The waveform data used is being partly rewritten at the time of transferring. Internal marks for the library are being created in the waveform in main RAM. Since the waveform has been altered for the library, it is recommended that sections of the waveform used by the library be reloaded into main RAM from the CD before transfer to SPU RAM for a second time.

During transfer, the contents of all waveform data in the main memory are rewritten by stream buffer half-size units and destroyed.

Four States in the SPU Streaming Library

There are four states in SPU streaming:

Idle

No streams are being processed; therefore, the library puts no load on the system.

The idle state follows the termination state.

Preparation

To eliminate time-lag of sound generation, some waveform data for each stream must be transferred into SPU RAM before stream processing begins. The data must be half the size of the stream buffer. The end of data transfer in the preparation state can be detected by the preparation finished callback function.

Transfer

This state is where sound generation is actually performed for the designated voice. Half of the stream buffer is processed, and end of processing can be detected by the transfer finished callback function.

Requests for the preparation state for other voices can be performed in this state, but the state does not change to preparation. The preparation for these other voices is performed in the transfer state.

Termination

Termination is designated for all the streams, and the transfer is completed. Any requests for preparation or transfer are not accepted. On completion, processing returns to the idle state, and the next request for preparation can be accepted.

Figure 16-2: Four States and their Transitional States

Idle ⟶ Preparation ⟶ Transfer ⟶ Termination

Callback Functions

The library provides three types of callback functions. Each function is called with the same timing in multiple streams. The requested stream can be recognized by the argument of the callback function.

Preparation Finished Callback Function

Called when the initial transfer is completed for the preparation state. If the stream is to be played immediately after preparation, the attributes for the next transfer must be set here.

Transfer Finished Callback Function

Called when the transfer of waveform data half as large as the stream buffer is completed in the transfer state. In this function, the attributes for the next transfer must be set.

CONFIDENTIAL

Stream Finished Callback Function

Called when playback of the termination-designated stream is completed.

Figure 16-3: Four Callback Functions and Transitional States

[Idle]

← SpuStTransfer (SPU_ST_PREPARE)

→ Preparation finished callback function

[Preparation]

← SpuStTransfer (SPU_ST_PLAY)

(Transfer to the stream buffer)

[Transfer]

→ Transfer finished callback function

(Transfer to the stream buffer)

→ Transfer finished callback function

(Transfer to the stream buffer)

Stream A
is completed   → Transfer finished callback function

(Transfer to the stream buffer)

→ Transfer finished callback function

→ For the stream A
Stream finished callback function

(Transfer to the stream buffer)

→ Transfer finished callback function

(Transfer to the stream buffer)

Last stream
is completed   → Transfer finished callback function

[Termination]

→ Stream finished callback function

[Idle]

Stream Processing

Stream Preparation and Start

Since all streams are processed at the same time, they are all transferring data to the same half of the stream buffer and playing back from the other half of the stream buffer. This can affect the way that new streams are added.

Preparation for each stream is always done by transferring waveform data to the first half of each stream buffer. If preparation is requested in the idle state, it is processed promptly. However, if a new stream's preparation is requested in the transfer state, it must wait until the other streams are ready to transfer data to the first half of their buffers. Therefore, a lag occurs; the larger the stream buffer, the longer the potential lag for this initial data transfer.

When data transfer to the second half of a stream buffer begins, playback of the first half of the stream buffer begins. If transfer to the second half of the stream buffer is requested during preparation, it is

CONFIDENTIAL

processed promptly. However, if transfer to the second half of a new stream buffer is requested during the transfer state, playback doesn't occur until the other streams transferring to the second half of the stream buffer are finished. As in the case of preparation, the lag time for playback lengthens as the stream buffer size increases.

Attributes for the Next Transfer

You should specify the following attributes for the next transfer in each stream in the transfer finished callback function:

- The start address in main RAM of the waveform data area (half as large as the stream buffer) for the next transfer.
- If the stream is completed in the next transfer, specify the termination in *status* and specify the size of the data for the last transfer (half the stream buffer size or less.)

Stream Termination

Termination of each stream is specified by setting termination for the attribute status and the size of the last-transferred waveform data (half as large as the stream buffer or less) when setting the attributes for the next transfer. The stream is terminated when playback of the stream specified in this setting is completed.

Key on/Key off

Only sound generation (key on) is done automatically, at the start of streaming. Sound cancellation (key off) must be done by the program. Be sure to perform key off after stream termination; otherwise the state of the sound library may be unstable. Since this method may result in some lag time between desired key off and actual processing completion, the recommended solution is to set the volume of the voice to 0.

Actual Flow of Stream Processing

A typical flow of SPU streaming is as follows:

Initialization

Initialize the library with SpuStInit().

```
SpuStEnv *stenv;
stenv = SpuStInit (0);
```

It returns the structure SpuStEnv, shown below.

```
typedef struct {
    char status;                    /*stream status*/
    char pad1;                      /*padding*/
    char pad2;                      /*padding*/
    char pad3;                      /*padding*/
    long last_size;                 /*the size of last transfer at
                                    termination stage
                                    (last_size <= (size / 2))*/
    unsigned long buf_addr;         /*The start address of stream
                                    buffer in SPU RAM*/
    unsigned long data_addr;        /*The start address of SPU
                                    streaming data in main RAM*/
} SpuStVoiceAttr;

typedef struct {
    long size;                      /*The size of stream buffer*/
    long low_priority               /*Priority of the stream; added in lib
                                    3.6*/
    SpuStVoiceAttr voice [24];
} SpuStEnv;
```

CONFIDENTIAL

Streams are processed by specifying the attributes for this structure.

Attribute Initialization

The SpuStEnv structure member *size* sets the size of the stream buffer for all streams. It should be chosen carefully to meet the needs of the program, considering the available area in SPU RAM and the number of stream buffers needed. As the size of the stream buffers increases, the lag time for new streams to be processed and for termination of streams increases. However, with smaller stream buffers, more time is consumed by callbacks, from both the SPU DMA during each data transfer and from the SPU IRQ when the end of each stream buffer is reached and playback needs to continue from the start of the stream buffer.

Example:

```
stenv->size = 0x8000;
```

low_priority is also a member of the SpuStEnv structure. Set SPU_ON to lower the priority level of SPU streaming processing compared to other processing (e.g. graphics processing will have higher priority than SPU streaming). The default value is SPU_OFF, where the priority level is not lowered.

The attributes that must be initialized for each stream are:

1. The start address of the stream buffer

```
= voice[].buf_addr in SpuStEnv structure
```

  Example:

```
unsigned long buf_addr;
if ((buf_addr = SpuMalloc (0x8000)) == -1) {
  /* ERROR */
}
stenv->voice [n].buf_addr = buf_addr;
```

2. The start address in main RAM of the waveform data to be transferred during the preparation stage.

```
= voice[].data_addr in SpuStEnv structure
```

  Example:

```
stenv->voice [n].data_addr = 0x80yyyyyy;
```

The subscript (n in the above example) of the array stenv->voice corresponds to the voice number.

Callback Functions Setting

All the callback functions have the following syntax:

```
SpuStCallbackProc callback_proc (unsigned long voice_bit, long c_status)
```

When the function is called, the voices to be processed in each callback function are passed to *voice_bit* by setting the bits SPU_0CH to SPU_23CH. The state in which the callback function is called is passed to the argument *c_status*. The program must analyze these arguments and process them appropriately.

At a minimum, the transfer finished callback function must be called in order to process the stream. This function specifies the start address of the next section of waveform data to be transferred, and when ready to terminate, species information about terminating the stream.

Voice Setting

The attributes for each streaming voice are set. For the start address of the waveform data in the voice attributes, the same value as the start address of the stream buffer is set.

Example:

```
SpuVoiceAttr s_attr;
        :
s_attr.voice = SPU_3CH;
s_attr.addr = stenv->voice [3].buf_addr;
        :
SpuSetVoiceAttr (&s_attr);
```

Preparation for the Stream

As preparation for starting a stream, waveform data half as large as the stream buffer is transferred to the stream buffer, in order to eliminate time-lag in sound generation. Preparation is done by calling SpuStTransfer(), specifying SPU_ST_PREPARE as the first argument. The second argument specifies the voices used for the stream by setting the bits of SPU_0CH … SPU_23CH. For example, to prepare voices 0 and 1 for streaming:

```
SpuStTransfer (SPU_ST_PREPARE, (SPU_0CH | SPU_1CH));
```

When data transfer for the requested voices is completed, the preparation finished callback function is called. After preparation, but prior to starting playback of the stream, you must specify the attributes for the next transfer; if playback is to immediately follow preparation, attributes must be set in the preparation finished callback function.

In particular, you must specify the start address of the next section of waveform data, to be copied to the second half of the buffer. (This section need not be contiguous in memory with the data which was transferred during the preparation stage.) Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
```

Start of the Stream

When preparation is completed for each stream, start the stream by calling SpuStTransfer() with the first argument SPU_ST_PLAY. The second argument specifies the voices used for the stream (by ORing the corresponding bits SPU_0CH to SPU_23CH); this value must be the same as the value specified during the preparation stage. Example:

```
SpuStTransfer (SPU_ST_PLAY, (SPU_0CH | SPU_1CH));
```

As soon as the stream is started, sound generation (Key on) is performed.

When transfer is completed for a stream, the transfer finished callback function is called. You use this function to set the attributes for the next transfer. You must specify the start address in RAM of the next section of waveform data (which is half as large as the stream buffer). This section need not be contiguous with the data for the previous or next transfer. Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
```

Stream Termination

To terminate a stream, set *voice[ ].status* in the SpuStEnv structure to SPU_ST_STOP in the transfer finished callback function. Set *voice[ ].last_size* to the size of the remaining waveform data (which must be half as large as the stream buffer or less). After transferring the waveform data area represented by *voice [ ].data_addr*, the stream is terminated.

Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
stenv->voice [n].status = SPU_ST_STOP;
stenv->voice [n].last_size = 0x4000;
```

CONFIDENTIAL

When playback of this stream completes, the stream finished callback function is called (immediately before the start of the next transfer if other streams are still being processed.)

Completion

When streaming is completed, call SpuStQuit(). Before calling, processing must be completed for all streams, and the status must be idle.

Basic Sound Library and Extended Sound Library Common Uses

The following is some information regarding using both libspu and libsnd.

Initialization

When using SsInit() in libsnd, it is not necessary to call SpuInit(), because SsInit() internally calls SpuInit().

Sequence Data

When using sequence data such as SEQ/SEP, normally it is necessary to use libsnd, since libspu has no functions to handle them.

When creating an individual driver to analyze and generate sequence data using libspu, it is necessary to use the libapi root counter and event processing functions for time management.

Sound Generation/libsnd Voice Manager Function

libsnd dynamically controls the voice ratio, and it generally controls the on/off of all 24 voices. Since libspu cannot use these controlled voices by setting the voices assigned to libspu to SsSetReservedVoice() (with a setting value less than 24) it will be possible to divide the voices controlled by libsnd from the voices which libspu can use.

Transfer to the Wave Pattern Data Sound Buffer

Although it is possible to use libspu to transfer VAB data to the sound buffer, header and attribute information isn't preserved.

Therefore, you should use libsnd routines:

- Use SsVabOpenHead(), SSVabOpenHeadSticky(), SsVabTransBody() or SsVabTransBodyPartly() to transfer the wave pattern
- Use SsUtGetVabHdr() to get header information
- Use SsUtGetProgAtr() or SsUtGetVagAtr() to select information to find out what location in the sound buffer the wave pattern data has been transferred
- Use SpuSetVoiceAttr() to set the voice attributes used in libspu.

Libsnd's default transfer mode is DMA transfer. Using libsnd, the transfer mode can be changed using SpuSetTransferMode(); it should be called before SsVabTransBody()/SsVabTransBodyPartly().

A transfer completion callback function can be used in libspu, but not in libsnd. Before calling the wave pattern transfer function (SsVabTransBody()/SsVabTransBodyPartly()) in libsnd, the transfer completion callback function must be set to NULL. Use SsVabTransCompleted() to determine completion status.

