Run-time Library Overview

Beta Release 3.6 - Dec 1996

Summary Table of Contents

About this Manual

This manual is a **beta** release of the Overview manual for Run-time Library 3.6. For related descriptions of the PlayStation run-time library functions and structures, please refer to the *Run-time Library Reference* in the PlayStation Developer Reference Series.

Changes Since Last Release

This manual has been significantly revised since release 3.0 of the Run-time Library Overview. Numerous corrections and additions have been made throughout the text.

Related Documentation

This manual should be read in conjunction with the Run-time Library Reference, since it defines all structures and functions available within the Run-time library.

The complete set of the Developer Reference Series includes the following:

- Programmer Board Set (DTL-H2000)
- PlayStation Operating System
- PlayStation Hardware
- Run-time Library Overview
- Run-time Library Reference
- Psy-Q Development System
- CD Emulator
- CD Generator
- 3D Graphics Tool
- Sprite Editor
- Sound Artist Tool
- File Formats

Note that the Developer Support BBS posts late breaking developments regarding the Libraries and also provides notice of forthcoming documentation releases and upgrades.

Manual Structure

The Library Overview contains eighteen chapters providing general descriptions of each of the high-level and low-level runtime ibraries.

Chapter 1	Describes the general features of the PlayStation operating system and applications as well as the structure of the run-time libraries and their relationship to the PlayStation operating system.
Chapter 2	Describes the Kernel library (libapi) which provides an interface between the PlayStation operating system and applications.
Chapter 3	Describes the “Standard” C library (libc/libc2); a subset of the standard C library. It includes, in part, character functions, memory operation functions, character class tests, non-local jumps, and utility functions.
Chapter 4	Describes the Math library (libmath) which contains ANSI/IEEE754 compliant math functions that include a software floating point computation package.
Chapter 5	Describes the Memory Card library (libcard) for reading/writing to the PlayStation memory card and calling the memory card BIOS service.

Chapter 6	Describes the Data Processing library (libpress) for compressing (encoding) and expanding (decoding) image and sound data.
Chapter 7	Describes the Basic Graphics library (libgpu) for executing drawing engine commands to handle data entities such as sprites, polygons and lines.
Chapter 8	Describes the Basic Geometry library (libgte) which uses the PlayStation GTE as a co-processor to handle high-speed geometry operations.
Chapter 9	Describes the Extended Graphics library (libgs) which uses the Basic Geometry library and the Basic Graphics library to construct a 3-dimensional graphics system. The data handled here is more abstract entities like objects, background surfaces etc
Chapter 10	Describes the CD-ROM library (libcd), for controlling the built-in PlayStation CD-ROM drive as well as the Streaming library (libcd) for “streaming” real-time data such as movies, sounds or vertex data stored on high-capacity media.
Chapter 11	Describes the Controller/Peripherals library (including libetc, libgun, libtap and other library functions) for performing low-level interrupt processing and controller-related functions.
Chapter 12	Describes the Link Cable library (libcomb) which provides a service for connecting PlayStation units via a “link” cable for linked operation.
Chapter 13	Describes the Extended Sound library (libsnd) for sound data with the PlayStation.
Chapter 14	Describes the Basic Sound Library (libspu) for controlling the SPU (sound processing unit).
Chapter 15	Describes the Serial Input/Output Library for supporting communications between the PlayStation and a PC.

Typographic Conventions

Certain Typographic Conventions are used through out this manual to clarify the meaning of the text. The following details the specific conventions used to represent literals, arguments, keywords, etc.

The following conventions apply to all narrative text outside of the structure and function descriptions.

Convention	*Meaning*
|	A revision bar. Indicates that information to the left or right of the bar has been changed or added since the last release.
`courier`	Indicates literal program code.
Bold	Indicates a chapter or section title.

The following conventions apply within structure and function descriptions only:

Convention	*Meaning*
Medium Bold	Denotes structure or function types and names.
Italic	Denotes function arguments and structure members.
`{ }`	Denotes the start and end of the member list in a structure declaration.

Ordering Additional Manuals

Additional printed copies of this documentation can be ordered by contacting Developer Support for your territory.

Chapter 1: Overview of the Playstation OS

The PlayStation OS is flexible and powerful. Using this OS, the PlayStation's capabilities can be fully exploited, and applications can be developed that make the most of these capabilities.

Table of Contents

The PlayStation OS

The PlayStation OS is an operating system that 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 installed in the machine. If the CPU and peripheral devices can be depended upon to be fast enough, then you can focus on using and developing the services provided by the OS. Since there will be no need to spend time considering how to get the most out of the hardware's capabilities, you can concentrate on coming to terms with the programming.

The design concept of the PlayStation OS is 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 is a combination of services to control PlayStation hardware and the R3000.

Moreover, each service is provided as a C language function. By using C, it is possible to program with improved maintainability while also making the most of special features such as block structure description and function calling.

Features of the PlayStation OS

Many features are realized in the PlayStation design concept.

Programming with C

Most services, such as controlling the R3000 CPU and the PlayStation suite of hardware, are provided as C language functions. Therefore programming can be carried out consistently with C.

R3000 Functions Are Made Easy to Use

Interrupt control procedure with the R3000 is said to be complicated, but the PlayStation OS adopts a “despatcher” system which the OS processes on its behalf and provides an easy interface. The overhead which accompanies the despatch is naturally kept to the minimum. Moreover its usage at a native level gives support which is not available publicly with the normal OS. Because of this, the chip's capability can be fully exploited and high quality tuning can be carried out. Since everything can be controlled with C, it is not necessary to master R3300 assembler, which is difficult to understand.

Lightweight-Small Size, Emphasis on Performance

Even high-speed games can be sluggish if the OS is slow. Because of the importance placed on the application's performance, the resident memory size of the PlayStation OS is kept to a minimum (64K bytes), and designed so that the CPU's exclusive time is also kept to a minimum. In addition, the OS system table is disclosed, and consideration given to such matters as the OS's extension and the acquisition of internal data.

Moreover, the checks seen in the typical OS have been drastically cut in order to improve the speed of the PlayStation OS. These checks do need to be carried out in the case of applications that are responsible for doing so. Therefore, there is a danger in carrying out an operation which is normally prohibited. However, a high level of tuning is possible by careful programming using the functions within the OS.

Provision for Hardware Functions

Up to now, when we have talked about controlling video game machine hardware, we have had to analyze it and worry about every last detail of the assembler. This burden is lightened by the PlayStation OS providing each function as a C language function, and the overhead of each function is kept to the minimum.

Single and Multitasking

The PlayStation OS is basically a multitasking OS which can carry out many tasks asynchronously. The multitasking control is suitable for controlling a CD-ROM drive, which is a comparatively slow device, and for the BGM music player.

Those who are used to traditional programming or single tasking can choose whether or not to use multitasking. The OS is in single tasking mode immediately after activation and the tasking mode can be selected by setting at this point.

The File System Device Driver

The PlayStation OS adopts a device known as the file system device driver which has not been used in game machines until now. The device can be attached or detached freely and multiple file systems can co-exist. With this development efficiency can be improved.

Activation and Operation of 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).

Activation of the OS

It jumps first to a special address in ROM and carries out a check on connected hardware (such as a CD-ROM drive).

It then carries out disk class definition on the CD-ROM drive. If the disk is suitable, it then searches for the system configuration file (`SYSTEM.CNF`) and executes it.

If the disk is not installed, a ROM resident program [OSD] (On Screen Display) equipped with both CD player and memory card control functions is executed.

Please refer to the Hardware Guide for the boot sequence

System Settings File (SYSTEM.CNF)

The PlayStation system configuration is now described. The parameters are described from the start of the line in the format of “keyword = content”. The entire description is English numbers and capital letters, and there must be a space on either side of the equal sign.. Half characters cannot be used. Also, the values are all hexadecimals. If there is more than one of the same parameter in the file, priority is given to the value which is found first.

The following parameters can be described:

Table 1–1: List of system setting file key words

Key word	Contents
BOOT	Give the file name to activate. Default is cdrom:PSX.EXE;1 Example: `BOOT = PSXAPP.EXE;1`
TCB	Give the task management block number. Default is 4. Example: `TCB = 5`
EVENT	Give the event management block number. Default is 16 Example: `Event = 5`
STACK	The value of the stack pointer at the time of activation of the file designated by boot. Default is 0x801ff00. Example: `STACK = 800fff0`.

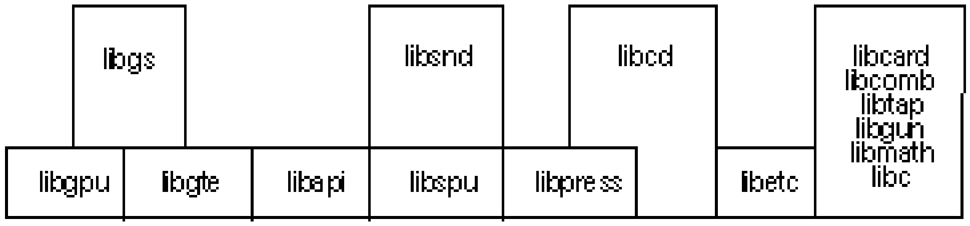
Structure of the PlayStation OS

Various libraries are provided in the PlayStation documentation, beginning with the kernel library that provides application interfaces (API) with the PlayStation OS. In this chapter we will explain the configuration format of these libraries and briefly introduce them.

PlayStation OS Library Components

There are two types of PlayStation libraries, low level libraries and high-level libraries, depending on their relationship to the PlayStation OS. These form a two-level library structure. Programs may use any level of library as needed, and may use both level libraries concurrently. This is expressed graphically as follows:

Figure 1–1: Playstation library structure



An overview of each library follows below.

libapi (Kernel Library)

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

libetc (Peripheral Library)

A library that performs callback control for using controllers and other peripheral devices and processing low-level interrrupts.

libcd (CD-ROM Library)

This library reads a program and image and sound data from a CD-ROM drive, and performs playback of DA (digital audio) and XA sound.

This library includes fast disk access through a file name key, and a support function for simultaneous data reading and processing streaming techniques.

libgpu (Basic Graphics Library)

A library for creating commands for the drawing engine and for building a drawing command list. The data handled here is for simple entities like Sprites, polygons and lines.

libgte (Basic Geometry Library)

This is a library for controlling the GTE (geometry transformation engine). The data handled here is simple, like matrices and vertices.

libgs (Extended Graphics Library)

libgs is a three-dimensional graphics system which uses libgpu and libgte. The data handled here are larger entities like objects, background surfaces etc.

libspu (Basic Sound Library)

This library controls the SPU (sound processing unit).

libsnd (Extended Sound Library)

This library plays as background, sound production sequences that have been prerecorded as score data.

libpress (Data Processing Library)

A library for compressing (encoding) and expanding (decoding) image and sound data.

libcard (Memory Card Library)

This library controls the memory card which saves data after reset and power off. It includes the memory card, the file system, and drivers.

libcomb (Link Cable Library)

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

libtap (Multi-tap Library)

This library allows for access to 3-8 controllers and memory cards throughthe option peripheral multi-tap

libgun (Gun Library)

This library provides access to the Light Pen type input equpment which can connect to the controller connector.

libc/libc2 (Standard C Libraries)

This is a subset of the standard C library. It includes, in part, character functions, memory operation functions, character class tests, non-local jumps, and utility functions.

libmath (Math Library)

This library contains ANSI/IEEE754 compliant math functions that include a software floating point computation package.

Chapter 2: Kernel Library

The Kernel library provides an interface (API) between applications and PlayStation OS. It controls the CPU and all other PlayStation basic hardware features. This chapter presents an overview of the kernel library services and explains them.

Table of Contents

Overview

The kernel library provides various types of services that control the PlayStation hardware, including the CPU, and C language functions that focus on jump commands to the kernel.

The services the kernel library provides are listed below.

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

Library Files

The file name of the kernel library is `libapi.lib`. Every program calling services must link with this library.

Header Files

The kernel library header file is `kernel.h`. Data structure and macros for the kernel (except for macros related to input/output service and those specifying registers) are defined in this file.

Programs that call services (except for input/output services and standard C language functions) must always include this header file.

Table 2–1

Content	Filename
Library	libapi.lib
Header	Kernel.h

Kernel Setting

The system designation file `system.cnf` is read at boot time. Memory etc. may be reserved according to the parameters described here. The boot device will differ with the hardware.

Parameters are described from the beginning of a line in “<key word> = <content>” format. Description is done entirely in 1 byte alphanumeric, uppercase characters, and there must always be a space inserted on either side of a sign. Lowercase characters cannot be used. If there are multiple generations of the same parameter within a file, the first found will take precedence. The parameters that may be described are as follows:

Table 2–2: SYSTEM.CNF Parameter Overview

Key word	Setting contents	Default	M inimum
BOOT	Boot file name	cdrom:PSXEXE;1	none
STACK	Stack pointer value when booted	0x801FFF00	0x8010000
TCB	Number of elements of task control block (hex)*	4	1
EVENT	Number of elements of event processing block (hex)*	16	0

* Task management block number X 192 + Event management block number X 32 + 544<4096.

System Table Information ToT

In order to allow each type of system table used in the kernel to be accessed uniformly, system table information is represented by the structure ToT (Table of Tables). ToT is placed at address 0x00000100.

The entries and content of ToT are as follows:

Table 2–3: ToT Entries

Entry	Contents
0	System Reserved
1	Task status queue header
2	Task control block
3	System reserved
4	Event control block
5-31	System reserved

The data structure of ToT is as follows. The structure is defined by the header file `kernel.h` included with the library.

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

It is used as follows:

Example 1: Getting the Pointer to the Execution TCB (Task Management Block)

```
struct ToT *t = (struct ToT *)0x100;
struct TCBH *h = (struct TCBH *)((t + 1)->head);
struct TCB *tcb_exec = (struct TCB*)(h->entry);
```

Example 2: Getting the Pointer to the Start of TCB

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

Example 3: Getting the Pointer to the Start of the Event Management Block

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

Descriptors

Unsigned 32-bit integer descriptors are used to specify system resources such as files and threads in the kernel library. The descriptors have the following bit assignments.

Table 2–4: 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 2–5: 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);
```

Events

An event is a mechanism that tells an application about either external interrupts from a device or exceptions that have arisen within the CPU. Interrupt generations can be searched and processed by polling or a subroutine hook.

Hooked subroutines execute in Interrupt Context (last out), using the interrupt stack maintained within the kernel. This stack is maintained within the kernel reserved memory area (last out). The size is fixed at 4K.

GetSysSp() can obtain an interrupt stack area higher-order address.

An event is specified by two 32-bit integers called the cause descriptor and event class. Designating these indicates the function which creates the particular event. In this way the event can be used as a kind of inter-process communication.

Callbacks

In libraries (libgpu, libsnd etc) that handle devices using DMA, there is a function for registering callback functions in the kernel. Callback functions execute after the testing and processing of first-out events is done.

Callback functions are executed in 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

The execution of the following listed operations or the generation of these phenomena will inhibit all interrupts. A session in which interrupts are prohibited is called a critical section.

System Start

Interrupts are inhibited immediately following system start.

Calling EnterCriticalSection() Function

Interrupts are inhibited when the EnterCriticalSection() function is called. To reenable interrupts, call the ExitCriticalSection() function.

Immediately Following Event Handler Start

Event handler functions are activated by the setting of an interrupt context. Interrupts are inhibited immediately after this activation. Interrupts may be permitted by the function ExitCriticalSection() or by not returning to the original generation, or they may be permitted by calling the function ReturnFromSection() and returning to the original generation.

Service Functions and Critical Section

Services modifying data within the kernel must be executed in code where interrupts are inhibited. Applicable services are described in the explanatory section of function.txt. .

Call the function EnterCriticalSection() to start a critical section.

Interrupt Context

When an interrupt or exception occurs, the program which has been executed up to that point (referred to as Main Flow) is interrupted, and processing begins after address 0x80. The jump code to the kernel interrupt despatcher is at this location; it carries out processes like event generation.

To return to the Main Flow after completing the process, the contents of the register immediately after an interrupt generation are saved in Execution TCB (see Thread Management Service) as Main Flow Context. The status after saving is called Interrupt Context.

Functions such as Event Handlers and Callbacks are executed in Interrupt Context (the former uses the interrupt stack, the latter uses the callback stack). Keep the following cautions and prohibitions in mind.

Cautions

- Halting interrupts for a long time may adversely affect the system. Design the function to be executed in the interrupt context in such a way as to enable completion of the process within the shortest time possible.
- Execution of functions that generate internal exceptions (e.g., ExitCriticalSection()) causes 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 the Event Handler. However, since this breaks off the action of the Interrupt Despatcher and interrupt management returns to Main Flow as incomplete, device related malfunctions may occur. Use ReturnFromException() only for error management functions.

Prohibitions

- 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 the kernel service, are not re-entrant. Re-entrant functions are described in the explanatory section of part 2.
- Execute the function ReturnFromExeption() 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 System

This provides time restrictions and timing adjustments etc., an indispensable feature with game programs.

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

(a) System block

(b) System block (8 spaces)

(c) Vertical synch.

A 16-bit target value may be set in each of each of these counters except (c) .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 (c) 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 2–6: List of Root Counters

Macro	Root Counter	Notes
RCntCNT0		
RCntCNT1	System clock	Target value more than 2
RCntCNT2	System clock (8 spaces)	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. This is an interval of 8 spaces, with 8 divisions. The count for the graphics display has the following timings.

Table 2–7: 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 2–8: Pixel Display Timing and Display Width

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

Upcounting

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

The function StopRCnt() will not stop upcounting. 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 upcounting.

Mode

For each counter the following modes may be set. Modes are defined by macros. The macros in Tables 3-9, 3-10, and 3-11 below can be set by logic.

Table 2–9: Root Counter Mode (1)

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

Table 2–10: Root Counter Mode (2)

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

Table 2–11: 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 2–12: 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

The root counter does not count up using an interrupt. 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 (re)set 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.

Event Control Service

Event control services control execution of programs with events that occur asynchronously during execution of the program, e.g. interrupts.

When using events, events are opened and an event descriptor is obtained by specifying the following:

- The descriptor that is the cause of the event.
- The type of event.
- The event mode.
- The pointer to the handler function.

Event generation test and so on set and execute this event descriptor.

Event Status

There are four statuses of events (defined by macros). Events occurring when event generation is prohibited are ignored. Events immediately after opening are in EvStWAIT status. When this is the status, the generation of the event will be reflected via the status changes to EvStACTIVE after the EnableEvent() function.

The DisableEvent() function switches EvStACTIVE and EvStREADY event states to an EvStWAIT state. Once in the EvStWAIT state, the next event activated by the EnableEvent() function 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

There are the following two modes of events. Modes are defined by macro.

Table 2–14: Event Modes

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

Event Handler

A function may be called when an event is triggered. This function is called the event handler.

The event handler function is executed on an interrupt stack reserved in the kernel. When the program returns from the function it returns to the interrupt source. By calling the ReturnFromException() function, the kernel interrupt processing can be terminated and return forced.

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. Changing the stack can be executed with the SetSp() function.

Cause Descriptor and Type of Event

The relationship between the cause descriptor and type of event is as follows:

Table 2–15: 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	EvSpUNKNOW*
HwSPU	SPU interrupt	EvSpTRAP
HwGPU	GPU interrupt	EvSpTRAP
HWPIO	Extension parallel port interrupt	EvSpTRAP
HwSIO	Extension serial port interrupt	EvSPTRAP
HwCPU	Exceptions	EvSpTRAP
DescUEV | m	User-defined event (m=0~0xffff)	Optional

*Other events are described the individual libraries.

Data Structure

The data structure of an event control block is as follows:

```
struct EvCB {               /*event control block*/
       unsigned long desc; /*cause descriptor*/
       long status;        /8 status*/
       long spec;          /*event type*/
       long mode;          /*mode*/
       (long *FHandler)(); /*function format handler*/
       long system [2];    /*system reserve*/
};
```

Event Creation

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

Event Clear

Switching the state of an open event from EvStALREADY to EvStACTIVE is known as an "event clear," and may be performed by UnDeliverEvent() or TestEvent() functions.

The UnDeliverEvent() function makes a request using source descriptor and event classification arguments, and clears all applicable events.

TestEvent uses an event descriptor argument; if an event related to an event descriptor is in the EvStALREADY state, the event is switched over to the EvStACTIVE state. 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.

```
longev;
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.

Thread Control Service

This is a service that performs thread management for implementing multi-tasking.

Thread control service provides a multi-tasking feature which switches contexts 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 is the complete contents of the register set. A task control block is provided as a data structure to store the thread context. You may move to another thread from the present thread by storing the current context in a TCB and assigning the contents of another TCB to the register set.

The context at a 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 linked to the task status queue (TCBH).

For registers, please refer to the section on *Register Specification Macros* on page 2-16 or to the *PlayStation™ Hardware Guide*.

Status Immediately After Kernel is Started

When the kernel starts, the TCB array is allocated and the zero element will be opened with OpenTh() and linked in the task status queue as execute TCB. The thread descriptor value of the default thread is as follows:

```
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. An example follows below.

```
unsigned long new_th;
new_th=OpenTh(0x80020000,0x1ffff0,0x00);
ChangeTh(new_th);
```

Operation proceeds as indicated below when the ChangeTh() function is called.

1. Jump to kernel

   A software interrupt is issued from an R3000 system call command. This causes a jump into an internal kernel interrupt dispatch routine. Other interrupts are not allowed at the same time.
2. ChangeTh() context shunt

   Context of the ChangeTh() function being executed is shunted into the previously executed TCB.
3. Link to the task status.

   The specified TCB is linked to the task status queue
4. Starting a new context

   The context read from the executing 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

There are two statuses in a TCB (defined by macros). With the OpenTh() function, status becomes TcbStACTIVE and you may execute the TCB with ChangeTh().

Table 2–16: TCB status

Macro	Status
TcbStUNUSED	Not used
TcbStACTIVE	Execution possible

Data Structure

The data structure of TCBH (Task status queue) and TCB (Task control block) is as follows:

- TCBH structure

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

- TCB structure

```
                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*/
                };
```

Register Specification Macros

The register-specified macros specify which register to shunt to in TCB. The register-specified macros are defined in `r3000.h` and `asm.h`.

Table 2–17: 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
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 Service

This is a service which supports low level access to files and logical devices. All PlayStation file management is performed using this service.

In addition, structures used by the input/output management service and macros are defined by `sys/file.h`.

Devices

The input/output management service supports the following devices.

Table 2–18: IO Device

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

Block Size

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

CD-ROM File System

The CD-ROM file system manages the files on the CD-ROM disk set in the CD-ROM drive, the medium for providing games. In the PlayStation, the file system corresponds to the level 1 format of ISO-9660. File system details are as follows:

Table 2–19: CD-ROM File System (Conforming to ISO 9660 Level 1)

Device name	cdrom
File format	<basename>.<extension name>;<version number> <base name>in 8 letters<extension>up to 3 letters. <base name> and <extension> to be separated by "." (period). <extension> and <version number> to be separated by a ";" (semicolon). Only English capital letters, numbers and "_" (underscore) may be used.
Directory name format	<base name> <base name> in 8 letters. Only English capital letters, numbers and "_" (underscore) may be used. <extension> cannot 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

However, the list of files and directories is only supported as far as it can be stored in one sector (2048 bytes). Accordingly, the benchmarks for PlayStation's inherent limitations are shown below.

Note: The file and directory control data structure in ISO-9660 is variable length. When there is a large number of short names, it is possible to use multiple directories and files from the numerical values below:

Table 2–20: PlayStations Inherent Limitation with Respect to CD-ROM

Total number of entries	Total number of files per directory

45 maximum	30 maximum

Memory Card File System

The memory card file system manages the files on the removable memory card used for saving game data. Carry out the fit and iniitialization test on the BIOS provided by libcard.

Details of the file system are as follows:

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

File Descriptors 0 and 1 are handled by every standard I/O stream.

On the game unit, the standard I/O stream is assigned to a NULL device. In the DTL-H2000 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 Service

This is a basic service that loads and executes the user application of each module.

Execute File Loading, Execution

The execution file should conform to the PS-X EXE format.

This file includes:

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

'Load' refers to the defining of the code and data in the execution file when linking occurs and the reading to the address.

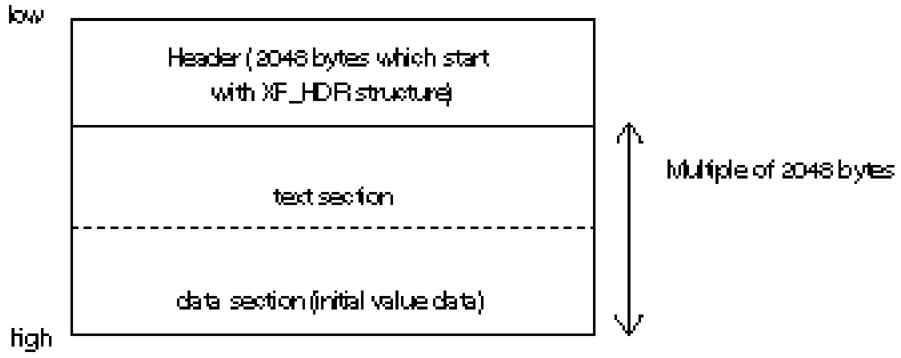
In order to execute the image loaded one of the following items must occur:

a. The stack area must be explicitly defined.
b. The current thread may be used as is.

Execution File Memory Map

The execution file is divided into the following three sections:

Figure 2–1: Execution File Memory Map


low
Header (2048 bytes which start with XF_HDR structure)
text section
data section (initial value data)
high
Multiple of 2048 bytes

Execute File Information Data Structure

The data structure of the execute file is as follows:

```
struct EXEC {                  /*execute file information*/
   unsigned long pc0;          /*execute 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*/
   unsigned long b_size;       /*data section size with no initial value*/
   unsigned long s_addr;       /*stack area lead address (user specified)*/
   unsigned long s_size;       /*stack area size (user specified)
   unsigned long sp, fp, gp, ret, base;  /*register shunt area*/
   };

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

Controller Service

This service controls certain controllers on the PlayStation's main input device. Applications may directly process received data; each type of controller may be identified dynamically.

CAUTION: It is suggested that the libetc function be used only when using standard controllers. This service provides a low-level interface.

Initialization

The normal procedure for initializing the controller is shown below.

When buf0 and buf1 and their byte lengths len0 and len1 are maintained by the buffer, initialization of the controller is made by the following two steps.

InitPAD() registers the receive buffers buf0 and buf1 and their byte lengths len0 and len1 in the kernel.

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

InitPAD() registers the two receive buffers and StartPad() begins automatic access to a controller that triggers a vertical blanking interrupt, and stores received data to buf0 and buf1.

The presence or absence of the device, as well as its state, may be determined by a direct test of the receive buffer's contents.

Buffer Data Format

Data stored in the receive buffer has the following format.

Table 2–21: Received Buffer Data Format

Byte	Contents
0	Receive result 0: success, 0xff: failure
1	Upper 4 bits: terminal type 0x1: mouse 0x2: 16-button analog controller 0x3: gun controller 0x4: 16-button controller 0x5: analog joystick 0x8: multi-tap
	Lower 4 bits : number of received data bytes/2
2,3	<16-button gun controller> Button state 1: released 0: pushed
	<mouse>
3	Button state 1: released 0: pushed Bit 2: right bit 3: left
4	Amount of movement in the X direction -128~127
5	in the Y direction -128~127
6	in the Z direction -128~127
7,-	in the ? direction -128~127
	<16-button analog controller>
2,3	Button state 1: released 0: pushed
4,5,6,7-	Analog channel value

Multi-tap received data structure

Byte	Contents	
0	Received result	0x00: Success 0xff: Failure
1	0x80 Fixed	
2	Connector #1	Received Result 0: Success 0xff:Failure
3	Same as above	(Terminal Classification<<4) / (Received data byte number/2)
4-9	Same as above	Received Data

10	Connector #2	Received Result 0:Success	0xff:Failure
11	Same as above	Received Data	
12-17			
18	Connector #3	Received Result 0:Success	0xff:Failure
19	Same as above	(Terminal Classification<<4)/(Received byte number/2)	
20-25	Same as above	Received Data	
26	Connector #4	Received Result 0:Success	0xff:Failure
27	Same as above	(Terminal Classification<<4)/(Received Data byte number/2)	
28-33	Same as above	Received Data	

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.

Existing Terminal Types

The 16-button terminal type is the PlayStation standard controller.

The mouse produced by our company is a mouse terminal that only supports movement in the X and Y directions. The Namco, Inc. Nejicon is a 16-button analog terminal with 4 channels. The lower 4 bits of the 1st byte of the buffer is 3 ((2 bytes per button + 4 bytes per 4 channels)/2).

Kanji Font Service

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.

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)
Code System	Shift-JIS
Access Method	The starting address in ROM of the font pattern of a specified character 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 2–22: 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(sjis)
unsigned char *sjis;
{
       unsigned short sjiscode;
       sjiscode = *sjis << 8 | *(sjis+1);
       return Krom2RawAdd(sjiscode);  /* get kanji font pattern address */
}
#define COLOR 0x4210
#define BLACK 0x3000
_get_font( s, data )
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;
                     }
       }
```

Other Services

There are other services such as cache, and interrupts that specify and perform processing related to the R3000 .

Chapter 3:
“Standard” C Library

This is a subset of the Standard C library that includes functions like character functions and memory operation functions. libc calls kernel ROM code and libc2 links applications. This chapter is an overview of these libraries.

Run-time Library Overview

Table of Contents

Overview

The C standard libraries (libc/libc2) are a subset of the K & R-based C standard libraries, and includes character functions and memory operation functions and the the like.

libc calls kernel ROM code, so it is a service that gives serious consideration to size.

libc2 links applications, so it is a service that gives serious consideration to speed.

Library File

The memory card library files are `libc.lib` and `libc2.lib`. Every program calling services must link with this library.

Header Files

The following header files must be included to use the C standard library. See a C language reference book for information on each function and header file.

Table 3–1

Contents	File name
library	libc.lib libc2.lib
header	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

The Math library provides a floating point operation package and standard C library math functions.

Table of Contents

Overview

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

Library File

The Math library file is `libmath.lib`. Every program calling services must link with this library.

Header Files

The following header files must be included to use the math library. See a C language reference book for information on each function and header file.

Table 4–1

Contents	File name
library	libmath,lib
header	libmath.h 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 CPU for the PlayStation unit is provided with no floating-point value operation coprocessor so float and double operations are not supported by the hardware. Accordingly if this library is linked and a sub-routine call format floating point arithmetic operations package is included in the application it will be possible to use float type and double type.

Table 4–2: Float Format

Item	Specification
Size	4 bytes
Significant digit count	6 (Decimal number)
Overflow limit value	2.0**128 = 3.4e38
Underflow limit value	0.5**126 = 2.2e-38

Table 4–3: Double Format

Item	Specification
Size	8 bytes
Significant digit count	15 (Decimal number)
Overflow limit value	20**1024 = 1.8e308
Underflow limit value	0.5**1022 = 2.2e-308

Error Processing

An event is used to report an error in a floating-point operation. Error status recording by C standard style external variables is also supported.

Error Types

Math library functions in the library 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

The detection of an error in a function belonging to the mathematical library or a floating-point operation package (for double-precision arithmetic operation) leads to the occurrence of an event stemming from cause descriptor SwMATH. Thus, an overflow and division by zero can be detected and a corresponding error generated.

Error Variable

The external variable math_errno for storing error codes is defined in libmath.lib. Also, in the variable is declared as extern in the header file `libmath.h`. The variable is initialized to zero. When an error arises in the library, however, macro EDOM or ERANGE (defined by sys/errno.h) is stored according to the error contents. math_errno is not automatically reset to zero; it requires the explicit assignment of zero after error processing.

Table 4–4: 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

The Memory Card library enables smooth access to the memory card, and does things like data reading and writing and calling the memory-card BIOS service. This chapter is an overview and explanation of the memory card library.

Table of Contents

Overview

The memory card library provides a service that enables smooth access to the memory card in a realtime environment.

Library File

The memory card library filename is `libcard.lib`. All programs that call services must be linked to this file.

Header Files

The memory card library has no unique header file. The kernel library header file `kernel.h` and the standard header file `sys/file.h` are required.

Table 5–1

Contents	File name
Library	libcard.lib
Header file	kernel.h
Header file	sys/file.h

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:

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
SwCARD	EvSpIOE	Processing complete
	EvSpERROR	Card no good
	EvSpTIMOUT	No card
	EvSpNEW	New card or uninitialized card

NOTES: 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.

libgpu VSync(), etc., functions which wait for a vertical interrupt, 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)and the like are provided by the BIOS.

In order to support concurrent controller reading and the accessing of 2 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:

Start Timing	After a vertical blanking interrupt, controller reading occurs, the card connection is checked 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 occurred, a card whose connection has already been confirmed continues to be connected. Go to (5).

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

   If a TIMOUT event occurred, a card is not connected. Operations other than this are unnecessary. A communication error is possible, so do a retry.
2. Do 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 do a retry.

3. Test logical format using _card_load().

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

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

   In other cases, either the card was removed or a communication error occurred. In these cases, return to (1) and do a retry.
4. Do 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 these cases, return to (1) and do a retry.
5. Do 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() or data reading. 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;

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

_clear_event_x()
{
      TestEvent(ev10);
      TestEvent(ev11);
```

```
        TestEvent(ev12);
        TestEvent(ev13);
}
```

File System

The file system as it relates to the memory card is as follows:

Device Name	buX0 X: 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 in case insufficient capacity or an unformatted card is detected 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 doing 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, query the user about whether it is okay to continue processing.

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 5–3: 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 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 5–4: 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
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.

Table 5–5: 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 */
_set_csum(buf)
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];

/* get data from memory card with checksum test */
get(num,data)
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);

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

/* get data from memory card with checksum test */
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

The following bugs have been confirmed at the present time. Take them into account when coding.

- 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() or write() the file pointer is updated by 128 bytes too few. It has to be corrected using lseek().
- For memory card A (facing left), access to controller A (facing left) fails during a frame in which a timeout event occurred, and all buttons go into release status.

If requesting asynchronous access to card A, the problem does not occur if the controller releases all the buttons, even if this was the user’s (game-player’s) intent, or set it to use card B (facing right).

Example: The Problem Occurs if a Card is not Set in Slot A

```
vertical sync
      ½     _card_info(0x00) etc. requests
      ½     async access
      ↓     pad = PadRead();            /* Normal */
vertical sync
      ½     Start comm to card A
      ↓     pad = PadRead();            /* Normal */
vertical sync
      ½     Card A timeout event occurs
      ½     pad = PadRead();            /* Controller A */
      ↓                                 /* All buttons released */
vertical sync
      ↓     pad = PadRead();            /* Normal */
vertical sync
```

Chapter 6: Data Processing Library

The Data Processing library provides a service for compressing (encoding) and expanding (decoding) image and sound data.