```
(void)  SpuSetTransferCallback  ((SpuTransferCallbackProc) NULL);
        :
        SsVabTransBody (. . .);
```

Sound Buffer Memory Control and Reverb

When using the libsnd function SSInit(), SpuInitMalloc() is called internally, so you don't need to call it yourself. It specifies 32 as the maximum number of memory areas that can be allocated by SpuMalloc() and SpuMallocWithStartAddr().

When a VAB is opened using SsVabOpenHead(), SpuMalloc() is called internally, and when it is closed SpuFree() is called. At any time, the number of additional memory areas the user can allocate with SpuMalloc()/SpuMallocWithStartAddr() is (32- number of VAB openings). If you need to allocate more areas, call SpuInitMalloc() after SsInit() and pass a greater number. The size of the control table you must allocate is SPU_MALLOC_RECSIZ x ((number of VAB openings + number of SPUMalloc() calls by user) +1)

Alternatively, you can do your own sound buffer memory management and use SsVabOpenHeadSticky() to specify a particular address in the sound buffer where the VabBody is to be transferred.

Libsnd reverb has been provided using almost the same functions as in libspu. Therefore, SpuReserveReverbWorkArea() in libspu can be used in the same way.

Refer to the *Run-Time Library Reference* for more information on these functions.

Applying Reverb to Voices Using Libspu and Libsnd

SpuSetReverbVoice() cannot be used to apply reverb to voices controlled by libsnd (that is, those set by SsSetReservedVoice()). Instead, use SsQueueReverb().

Prior to library 4.5, when libsnd voice allocation was used, voices set by SpuSetReverbVoice() would not work correctly, because they would be turned off during the libsnd tick callback. This problem has been fixed in library 4.5

Noise during CD-DA/XA playback

When noise occurs during CD-DA/XA playback, check the following points:

Is the converted data correct?

The sound tool assumes that data is 16-bit straight PCM data. Note that it is not compatible with AIFF. When converting AIFF, since the header and footer information which appears at the beginning and end is converted into sound, noise will be produced. The SoundDesignerII 2.5 sampling data format is 16-bit straight PCM, so it can be used as is.

Does the volume decrease when playback is paused or a seek is performed?

Pausing a CD or performing a seek while sound is playing can cause clip noise to be produced. When pausing a game where the CD also pauses, issue the CD command after performing a fade out.

Does the XA data contain a large number of high pass components?

With XA data, sound is compressed to 1/4, so noise is sometimes produced. The noise can become particularly evident when there are a large number of high pass components. Perform a pre-process such as installing a filter in advance to avoid this.

Chapter 17:
Serial Input/Output Library

Table of Contents

Overview

This library (libsio) provides standard input/output functions for connecting the PlayStation to the PC. It supports the output of debug information to the PC.

Library and Header Files

To use the standard I/O library, you must link with the library file `libsio.lib`.

Your source files must include the header file `libsio.h`.

Driver and BIOS

The serial I/O library consists of the serial I/O driver and the serial I/O BIOS.

Serial I/O Driver

The serial I/O driver provides standard I/O using standard C-language procedures. By including the driver, you can easily allocate standard I/O to the communications port. The BIOS should be used when performing complex communication with a PC and modem, etc.

To include the serial I/O driver, call AddSIO( ). To delete it, call DelSIO( ).

BIOS

The serial I/O BIOS provides low-level driver control and information acquisition functions that cannot be covered by standard C functions. The interface function is _sio_control( ). Since debugging data is normally output from the library in standard I/O, when performing data communication with a PC, unexpected data is output. To avoid this, communication must be performed using the BIOS, without attaching the driver and with the standard I/O in NULL mode. The BIOS will also operate when the serial I/O driver is not included. The features provided in sio_control are shown below:

Syntax:

```
long _sio_control (
        unsigned long cmd,          /* command */
        unsigned long arg,          /* subcommand */
        unsigned long param)        /* argument */
```

For details, see the discussion of _sio_control( ) in the *Run-Time Library Reference*.

Chapter 18:
HMD Library

Table of Contents

CONFIDENTIAL

Overview

The HMD file format integrates several types of data, including modeling, animation, texture, and MIMe data. The HMD Library (libhmd) provides functions and structures for using HMD. It uses the jump table method used in libgs to handle TMD data. HMD data consists of a number of primitive types, which have corresponding primitive drivers (functions that handle the data). SCE provides standard primitive drivers for handling a variety of types, including modeling data with hierarchical structures, one-skin models, key-frame animation and vertex/joint MIMe.

You use standard public APIs to call primitive drivers within the HMD framework. You can define your own primitive drivers to provide optimization and expansion for individual game titles.

In PlayStation library versions between 4.0 and 4.2, HMD-related functions were part of the libgs and libgte libraries, but are now offered as a separate library. These functions have been removed from the libgs and libgte libraries, and their declarations have been removed from the corresponding header files.

Note: The environment map is provided as a Beta version with this release. Because future releases may introduce format changes not compatible with the current release, the Beta version primitives are currently not supported by SCE and should be used only at the licensee's discretion.

Library and Header Files

Programs that make calls to the HMD library must link with the library file `libhmd.lib`. Because libhmd is currently dependent on libgs, `libgs.lib` must also be linked in.

Source files must include the header files `libhmd.h` and `libgs.h`.

Basic Architecture

The basic HMD architecture is shown in the figure below.

Figure 18-1: HMD Basic Architecture
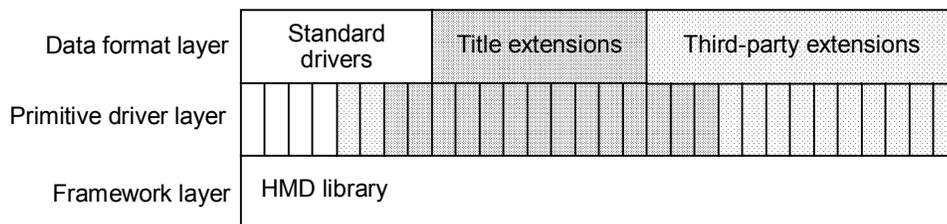
The framework layer is provided by libhmd. There is a standardized API between the framework layer and the primitive driver layer, allowing user-defined primitive drivers and third-party products to be used. The data format layer refers to data formats based on combinations of primitive drivers.

The framework layer performs operations such as:

- Mapping HMD data loaded in memory
- Scanning, which binds primitive drivers to their corresponding primitive types
- Sorting, to traverse the data structures and call the actual primitive drivers.

Each of these operations is described in a later section.

Part of the HMD data format is defined by the framework, and cannot be modified or extended independently by the user. Another part is defined by the implementation of the primitive driver, and can be defined freely based on the fixed set of rules provided by the framework.

For example, the primitive drivers provided by SCE for processing polygons, because of their general-purpose design, may be inadequate in terms of speed. Performance improvements can be obtained by defining new drivers specifically for a particular title.

HMD Features

The HMD framework and the primitive drivers are loosely coupled using a standardized API. SCE provides a group of general-purpose primitive drivers, and users can write their own specific primitive drivers.

This section describes the following features that are available from the SCE primitive drivers:

- Hierarchical structures
- Polygon/MESH
- Shared polygons (one-skin model)
- Animation
- MIMe
- Other

Hierarchical structures

In the TMD/PMD formats used by libgs, data in hierarchical structures have to be described in the program code. Hierarchical structures can be represented with TOD, but this increases code size and can make communication between designers and programmers more difficult. HMD overcomes this problem by implementing hierarchical structures in the data format itself.

Figure 18-2: Hierarchical Structure

A's Child 1

A's Child 2

Parent coordinate system A

CONFIDENTIAL
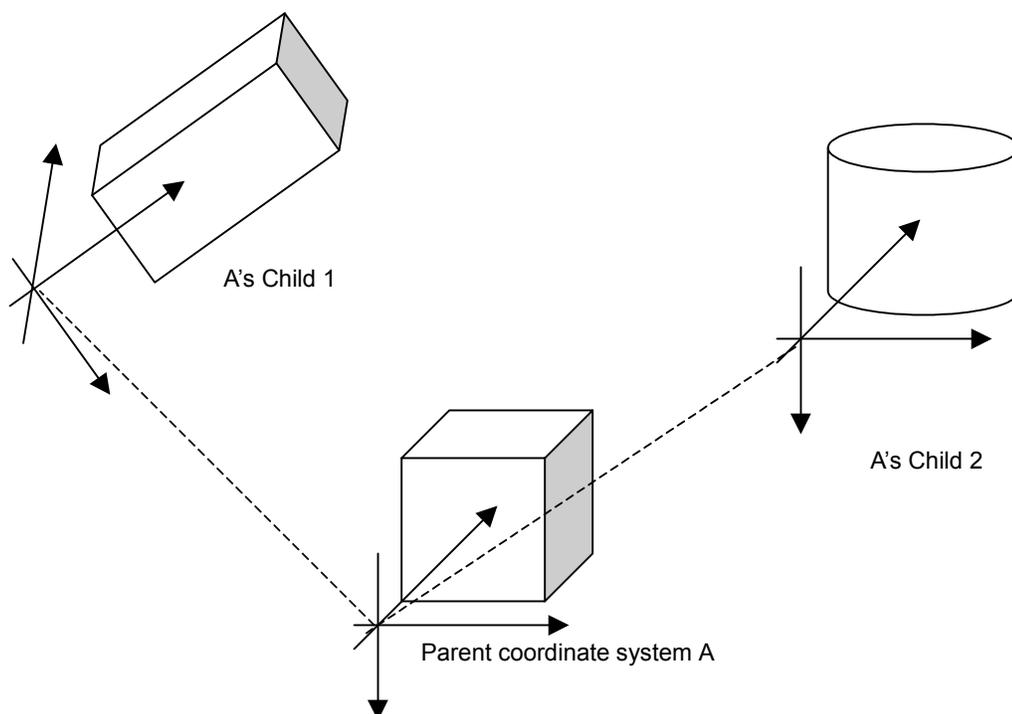
Polygon/MESH

Polygon/MESH is a high-order set of what was provided in TMD/PMD. In addition to standard polygons, polygons using MESH (Strip Mesh) can also be described with this structure. In some cases, the use of MESH can improve performance in terms of both size and speed.

Figure 18-3: Strip Mesh
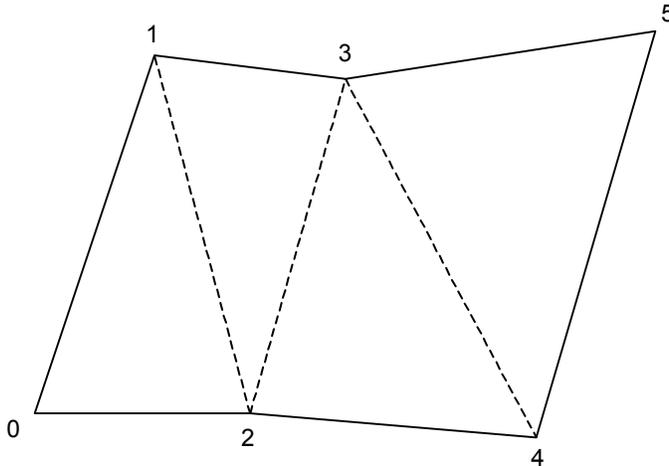
In the figure above, if polygons were used, four triangles and 4 x 3 = 12 coordinate transformations would be required. However, with Strip Mesh, only 6 coordinate conversions are necessary. In terms of the data structure, four triangles would normally require 12 indexes to specify the vertices, while Strip Mesh would only require 6. The more complex the figure, the greater are the advantages in speed and memory efficiency offered by Strip Mesh. Rendering speed is improved, since GPU packets are generated as connected triangles.

Shared polygons (one-skin model)

With HMD, a polygon spanning the hierarchical structures described above can be pasted, allowing one-skin models to be represented. These can be used at joints so that more natural, smoother figures can be represented.

Figure 18-4: Shared Polygons

Polygon c of coordinate system C

P

R

Q

Polygon a of coordinate system A

Polygon b of coordinate system B
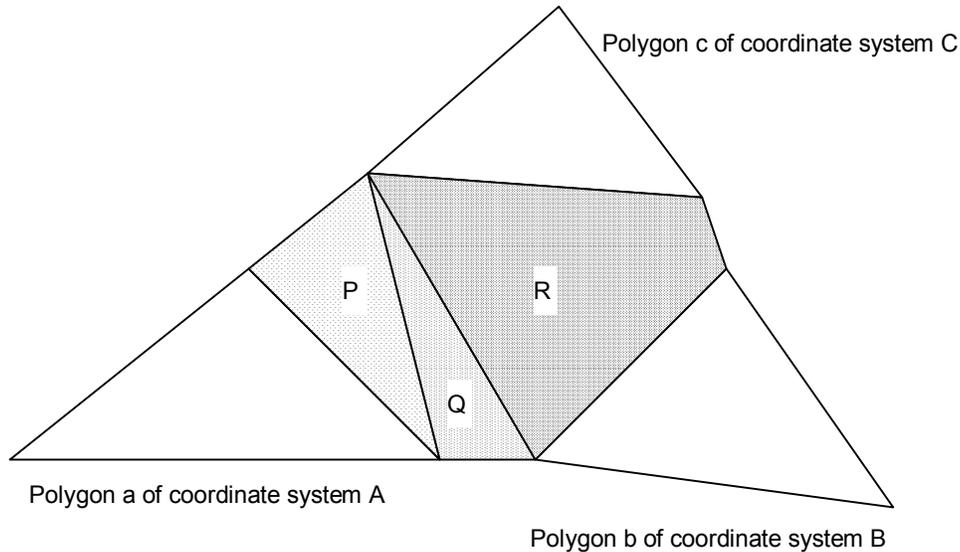
In the figure above, three shared polygons (P, Q, R) are defined. Polygon P has two vertices belonging to coordinate system A and one vertex belonging to coordinate system C. Thus, the polygon is shared between coordinate system A and coordinate system C. The vertices of the triangle in polygon Q belong to coordinate systems A, B and C. This polygon is shared by three coordinate systems. R is a quadrangle polygon shared by coordinate systems B and C.

With this arrangement, the values of coordinate systems A, B, and C could be changed without losing the connectivity between the polygons, since polygons P, Q and R follow the coordinate systems to which their vertices belong.

Animation

Key-frame animation can be performed with three types of interpolation curves (Linear, Bezier, and B-Spline). In addition, multiple motions can be saved in HMD data and switched during execution.

Key-frame animation can be applied to standard coordinate data. If the data is in memory, arbitrary values can be animated to provide more varied expressions, e.g. moving MIMe parameters and controlling polygon attributes such as color.

MIMe

MIMe is suited for facial animation, since the high-speed multi-layer interpolation performed by the GTE can be used to combine multiple key frames. In the past, sample programs have been provided to implement vertex/normal MIMe. These have now been incorporated into the HMD format, along with "joint MIMe", which interpolates joint angles. Compared to vertex/normal MIMe, joint MIMe provides reduced data size and improved speed. By combining vertex and joint MIMe, various movements can be efficiently expressed, e.g., the muscle formed when flexing an elbow.

CONFIDENTIAL

Figure 18-5: Combining vertex and joint MIMe
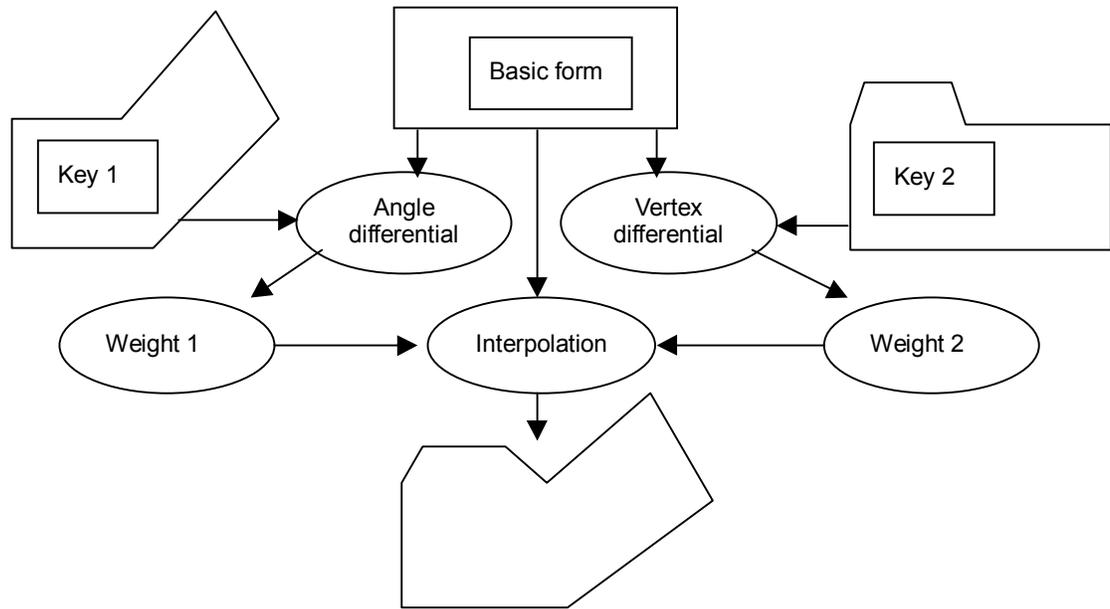
The basic form, angle differential, and vertex differential shown in the figure above can be specified in HMD data. Weight can be controlled via application code or by using HMD animation.

Other Features

Libhmd provides camera/light primitives and experimental primitives, such as ground primitives containing terrain data and simulated environment mapping primitives (Beta version).

Samples of primitive driver source code are also provided as a reference for implementing user-defined primitives.

Hierarchical Coordinate Systems and Process Flow

The following figure shows the operations that are performed starting with the hierarchical coordinate system.

Figure 18-6: Process flow and data structures
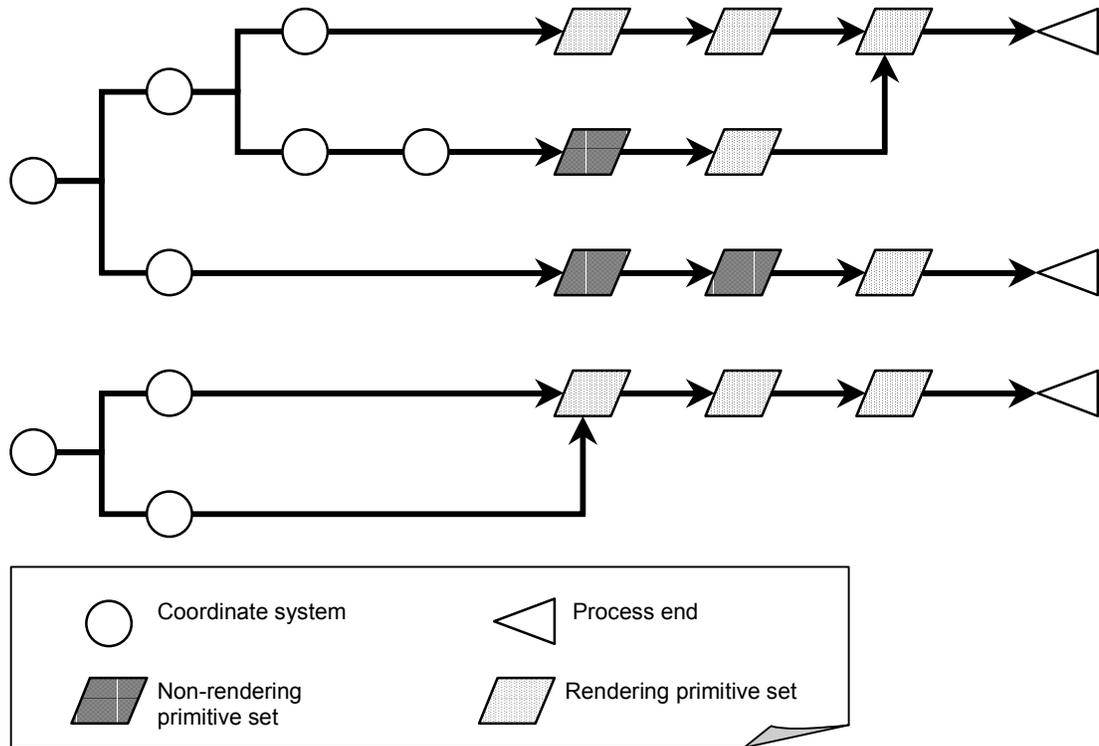

Primitive sets come in two varieties:

- Non-rendering primitive sets handle operations where direct GPU packets are not generated, e.g., loading animation or texture data to VRAM.
- Rendering primitives generate GPU packets, based on the results from non-rendering primitives or values specified in data, which are then entered into an ordering table.

The process begins from the pointers to the primitive sets that correspond to each of the root coordinate systems.

The figure above gives a schematic representation of the flow through the process.

Pointers to primitive sets are placed near the top of HMD data. The first word is the number of pointers: the value N+2 means that there are N primitive sets, each corresponding a coordinate system, as well as pointers to pre-processing (such as loading textures) and post-processing (such as rendering shared polygons) primitive sets. Pointers to the second primitive set through the (N+1)st primitive set correspond to the N coordinate systems. Primitive sets can also contain pointers for linking to the next primitive set. This enables completely different types of primitive sets to be processed one after another in the same coordinate system. Coordinate system data also contains pointers for specifying the parent coordinates. These pointers are used to link the hierarchical coordinate system with their related primitives.

CONFIDENTIAL

Figure 18-7: Linking primitive sets and coordinate systems
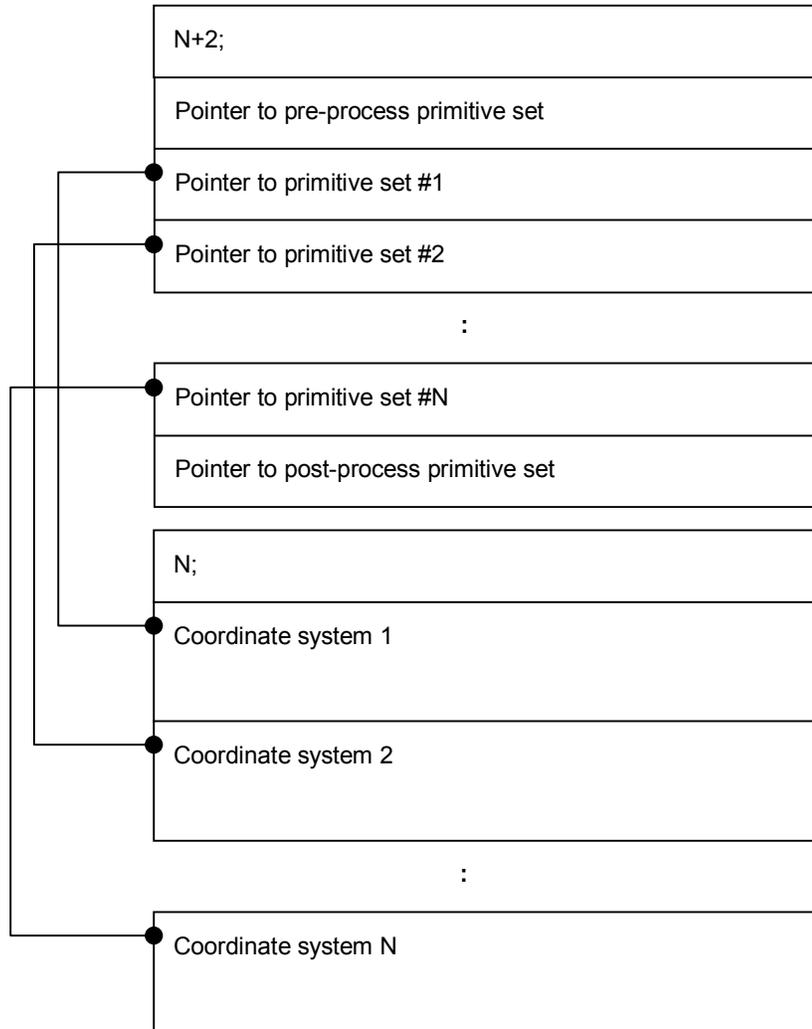
In order to use HMD, parsing must be performed according to the data structures described above, and the following three operations must be performed.

- **Mapping** - When HMD data is created, embedded pointer values contain offsets from the start of the data. During mapping, these offsets are converted into the real addresses where the data is loaded. In general, mapping is performed only once after HMD data is read into memory. For more information, refer to the description of GsMapUnit() in the *Run-Time Library Reference*.

- **Scanning** - Primitive types are represented as 32-bit values. During scanning, they are replaced with pointers to the corresponding primitive driver functions. In general, scanning is performed only once after mapping. For more information, refer to the description of GsScanUnit() in the *Run-Time Library Reference*.

- **Sorting** - Sorting involves calling the actual primitive drivers set up during scanning. In general, sorting operations are called for each Vsync. For more information, refer to the description of GsSortUnit() in the *Run-Time Library Reference*.

Basic Data Structures

This section describes basic data structures used in HMD. See the HMD section of the *File Formats* manual for further details.

Primitives

The primitive is the smallest unit in the HMD format. Primitives contain primitive types, and are called when the corresponding primitive driver performs a sorting operation. The structures below are defined by the framework (shown in the HMD assembler "LAB" format).