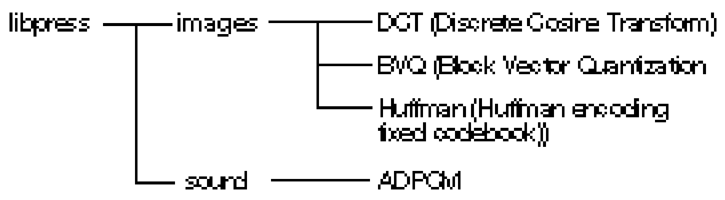
Table of Contents

Overview

The data processing library (libpress) is a low level function library for compressing (encoding) and expanding (decoding) image and sound data.

The library functions can be divided up according to the type of data they handle, that is the image or sound. They can also be divided according to the nature of the processing into Compressor functions (which carry out compression) and Decompressor functions (which carry out decompression).

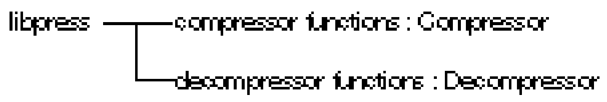
Figure 6–1: Classification by Type of Data Handled


libpress
images
DCT (Discrete Cosine Transform)
BVQ (Block Vector Quantization
Huffman (Huffman encoding fixed codebook))
sound
ADPCM

The image library compresses and decompresses the image data. Three methods of compression are available: one is a method which compresses direct color images using DCT, one is BVQ (Block Vector Quantization) which likewise combines the number of colors in the direct color image together to create 256/16 colors, and finally Huffman Encoding which reversibly compresses 4 bit index colors.

The sound library uses ADPCM to compress a 16bit straight PCM to about 1/4. The compressed sound data can be used as SPU sound source data. Also, each library's functions can be divided into a compressor library for performing compression and a decompressor library for performing decompression.

Figure 6–2: Classification by Nature of Processing


libpress
compressor functions : Compressor
decompressor functions : Decompressor

Library File

The filename of the data processing library is `libpress.lib`. Programs that call individual services must always be linked to this library.

Header File

The data processing library header is libpress.h.

Table 6–1

Content	Filename
Library	libpress.lib
Header	libpress.h

Compressor Function and Decompressor Function

Compressor functions compress image and sound data which exist decompressed in the main memory, and return the results to the 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.

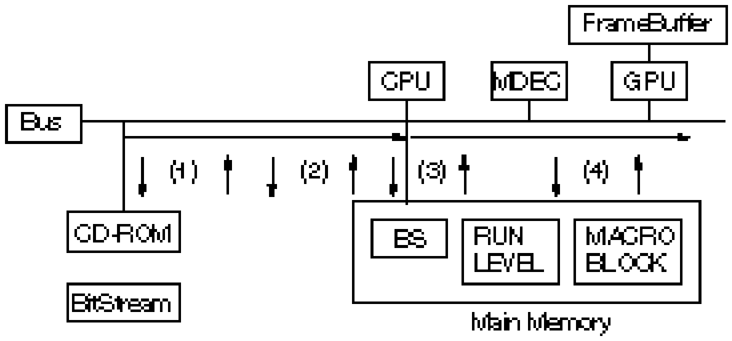
Decompressor functions decompress, in real time, data compressed by the Compressor functions. However, in some cases Compressor functions output data in formats which can be processed without conversion but rather via local environment hardware, like BVQ. Data in these formats cannot be handled by Decompressor functions.

MDEC

The PlayStation provides a specialized data display engine (display) to realize high-speed image data expansion. This is called MDEC (Motion DECoder).

MDEC expands data compressed 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 6–3: 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

Overview

The compression algorithms used in the compression of image data vary according to the type and intended use of the data being handled.

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 10% and 5%.

In DCT, the basic processing unit is a 16x16 24-bit direct-color image. This cell is specifically termed a macroblock.

All the images are first broken down into 16x16 macroblocks before being compressed into what is known as bitstream format. The results of decompression are also output in macroblock units.

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

Figure 6–4: 320x240 Image Breakdown

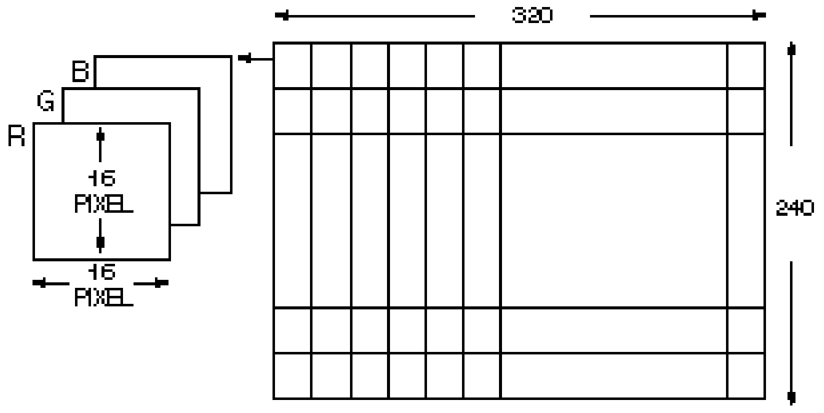
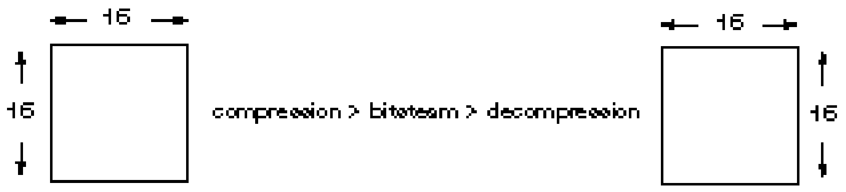


Figure 6–5: DCT Processing



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 have a data format expressed as a two-dimensional array consisting of the table (called a CLUT or 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 not lossy. Therefore, a Huffman encoding function is provided for reversible compression of 4-bit index colors. The Huffman encoding used 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 correspondence 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.

Following is a summary of all the above mentioned compression and decompression algorithms.

Table 6–3: 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.

The decompression of the bitstream therefore consists of two passes:

1. VLC decoding
2. IDCT

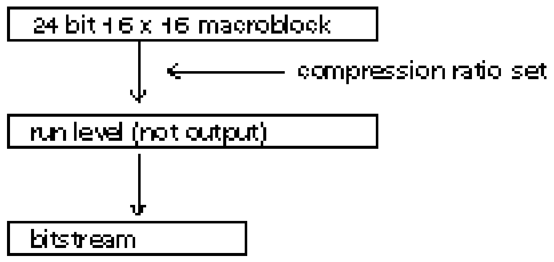
Methods Supported

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.

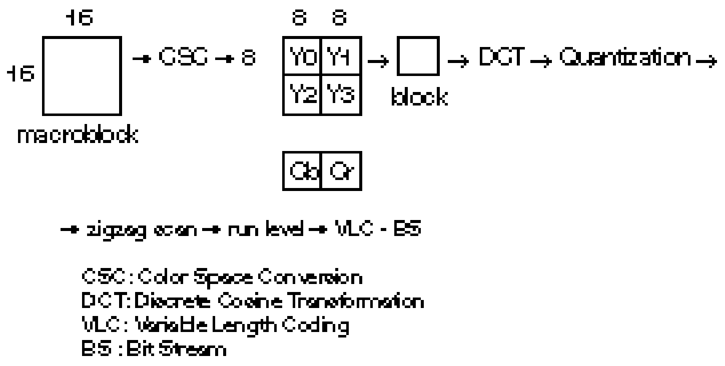
When the actual compression is carried out, the run level (the intermediate data) is not output.

Figure 6–6: DCT Compression



Macroblock encoding is performed in the following fashion.

Figure 6–7: Macroblock Encoding Flow

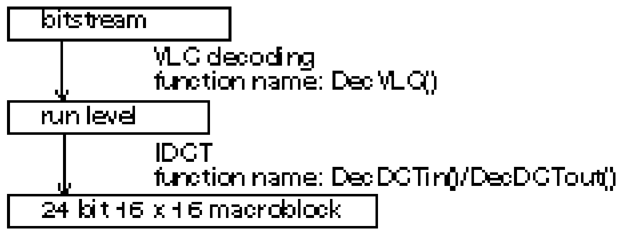


Decompression is carried out by operations which are the reverse of those used in compression.

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.

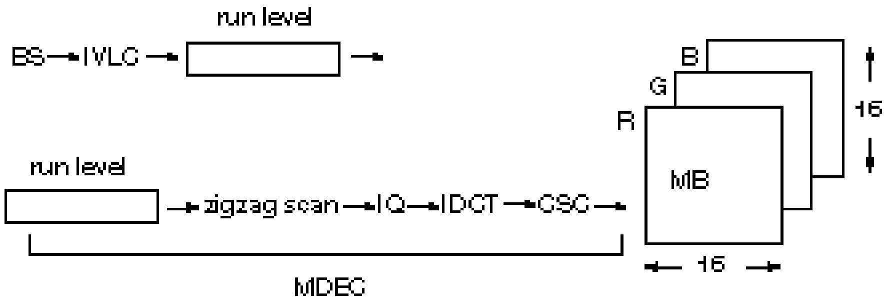
Figure 6–8: DCT Decompression


bitstream
VLC decoding
function name: DecVLC()
run level
IDCT
function name: DecDCTin()/DecDCTout()
24 bit 16 x 16 macroblock

A decompression block diagram is shown below.

MDEC performs decompression from runlevel to macroblock

Figure 6–9: Decompression Block



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() “sews together” 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

The functions DecDCTin() and DecDCTout() are both non-block functions, and return their result without waiting for data transmission/reception to terminate.

Therefore, to detect the termination of the transmission, it is necessary either to carry out polling using the functions DecDCToutSync() and DecDCTinSync(), or to register a callback function and arrange for this function to be called when the transfer terminates.

The registration of the callback function is carried out using the functions DecDCToutCallback() and DecDCTinCallback(). You can arrange for the decompression of images to be carried out asynchronously in the background by designing the function in such a way that the next data transmission/reception is activated within the function.

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

Example

```
/*Main function*/
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*/
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

By reading in and playing bitstreams continuously from the CD-ROM, moving pictures can be played.

The moving picture 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. This corresponds to decompressing 30 320x240 images in one second. 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 continuouly reading data from a CD-ROM is called streaming. The library used for performing streaming (streading library) is supplied separately in a more general purpose format.

Movie playback is realized by placing the bit stream in the containers supplied by the streaming mechanism. Also supplementary information such as movie size, etc., is not included in bit stream. As a result of this, the infomation needed to play a movie is defined separately in the data format (STR format) added to the header.

Table 6–4: 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, however, 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 6–5: 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 sound data used on the PlayStation is data which 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.

On the SPU, a function called looping is provided so that periodic sound data can be recorded using a small number of samples. The sound compression functions also include a function for setting a suitable loop point when the data is compressed.

The PlayStation Bitstream Format

This text discusses the image data format used in MDEC and libpress, as well as how to create data in this format.

Original image data

Original MDEC image data consists of a series of RGB 24-bit (R:G:B=8:8:8-bit) images arranged over time, where each image is made up of 16m x 16n pixels (where m,n are natural numbers). Each of these images is called a frame. Because correlations between frames are not used in MDEC, each frame can be processed independently.

MDEC treats each frame as a collection of small regions of 16x16 pixels. These small regions are called macroblocks. MDEC takes the decoded results and rewrites these to main memory as macroblocks. The reconstruction of these into a single frame is performed by the CPU and the GPU.

For example, data for a 320x240 image is split up into a number of 16x16 macroblocks, and each macroblock is then compressed.

macroblock   playback image

```
                       |‾‾|‾‾|‾‾|‾‾|‾‾|‾‾|‾‾‾‾‾‾‾‾‾‾‾‾|‾‾|
    B ----------  <----|  |  |  |  |  |  |            |  |
  G ---------- |       |--|--|--|--|--|--|------------|--|
  ---------- | |       |  |  |  |  |  |  |            |  |
R| ^       | | |       |--|--|--|--|--|--|------------|--|
 | 16      | | |       |  |  |  |  |  |  |            |  |
 |  pixels | |-|       |  |  |  |  |  |  |            |  |
 | V       |---|       |  |  |  |  |  |  |            |  |
 ----------            |  |  |  |  |  |  |            |  |
   <- 16 ->            |  |  |  |  |  |  |            |  |
    pixels             |--|--|--|--|--|--|------------|--|
                       |  |  |  |  |  |  |            |  |
                       |__|__|__|__|__|__|____________|__|
```

The vertical and horizontal frame sizes should both be multiples of 16. If this is not the case, some additional processing is necessary.

Splitting into macroblocks

The pixels in a frame are generally ordered from top left to bottom right. When encoding, the pixels are re-ordered so that they can be unified in a macroblock.

```
#define WIDTH        320
#define HEIGHT        160

typedef struct {
        u_char r, g, b, pad;
} PIXEL

make_macro_block(frame, macroblock)
PIXEL frame[HEIGHT][WIDTH];
PIXEL macroblck[][16][16];
{
        int ox, oy, x, y, i = 0;

        for (ox = 0; ox < WIDTH; ox += 16) {
            for (oy = 0; oy < HEIGHT; oy += 16, i++)
                for (y = 0; y < 16; y++)
                    for (x = 0; x < 16; x++)
                        macroblock[i][y][x] = frame[oy+y][ox+x];
                }
}
```

In this case, the macroblock is ordered vertically from the top left. This makes it possible to reduce the number of times the frame buffer transfer command (LoadImage) is executed.

If the macroblocks are to be ordered from left to right:

```
for (y = 0; y < HEIGHT; y += 16)
        for (x = 0; y < WIDTH; x += 16) {
                setRECT(&rect, x, y, 16, 16);
                LoadImage(&rect, p);
                p += 16*16;
}
```

If the macroblocks are to be ordered from top to bottom:

```
for (y = 0; y < HEIGHT; y += 16) {
       setRECT(&rect, x, 0, 16, HEIGHT);
       LoadImage(&rect, p);
       p += 16*HEIGHT;
}
```

RGB-YCbCr conversion

MDEC performs its internal processing using the YCbCr color system. Macroblocks in the RGB color system (RGB macroblocks) must be converted to the YCbCr color system. This process is called CSC (Color Space Conversion). The luminance values of the pixels can be expressed as a point in the three-dimensional space formed by the R,G,B components. CSC can be understood as a coordinate transformation for this coordinate system (color space).

In MDEC, the conversion formula below is used to convert the luminance of a pixel to the RGB color system.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.3437 & -0.7143 \\ 1.0 & 1.772 & 0 \end{bmatrix} \times \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

The inverse of this matrix is generally used to convert from the RGB system to the YCbCr system. The movconv encoder uses this matrix.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16871 & -0.33130 & -0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Physically, the Y signal is the luminance signal, and Cb, Cr are the color-difference signals. A black-and-white TV set uses only the Y signal. For a black-and-white screen, the Cb and Cr components are all 0.

The luminance values, recorded in the frame buffers according to the RGB system, are converted in the PlayStation to the YCbCr system and are output from the video terminal (the S terminal). The receiver (TV monitor) receives this video signal and converts it back into RGB. Voltages corresponding to the separate RGB components are sent to the electron beam, which lights up the phosphors arranged on the picture tube. This creates the final image seen by the user.

Because numerous color space conversions are performed, the color shade of the final output can sometimes differ from the expected color. These variations can be due to shifts caused by gamma correction coefficients, or they can be due to shifts caused by the conversion matrices.

Electron beam voltages and luminance values are generally not directly proportional, so some correction is made to luminance ahead of time. This is known as gamma correction. The luminance B of the phosphors on the monitor and the input voltage E have the exponential relationship shown below.

B = pow(aE, gamma);

Thus, the signal source raises the RGB signals to the power of (1/gamma) beforehand. The value of gamma varies slightly from monitor to monitor.

For images composed with computer graphics (CG), gamma correction may be set for a high-definition computer monitor, or the file on a hard disk may have no correction at all. If gamma correction is improper or not present, proper correction must be performed at this encoding stage. If the luminance of the final CG image is darker (or brighter) than expected, improper gamma correction could be the problem.

The characteristics of the phosphors used in a monitor can also vary from monitor to monitor. Characteristics may also vary from country to country based on the users' tastes.

These factors in a video monitor's handling of video signals can be evaluated to some extent by comparing the image from the Playstation's RGB analog output to the image from the video output.

Differences in color shade can be due to the image source, but if this difference is noticeable enough, some sort of counter-correction needs to be applied at the encoding stage. Differences in the displayed color system and the color system used during encoding can result in mishandling of errors (noise) during the encoding. This could be the problem if the image quality changes only for image sources having a specific color shade. A minor discrepancy in the color space used in encoding could come out as a significant color difference on the display.

Creation of macroblocks

Since the luminance signal (Y) generally does not require as high a resolution as the color-difference signals, its data is reduced by 1/4 (1/2 in the x direction and 1/2 in the y direction) relative to the Cb and Cr elements. At the same time, the Y element is split into four 8x8 blocks. Thus, a 16x16 macroblock is split into two color-difference blocks (Cb,Cr) and four luminance blocks (Y0,Y1,Y2,Y3). This collection of small 8x8 units is called a block. A macroblock is arranged in the following order, beginning with a color-difference block.