```
DEV_ID(dev_id)|CTG(ctg)|DRV(drv)|PRIM_TYPE(type);

H(size); M(H(data));

DATA;   /* for "size - 1" long words */
```

The first line (`DEV_ID(dev_id)|CTG(ctg)|DRV(drv)|PRIM_TYPE(type)`) is a 32-bit value representing the primitive type (and other data). During scanning, it is replaced by a pointer to the corresponding primitive driver.

- `dev_id`: a 4-bit value for the vendor that defined the primitive. 0x0 and 0x01 are SCE. 0xf is reserved for user-defined primitives. Other values are planned for third-party primitive drivers.
- `ctg`: a 4-bit value indicating the major category of the primitive. The standard primitives provided by SCE are categorized in the following manner: polygons (CTG_POLY: 0); shared polygons (CTG_SHARED: 1); images (CTG_IMAGE: 2), Animation (CTG_ANIM: 3), MIMe (CTG_MIME: 4), Ground (CTG_GND: 5), and Equipment (CTG-EQUIP: 7)
- `drv`: an 8-bit value used when actions need to be modified, without needing to redefine the data structures used by the primitive. With polygon primitive types provided by SCE, these bits are used to specify double-sided/single-sided polygons, for example.
- `type`: a 16-bit value specifying the primitive type.

`H(size)` is a 16-bit value representing the size of the primitive. "H" stands for half-word.

`M(H(data))` contains 15 bits of data that are used by the primitive driver. "M" refers to the fact that the high bit is changed from 1 to 0 during scanning in order to prevent double-scanning. Thus, only the low 15 bits are valid as data.

`DATA` contains `size` words. The values depend on the primitive type.

For example, the Gouraud triangle primitives provided by SCE would be:

```
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|IIP);

H(2); M(H(20));                    /* size: always 2 for SCE's standard
                                      polygon primitive data: interpreted as a
                                      polygon count for this primitive driver */

(Poly_0010 - Poly_0000) / 4;  /* Offset from the start of the polygon
                                 section contained in the corresponding
                                 polygon header. In this case, data for
                                 20 Gouraud triangles are arranged
                                 continuously from the offset position. */
```

The primitive driver is called at least once for each primitive. For efficiency, it is advantageous to perform many operations for a single primitive. In the polygon primitive type described above, it would be inefficient to prepare the data as 20 primitives containing one polygon each, in terms of instruction cache hit rate and memory access.

CONFIDENTIAL

Primitive Sets

A primitive set contains multiple primitives. It consists of the following data:

```
PrimSet:
      next_prim_set;
      PrimHdr;
      M(num_of_types);
```

- **next_prim_set;**

  The pointer to the next primitive set. This allows manipulating different types of primitive sets in a single coordinate system, in an explicit sequence. Depending on how the link is set up, the size of the data can be reduced through instancing. When there are no more primitive sets, the value "TERMINATE" (0xffffffff) is specified.

- **PrimHdr**

  Pointer to a primitive header. It must be in a format matching the primitive type contained in the primitive set.

- **M(num_of_types);**

  The number of primitives contained in the primitive set. Double-mapping is prevented by setting the high-order bit to 0 when `next_prim_set` and `PrimHdr` are mapped.

As an example, a primitive set containing the Gouraud triangle primitives described above is shown below.

```
PolyPrimSet:

      TERMINATE;               /* next prim; nothing more to process */
      PolyPrimHdr / 4;         /* header containing pointer to section in
                                  which polygon data, etc. are saved */
      M(2);                    /* num of types; one more in addition to
                                  Gouraud triangles */
```

Primitive Headers

A primitive header contains pointers to sections in the HMD data, as well as numeric data. Its format depends on the implementation of the primitive driver that corresponds to the particular type.

The header consists of a series of 32-bit words:

```
PrimHdr:

      hdr_size;

      M(ptr);

      num;

          :
```

- **hdr_size** - The size of the primitive header in 32-bit words (not including the space taken by this value itself.)

The remaining words are either:

- **M(ptr)** - If the high-order bit is 1, saves a pointer to a particular section which is then mapped.
- **num** - If the high-order bit is 0, the value is interpreted as a standard numeric value and mapping is not performed.

CONFIDENTIAL

The primitive header corresponding to the Gouraud triangles is shown below.

```
PolyPrimHdr:

3;                      /* hdr_size */
M(Poly_0000 / 4);       /* base address for polygon data */
M(Vert_0000 / 4);       /* vertex data; referenced in polygon
                           data via indexing */
M(Norm_0000 / 4);       /* normal data; referenced in polygon
                           data via indexing */
```

Sections

A section is used to group together data of a type other than what is described above. The primitive header itself is also set up as a "primitive header section".

The figure below shows the relationship between the different components.

Figure 18-8: Primitive sets, primitives, primitive headers, sections

CONFIDENTIAL
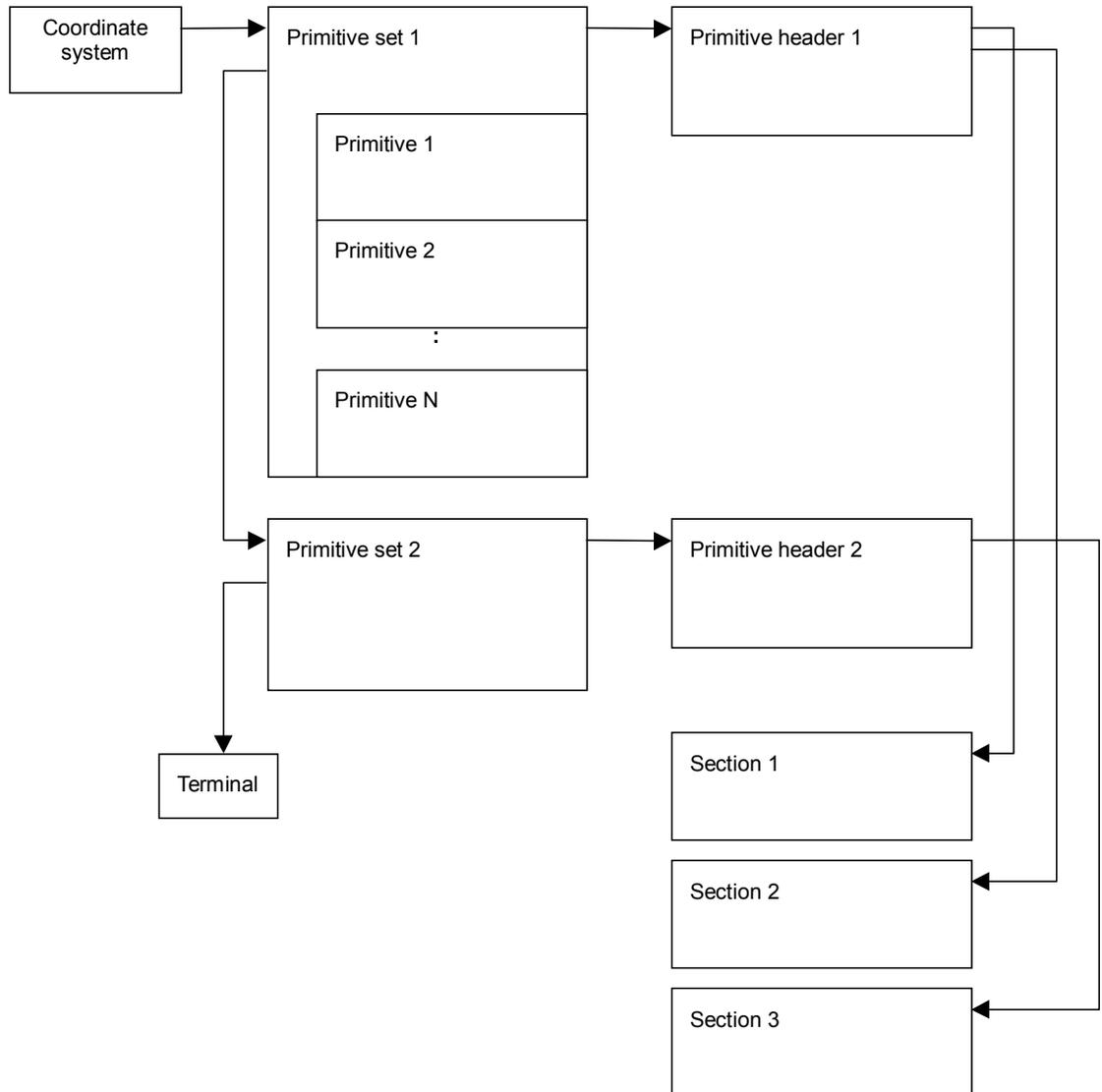
In the example shown in the figure, primitive set 1 contains N primitives, and these refer to primitive header 1. Primitive header 1 contains pointers to sections 1 and 2. Primitive set 2 is linked from 1 and refers to primitive header 2, which is in a different format (or simply contains different values) from primitive header 1. Primitive header 2 contains a pointer to section 3.

Primitive drivers

Primitive drivers, corresponding to the different types, are called during sorting operations.

Information that can be accessed from the primitive driver

The primitive driver can receive the following information from the framework:

- A copy of the primitive header, which contains pointers to sections and numeric data.
- Pointers to primitive data.
- Pointers to the ordering table specified by GsSortUnit(). For rendered primitives, the generated GPU packets are registered.

Information that should be returned to the framework

Primitive drivers return a pointer to the next primitive.

The location of the next primitive can be determined by referencing the size of the primitive. Depending on the defined primitive, it may be possible to determine the pointer to the next primitive without referencing the size (for example, with the standard polygon primitive driver from SCE, the fact that size is always 2 can be used to return the pointer to the next primitive).

MIMe Primitive Structure

This section uses diagrams to provide additional explanation on HMD MIMe primitives with complicated structures. Refer to the HMD section of the *File Formats* manual for details.

Notations used in diagrams

Figure 18-9: Index starting point

Point indicating section starting point, origin of index references including within a primitive, etc. In "index reference" below, it is added to the arrow shown with a dotted line.

Figure 18-10: Index reference
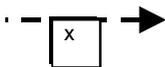
Indicates an index reference with the "index starting point" "x" as the origin.

Figure 18-11: Pointer reference

Indicates the absolute address reference within the HMD file. Values such as this are only described in primitive headers.

Figure 18-12: Vertex MIMe
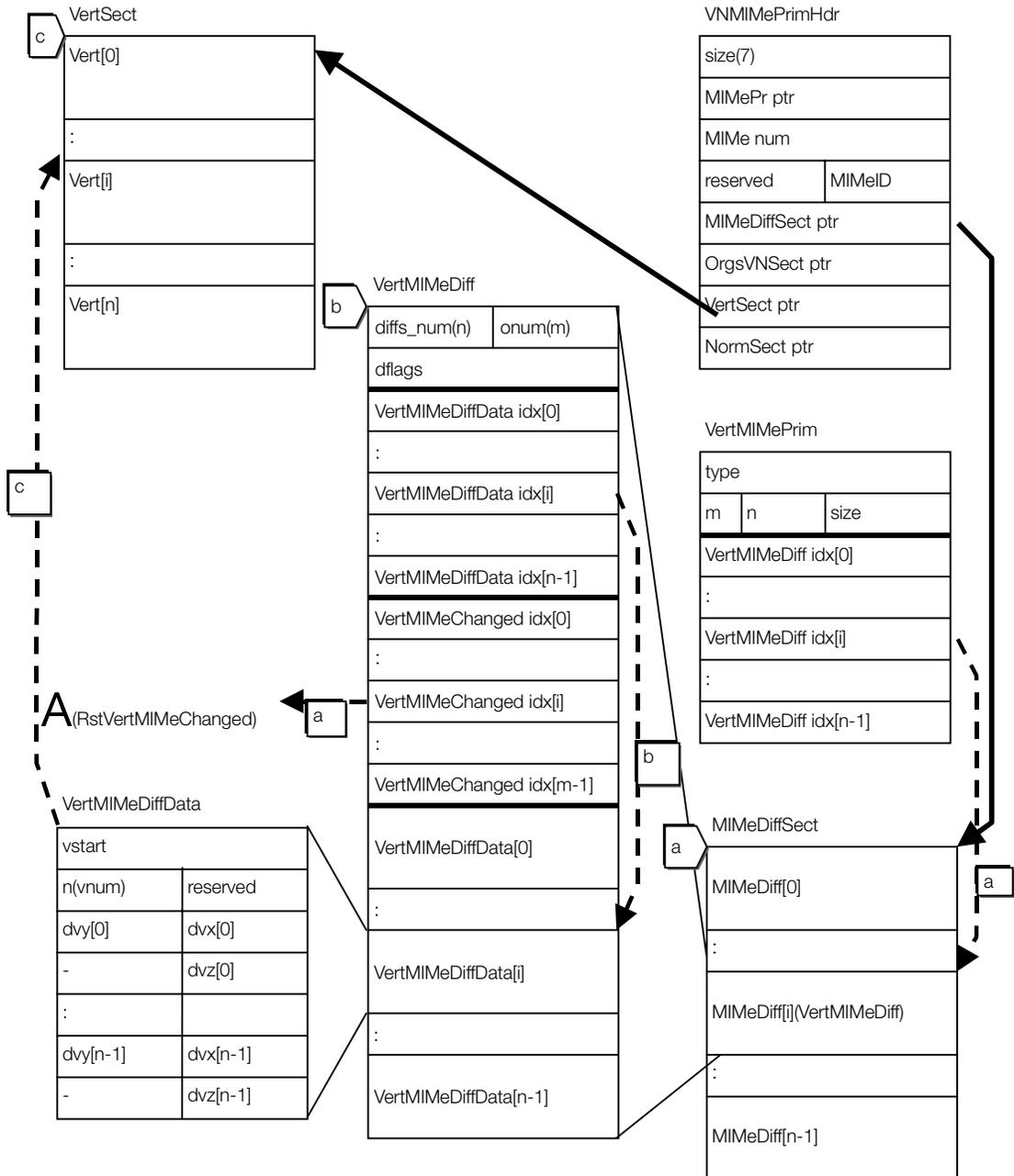
Except for when the vstart starting point becomes NormSect ptr rather than VertSect ptr, normal MIMe reset has the same structure as vertex MIMe.

Figure 18-13: Vertex MIMe reset
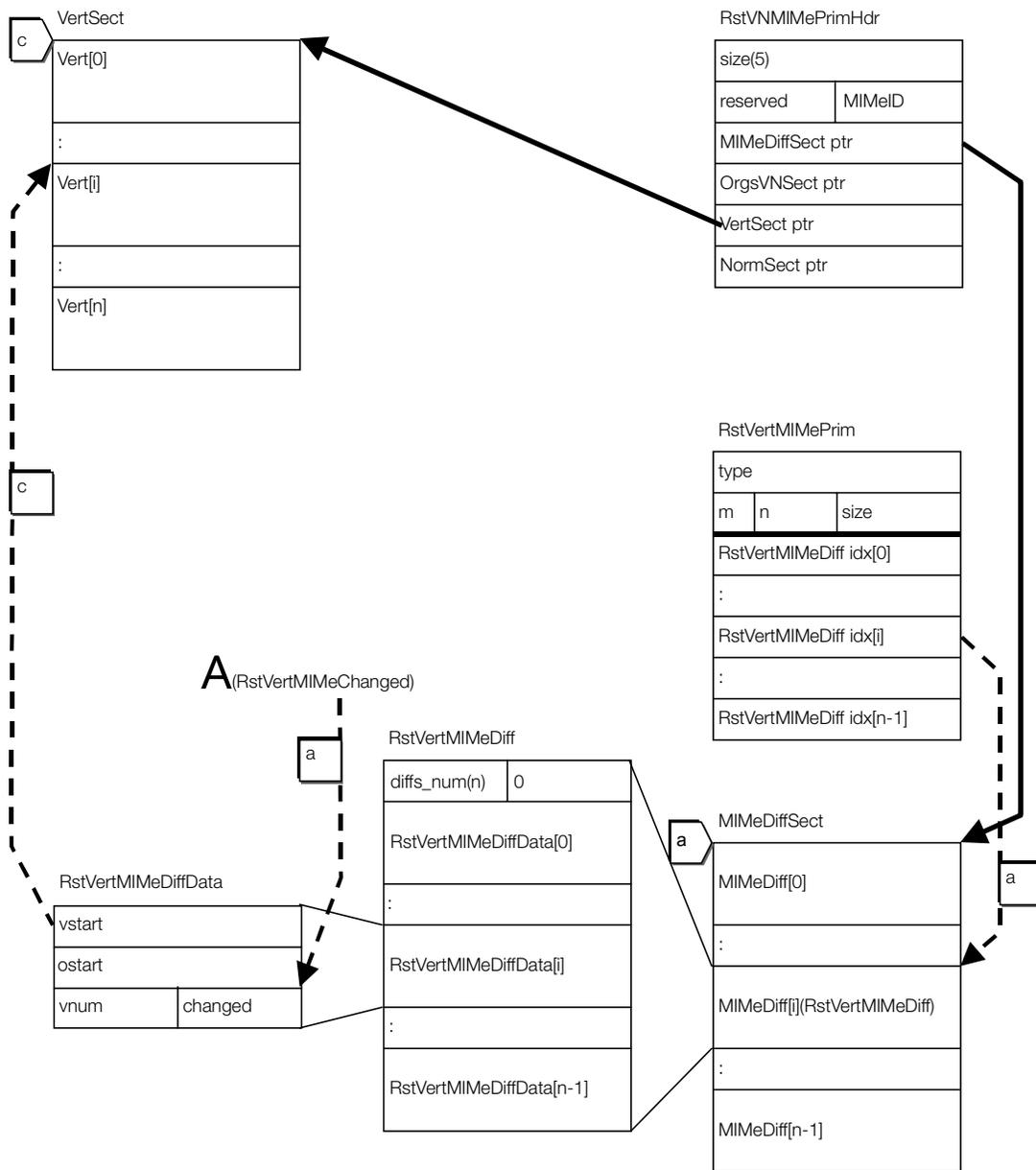
Except for when the vstart starting point becomes NormSect ptr rather than VertSect ptr, normal MIMe reset has the same structure as vertex MIMe reset.

Figure 18-14: Joint Axes MIMe

JntMIMePrimHdr

size(5)	
CoordSect ptr	
MIMePr ptr	
MIMe num	
reserved	MIMeID
MIMeDiffSect ptr	

RstJntMIMePrimHdr

size(3)	
CoordSect ptr	
reserved	MIMeID
MIMeDiffSect ptr	

(Rst)AxesMIMePrim

type		
m	n	size
AxesMIMeDiff idx[0]		
:		
AxesMIMeDiff idx[i]		
:		
AxesMIMeDiff idx[n-1]		

AxesMIMeDiff

diffs_num(n)	coord_id
dflags	
AxesMIMeDiffData[0]	
:	
AxesMIMeDiffData[i]	
:	
AxesMIMeDiffData[n-1]	
RstAxesMIMeDiffData	

AxesMIMeDiffData

dvy	dvx
dtp	dvz
dtx	
dty	
dtz	

RstAxesMIMeDiffData

m[0][1]	m[0][0]
m[1][0]	m[0][2]
m[1][2]	m[1][1]
m[2][1]	m[2][0]
changed	m[2][2]
dtx	
dty	
dtz	

MIMeDiffSect

MIMeDiff[0]
:
MIMeDiff[i](AxesMIMeDiff)
:
MIMeDiff[n-1]

CONFIDENTIAL

Figure 18-15: Joint RPY MIMe
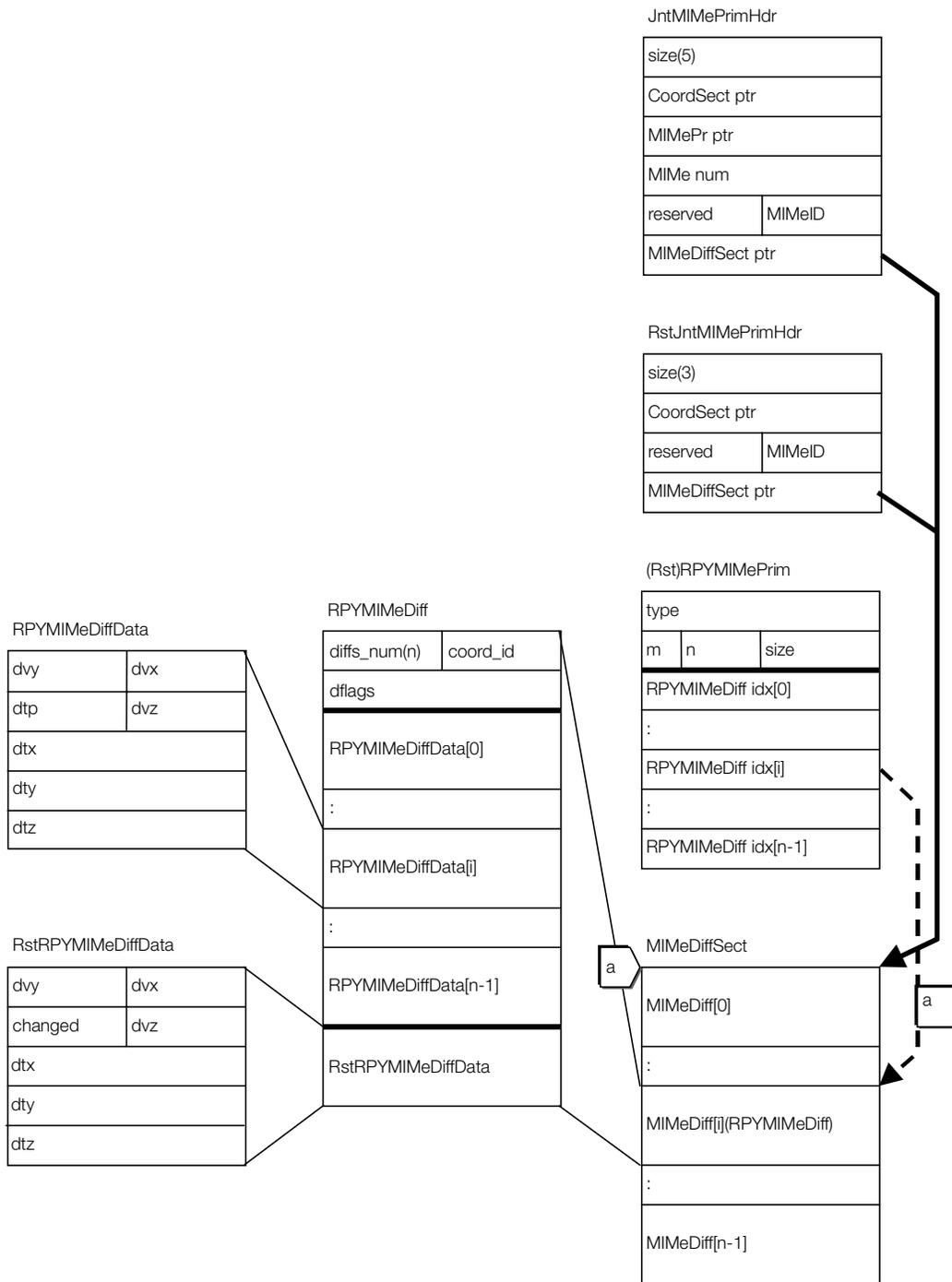
Addendum A: Migrating from TMD to HMD

The following are some differences between using libgs with the TMD format and libhmd with the HMD format:

- The GsSortObject…() initialization functions are replaced be GsSortUnit().
- Object handler structures change from GsDOBJ… to GsUNIT. A primitive's behavior is not controlled with attributes but by switching to a different primitive driver.

  GsUNIT contains two members. *coord* points to the coordinate system. *primtop* points the start of the primitive block. As before, the local-world matrix should be calculated from *coord* and GTE should be set before calling GsSortUnit(). *primtop* is passed on to GsSortUnit().

- Pointers specified in the HMD data are converted to real addresses via GsMapUnit().

  GsScanUnit() is used to get addresses and types for embedding pointers to the primitive drivers. The application program looks at the type bit, determines which primitive driver should be linked, and sets up the obtained address. When the INI bit in the type field is on, a function for initializing the sections that are defined locally--for example, GsMapCoordUnit()--is called.

Addendum B: Installation status of HMD primitive drivers

An Excel spreadsheet is provided on the Technical Reference CD, containing the current installation status of primitive drivers in libhmd. To access it, open this chapter (Chapter 18, Addendum B) in the *Run-Time Library Overview* on the CD and click the link.

CONFIDENTIAL

Chapter 19:
PDA Library (libmcx)

Table of Contents

CONFIDENTIAL

Overview

The PDA library provides access to various functions of the PDA when the PDA is inserted into a Memory Card slot. This chapter provides an overview of these functions.

Library and Header Files

The PDA library file `libmcx.lib` must be linked to any programs that call PDA library services.

Source code must include the header file `libmcx.h`.

Library functions

- Get PDA status
- Switch PDA application
- Access PDA memory
- Turn file transfer display on/off
- Get/set real time clock
- Get/set alarm clock

Checking PDA status

The PDA can be inserted and removed at anytime with the power on, and a user application must be prepared to handle this situation. When a card is inserted, the libmcx functions McxCardType(), followed by McxSync(), can be called to check for the presence of and identify a PDA or Memory Card.

If the value of *result from McxSync() is McxErrNoCard, then no card (neither PDA nor Memory Card) is present. A *result value of McxErrInvalid means that some card is connected to the Memory Card connector but a communications failure has occurred. *result values of either McxErrSuccess or McxErrNewCard mean that a PDA was detected. For McxExecFlag() only, if a PDA or a Memory Card is inserted, a subsequent call to McxSync() will return McxErrSuccess or McxErrNewCard. Therefore, McxExecFlag() cannot be used to distinguish a PDA from a Memory Card.

Detecting new cards

A PDA that has just been swapped is considered to be a new card, and is handled just like a Memory Card. After the card is inserted and all process registration functions have been called, calling McxSync() will return McxErrNewCard in *result*, reporting that PDAs have been swapped. Subsequently, processes can exit normally and their results can be used.

However, when *result* is McxErrNewCard, processing will be interrupted for McxExecFlag(). Unverified flags should first be cleared using MemCardAccept() and _card_clear(), then process registration should be performed again.

Run-Time Library Overview

Conflicts with other libraries

Conflicts with use of libcard, libmcrd and card BIOS

Since interrupts from libmcx conflict with libcard and libmcrd, libmcx cannot be used when asynchronous functions from libcard or libmcrd are pending. Otherwise, libmcx does not conflict with libcard or libmcrd since libmcx does not use any libcard, libmcrd, card BIOS functions, or HwCARD/SwCARD events.

Constraints between libmcx and libpad, libapi controller functions and libcard

The initialization and start-up functions in the libraries must be called in a specific sequence. Otherwise, improper operation may occur during program execution. In the figure below, the function at the starting point of the arrows must be called first.

Figure 19-1: Function calling sequence
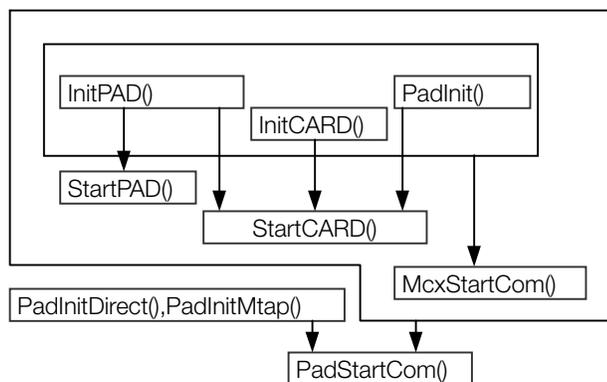
A standard sequence in a program would be as follows.