```
         16                      8   8               8   8
       ------                 |----|----|          |----|----|
      |      | -> CSC ->   8 | Cb | Cr |        8 | Y0 | Y1 |
   16 |      |                |____|____|          |----|----|
      |      |                                   8 | Y2 | Y3 |
      |______|                                     |____|____|
    rgb macroblock      color-difference block   luminance block
```

The following methods are possible ways to reduce color-difference blocks by 1/4:

a) Make the block using only even-numbered x,y points from the original 16x16 blocks.

b) Use the average of four adjacent points from the original block as one point in the block.

In a), elements at or over 1/4 of the sampling frequency (fs) are carried over as noise (aliasing noise).

In b), the processing used to determine the average acts as a sort of two-dimensional lowpass filter (LPF), and so provides better results than a). This method is used in MOVCONV. This method is still unable to completely eliminate elements over 1/4 fs, however, so some aliasing remains. To improve on this a filter having more dimensions and better cut-off properties would be needed, but this would require greater encoding time. Use a filter having a number of dimensions suited to the application.

Block offset

A color space expressed in 8-bit (0-255) RGB values would have the following range in the YCbCr color space:

Y	0 - 255
Cb	-128 - +127
Cr	-128 - +127

In order to unify the ranges for the luminance blocks and the color-difference blocks, -128 is added to all the luminance values in the luminance block. This allows all internal processing to be performed using unsigned chars. During decoding in MDEC, a block is checked to see if it is a luminance block or a color-difference block. If it is a luminance block, a value of +128 is added, which returns the value to its original mode.

DCT

DCT (Discrete Cosine Transformation) is applied to the blocks making up the macroblock. DCT is generally a type of similarity transformation called an orthogonal transformation. Taking an 8x8 matrix X, where the elements are the luminance values in a block, the transformation defined by

Y = P * X * Pi

is called a similarity transformation (P is a matrix having an inverse matrix, and Pi is the inverse of P). When Pt is the transpose of matrix P (Pt(x,y)=P(y,x)), and

Pi = Pt

then this matrix is called an orthogonal matrix, and this transformation is called an orthogonal transformation. Using orthogonal matrix P, the orthogonal transformation can be written as

Y = P * X * Pt

DCT is this orthogonal transformation, where P has the values shown below.

```
         --                                                --
        | 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096,   |
        | 5681, 4816, 3218, 1130,-1130,-3218,-4816,-5681,   |
        | 5532, 2217,-2217,-5352,-5352,-2217, 2217, 5352,   |
        | 4816,-1120,-5681,-3218, 3218, 5681, 1130,-4816,   |
 P =    | 4096,-4096,-4096, 4096, 4096,-4096,-4096, 4096,   | x 1/64
        | 3218,-5681, 1120, 4816,-4816,-1130, 5681,-3218,   |
        | 2217,-5352, 5352,-2217,-2217, 5352,-5352, 2217,   |
        | 1130,-3218, 4816,-5681, 5681,-4816, 3218,-1130,   |
        |__                                               __|
```

In this case, Pt is as follows, so that P * Pt = E (E: unit matrix).

```
         --                                                --
        | 4096, 5681, 5352, 4816, 4096, 3218, 2217, 1130,   |
        | 4096, 4816, 2217,-1130,-4096,-5681,-5352,-3218,   |
        | 4096, 3218,-2217,-5681,-4096, 1130, 5352, 4816,   |
        | 4096, 1130,-5352,-3218, 4096, 4816,-2217,-5681,   |
 Pt =   | 4096,-1130,-5352, 3218, 4096,-4816,-2217, 5681,   | x 1/64
        | 4096,-3218,-2217, 5681,-4096,-1130, 5352,-4816,   |
        | 4096,-4816, 2217, 1130,-4096, 5681,-5352, 3218,   |
        | 4096,-5681, 5352,-4816, 4096,-3218, 2217,-1130,   |
|__                                                              __|
```

Based on P * Pt = E, the inverse transformation of DCT can be expressed as

X = Pt * Y * P

Thus, it can be seen that IDCT is simply DCT with matrix P replaced by the transpose matrix of P.

Physical significance of DCT

Physically, DCT signifies a frequency transformation. The upper left element (element (0,0)) of the 8x8 matrix obtained from a DCT transformation expresses the DC (direct current) element of the original image block X, and is equivalent to the average of all the elements in image block X. The other elements express the AC (alternating current) elements, and the frequency elements increase to the right and down in the matrix.

In natural images, the frequency elements are generally concentrated in the lower regions. Thus, performing a DCT transformation results in smaller values toward the bottom right. Compression using DCT takes advantage of this tendency in the elements.

Quantization

After DCT transformation, each element in a block is quantized according to different resolutions. A quantization table (Q table) is used to indicate the quantization widths (steps) for each element.

MDEC uses the quantization table shown below. The same table is currently used for both the luminance blocks and the color-difference blocks.

Luminance block:

```
        --                       --
       |   8 16 19 22 26 27 29 34  |
       |  16 16 22 24 27 29 34 37  |
       |  19 22 26 27 29 34 34 38  |
       |  22 22 26 27 29 34 37 40  |
Qtab = |  22 26 27 29 32 35 40 48  | x 1/16
       |  26 27 29 32 35 40 48 58  |
       |  26 27 29 34 38 46 56 69  |
       |  27 29 35 38 46 56 69 83  |
       |__                       __|
```

Color-difference block:

```
        --                       --
       |   8 16 19 22 26 27 29 34  |
       |  16 16 22 24 27 29 34 37  |
       | 19 22 26 27 29 34 34 38  |
       | 22 22 26 27 29 34 37 40  |
Qtab = | 22 26 27 29 32 35 40 48  | x 1/16
       | 26 27 29 32 35 40 48 58  |
       | 26 27 29 34 38 46 56 69  |
       | 27 29 35 38 46 56 69 83  |
       |__                      __|
```

The actual quantizing is performed for each element by dividing it by the product of the corresponding Q table value and QUANT, which determines the quantizing step for the entire block. DC elements are not affected by QUANT.

```
y[0] = x[0]*16/(iqtab[0]*8);
for (i = 1; i < 64; i++)
    y[i] = x[i]/(QUANT*Qtable[i]);
```

Q table values increase toward the bottom right of the matrix. This is because the higher frequency elements of the image do not need as much accuracy as the lower frequency elements.

Making the QUANT (the overall quantization step) value large increases the amount of lost data, thus decreasing image quality after decoding. However, since the number of 0 elements in the block is increased, the size of the data used in the run-level transformation is decreased.

Zigzag transformation

The quantized block is numbered one-dimensionally in a type of ordering called zigzag ordering. Quantization and zigzag transformation are performed together in the following manner.

```
static int zscan[] = {
       0 ,1 ,8 ,16,9 ,2 ,3 ,10,
       17,24,32,25,18,11,4 ,5 ,
       12,19,26,33,40,48,41,34,
       27,20,13,6 ,7 ,14,21,28,
       35,42,49,56,57,50,43,36,
       29,22,15,23,30,37,44,51,
       58,59,52,45,38,31,39,46,
       53,60,61,54,47,55,62,63,
};
static block_t iqtab[] = {
       2,16,19,22,26,27,29,34,
       16,16,22,24,27,29,34,37,
       19,22,26,27,29,34,34,38,
       22,22,26,27,29,34,37,40,
22,26,27,29,32,35,40,48,

       26,27,29,32,35,40,48,58,

       26,27,29,34,38,46,56,69,

       27,29,35,38,46,56,69,83,

}
```

```
blk_zig[0] = blk_dct[0]/iqtab[0];

for (i = 1; i < 64; i++) {

       j = zscan[i];

       blk_zig[i] = blk_dct[j]*16/(iqtab[j]*q_scale);

}
```

By arranging the blocks in zigzag order, the elements (coefficients) in the block are arranged starting from the elements corresponding to lower frequency elements. There are more 0 elements further back in the series to make run-level compression easier.

The example below shows a macroblock going from DCT transformation to zigzag transformation. These transformations are for a 16x16 black-and-white block. Because the data is black and white, the elements in the color-difference block are all zero.

```
(QUANT = 8)
 Cb ----------------------------------------------------------------

 src:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0

 dct:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0

 zig:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0

 Cr ----------------------------------------------------------------

 src:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0

 dct:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0

 zig:
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
 Y0 ----------------------------------------------------------------

 src:
    -128    -128    -128    -128    -128    -128    -128    -128
    -128    -128    -128    -128    -128     -68     -51     -39
    -128    -128    -128     -83     -59     -42     -28     -17
    -128    -128     -83     -56     -38     -23     -11       0
    -128    -128     -59     -38     -22      -8       5      14
    -128     -68     -42     -23      -8       6      18      28
    -128     -51     -28     -11       5      18      30      40
    -128     -39     -17       0      14      28      40      51

dct:
    -229    -155     -23     -21     -10      -4      -1      -4
    -155      42      34      23      12      12       7       0
     -23      34       7      -6     -13     -10      -7      -5
     -21      23      -6      -5      -2       1       3       3
     -10      12     -13      -2       5       4       5       6
      -4      12     -10       1       4      -4      -4       1
      -1       7      -7       3       5      -4      -6      -1
      -4       0      -5       3       6       1      -1       2

zig:
    -229     -19     -19      -2       5      -2      -2       3
       3      -2      -1       2       1       2      -1       0
       1       0       0       1       0       0       1      -1
       0      -1       1       0       0       0      -1       0
       0      -1       1       0       0      -1       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0

Y1 --------------------------------------------------------------
src:
    -128    -128    -128    -128    -128    -128    -128    -128
     -29     -21     -16     -15    -128    -128    -128    -128
      -7       1       7      11      12       5    -128    -128
      10      18      25      30      33      32      23    -128
      25      33      40      45      49      50      46    -128
      38      46      53      59      63      65      64      56
      50      58      65      71      74      78      78      73
      60      69      75      82      86      89      89      85

dct:
      -1      55     -57      20     -23      16      -5      -4
    -263      57      -8      -7       2       2       6      -9
     -57     -34      48     -40      26     -14      10      -8
     -38     -35      22      -5      -2       7      -9       8
     -22     -18     -10      27     -20      14     -16      14
      -3     -30      -2      14       2     -11       3       5
       1     -19      -6       9      12     -23      13      -1
     -12       8     -19      17      -1      -9       6       0

zig:
      -1       7     -33      -6       7      -6       2      -1
      -3      -3      -2      -3       4      -1      -2       1
       0      -3       2      -1       0       0      -2      -1
       0       2       0       0       0       0      -1       0
       2       0      -1      -1       1       0       1      -1
       0       1       0       0       0       1       0       1
      -1       1       1      -1      -1       0       1       0
      -1       0       0       0       0       0       0       0

Y2 --------------------------------------------------------------
src:
    -128     -29      -7      10      25      38      50      60
    -128     -21       1      18      33      46      58      69
    -128     -16       7      25      40      53      65      75
    -128     -15      11      30      45      59      71      82
    -128    -128      12      33      49      63      74      86
    -128    -128       5      32      50      65      78      89
    -128    -128    -128      23      46      64      78      89
    -128    -128    -128    -128    -128      56      73      85

dct:
      -1    -263     -57     -38     -22      -3       1     -12
      55      57     -34     -35     -18     -30     -19       8
     -57      -8      48      22     -10      -2      -6     -19
      20      -7     -40      -5      27      14       9      17
     -23       2      26      -2     -20       2      12      -1
      16       2     -14       7      14     -11     -23      -9
      -5       6      10      -9     -16       3      13       6
      -4      -9      -8       8      14       5      -1       0

zig:
      -1     -33       7      -6       7      -6      -3      -3
      -1       2      -2      -1       4      -3      -2       0
      -1       2      -3       0       1       0       0       2
       0      -1      -2       0      -1      -1       0       2
       0      -1       0       0      -1       1       0      -1
       1       0       0      -1       0       0       1      -1
       0       0      -1      -1       1       1       0      -1
       0       1       0       0       0       0       0       0

Y3 ---------------------------------------------------------------
src:
      70      78      85      91      96      99      99      94
      78      86      94      99     104     106     106     103
      85      94     101     106     110     112     112     107
      91      99     106     112     115     118     117     109
      96     104     110     115     120     120     118    -128
      99     106     112     118     120     120     112    -128
      99     106     112     117     118     112    -128    -128
      94     103     107     109    -128    -128    -128    -128

dct:
     300     109     -84      28     -33      26      -6     -10
     109    -150      53      -7      14     -18      -5      20
     -84      53      23     -52      28     -11      21     -26
      28      -7     -52      65     -32       8     -14      19
     -33      14      28     -32      -1      22     -12      -3
      26     -18     -11       8      22     -39      27      -8
      -6      -5      21     -14     -12      27     -21       7
     -10      20     -26      19      -3      -8       7      -1

zig:
     300      14      14      -9     -19      -9       3       5
       5       3      -3      -1       2      -1      -3       2
       1      -4      -4       1       2       0      -1       2
       5       2      -1       0      -1       0      -1      -2
      -2      -1       0      -1       1       1       0       0
       0       1       1      -1      -1       1       1      -1
      -1       1      -1      -2      -1       1       0       1
       1       0       0      -1       0       0       0       0
```

Runlevel conversion

Because there tend to be sequences of zeros in zigzag ordered blocks, data is compressed by combining two or more continuous zeros. This is called runlevel transformation. Runlevel consists of the following two-dimensional data:

(run,level) = (the number of zeros preceding level, value of the element)

An actual runlevel has 16 bits and is in the following format:

```
 15¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡   10  9¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡ 0¡¡
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|¡¡¡¡¡¡¡¡RUN¡¡¡¡¡¡¡¡    |¡¡¡¡¡¡¡¡¡¡¡¡¡¡LEVEL¡¡¡¡¡¡¡¡¡¡¡¡¡       |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

RUN The number of zeros preceding a non-zero coefficient (unsigned, 6 bits)

LEVEL The non-zero coefficient (signed, 10 bits)

A runlevel at the beginning of a block always has a run field of 0. (The runlevel is (0,0) even if the DC element is 0.) Therefore, the quantization step (QUANT) is always placed in the run field of the first runlevel in a block.

```
 15¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡   10  9¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡ 0¡¡
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|¡¡¡¡¡¡¡¡QUANT¡¡¡¡¡¡¡¡  |¡¡¡¡¡¡¡¡¡¡¡¡¡¡LEVEL¡¡¡¡¡¡¡¡¡¡¡¡¡       |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

QUANT quantization step (unsigned, 6 bits)

LEVEL DC coefficient (signed, 10 bits)

A runlevel pair RL is transferred as a two-set 32-bit pack.

```
 31¡¡¡¡¡¡¡¡    23¡¡¡¡¡¡          15           7               0

|--------------------------------|--------------------------------|
|¡¡¡¡¡¡¡¡RL(1)¡¡¡¡¡¡¡¡           |¡¡¡¡¡¡¡¡¡¡¡¡¡¡QL                |
|--------------------------------|--------------------------------|
|¡¡¡¡¡¡¡¡RL(3)¡¡¡¡¡¡¡¡           |¡¡¡¡¡¡¡¡¡¡¡¡¡¡RL(2)             |
|--------------------------------|--------------------------------|
|¡¡¡¡¡¡¡¡RL(5)¡¡¡¡¡¡¡¡           |¡¡¡¡¡¡¡¡¡¡¡¡¡¡RL(4)             |
|--------------------------------|--------------------------------|
                                 :
                                 :
|--------------------------------|--------------------------------|
|¡¡¡¡¡¡¡¡RL(N)¡¡¡¡¡¡¡¡           |¡¡¡¡¡¡¡¡¡¡¡¡¡¡RL(N-1)           |
|--------------------------------|--------------------------------|
```

For example,

```
-229     -19     -19      -2       5      -2      -2       3
   3      -2      -1       2       1       2      -1       0
   1       0       0       1       0       0       1      -1
   0      -1       1       0       0       0      -1       0
   0      -1       1       0       0      -1       0       0
   0       0       0       0       0       0       0       0
   0       0       0       0       0       0       0       0
   0       0       0       0       0       0       0       0
```

A matrix having this data would be organized as follows:

(0,-229)(0,-19)(0,-19)(0,-2)(0,5)(0,-2)(0,-2)(0,3)(0,3)(0,-2)(0,-1)

(0,2)(0,1)(0,2)(0,-1)(1,1)(2,1)(2,1)(0,-1)(1,-1)(0,1)(3,-1)(2,-1)

(0,1)(2,-1)(END)

A (run,level) = (63,128) = 0xfe00 is inserted at the end of a block.

The previous block would be converted as follows (QUANT=8):

```
0x03ED231B,
0x03FE03ED,
0x03FE0005,
0x000303FE,
0x03FE0003,
0x000203FF,
0x00020001,
0x040103FF,
0x08010801,
0x07FF03FF,
0x0FFF0001,
0x00010BFF,
0xFE000000,
```

The RL data format

RL data combines all the runlevel transformed block data required for decoding. RL data includes a one-word (32-bit) header and a footer for maintaining 32-word boundaries.

Each runlevel making up a block is delimited with a NOP (0xfe00). NOPs indicate the following:

When a NOP appears at a position other than between blocks:

That block is ended and the subsequent block elements are filled with 0.

When a NOP appears between blocks:

The NOP is skipped.

When two NOPs appear in the middle of a block, the first NOP functions as a delimiter and the second NOP is simply skipped.

The blocks are assumed to be in the following order: Cb, Cr, Y0, Y1, Y2, Y3. Six blocks make a single macroblock. The number of blocks in an RL must be a multiple of a macroblock unit (6).

A header word has the following format:

```
 31¡¡¡¡¡¡¡¡      ¡¡¡¡¡¡          15                            0
|------------------------------|------------------------------|
|¡¡¡¡¡¡¡¡MAGIC¡¡¡¡¡¡¡¡          |¡¡¡¡¡¡¡¡¡¡¡¡¡¡SIZE            |
|------------------------------|------------------------------|
```

MAGIC	magic number (constant 0x3800)
SIZE	data size (word), not including the header

The data size must be a multiple of 32. Consequently, a footer containing an appropriate number of NOPs is added as padding to make the data a fixed length.

The results of a runlevel transformation of the previous macroblock is shown below. Because all the elements in the Cb and Cr blocks are 0, it can be seen that both blocks are contained in one word that includes the delimiter.