```
PadInit();
InitPAD();
StartPAD();
InitCARD();
StartCARD();
McxStartCom();
PadInitDirect();
PadStartCom();
```

However, the following three sets of function pairs cannot be called simultaneously:

[PadInit()],[InitPAD(),StartPAD()],[PadInitDirect(),PadStartCom()].

An appropriate set from the three sets should be selected when writing programs.

MemCardInit() and MemCardStart() can be replaced with InitCARD() and StartCARD(). InitPAD() and StartPAD() can be replaced with InitTAP() and StartTAP() or InitGun() and StartGUN().

PDA

The PDA is a device that functions like a Memory Card, allows saved files to be executed as ARM7 programs, and provides LCD, LED, speaker, and IR communication functions.

Hardware

The following table gives the specifications of the PDA hardware.

Table 19-1: PDA Memory Card specifications

Item	Description
Capacity	120 KBytes formatted (accessed as 128 byte sectors)
Communications	Synchronous serial communications using the controller port.
Access speed	(1) Access inhibited for 20 msec after 1 sector written. (2) Maximum continuous read speed: approximately 10 KBytes/sec.
Other	Can be hot swapped without turning off the PlayStation. Guaranteed for 100,000 writes.

Guidelines for using the PDA

The PDA is a resource that can be shared by multiple applications. Therefore it should be used in accordance with guidelines that allow it to be used as a common resource.

Terminology

Memory capacity is expressed in units of blocks, as needed for the product catalog.

Each block contains 64 128-byte sectors for a total capacity of 8,192 bytes.

File names

When using Memory Card file header extensions (described later), Memory Card filenames must be defined so that the hyphen in SLPS-xxxxx is replaced with a capital "P", i.e., SLPSPxxxxx. Otherwise, the application will not be started as a PDA application by the start-up application and file list animation icons will not be displayed in the file list.

File header

Always place a standard header at the start of each file. Two types of file headers are available: Memory Card extended file headers and existing Memory Card file headers . These are described in more detail below. The extended file headers are upward compatible with existing file headers.

Table 19-2: Existing File Header (non-PDA compatible)

Item	Size	Notes
Magic	2	Always 'SC'
Type	1	0x11/0x12/0x13 *1
No. of blocks	1	
Document name	64	Shift-JIS code *2
Pad	28	All 0x00
CLUT	32	CLUT entry x 16 *3
Icon image 1	128	Required (16 x 16bit x 4plane)
Icon image 2	128	Only when Type=12,13
Icon image 3	128	Only when Type=13

*1: Type: Indicates the number of icon images. Preset icon images are displayed sequentially to provide animation.

*2: 32 full-width characters, non-kanji and Level 1 kanji only. However, 0x84bf through 0x889e cannot be used. If the string is less than 32 characters, it must be terminated with 0x00.

*3: CLUT: Actual display color corresponding to a color number
CLUT= (B[4:0] << 10) | (G[4:0] << 5) | R[4:0]

Table 19-3: Memory Card extended file header

Item	Size	Notes	Changes
Magic	2	Always 'SC'	
Type	1	0x11/0x12/0x13	
No. of blocks	1	No. of blocks occupied by the file	
Document name	64	Shift JIS code	
Pad	12	All 0x00	*
File list icons No. of animation icons	2	Displayed when files are listed No. of icons used for animation (=n1)	*
File type	4	"MCX0", "CRD0"	*
Game selection icon No. of pages	1	No. of animation pages switched on the game selection screen (=n2)	*
User-defined device drivers No. of entries	1	Total number of user-defined device drivers (=n3)	*
Reserved	4	All 0x00	*
Program starting address	4	unsigned long (ARM7)	*
CLUT	32	CLUT entry x 16	
PlayStation Memory Card set-up screen icon image 1	128	Required (16 x 16bit x 4plane)	
PlayStation Memory Card set-up screen icon image 2	128	Only when Type=12,13	

CONFIDENTIAL

Item	Size	Notes	Changes
PlayStation Memory Card set-up screen icon image 3	128	Only when Type=13	
Entry table for device drivers	128×n4	4bytes x 2 for each device driver (n4=(n3/16) rounded up)	*
File list icon images	128×n1	32 x 32bit x n1 icons	*
Entry table for game selection icons	128×n5	4bytes x 2 for each animation page (n5=(n2/16) rounded up)	*
Game selection icon image	128×n6	32 x 32bits x n6 icons (n6 is the total sum of all the animation icons appearing in the game selection icon entry table). Put the icon image starting address on the 128 byte boundary.	*

*An asterisk appears in the "Changes" column for items which change when they become PDA file headers.

Extensions provided with PDA file headers

Previously, information had been added to the pad area in the file header. With the PDA file header, device driver entry table, and icon data are added immediately after the icon images.

This additional information is used for the file list and game selection functions of the PDA.

Please refer to the "Icons" section for a detailed description of icon-related data.

File list icon images, animation icons

File list icon images are used for icon animation when the list of files stored in the PDA is displayed. Animation is performed by switching 32 x 32 dot black-and-white icon images in sequence.

Generally, when the number of file list animation icons is one or more, animation is performed using the icon images that have been prepared for the file list. Files having file type "CRD0" are not PDA applications but they do contain icons for PDA file list animation. For these files, animation is performed using the file list icon images.

If the file type is "MCX0" (i.e., the file is a PDA application) and if no file list icons are available (i.e., the animation icon count is 0), then the page 1 icon of the game selection icon is used for animation.

For files with an existing file header, file list animation is performed by converting the standard icon image to a 32 x 32 dot black-and-white image.

If the file type is "CRD0", the game-selection icon page count, the user-defined device driver entry count, the reserved area, and the program starting address fields must all be filled with NULLs (0x00). Thus, there will be no device driver entry table, game selection icon entry table, or game selection icon image fields.

File types

If a file is a PDA application, the file type will be "MCX0", indicating that the file can be executed as an ARM program. For all other file types, it is assumed that the file is not a PDA application and that it cannot be executed from the PDA start-up application.

File type "CRD0" denotes a file that is not a PDA application, but does contain 32 x 32 dot icons for performing animation on the LCD display in the PDA file list. The file list animation icon count is meaningless for file types other than "MCX0" and "CRD0".

Game selection icons

The game selection icon image is icon animation data that will be displayed as part of the PDA's game selection function (see the section on icons). The icon page count contains the number of animation pages for the application, such as the "game title" animation page, the "game summary" animation page and the "developers list" animation page. The actual icon images for the game selection icons are arranged in sequence beginning with the start of the "game selection icon image".

The icon entry table contains the icon page count, the animation icon switching speed, and the icon image starting address for each animation page. The actual frequency at which animation is updated is (30 / "animation speed for page n") icons per second.

The icon image address must be specified as an absolute address where the start of the header is defined to be 0x2000000. Put the icon image starting address on the 128 byte boundary (LSD is 0x00 or 0x80).

The n6 at the end of Table 19-3 indicates the total number of icons that have been prepared for game selection. n6 holds the value for "page 1 animation icon count" + "page 2 animation icon count" + ... + "page n2 animation icon count".

Table 19-4: Game selection icon entry table

MSB	Byte	Byte	LSB
Word (32bit)			
Reserved	Page 1 animation speed	Page 1 animation icon count	
Icon address for page 1 animation			
Reserved	Page 1 animation speed	Page 2 animation icon count	
Icon address for page 2 animation			
…….	…….	…….	
…….			
Reserved	Page 1 animation speed	Page n2 animation icon count	
Icon address for page n2 animation			

Note: Put each icon address on the 128 byte boundary

User-defined device driver entry count

The user-defined device driver entry count is the total number of user-defined device drivers set up to call special functions of the PDA from the PlayStation. If there is no device entry table, this field must be set to 0.

Device entry table

Devices are numbered as 128, 129, 130, ... from the beginning of the table. If the number of devices exceeds 16, the next 128 bytes are used as an additional table to hold entries for 136, 137, ...

Table 19-5: Device entry table

Word (32bit)
Fixed part of device 1 data length
Device 1 data handling routine
Fixed part of device 2 data length
Device 2 data handling routine
……..
Fixed part of device n3 data
Device n3 data handling routine

Program starting address

The address specified here is interpreted as an absolute address used for executing the program. No errors are reported if the specified address is invalid. The file header must start at absolute address 0x2000000. When switching applications, execution of the application will always be started from this address.

Icons

Icon types

The PDA uses three types of icons: PlayStation Memory Card set-up screen icons, (PDA) file list icons, and game selection icons. The table below describes the functions of these icons.

Table 19-6: Icon Types

	PlayStation Memory Card set-up screen icon	PDA file list icon	Game selection icon
Icon size	16×16	32×32	32×32
Color	16 colors selected from 32,768	Black and white	Black and white
Page count	1	1	n2 (game-selection icon page count)
Icon page count	1-3 icons (specified by Type in file header)	n1 file list icon animation count)	n6 (total animation page count of all animation pages appearing in the game-selection icon entry table)
Minimum required icon count	1 needed for all files)	0 (if none, use other icon instead)	1 (needed only for PDA applications)

If the file is a PDA application but there are no PDA file list icons, the first icon page from the game selection icon is used.

If the file is not a PDA application, the PlayStation Memory Card set-up screen icon is used.

Even if the file is not a PDA application, it may still use the PDA file list icon by setting the file type to "CRD0".

File list function

The file list function allows files that are stored in the PDA to be verified using icon animation.

Figure 19-2: File list functions

File 1 (non-PDA)
PlayStation file set-up screen icon]

Data

File 2 (PDA data file)
[File list icon]

Data

File 3 (PDA application)
[Page 1 game selection icon]

Appli
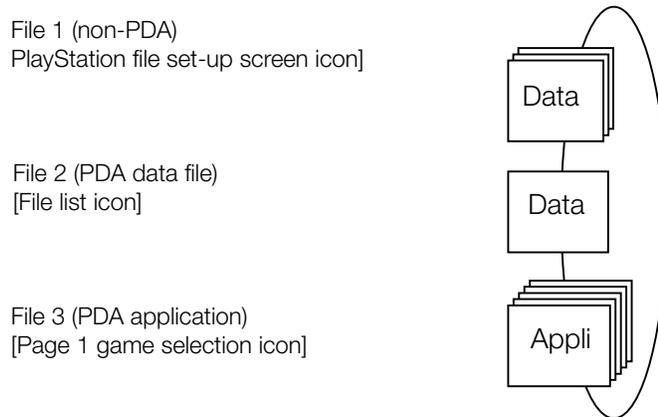
File list animation is performed using the file list icons. However, if no file list icon is available, file list animation will still be performed by automatically substituting PlayStation Memory Card set-up screen icons. This allows non-PDA files to be seen through the file list.

Table 19-7: Icons used in the file list

File type	File list icon animation icon count	Game selection icon page count	
MCX0,MCX1	1 or more	*1	File list icon
MCX0,MCX1	0	1 or more	Page 1 game selection icon
MCX0,MCX1	0	0	(setting at left is prohibited)
CRD0	1 or more	*2	File list icon
CRD0	0	*2	(setting at left is prohibited)
non-PDA	*2	*2	PlayStation file set-up screen icon

*1: This is the value equivalent to the icon page count.

*2: Based on the file header definition, this is always "00".
Note that the frequency at which the icon is updated depends on the type of icon used.

Table 19-8: Icon animation update frequency

Icon used in animation	Icon update frequency (fps)
PlayStation Memory Card set-up screen icons (2 icons)	2
PlayStation Memory Card set-up screen icons (3 icons)	3
PDA file list icon	6
Game selection icon	Frequency shown in game selection icon entry table. 30/f (fps)

Game selection function

The game selection function displays the game-selection icon and is used to start a desired PDA application. This function will only display PDA application files (i.e., files with type "MCX0"). Each PDA file can have more than one page, and each page can use animation with multiple icons.

Since multiple pages can be animated, it would be possible, for example, to have the first page be game title animation, the second page be game summary animation, and the third page be animation with the developers' names.

Figure 19-3: Game selection function
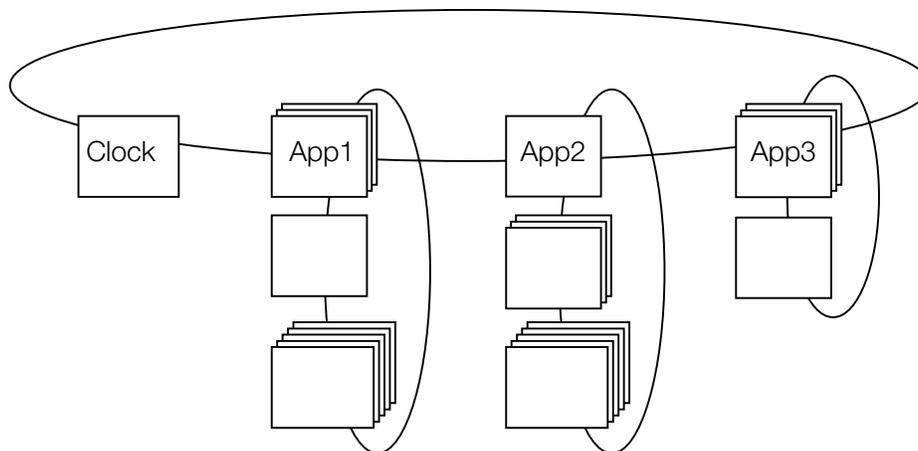
Notes on icon entry table and icon image position

Always place the icon image starting address on the 128 byte boundary (LSD is 0x00 or 0x80) and use an icon image starting address for addresses in the icon entry table.

Standard use of PDA functions

Initialization and termination

1. The PDA system is activated using McxStartCom(). PDA function-calling interrupts are enabled, and all PDA process registration functions are available for use.
2. The McxXXX() function group is called to access the PDA.
3. McxStopCom() is used to inhibit PDA function-calling interrupts and stop the PDA system.

Using asynchronous functions

All functions that access the PDA are asynchronous functions. To access the PDA, an operation must first be initiated (reserved) using a process registration function. McxSync() must then be used to check for process completion. If the operation was successful, the transmit data or the data in the receive buffer will be valid.

The following example uses McxGetTime() to get date and time information. The same procedure can be used for all other asynchronous functions.

1. Call `McxGetTime(port, buff)`
2. Wait for process to finish using `McxSync(0, &cmd, &result)` or (`!McxSync(1, &cmd, &result))`;
3. Look at result. If successful, the contents of buff are valid and can be used as calendar data.

Saving a PDA application or file list icon images file

Saving a PDA application to the PDA is similar to saving a file to a Memory Card, with a few exceptions. After saving, McxExecFlag() should be used to embed "PDA application flag" information in the FAT. Also, in order to avoid writing PDA application programs to alternate sectors, McxShowTrans() and McxHideTrans() must be used before and after the file save. ("PDA application flag" information should be embedded as necessary. It is not required. Calling McxShowTrans() and McxHideTrans() when reading a file is also not required.)

For McxExecFlag(), the DIRENTRY of the file is obtained with firstfile(), and member head is divided by 64 and passed as the second parameter, block. (Member head contains the header sector number of the saved file. By dividing this by 64, the header block number can be calculated.)

Example of saving a PDA application