```
0x38000060, 0xFE002000, 0xFE002000, 0x03ED231B,
0x03FE03ED, 0x03FE0005, 0x000303FE, 0x03FE0003,
0x000203FF, 0x00020001, 0x040103FF, 0x08010801,
0x07FF03FF, 0x0FFF0001, 0x00010BFF, 0xFE000BFF,
0x000723FF, 0x03FA03DF, 0x03FA0007, 0x03FF0002,
0x03FD03FD, 0x03FD03FE, 0x03FF0004, 0x000103FE,
0x000207FD, 0x0BFE03FF, 0x040203FF, 0x040213FF,
0x03FF07FF, 0x04010001, 0x040103FF, 0x04010C01,
0x000103FF, 0x03FF0001, 0x040103FF, 0xFE0007FF,
0x03DF23FF, 0x03FA0007, 0x03FA0007, 0x03FD03FD,
0x000203FF, 0x03FF03FE, 0x03FD0004, 0x07FF03FE,
0x03FD0002, 0x08020401, 0x03FE07FF, 0x03FF07FF,
0x07FF0402, 0x00010BFF, 0x000107FF, 0x08010BFF,
0x0BFF03FF, 0x000103FF, 0x07FF0001, 0xFE000401,
0x000E212C, 0x03F7000E, 0x03F703ED, 0x00050003,
0x00030005, 0x03FF03FD, 0x03FF0002, 0x000203FD,
0x03FC0001, 0x000103FC, 0x07FF0002, 0x00050002,
0x03FF0002, 0x07FF07FF, 0x03FE03FE, 0x07FF03FF,
0x00010001, 0x00010C01, 0x03FF03FF, 0x00010001,
0x03FF03FF, 0x03FF0001, 0x03FF03FE, 0x04010001,
0x0BFF0001, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00,
```

The syntax for RL data is as follows:

```
RL data ---|---------- header
           |--------   macroblock
           |--------   ..
           |--------   macroblock
           |--------   footer

header    ---|---- 16bit: MAGIC (0x3800)
             |---- 16bit: SIZE

macroblock --|------   Cb block
             |------   Cr block
             |------   Y0 block
             |------   Y1 block
             |------   Y2 block
             |------   Y3 block

Block      --|------   5-bit: QUANT, 10-bit: DC
             |------   5-bit: RUN,   10-bit: LEVEL
             |------   ...
             |------   5-bit: RUN,   10-bit: LEVEL
             |------   NOP

header    ---|---- NOP
             |---- ...
```

VLC

VLC (Variable Length Coding) is based on Huffman coding theory with reversible data compression using a fixed dictionary (code book). Huffman coding takes advantage of the probabilities of occurrence of data (word: in this case, a runlevel). Data is compressed by assigning the shorter codes to words having higher probabilities of occurrence.

For example, the word (run,level) = (0, 1) has a high frequency of occurrence, and so is assigned the code 01 (2 bits). On the other hand, (run,level) = (23,1) has a low frequency of occurrence and so it is assigned a code of 00000000111101 (14 bits).

The look-up table for words and codes is called a dictionary (code book). Huffman coding assumes that the probabilities of occurrence of data is known ahead of time, and that these probabilities do not change over time (non-memory data source). In this case, the code book is established in the beginning, and this code book is called a static dictionary.

Code books vary according to the characteristics of the image source. For this reason, code books should actually be prepared for each image according to the specific characteristics of that image. To simplify things, libpress prepares a single standard code book that is applied to standard images. This code book is based on a runlevel distribution in a typical natural image. Different code books should be used for applications such as animation, where the probability distribution in the runlevels is different from that of natural images.

The contents of the standard code book are appended at the end of this text. The bit sequences in the left-hand column correspond to the runlevels in the right-hand column. EOB is an abbreviation of EndOfBlock and represents an NOP (0xfe00).

Word (32-bit) boundaries are not significant in the coded data. Because byte ordering in the PlayStation is little-endian, coded bit sequences are ordered starting with low-order bits. Code sequences packed in this way are called bitstreams.

If a runlevel that is not in the code book appears, an escape code 000001 followed by an FLC (fixed-length code) is used. The (run, level) of an FLC are described at the end of this section.

VLC transformation during decoding is performed with software. Decoding operations for runlevel and after are performed with hardware by MDEC. Thus, VLC decoding and runlevel decoding are generally performed with pipelines in order to decrease processing time. In the library, the DecDCTvlc() function performs VLC decoding, and the DecDCTin()/DecDCTout() functions perform DCT processing for runlevel and after.

BS format

RL format data on which VLC transformation has been performed is called BS data. The BS format is made up of bitstreams in which headers and blocks are coded.

The header is made up of two words and has the format shown below. In the current BS format, it is assumed that all the blocks contained have identical quantization steps
(QUANT). Consequently, QUANT is stored only once in the header.

```
 31¡¡¡¡¡¡¡¡      ¡¡¡¡¡¡          15                             0
|-------------------------------|-------------------------------|
|¡¡¡¡¡¡¡¡MAGIC¡¡¡¡¡¡¡¡          |¡¡¡¡¡¡¡¡¡¡¡¡¡¡RLSIZE           |
|-------------------------------|-------------------------------|
|¡¡¡¡¡¡¡¡VER¡¡¡¡¡¡  ¡¡          |¡¡¡¡¡¡¡¡¡¡¡¡¡¡QUANT            |
|-------------------------------|-------------------------------|
```

MAGIC	magic number (constant 0x3800)
RLSIZE	data size (word) of runlevel, not including the header
VER	version number (constant 2)
QUANT	quantization step

A block contains a 10-bit DC element followed by a bitstream converted from a runlevel representing the AC element. The block boundaries are not necessarily at the word boundaries.

```
     <--------------------------- 32 bits -------------------------->
     31¡¡¡¡¡¡¡¡      ¡¡¡¡¡¡          15                             0
    |--------------------------------|--------------------------------|
  0 |¡¡¡¡¡¡¡¡MAGIC¡¡¡¡¡¡¡¡           |¡¡¡¡¡¡¡¡¡¡¡¡¡¡RLSIZE          |
    |--------------------------------|--------------------------------|
  1 |¡¡¡¡¡¡¡¡VER¡¡¡¡¡¡   ¡¡          |¡¡¡¡¡¡¡¡¡¡¡¡¡¡QUANT           |
    |--------------------------------|--------------------------------|
  2 |¡¡¡¡¡¡¡¡                    Block0(Cb)                         |
    |-----------------------------------------|                       |
    |                                         |-----------------------|
    |¡¡¡¡¡¡¡¡                    Block1(Cr)                         |
    |--------|                                                        |
    |        |--------------------------------------------------------|
    |¡¡¡¡¡¡¡¡                    Block3(Y0)                         |
    |¡¡¡¡¡¡¡¡                                                        |
    |¡¡¡¡¡¡¡¡                                                        |
    |¡¡¡¡¡¡¡¡                                                        |
```

The RL data footer padding (0xfe00) is not included in BS data. So when BS decoding is performed, DecDCTvlc() automatically adds footers to maintain 32-word boundaries.

The BS format does not contain data for the frame size or frame rate to be used in the final playback. These settings need to be indicated to the program in other ways.

The VLC transformation of the runlevel from the previous example is shown below.

```
0x38000060, 0x00020008, 0x02C60020, 0x8039C01C,
0x29284931, 0x6476A4F4, 0xE3D752BB, 0xE050977F,
0x860500BC, 0xE5960868, 0x3A7192C3, 0x278C2D1C,
0xFCDD3463, 0xB7EC8E6F, 0x005EF7FE, 0x60500508,
0x7D098659, 0xA5D0E185, 0x3A5F5B04, 0xE7CB8C75,
0xD9DA575F, 0x30004B00, 0x200EC006, 0x51304062,
0x7D1C9851, 0xD0DCB460, 0x311D8741, 0xEFB1DD29,
0xBFDDDBFD, 0x5CFF3F36, 0x00008000,
```

BS format version

The version numbers that are currently valid for BS data are 2 and 3. In BS version 3, Huffman coding is performed on the DC element (10 bits) at the start of the block. What is coded in this case is the difference from the DC element in the adjacent (previous) block. This makes it possible to decrease the bit-width of the DC element.

In version 3 encoding, the Huffman coding of the DC element improves the compression ratio, but this also increases the processing time. Version 2 should be used in applications where CPU processing time can be a bottleneck.

Decoding speed

The maximum MDEC decompression speed is 9000 macroblocks/second. This is equivalent to a rate of 30 frames of 320x240 images decompressed in a second. This decompression speed has no relation to the compression ratio. Of course, the resolution of the images is in inverse proportion to the playback frame rate. In other words, 320x240 images can be played back at 30 frames/second, and 640x240 images can be played back at 15 frames/second.

Decompression speed

Resolution	320x240	640x240	640x480....
Frames/second	30	15	7.5 ....

In practice, the CPU performs VLC decoding through DCT decompression in the background. If the CPU is performing other compute-intensive processing, VLC decoding can become a bottleneck.

The transfer rate for the CD-ROM can be selected as either 150KB/sec (standard speed) or 300KB/sec (double speed). With double-speed playback, data can be read from the CD-ROM at a rate of 30 frames/second if the bitstream for one frame is compressed to 10KB (=300KB/30) or less.

Transfer speed

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

The playback rate for animation is determined by these two factors. For example, double-speed playback can be performed if the bitstream for one frame (320x240) recorded on a CD-ROM is compressed to 10KB or less. As long as this condition is met, the frame rate, the image resolution and the compression ratio can be set as desired.

Improving image quality

The following methods are possible ways to improve the image quality in MDEC animations.

Preprocessing

Image noise at the initial stage, i.e. the image input stage, is the biggest factor in image quality during playback. For this reason, very high quality images should be used. Image sources need to be at least 24-bit images.

The S/N ratio for image sources can be decreased by horizontal filtering as well as filtering over time.

If a high-resolution image capturing system can be used, sampling should be done at a high resolution (high sampling rate), and then a lowpass filter should be applied for down-conversion. If 2x oversampling can be performed, then random noise (white noise) can be reduced by up to 1/2.

If possible, a 60 frames/second image source should be used in creating images for 15 frames/second playback, and one frame should be created from four sequential frames. Simply creating new frames from the average of four frames can reduce noise by up to 1/4.

When macroblocks are to be created by compressing 16x16 Cb, Cr blocks to 8x8, raising the filtering coefficient to a higher dimension can eliminate aliasing. Block noise (noise that appears in a 16x16 unit) during decoding is due to the Cb, Cr blocks. Block noise in an image source can be reduced by taking advantage of the correlation between adjacent blocks during Cb, Cr block creation.

Block noise is visually conspicuous when the encoded noise takes on different values between frames. This can occur when there are large changes in the quantization steps between adjacent frames. When BS data is recorded by interleaving with audio sectors, the audio sectors and the movie sectors are asynchronous, and the data sizes assigned to each frame vary. This leads to variations in the quantization steps and appears as noise. Noise caused by variation in quantization can often be seen in cases where the image source has a lot of static areas (animation backgrounds). In these cases, encoding must always be performed with fixed sectors.

Evaluation during encoding

The display of some home monitors gives more emphasis than necessary to the color-difference signal. In addition, there are cases where gamma correction is not performed correctly. This means that the error in the color-difference component during calculations is evaluated as being lower than it actually is, depending on the situation. There are also cases where the color-difference signal does not have a resolution of 1/4 to start with, and this can cause the playback image to appear deteriorated on specific TV sets. In these cases, the evaluation functions used in encoding needs to be modified.

Making changes in the code book

The current code book is based on natural images. The compression ratio is lower for images such as animation which have a probability distribution different from natural images. In these cases, a dedicated code book should be prepared. The current libpress does not allow switching of code books, but this feature is planned for future versions. Also, since VLC encoding is performed with software, it is possible for an application to perform its own coding.

Correlation between frames

In encoding that takes advantage of frame correlations, the benefits are outweighed by the penalties, such as the need for processing dropped frames and the CPU load during decoding. However, for image sources that obviously have a lot of fixed scenes, it would be possible to take solely 0-degree correlations. In such cases, it would be necessary to have data indicating correlation in addition to the BS format data.

Processing during playback

In general, better results are obtained by playing back in 24-bit mode compared to playing back in 16-bit mode. This is because a standard encoder truncates the low-order bits when going from 24 bits to 16 bits, and quantization errors are not taken into consideration in the evaluation functions. When playing back at 16 bits, the input image source should be rounded off to 16 bits in the manner shown below, rather than simply truncating to 16 bits.

```
pix16 = (pix24+0x10)&0xfe
```

Appendix

The VLC code book

The VLC code book is shown below. This code book is compatible with the one used in MPEG microlayers.

I.

bit pattern	(run,level)
10	(EOB)
110	0 +1
111	0 -1
0110	1 +1
0111	1 -1
01000	0 +2
01001	0 -2
01010	2 +1
01011	2 -1
001010	0 +3
001011	0 -3
001110	3 +1
001111	3 -1
001100	4 +1
001101	4 -1
0001100	1 +2
0001101	1 -2
0001110	5 +1
0001111	5 -1
0001010	6 +1
0001011	6 -1
0001000	7 +1
0001001	7 -1
000001	(ESCAPE)
00001100	0 +4
00001101	0 -4
00001000	2 +2
00001001	2 -2
00001110	8 +1
00001111	8 -1
00001010	9 +1
00001011	9 -1
001001100	0 +5
001001101	0 -5
001000010	0 +6
001000011	0 -6
001001010	1 +3
001001011	1 -3
001001000	3 +2
001001001	3 -2
001001110	10 +1
001001111	10 -1
001000110	11 +1
001000111	11 -1
001000100	12 +1
001000101	12 -1
001000000	13 +1
001000001	13 -1
00000010100	0 +7
00000010101	0 -7
00000011000	1 +4
00000011001	1 -4
00000010110	2 +3
00000010111	2 -3
00000011110	4 +2
00000011111	4 -2
00000010010	5 +2
00000010011	5 -2
00000011100	14 +1
00000011101	14 -1
00000011010	15 +1

II.

```
==========================
bit pattern       (run,level)
---------------------------
00000011011       15 -1
00000010000       16 +1
00000010001       16 -1
0000000111010     0 +8
0000000111011     0 -8
0000000110000     0 +9
0000000110001     0 -9
0000000100110     0 +10
0000000100111     0 -10
0000000100000     0 +11
0000000100001     0 -11
0000000110110     1 +5
0000000110111     1 -5
0000000101000     2 +4
0000000101001     2 -4
0000000111000     3 +3
0000000111001     3 -3
0000000100100     4 +3
0000000100101     4 -3
0000000111100     6 +2
0000000111101     6 -2
0000000101010     7 +2
0000000101011     7 -2
0000000100010     8 +2
0000000100011     8 -2
0000000111110     17 +1
0000000111111     17 -1
0000000110100     18 +1
0000000110101     18 -1
0000000110010     19 +1
0000000110011     19 -1
0000000101110     20 +1
0000000101111     20 -1
0000000101100     21 +1
0000000101101     21 -1
00000000110100    0 +12
00000000110101    0 -12
00000000110010    0 +13
00000000110011    0 -13
00000000110000    0 +14
00000000110001    0 -14
00000000101110    0 +15
00000000101111    0 -15
00000000101100    1 +6
00000000101101    1 -6
00000000101010    1 +7
00000000101011    1 -7
00000000101000    2 +5
00000000101001    2 -5
00000000100110    3 +4
00000000100111    3 -4
00000000100100    5 +3
00000000100101    5 -3
00000000100010    9 +2
00000000100011    9 -2
00000000100000    10 +2
00000000100001    10 -2
00000000111110    22 +1
00000000111111    22 -1
00000000111100    23 +1
00000000111101    23 -1

---------------------------
```

III.

```
===========================
bit pattern       (run,level)
---------------------------
00000000111010    24 +1
00000000111011    24 -1
00000000111000    25 +1
00000000111001    25 -1
00000000110110    26 +1
00000000110111    26 -1
000000000111110   0 +16
000000000111111   0 -16
000000000111100   0 +17
000000000111101   0 -17
000000000111010   0 +18
000000000111011   0 -18
000000000111000   0 +19
000000000111001   0 -19
000000000110110   0 +20
000000000110111   0 -20
000000000110100   0 +21
000000000110101   0 -21
000000000110010   0 +22
000000000110011   0 -22
000000000110000   0 +23
000000000110001   0 -23
000000000101110   0 +24
000000000101111   0 -24
000000000101100   0 +25
000000000101101   0 -25
000000000101010   0 +26
000000000101011   0 -26
000000000101000   0 +27
000000000101001   0 -27
000000000100110   0 +28
000000000100111   0 -28
000000000100100   0 +29
000000000100101   0 -29
000000000100010   0 +30
000000000100011   0 -30
000000000100000   0 +31
000000000100001   0 -31
0000000000110000  0 +32
0000000000110001  0 -32
0000000000101110  0 +33
0000000000101111  0 -33
0000000000101100  0 +34
0000000000101101  0 -34
0000000000101010  0 +35
0000000000101011  0 -35
0000000000101000  0 +36
0000000000101001  0 -36
0000000000100110  0 +37
0000000000100111  0 -37
0000000000100100  0 +38
0000000000100101  0 -38
0000000000100010  0 +39
0000000000100011  0 -39
0000000000100000  0 +40
0000000000100001  0 -40
0000000000111110  1 +8
0000000000111111  1 -8
0000000000111100  1 +9
0000000000111101  1 -9
0000000000111010  1 +10
---------------------------
```

```
IV.

==========================
bit pattern      (run,level)
---------------------------
0000000000111011  1 -10
0000000000111000  1 +11
0000000000111001  1 -11
0000000000110110  1 +12
0000000000110111  1 -12
0000000000110100  1 +13
0000000000110101  1 -13
0000000000110010  1 +14
0000000000110011  1 -14
00000000000100110 1 +15
00000000000100111 1 -15
00000000000100100 1 +16
00000000000100101 1 -16
00000000000100010 1 +17
00000000000100011 1 -17
00000000000100000 1 +18
00000000000100001 1 -18
00000000000101000 6 +3
00000000000101001 6 -3
00000000000110100 11 +2
00000000000110101 11 -2
00000000000110010 12 +2
00000000000110011 12 -2
00000000000110000 13 +2
00000000000110001 13 -2
00000000000101110 14 +2
00000000000101111 14 -2
00000000000101100 15 +2
00000000000101101 15 -2
00000000000101010 16 +2
00000000000101011 16 -2
00000000000111110 27 +1
00000000000111111 27 -1
00000000000111100 28 +1
00000000000111101 28 -1
00000000000111010 29 +1
00000000000111011 29 -1
00000000000111000 30 +1
00000000000111001 30 -1
00000000000110110 31 +1
00000000000110111 31 -1
---------------------------
```

EOB: End Of Block

ESC: Escape

FLC Code

ESC is followed by a FLC(fixed-length code). The (run, level) of an FLC is defined as follows:

```
============================
Fixed-length Code    run
----------------------------
```

000000	0
000001	1
000010	2
....	
111111	63

Fixed-length Code	level
1000 0000 0000 0001	-256
1000 0000 0000 0010	-255
1000 0000 0000 0011	-254
....	
1000 0000 0111 1111	-129
1000 0000 1000 0000	-128
1000 0001	-127
1000 0010	-126
....	
1111 1110	-2
1111 1111	-1
0000 0001	1
0000 0010	2
....	
0111 1111	127
0000 0000 1000 0000	128
0000 0000 1000 0001	129
....	
0000 0000 1111 1111	255

IDCT coefficients

The IDCT coefficients (the coefficients for matrix U) used in the decoder are shown below.

	0	1	2	3	4	5	6	7
0	4096	5681	5352	4816	4096	3218	2217	1130
1	4096	4816	2217	-1130	-4096	-5681	-5352	-3218
2	4096	3218	-2217	-5681	-4096	1130	5352	4816
3	4096	1130	-5352	-3218	4096	4816	-2217	-5681
4	4096	-1130	-5352	3218	4096	-4816	-2217	5681
5	4096	-3218	-2217	5681	-4096	-1130	5352	-4816
6	4096	-4816	2217	1130	-4096	5681	-5352	3218

```
7 |4096 -5681  5352 -4816  4096 -3218  2217 -1130

---------------------------------------------------
```

IQ coefficient

The IQ coefficients used by the decoder are shown below.

```
===========================

  | 0  1  2  3  4  5  6  7

--+------------------------

0 | 2 16 19 22 26 27 29 34

1 |16 16 22 24 27 29 34 37

2 |19 22 26 27 29 34 34 38

3 |22 22 26 27 29 34 37 40

4 |22 26 27 29 32 35 40 48

5 |26 27 29 32 35 40 48 58

6 |26 27 29 34 38 46 56 69

7 |27 29 35 38 46 56 69 83

---------------------------
```

CSC coefficients

The CSC (Color Space Conversion) coefficients are shown below.

|Red | |1.0 +0.0   +1.402 | |Y |

|Green| = |1.0 -0.3437 -0.7143| x |Cb|

|Blue | |1.0 +1.772 +0.0 | |Cr|

Chapter 7:
Basic Graphics Library

The Basic Graphics library executes drawing engine commands; the command list is registered in this library. This library handles basic data such as Sprite, polygon, and line data.

Table of Contents

Overview

The Basic Graphics library (libgpu) is a low level function library for expressing basic forms like primitives (triangles, rectangles, Sprites).

The functions of the library can be roughly classified as follows:

System functions	Control the entire graphics system such as graphics system reset etc.
Frame buffer access functions	Directly read and write the contents of the frame buffer.
Primitive functions	Manipulate the contents of primitive structures such as initializing primitives and setting the texture page.
Primitive manipulation/ordering table manipulation functions	Record primitives in an ordering table, draw primitives in the ordering table, and the like.
Synchronization control	Perform controls such as vertical retracing and drawing synchronization.

Library Files

The file name of the basic graphics library is `libgpu.lib`.

Every program calling services must link with this library. You must link `libapi.lib` and `libetc.lib` at the same time when you use libgpu.lib

Header Files

The basic graphics library header file is `libgpu.h`.

Every data structure and macro is defined by this file. Every program calling services must link with this library.

Include `libgte.h` when using `libgpu.h`.

Example

```
        #include      <sys/types.h>
        #include      <libetc.h>
        #include      <libgte.h>
        #include      <libgpu.h>
```

libgpu.h uses the following sys/types.h formats. When including libgpu.h, please include <sys/types.h> at the same time. Refer to libgpu.h for details.

```
        typedef unsigned char             unsigned char;
        typedef unsigned short            unsigned short;
        typedef unsigned int                     unsigned int;
        typedef unsigned long             unsigned long;
```

Table 7–1

Content	Filename
Library	libgpu.lib
Header	libgpu.h

Frame Buffer

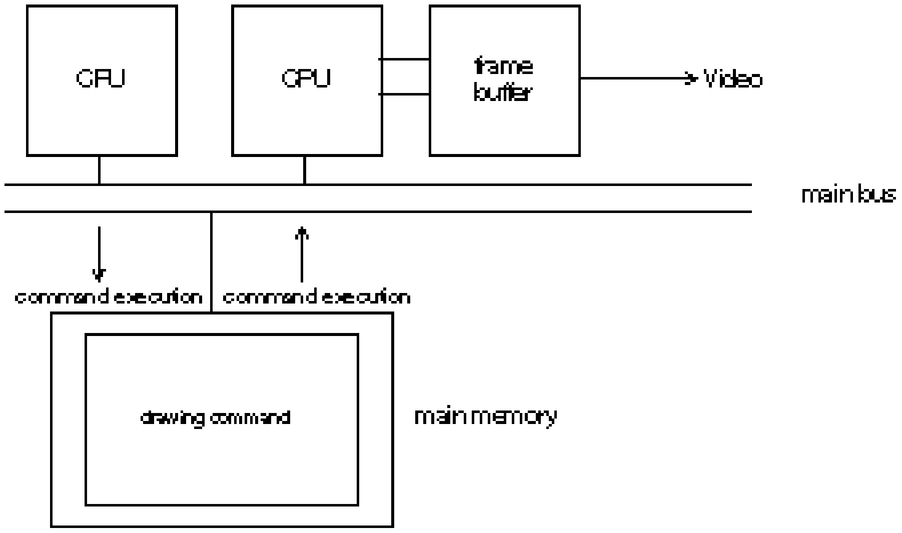
GPU

There is a specialized rendering engine (drawing device) that realizes high-speed graphics capability in the PlayStation. This is called the GPU (Graphics Processing Unit); the GPU and frame buffer together are called the Graphics System.

The GPU consecutively executes main memory instruction strings (primitives) with a frame buffer-type drawing device, and does not perform drawing in the frame buffer.

Frame buffer data is changed into a video signal, and displayed. A moving image is generated by consecutively rewriting the frame buffer contents at a display field rate of 60 fields/second.

Figure 7–1: Graphics System



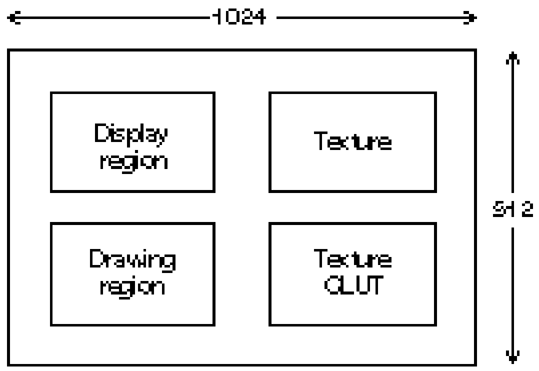
There is no special BG (background) plane for displaying image data with the graphics system after it is drawn temporarily in the frame buffer. These data structures designed to somehow correspond to main memory primitive strings; the specifics are entrusted to the applications.

Also, the GPU and frame buffer are generally termed the graphics system.

Frame buffer addressing

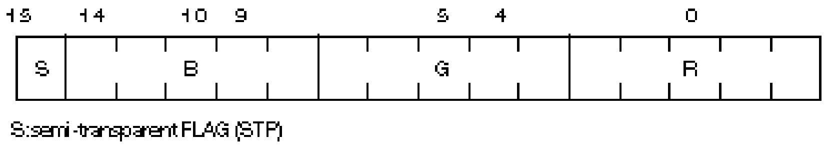
The graphics system has a local 1024 x 512 x 16-bit (1MB) video memory (frame buffer). The frame buffer is a buffer for drawing and displaying, and is also used as an area for texture patterns and texture CLUTs.

Figure 7–2: Frame Buffer



Pixels in the frame buffer are specified by 2-dimensional coordinates, (X, Y) (X = 0-1023, Y = 0-511). Each pixel has a 16-bit depth, with one semi-transparent bit and 5 bit RGB values. Pixels are configured as follows:

Figure 7–3: Pixels



Therefore the total volume of the frame buffer is 1024 x 512 x 16 16 bit = 1MB.

The contents of the frame buffer may always be accessed even if it is an area being displayed or an area where drawing is being done.

Display Area

A rectangular area, part of the frame buffer, is displayed as video images. The area is referred to as a display area.

The size of the display area depends on the display mode, ranging from 256 x 240 to 640 x 480 (709 x 488 during overscan). Any of the following combinations can be chosen.

Width	320, 360, 512, 640
Height	240, 480
Pixel mode	24-bit, 16-bit
Interlace	On, off

In the 480 height mode, however, non-interlace cannot be selected.

Drawing Area

Data is drawn in a rectangular area, part of the frame buffer. The area is referred to as a drawing area. The size of the drawing area is not limited if it is contained in the frame buffer.

Frame Double Buffer

The drawing area and dislay area may each be specified in the frame buffer, but if the drawing area is in the display area, the current drawing situation is reflected on the screen.

To achieve this effect, you usually prepare two drawing areas in advance. One area is for drawing and one for display, and, when drawing is finished, you exchange them. This is a frame double buffer.

Double buffer exchange, at the completion of drawing, in conjunction with vertical sync, exchanges the drawing environment and the display environment.

Drawing Environment

The general information related to 2D primitive drawing, such as the position of the display region and the drawing offset is called the drawing environment. Drawing environment information is held in the DRAWENV structure and may be set by calling the PutDrawEnv() function. The present drawing environment may be obtained with the GetDrawEnv() function. The DRAWENV structure is defined as follows:

```
struct DRAWENV{
    RECT clip;                  /*clipping 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;
```

Clipping (clip.x, clip.y, clip.w, clip.h)

This is a rectangular area for clipping. Drawing is carried out in the rectangular area within the frame buffer (clip.x, clip.y) - (clip.x + clip.w, clip.y + clip.h) and not outside the clip area.

Offset (ofs[0], ofs[1])

The drawing offset. ofs[0] and ofs[1] are added respectively to the X and Y values of primitives before drawing.

Texture Window (tw.x, tw,y, tw,w, tw.h)

It sets the texture window. (tw.x, tw.y) - (tw.x + tw.w, tw.y + tw.h) is the texture window. It is used to wrap round only a section of the texture page.

Texture Page (tpage)

Specifies the default texture pattern . One texture page has a size of 256 x 256.

Dither Processing Flag (dtd)

If this is set to 1, the drawing engine will perform dithering when drawing pixels.

Display Area Drawing Flag (dfe)

By default, the drawing engine blocks drawing in areas being displayed. When this is set to 1, drawing is permitted in areas being displayed.

Drawing Area Clear Flag (isbg), Background Color (r0, g0, b0)

If this is set to 1, the clipping area is cleared with the brightness value of (r0, g0, b0) when the drawing environment is set.

Display Environment

Information related to the frame buffer display, such as the position of the display region, is called the “display environment”.

The display environment may be set by using the DISPENV structure and calling the PutDispEnv() function. The present drawing environment may be obtained with GetDispEnv().

The following parameters can be set in the drawing environment.

```
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;
```

Display Area (disp.x, disp.y, disp.w, disp.h)

The rectangular area within the frame buffer (disp.x, disp.y) - (disp.x + disp.w, disp.y + disp.h) is the display area. A display width (disp.w) is given as 256, 320, 512 or 640. The display height (disp.h) is given as 240 or 480.

Screen Area (screen.x, screen.y, screen.w, screen.h)

This argument specifies where on the actual physical screen the display area is shown. The display is indicated in the display area/mode starting from the top left-hand corner, with a coordinate range of (0, 0) - (256, 240). A standard drawing is specified to be in the range of (0, 0) - (256, 240). It is an underscan if it is smaller, and an overscan if it is larger.

For example, if screen.w is set to a value greater than 256, more pixels than 256 cannot be displayed, even if it is 320 mode. The size of each pixel does not change.

Display offset (ofs[0],ofs[1])

This argument specifies the starting position on the display. The horizontal direction shows pixel units and the vertical direction shows luster units.

Interlace (isinter)

This forcibly changes the display to interlace mode. If the display mode area height is greater than 240, display will be in interlace mode regardless of the interlace setting.

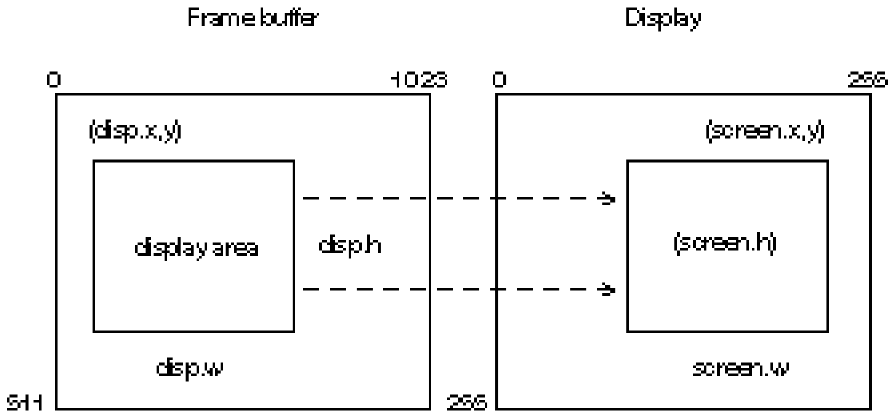
24-Bit Mode Flag (isrgb24)

This flag changes the display to 24-bit mode. When set to 24-bit mode the frame buffer is interpreted as 24 bit pixel format and displayed. Also the x address in the frame buffer and the pixel will not be 1 to 1.

Display Area and Screen Area

The relationshsip between the display area and the screen area is as indicated below.

Figure 7–4: Display Area and Screen Area



Switching Display and Drawing Environments (Realization of the Frame Double Buffer)

Double buffering provides both display and drawing environments and is implemented by switching alternately between these.

For example, when buffer 0 is (0,0)-(320,240) and buffer 1 is (0,240)-(0,480) the respective drawing and display environments would be set as follows. The PutDrawEnv() and PutDispEnv() functions are used to switch the drawing and display buffers set in the system.

Table 7–2: Double Buffer

Drawing environment

	Buffer 0	Buffer 1	Notes
(clip).x, clip.y)	(0,0)	(0,240)	Clip start point
(ofs[0], ofs[1])	(0,0)	(320,240)	Drawing offset

Display environment

	Buffer 0	Buffer 1	Notes
(disp.x, disp.y)	(0,240)	(0,0)	Display start position

In addition to using the PutDrawEnv() function to change environments, you may also dynamically switch the drawing environment in the midst of drawing by registering the DR_MODE primitive in the ordering table. The DR_MODE primitive and the ordering table will be explained later.

Basic drawing using double buffering is as follows:

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

while (1) {
      VSync(0);                    /*synchronize vertical retrace*/
      dispid = (dispid + 1) %2;    /*swap buffer ID*/
      PutDrawEnv(&drawenv[dispid]);
      PutDispEnv(&dispenv[dispid]); /*set primitive and execute*/
      }
```

The standard values of the drawing environment and display environment structures may be set with the fucntions SetDefDrawEnv() and SetDefDispEnv().

Also, when setting the drawing environment during drawing there is no effect on the primitive being executed at that time or on the primitive list. The drawing environment that has been set will be in effect from the next drawing. On the other hand settings made in the disply environment become effective immediately. Therefore, the display location and display area can be changed even whern drawing is being carried on in the background.