```
1. McxShowTrans(port, 1, TimeOut);
2. Open file ("filename");
3. Write file
4. Close file
5. McxHideTrans(port, 1);
6. firstfile("filename", &direntry);
7. McxExecFlag(port, direntry.head/64, 1);
```

When saving a file with PDA file list icon images rather than a PDA application, the "original copy" information is unnecessary, so 6.firstfile() and 7.McxExecFlag() would not be called.

Enabling the speaker, IR communication, and LED

Because power from the PlayStation to the Memory Card connector is limited, the enabling of the IR module, speaker, and LED, which are high-current consumption hardware devices on the PDA, needs to be controlled from the PlayStation.

The table below shows the amount of current consumed by these devices. The maximum current that can be supplied from the PlayStation to all the Memory Card connectors is 160 mA total. McxCurrCtrl() should be used to control the enabling of the devices so that this value is not exceeded. These restrictions must be followed, as exceeding the maximum current may result in improper operation of the PlayStation.

CONFIDENTIAL

Table 19-9: Device current consumption

Device name	Current consumption
CPU chip	10mA
IR module transmitter	70mA
Speaker	20mA
LED	10mA

Device enable status is obtained for all PDAs using McxAllInfo() (for a non-Multi Tap configuration, two places can be checked--Memory Card connectors 1 and 2, and for a Multi Tap configuration, there are eight places--A - D for each connector). The result from McxAllInfo(), together with the table above, can be used to calculate the current consumption for enabled devices.

If the sum of this current consumption and the current consumption of the devices to be enabled does not exceed 160 mA, the devices can be enabled using McxCurrCtrl(). If the value exceeds 160 mA, the devices cannot be enabled.

The user is informed that a device cannot be enabled through a message displayed on the TV screen. The total current consumption could then be kept from exceeding 160 mA by disabling the device using McxCurrCtrl() or disabling a device on another PDA.

CONFIDENTIAL

Chapter 20:
Memory Card GUI Module (mcgui)

Table of Contents

Overview

The Memory Card GUI module (McGUI) is a program that complies with the TRC (Technical Requirements Checklist) and provides Memory Card access functions such as load and save, along with support for the user interface. By incorporating McGUI in user game titles, the work involved in coding the Memory Card screen for saves, loads, etc. can be reduced.

Module and Header Files

The filename of the Memory Card GUI module is `mcgui.obj`. It must be linked to programs that call McGUI module services.

With languages other than Japanese (English, French, German, Italian, Spanish), mcgui_e.obj must also be linked.

Programs that call McGUI module services must include the Memory Card GUI header file `mcgui.h`, which defines structures, functions, and macros needed to use the module.

Required libraries

McGUI uses the following libraries, which must be linked to programs that use McGUI:

- Kernel library (libapi) 4.3 or later
- Basic graphics library (libgpu) 4.2 or later
- Controller/Peripherals library (libetc) 4.2 or later
- Memory Card library (libcard) 4.3 or later
- Extended Memory Card library (libmcrd) 4.5 or later
- Sound library (libspu) 4.3 or later
- Extended sound library (libsnd) 4.3 or later

Module functions

McGUI has 2 functions: McGuiSave(), which saves game data, and McGuiLoad(), which loads game data.

Before calling each function, a data structure is initialized with appropriate values. Some internal checking is done to make sure the values are valid. However, not all conditions are checked, so invalid values may not necessarily generate errors.

Save game data – McGuiSave()

This function invokes the save operation of the Memory Card screen to save game data.

Figure 20-1: Save Operation of the Memory Card Screen

Select Memory
Card Slot

Check Status of
Selected Slot

Check to see if Memory
Card is not installed,
not initialized, etc.

Check Contents of
Memory Card

See if previously saved
data is present; check if
empty blocks are
present, etc.

Save Game Data

Return to Title

CONFIDENTIAL
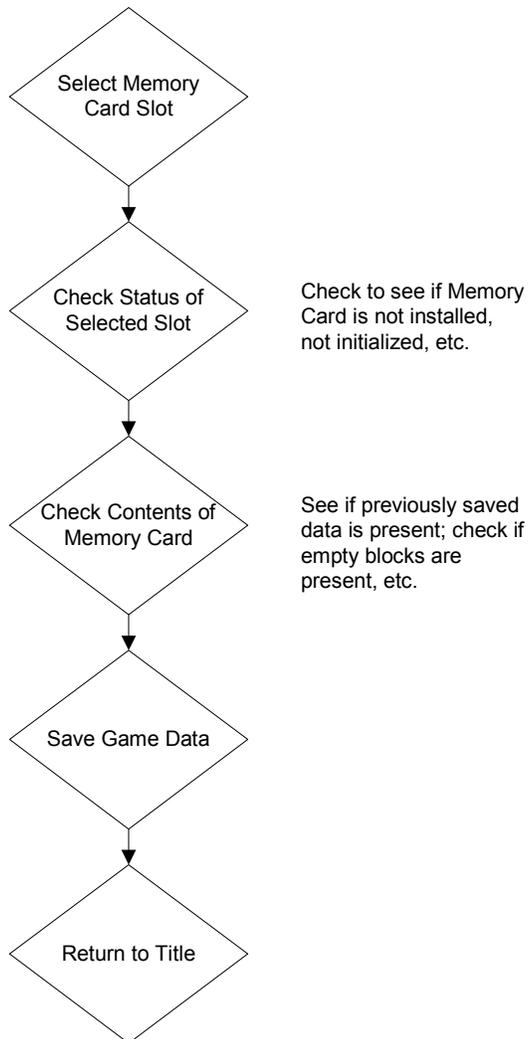
Load game data – McGuiLoad()

This function invokes the load operation of the Memory Card screen to load game data.

Figure 20-2: Load Operation of the Memory Card Screen

Select Memory
Card Slot

Check Status of
Selected Slot

Check to see if Memory
Card is not installed,
not initialized, etc.

Check Contents of
Memory Card

See if previously saved
data is present.

Load Game Data

Return to Title
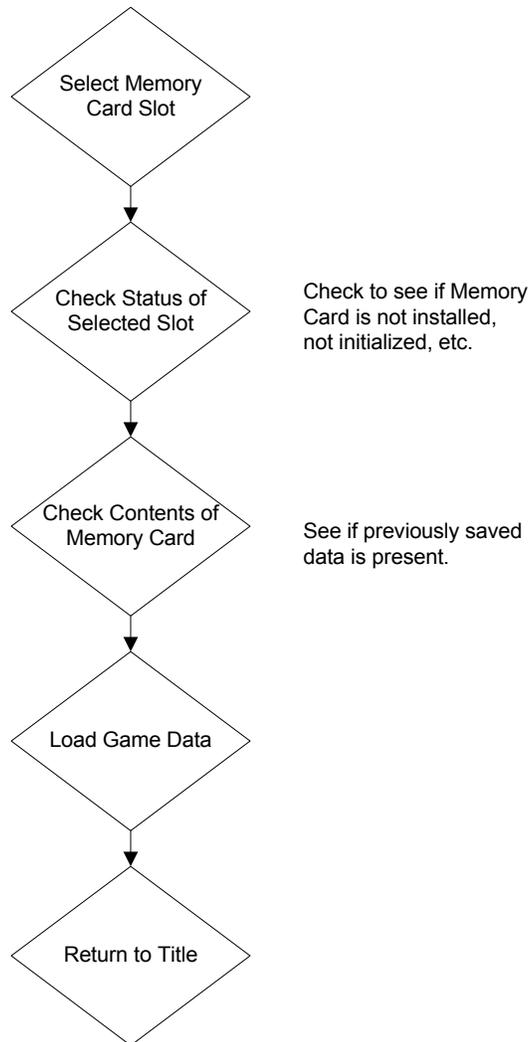
Supported controllers

The following controller terminal types are supported by McGUI.

Table 20-1: Supported controllers

Terminal type	Controller name	Main controller model number
1	Mouse	SCPH-1030
2	16-button analog	SLPH-00001(Namco Corp.)
4	16-button	SCPH-1080,1150,1200
5	Analog joystick	SCPH-1110
7	DUAL SHOCK	SCPH-1200

Gun controllers and Multi taps (except 1-A, 2-A) are not supported.

Language setting

McGui currently supports Japanese, English, French, German, Italian and Spanish.

The mode is initially Japanese and this can be switched to each language by executing the McGuiSetExternalFont() function.

Initialization and termination

McGUI internally calls the following initialization functions:

- SsInit()
- SsStart()
- ResetGraph()
- ResetCallback()

McGUI calls the following termination functions on exit:

- SsEnd()
- SsQuit()
- StopCallback()

Your application must do the following initializations before calling an McGUI function:

- Initialize libmcrd using MemCardInit(). MemCardStart() and MemCardStop() should also be executed when appropriate.
- Initialize controllers with functions that set up the receive data buffers on the application side such as InitPAD() and PadInitDirect(). The receive data buffer address should be set in the McGuiEnv structure. Functions that cannot set up the receive buffer, such as PadInit(), are not supported.

Limitations

This section describes guidelines and precautions relating to the use of McGUI.

Number of blocks of game data

The block count is fixed at the number of blocks specified when data was first saved to the Memory Card by McGUI. If saved data already exists, it cannot be overwritten by data that contains a greater number of blocks. If the amount of saved game data is expected to increase as the game proceeds, the initial save should be performed with the maximum block count.

Textures

Coordinates where texture data is loaded into the frame buffer (VRAM)

The coordinates where texture data is loaded are set in the TIM header information. Set the coordinates so that they appear at the start of the texture page along with the images and CLUTs.

Using the frame buffer (VRAM)

According to the coordinates specified in the header information of the texture data (mcgui.tim), McGUI loads the images and CLUTs to the frame buffer. The 67 lines below the start of the image loading position are used as an internal work area. In English mode, the 64x128 dot rectangular area to the right of the image loading area is used as the external font area.

Figure 20-3: Location where textures are loaded in the frame buffer
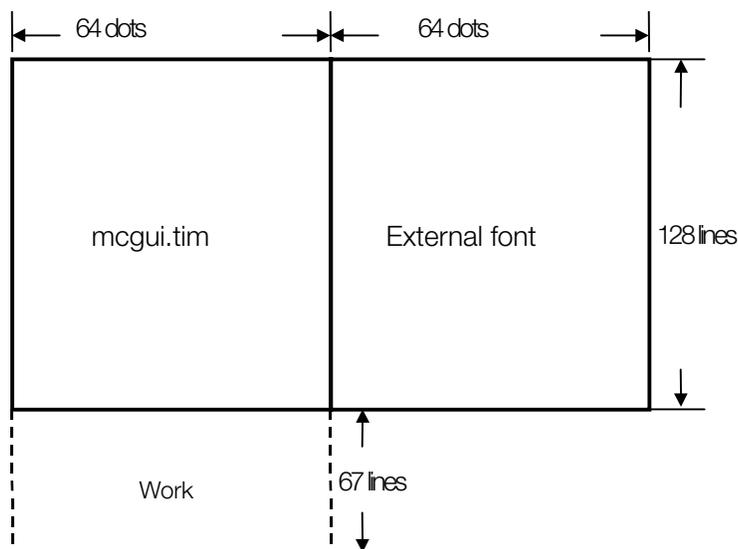
Texture data always uses 128 x 128 8-bit CLUT mode. The coordinates and sizes of titles and mouse cursors are also fixed.

The following is the texture data (mcgui.tim) used by McGUI.

Figure 20-4: mcgui texture data structure
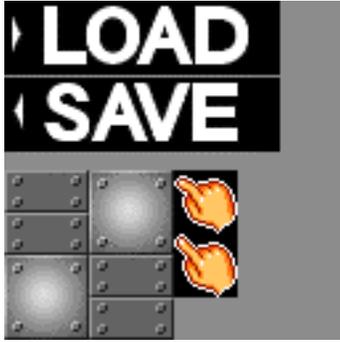
(128 x 128 pixels, 8-bit CLUT mode)

Graphics

The screen resolution is fixed at 320 x 240. Since drawing is performed with double frame buffers, the rectangular region (0,0)-(319,479) in the frame buffer will be destroyed.

Sound

When creating SEQ data:

- SEQ data will always be played at TICK60. SEQ data should be created presuming TICK60.
- The maximum number of simultaneous voices is 23. The 24th voice is reserved for special effects.

When creating sound effects (sfx) data: sfx data is played back using the SsUtKeyOnV() function with note:60 fine:0. SE data should be created so that it will be played back properly at note:60 fine:0.

Note: The contents of the sound buffer are not guaranteed once McGUI completes. After completion, waveform data should be transferred as needed.