Interlace Double Buffer

If you use interlace mode with a height of 480 (640 x 480), a single buffer may be used for drawing and display.

In interlace mode, the even field displays even lines, while the odd field alternately displays odd lines.

Drawing the odd field during the display of the even field, and the even field during the display of the odd field leads to the use of the usual display/drawing buffer. This is accomplished by making the dfe flag zero.

The completion of drawing in the usual 640 x 480 buffer at a rate of 1/60 seconds leads to the suppression of drawing to the field currently displayed (even or odd line). Thus, a double buffer is implemented automatically.

In this case, buffer selection does not require switching between the drawing environment and the display environment.

Frame Buffer Access Functions

The contents of the frame buffer may be accessed directly using the following functions.

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

Blocking Functions and Non-Blocking Functions

Functions that take a long time to process, such as the frame buffer action function, are processed in the background and return without awaiting completion. These are called non-blocking functions. The following functions are non-blocking.

- LoadImage()
- StoreImage()
- MoveImage()
- DrawPrim()
- DrawOTag()

In contrast to these, functions that wait for the system to complete processing before returning are called blocking functions. All functions other than those listed above are blocking functions.

You can detect whether non-blocking functions have finished or not with the DrawSync() function. For example, LoadImage(), a non-blocking function, could be used as follows:

```
LoadImage(&rect, pix);
DrawSync(0);
```

A maximum of 64 non-blocking functions to be executed may be queued. For example, suppose that one drawing and several OT are running together.

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

In this example, if DrawOTag(ot0) is not completed when DrawOTag(ot1) is invoked, the system will simply register the request to the queue, and will return without carrying out the process. DrawOTag(ot1) will wait until DrawOTag(ot0) has finished and then execute automatically.

However, the queue is a maximum of 64 items, so when the 65th request reaches the queue, it will be blocked until the queue is opened. The example shows that the 65th LoadImage is blocked until the first LoadImage is completed, and until the waiting queue is available.

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

Primitives

The smallest command that the graphics system will handle is called a “primitive (packet)”. Primitives are created in main memory, and the CPU and the drawing engine both refer to them at the same time.

Primitives may be classified as “drawing primitives” for actually drawing forms in the frame buffer or “special primitives” for changing the internal status of the drawing engine.

Special primitives are for switching the parameters of the drawing engine, such as the clipping area and texture page, while drawing is being done. They cannot directly change the contents of the frame buffer.

Both special primitives and normal primitives may be registered in an ordering table (to be discussed later).

Primitives may be executed with the DrawPrim() and DrawOTag() functions. The following example outlines the two alternative methods of drawing primitives.

```
POLY_F3 f3;                     /*reserve primitive area*/

SetPolyF3(&f3);                 /*initialize primitive*/
     setRGBO(&f3, 255, 0, 0);   /*set primitive members*/
     setXY3(&f3, 0, 0, 0, 100, 100, 100);

     if (direct execution)
     DrawPrim(&f3);
else if (execute after temporarily recording in ordering table) {
     AddPrim(ot, &f3);
     DrawOTag(ot);
}
```

Drawing Primitives

The drawing primitives are listed below.

Polygon Primitives

There are shading types (Gouraud shading primitives and flat shading primitives) that corresond to polygon primitives.

You may choose the texture mapping on/off setting for any primitive, or choose whether it is a three-sided or four-sided polygon. Therefore, polygon primitive combinations can be made from the following:

Table 7–3: Polygon Primitives

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

Line Primitives

Line primitives draw straight lines.

Table 7–4: Line Primitives

Primitive name	Contents
LINE_F2	Non-connected straight line
LINE_G2	Non-connected straight line w/gradation
LINE_F3	3 point connecting straight line*2
LINE_G3	3 point connecting straight line with gradation
LINE_F4	4 point connecting straight line*2
LINE_G4	4 point connecting straight line with gradation

Sprite and Tile Primitives

Sprite indicates a drawing command rectangular area; drawing is done with a solid color and accompanying texture-mapping.

Table 7–5: Sprite Primitives

Primitive Name	Contents
SPRT	Sprite (free size)
SPRT_8	Sprite (8 x 8)
SPRT_16	Sprite (16 x 16)
TILE	Solid clor tile (free size)
TILE_1	Dot (1 x 1))
TILE_8	Solid color tile (8 x 8)
TILE_16	Solid color tile (16 x 16)

Special Primitives

Special primitives that change all or part of the drawing environment during drawing are listed below.

Table 7–6: 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

A primitive is defined as a structure in the C language. For example, the definition of POLY_FT4 (four-sided polygon, flat, textured) primitive is as follows:

```
typedef struct {
    PRIM  *tag;
    unsigned char r0, g0, b0, code;
    short x0, y0;
    unsigned char u0, v0;
    unsigned short clut;
    short x1, y1;
    unsigned char ul, 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:                pointer to next primitive
code:               primitive identifier (system reservation)
r0,g0,b0:           display color (R,G,B)
tpage:              texture page ID
clut:               clut ID
x0,y0,...x3,y3:     polygon vertex coordinates
u0,v0,...u3,v3:     polygon texture coordinates
```

As stated above, a primitive has an internal pointer (tag) towards the next primitive. Therefore, by using this pointer to display primitives in list structure, multiple primitives can be executed together.

The first two words of primitives are shared by all primitive structures. Primitive structures may be defined as below.

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

Initialization of Primitives

Primitives have a primitive identifier and a pointer to the next primitive to be executed. Because of this it is necessary to initialize a primitive before executing it with a primitive initializing function.

When initializing a primitive use the initializing function which corresponds to the type of primitive. For example, before drawing a POLY_FT4 (rectangular polygon, flat texture) primitive, it is necessary to initialize it in the following manner.

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

Primitive Attributes

The following attributes may be set for primitives.

SemiTrans	Semi-transparent mode
ShadeTex	Inhibiits function that simultaneously performs texture mapping and shading

These attributes may be set or discontinued for each primitive with the SetSemiTrans() and SetShadeTex() functions (See the example below). These functions may be called at any time once primitive initialization has been performed.

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

Setting Primitive Members

Other than `tag` and `code`, the members of a primitive may be freely written. There are numerous macros provided for setting primitive members. For example, both example 1 and example 2 below generate the same code. For details, refer to libgpu.h.

Example a

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

```
POLY_F4 f4;
SetPolyF4(&f4);     /*initialize primitive*/
f4.r = 0;
f4.g = 0;
f4.b = 255;         /* = setRGBO()*/
f4.x0 = 0;          /* = setXY4()*/
f4.y0 = 0;
f4.x1 = 100;
f4.y1 = 0;
f4.x2 = 0;
f4.y2 = 100;
f4.x3 = 100;
f4.y3 = 100;
DrawPrim(&f4);      /*execute primitive*/
```

Combining Primitives

Many primitives may be used in combination with other primitives; two primitives may be brought together to form a single new primitive.

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

setTSPRT (TSPRT*p)
{
       SetDrawMode(&p->mode);
       SetSprt(&p->sprt);
       return(MargePrim(&tsprt->mode, &tsprt->sprt));
}
```

setTSPRT() is an example of a function that initializes a new TSPRT. A primitive TSPRT initialized in this manner could use the arguments AddPrim() and DrawPrim() as it uses other primitives.

Execution of 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)*/
setRGBO(&f4, 0xff, 0x00, 0x00);               /*RGB = (255, 0, 0)*/
DrawPrim(&f4);                                /*draw*/
```

However, usually multiple primitives are connected into a list and executed together using the DrawOTag() function.

Primitive Execution Priority

The priority for executing drawing primitives is the same priority as displaying primitives. In other words, the first primitive to be executed will be displayed the furthest back and the last primitive executed will be displayed furthest forward.

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]);
```

An ordering table must be used in order to more easily control the order of execution.

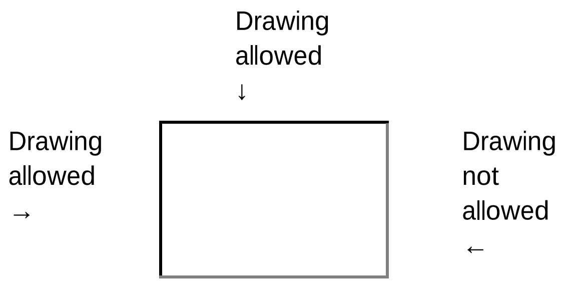
Primitive Drawing Rules

The pixels used for drawing are those where the center of the pixels lies inside the area (drawing area) linked by each vertex (x, y). When the center of the pixel is outside this area, the drawing rules are as follows:

Right pixel is inside the drawing area → can be drawn
Left pixel is inside the drawing area → cannot be drawn
Upper pixel is inside the drawing area → canbe drawn
Lower pixel is inside the drawing area → cannot be drawn

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

Figure 7–5



↑
Drawing
not
allowed

This ensures that the polygon boundary line is not drawn more than once when polygons are placed next to each other.

Primitive Drawing Speed

Primitive execution speed (drawing speed) differs depending on the primitive area and type. Drawing primitives are composed of repeated read/write access to the VRAM (video RAM) that constitutes the frame buffer. A larger primitive area results in more write access to the VRAM, requiring more drawing time.

Semi-transparent drawing requires reading access, so, the speed of drawing the same primitive is reduced by a half or more compared to opaque primitive drawing.

The drawing speed depends on the primitive type. The following shows the relative execution speed of primitives having the same drawing area.

Table 7–7: Primitive Drawing 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 on, while OffC indicates that the texture cache is off.

4bit/8bit/16bit show the texture mode. When the texture cache is missing condition, 4bit mode texture will be faster than 16bit.

Ordering Tables

Primitives are registered once in an execution queue called an ordering table (OT) using the AddPrim() or the AddPrims() functions. The primitives that are registered may be executed together with the DrawOTag() function. Using the OT has the following advantages.

1. The order of primitive execution can be easily changed.
2. Since the DrawOTag() function is a non-blocking function, the CPU can perform further processing concurrently without waiting for the completion of drawing.

The OT is 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 definition gives 256 levels of priority.

```
unsigned long ot[256];
```

It is necessary to initialize the OT prior to use, with the ClearOTag() function.

```
ClearOTag(ot, 256);
```

The following list will be formed in the OT by the ClearOTag() function. (EndofPrim) indicates the end of the list of primitives.

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

Registering to the OT

Before drawing, primitives must be registered once in the OT. Use the AddPrim() function to register primitives.

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

The execution priority of each primitive is determined by the position it is registered in the OT. The first primitive registered in the OT will be executed first, the last primitive registered in the OT will be executed last.

The display priority of each primitive is determined by its execution priority. Thus, the first primitive registered will be the lowest priority and the last primitive registered in the OT will be the highest priority. In the following example, p1 is executed first, so is displayed furthest back on the screen. p2 is executed last so is displayed furthest forward, i.e. it overwrites any primitives already drawn.

Note that primitives begin to execute, not when AddPrim() is called, but when DrawOTag() is called.

```
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 switched during the drawing of all or part of the drawing environment and the following exist.

Special primitives, like normal primitives, may be registered once in the OT, then executed together with normal primitives using the DrawOTag() function.

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);
```

Primitive Links

OT can also be regarded as a null primitive having only a tag. Thus, primitives can be connected directly using no OT, followed by DrawOTag function batch execution.

For example, the following provides the same operation as DrawPrim().

```
myDrawPrim(void *p)
{
       TermPrim(p);
       DrawOTag(p);
}



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

OT and Z Sort

Z sorting is a method of eliminating hidden surfaces by creating a list of primitives ordered by their depth (z value). If the primitives in the list are then drawn starting with those furthest in the distance, the nearer primitives will overwrite the more distant ones. The OT executes the primitives registered in the higher entries first and the primitives registered in the lower entries later. Thus you may implement Z sorting using the OT if you calculate the position of OT registration from the primitive Z.

In the following example, you can calculate the position of OT registration for primitive p0 using its Z value. Registering all primitives in the OT in this way will implement a distance ordered list.

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

OT and geometry functions

Many of the functions in the basic geometry library calculate an otz value (to help create a Z ordered OT) while performing 3 dimensional coordinate conversion.

```
SVECTOR x3, x2;
int flag, 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, with 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.

otz is the Z coordinates divided by 4, and may be used as is for calculation of the index to the OT. Because otz divides the Z coordinates by 4, it is sufficient to provide an OT with 1/4 of the dynamic range of actual Z. By making use of otz, a 3-dimensional Z sort can be performed at high speed.

Reverse OT

otz takes a large value for distant objects; as they get nearer the value approaches zero. Because of this, it is necessary to invert otz's sign to use it as an OT index.

To avoid this, there is a function provided (ClearOTagR()) which initializes the OT in advance so that the order of OT execution will be reversed.

The usual ClearOTag() functionwill initialize the OT as follows:

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

ClearOTagR() functionwill initialize the OT as follows:

```
ClearOTagR(ot, OTSIZE)
ot[OTSIZE-1]->ot[OTSIZE-2]->ot[OTSIZE-3]->
.....ot[0];
```

You can use the function ClearOTagR() and the initialized OT as an otz index. Note that the OT starting pointer address passed to the function DrawOTag() is also passed.

The table below shows the preceding information in table format.

Table 7–8: OT

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

In this way, in the case of reverse order OT, the OT starting pointer address that is passed to the DrawOTag() function is also passed.

The normal order OT is a 2-dimensional application, and the reverse order OT is a 3-dimensional application. Shared number entries in the reverse order OT greatly increase the size of the array; when a large OT is cleared, the reverse order OT clear is performed at high speed.

Combining with Geometry Functions

To display three dimensional objects, an object is broken up into triangles and quadrilaterials, and the coordinates of each polygon are calculated. This determines the position of the corresponding primitive. In other words, the location of the primitive in the frame buffer—the (x,y) member of the primitive—is obtained from the 3D coordinates of a polygon comprising 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.

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

Figure 7–6: Polygon Vertex Format

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

The drawn figure 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 7–7: Perspective Transformation

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

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.

Example: Perspective Transformation

```
rotTransPersAddPrim(pos, s, ot, ot_size)
SVECTOR *pos;              /*position*/
SPRT *s;                   /*Sprite primitive*/
unsigned long *ot;         /*OT*/
int ot_size;               /*size of OT*/
{
       long otz, dmy, flg;
       otz = RotTransPers(&pp->x[0],(long*)&sp
                             ->x0,&dmy,&flg);
       if (otz > 0 && otz < ot_size)
             AddPrim(ot+otz, sp);
}
```

Multiple OTs

OT is a simply linked null primitive array. An OT can be cataloged as one entry for a parent OT. This cataloging is valid for using more than one hierarchical coordinate system.

The following example connects sub OT ot1 with a length of n to ot0.

```
AddOT(unsigned long *ot0, unsigned long *ot1, int n)
{
```

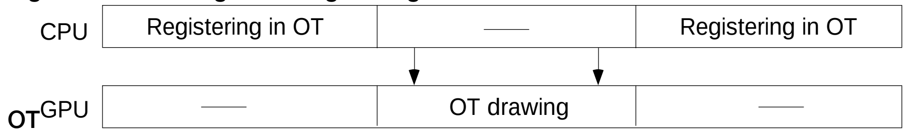
```
                AddPrims(ot0, ot1, ot1+n-1);
        }
```

Packet Double Buffer

The general term for the OT and primitive area is packet buffer.

Waiting for a primitives to be drawn after they have been registered in OT makes it impossible to operate the CPU and the graphic system in parallel.

Figure 7–8: Drawing After Registering in



Operating the graphic system and the CPU in parallel requires two packet buffers, a setting packet buffer and an execution packet buffer. The two packet buffers assume the tasks of drawing and execution alternately. This is referred to as a packet double buffer system.

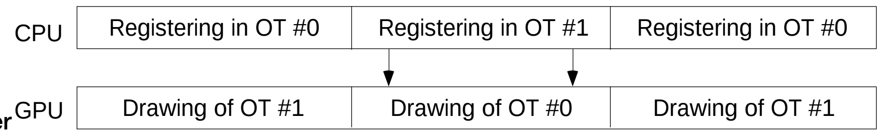


Figure 7–9: Packet Double Buffer

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;
        while (1) {
             cdb=(cdb==db)? db+1: db;   /*switches buffers*/
             ClearOTag(cdb->ot);        /*clears OT*/
             for(j=0; j<256; j++){      /*registers Sprite*/
                            /*at this point, calculate the Sprite position*/
                            AddPrim(cdb->ot, cdb->sprt[j];
        }
        DrawOTag(cdb->ot);              /*Drawing*/
    }
}
```

Synchronization and Reset

There are the polling and callback methods that may be used to identify the completion of the drawing process presently carried out in the background.

Polling

DrawSync (0)	Block until all requests remaining in the queue are finished.
DrawSync (1)	Find queue status and execution status

There are numerous levels of reset when resetting the graphics system: current requests may be stopped in mid-process and the queue flushed, current requests only may be stopped, etc. The ResetGraph() function argument controls the reset level.

Callback

Functions that are defined to be called when a vertical sync occurs or a drawing finishes (callbacks) also have a way of detecting the end of a background process. There are two graphics library callback registration functions that do this: VSyncCallback() and DrawSyncCallback().

Table 7–9: libgpu Callback Registration Functions

Function Name	Trigger
VSyncCallback()	Vertical sync
DrawSyncCallback()	End of drawing

VSyncCallback() specifies a function to be called at the beginning of vertical sync.

```
main() {      /*initialization routine entered here*/

VSyncCallback(1, callback);      /*defines callback*/
callback();                /*cannot return*/
}
callback() {  /*processing within the frame entered here*/

while (1);    /*cannot return*/
}
```

Texture Mapping

Texture mapping may be performed while drawing polygons and Sprites, using patterns placed in texture pages.

Texture mapping is a method of mapping a 2-dimensional image known as a texture pattern onto the surfaces of triangles and quadrilaterals.

Texture patterns are normally 256 x 256 pixel units placed in areas other than the display areas and drawing areas of the frame buffer. These are called texture pages.

There are three types (modes) of pixel format in texture patterns: 4, 8, and 16-bit as shown below. Each primitive may have a different mode.

Table 7–10: Texture Pattern Modes

Mode	Type	No. of colors
4-bit	Pseudo	16
8-bit	Pseudo	256
16-bit	Direct	32767

When using texture mapping, it is necessary to specify the texture page to be used, the offset value in the texture page corresponding to each vertex and when needed, the texture CLUT.

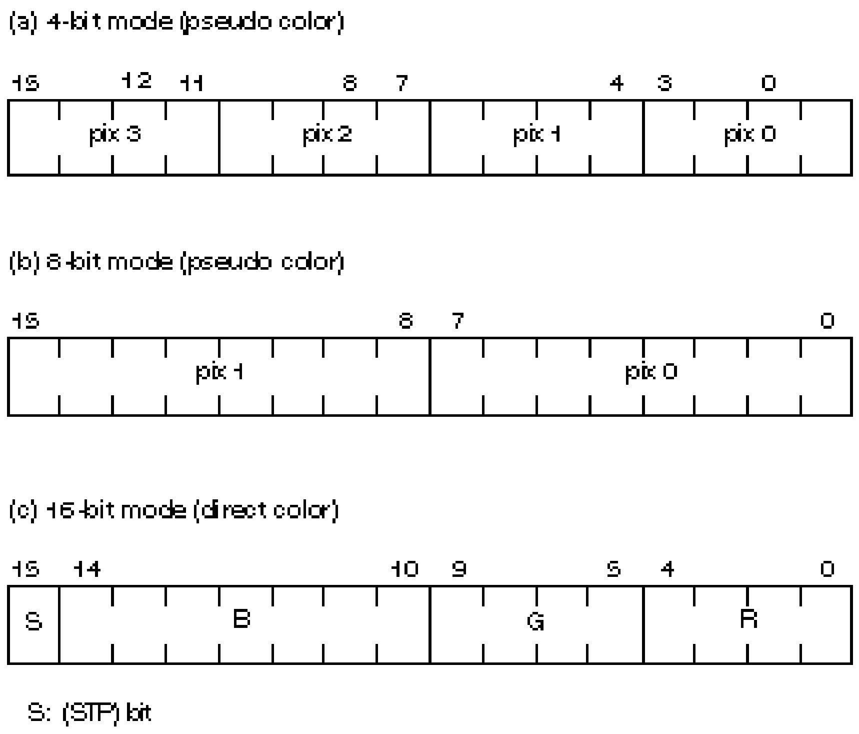
Texture page offsets normally use the symbols `U` and `V` to distinguish them from the coordinate values (X, Y) in the frame buffer.

Texture Pattern Format

The 4-bit, 8-bit and 16-bit texture patterns are placed in the frame buffer in a format in which varying numbers of pixels are packed in one word. Thus, the texture pattern pixels (U, V) and the frame buffer addresses will not be 1 to 1. Because of this care must be taken with specifying size when loading texture patterns into the frame buffer using the LoadImage() function.

The pixel format for each mode is shown below.

Figure 7–10: Texture Pattern Format


(a) 4-bit mode (pseudo color)
15
12
11
8
7
4
3
0
pix 3
pix 2
pix 1
pix 0
(b) 8-bit mode (pseudo color)
15
8
7
0
pix 1
pix 0
(c) 16-bit mode (direct color)
15
14
10
9
5
4
0
S
B
G
R
S: (STP) bit

With 4-bit and 8-bit modes, the pixel value is not the actual brightness value, but is an index to the CLUT. The CLUT pixel pattern format is the same as the 16-bit mode format.

Transparent Pixels and Semi-Transparent Pixels

You may select transparent, opaque or semi-transparent for each pixel when performing texture mapping.

Only when the pixel value (the corresponding CLUT value in 4 and 8-bit mode) of the texture pattern is 0x0000 (STP, R, G and B are all zero) are the pixels transparent and therefore not drawn. These are called transparent pixels.

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 7–11: 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 black	Non-transparent	Semi transparent

The STP bit is the most significant bit in each entry in the CLUT in 4-bit and 8-bit modes. In 16-bit mode, the most significant bit of the texture pattern is the STP bit.

Untexture mapped (flat and Gouraud) primitives may be set in semi-transparent mode using the SetSemiTrans() function. In these cases, all pixels drawn will be semi-transparent.

Semi-Transparent Pixel Mode

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

Table 7–12: 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 of a semi-transparency process is clipped when it exceeds the range (0-255). Semi-transparency rates may be used by a current texture page mentioned later. The same rate is applied to primitives that do not perform texture mapping.

Texture-Mapping Primitive Brightness Values

In the case of a texture mapping 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 drawing is calculated from the corresponding texture pattern pixel value T and brightness value L, as shown below:

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

In other words, if the (r, g, b) fields are all set to 0x80 (=128), a drawing will be done as is, at the source texture brightness value.

For this reason, 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*/
set RGB0(&fT4, 0x80, 0x80, 0x80);/*Initializes the brightness value*/
```

Texture Pages

Texture pages may be specified anywhere in the frame buffer but the X of the page's upper left address is restricted to a multiple of 64 and Y a multiple of 256.

A 256 x 256 pixel texture pattern is placed in 1 texture page. The area occupied by a texture page in the frame buffer differs, depending on the mode, from 256 x 256 (16-bit mode) to 64 x 256 (4-bit mode).

Texture pages are specified in primitives not by the actual page address (X, Y) but by the texture page ID. The texture page ID may be obtained with the GetTPage() function.

Texture CLUT

When performing texture mapping in 4-bit and 8-bit mode, it is first necessary to specify the texture CLUT that will be used. Texture CLUTs may be set independently for each primitive regardless of the texture to be used.

CLUTs are arranged in narrow and long form horizontally in the frame buffer, 256 x 1 (8-bit mode) or 16 x 1 (4-bit mode).

CLUTs are specified in primitives not by the actual CLUT address (X, Y) but by the texture CLUT ID. The texture CLUT ID may be obtained with the GetClut() function.

Specifying Texture Page ID and Texture CLUT ID

Texture page ID and texture CLUT ID are specified by assigning respective values to a primitive tpage and clut members. For example, it would be done as follows:

```
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 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*/
```

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

Current Texture Page

The initial current texture page may be specified in the drawing environment.

Because Sprite primitives (SPRT) do not have tpage members, they may not specify texture pages. Thus, when a Sprite is being executed, the current texture page is used.

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, 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 the same OT entry is executed from the latest registered primitive.

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 or by using the DR_MODE primitive. Please refer to the following example. Texture windows are normally set within (0,0) - (255, 255).

```
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 of drawing mode primitive*/
SetDrawMode(&dr_mode,0,0,GetTPage(0, 0, 640, 0), tws, twe):
        :
AddPrim(ot+n, &dr_mode);
```

Texture Cache

Texture patterns are internally cached. The texture cache size varies depending on the texture mode, as shown below.

Table 7–14: Texture Cache Size

Texture	Cache size
4-bit texture	64 × 64
8-bit texture	32 × 64
16-bit texture	32 × 32

If the texture pattern used by a primitive is within this range, the texture pattern is cached to enable high-speed drawing.

Only one CLUT entry is cached.

As long as the CLUT entry used by a primitive does not change, it can be cached to enable high-speed drawing.

Both the texture cache and CLUT cache are automatically flushed when LoadImage() and MoveImage() are called.

Debug Environment

Debug Mode

Once 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 described later.

Debug String

When debug mode is set by the SetGraphDebug() function, or the content of the 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 as far as possible not to depend on any particular data structure and paradigm. So there are no functions that can directly handle either TIM format, a two-dimensional image related data structure, or TMD format, a three-dimensional object data structure. In order to handle these, it is necessary to use the functions of the integrated environment of the extended graphics library.

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 the high level library and the low level library.

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

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

Notes when programming

This section explains the following items when creating an application:

Texture polygon coordinate specification
Drawing processing high speed evaluation rule
Texture cache configuration
Correspondence to PAL mode
Frame buffer switching timing

Texture Polygon Coordinate Specification

The following problems have been reported when drawing textured polygons.

1. When attempting to display 16x16 texture mapping on a 16x16 polygon, entering 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.

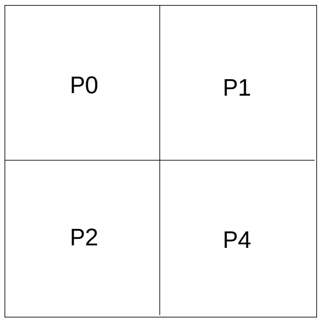
Cause and Effect

According to the drawing rules for PlayStation POLY_... primitives, drawing cannot be performed along the right and bottom edges.

This rule prevents the polygon boundary lines from being written upon 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 7–14: Drawing rule



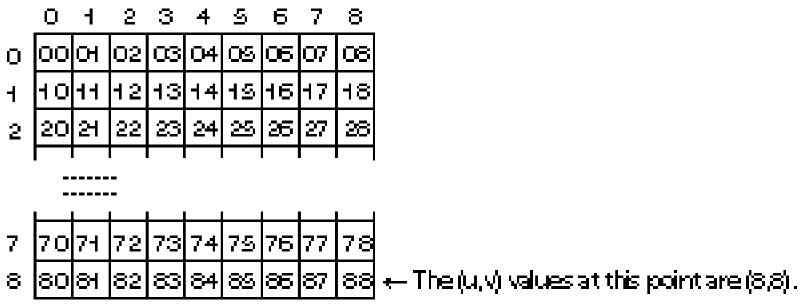
The above example assumes that a square specified by the coordinates (x, y) = (0, 0) to (8, 8) and (u, v) = (0, 0) to (8, 8) is being drawn as POLY_FT4. In other words, the following is assumed.

```
POLY_FT4 ft4;          ft4.x0=0, ft4.y0=0;   ft4.x1=8, ft4.y1=0;    ft4.x2=0, ft4.y2=8;
                       ft4.x3=8, ft4.y3=8;   ft4.u0=0, ft4.v0=0;    ft4.u1=8, ft4.v1=0;
                       ft4.u2=0, ft4.v2=8;   ft4.u3=8, ft4.v3=8;
```

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 7–15: Mapping



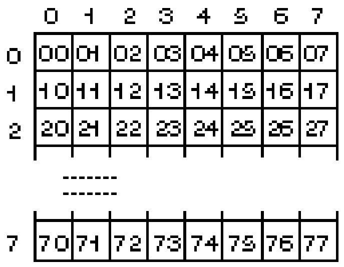
	0	1	2	3	4	5	6	7	8
0	00	01	02	03	04	05	06	07	08
1	10	11	12	13	14	15	16	17	18
2	20	21	22	23	24	25	26	27	28

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 7–16: Displayed contents



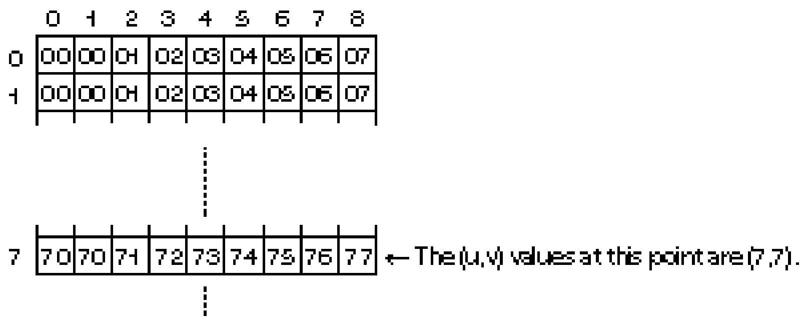
	0	1	2	3	4	5	6	7
0	00	01	02	03	04	05	06	07
1	10	11	12	13	14	15	16	17
2	20	21	22	23	24	25	26	27

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).

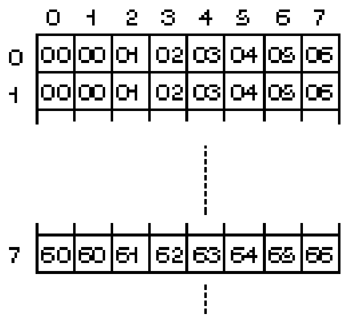
Next, if (u, v) = (0, 0) to (7, 7), the mapping is as shown below.

Figure 7–17: Mapping



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 7–18: Displayed Contents



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

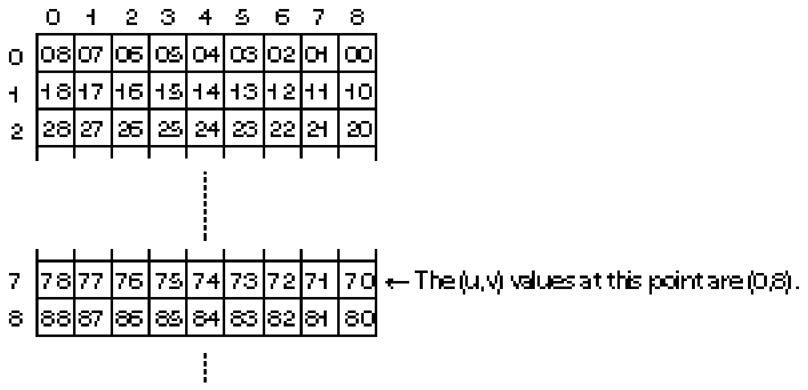
For example, suppose you want to rotate the texture described above 180 degrees in the XY direction and then display the rotated texture. Without changing the (u, v) values for POLY_FT4, you can specify the following for (x, y).

```
ft4.x0 = 8, ft4.y0 = 0;
ft4.x1 = 0, ft4.y1 = 0;
ft4.x2 = 8, ft4.y2 = 8;
ft4.x3 = 0, ft4.y3 = 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.

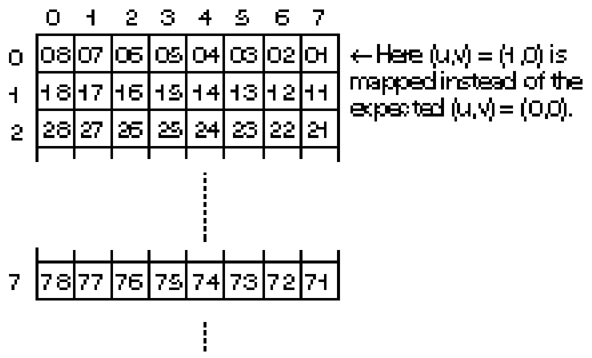
Figure 7–19: Mapping



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 7–20: Mapping


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

This effect may occur when the texture pattern is inverted vertically or when the polygon is rotated 90 or more degrees.

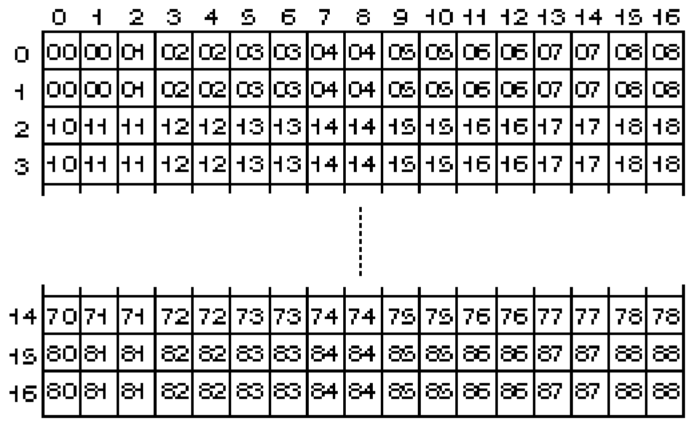
In particular, when the polygon is rotated, the mapped texture pixels change depending on the angle of rotation.

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).

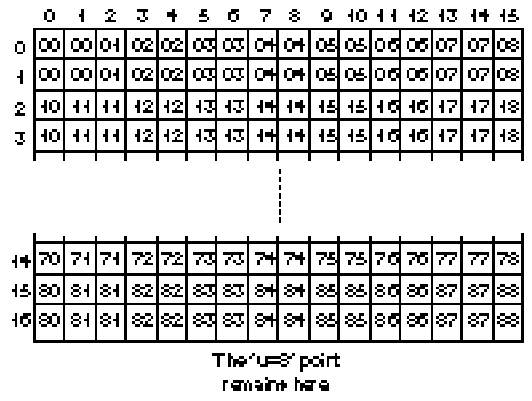
Figure 7–21: Mapping



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	00	00	01	02	02	03	03	04	04	05	05	06	06	07	07	08	08
1	00	00	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 7–22: Mapping



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	00	01	02	02	03	03	04	04	05	05	06	06	07	07	08
1	00	00	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
16	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.

```
ft4.u0 = 248,  ft4.v0 = 248;
ft4.u1 = 256,  ft4.v1 = 248;
ft4.u2 = 256,  ft4.v2 = 256;
ft4.u3 = 248,  ft4.v3 = 256;
```

But because the resolution of (u, v) is 8 bits, these values are rounded as shown below.

```
ft4.u0 = 248,  ft4.v0 = 248;
ft4.u1 = 255,  ft4.v1 = 248;
ft4.u2 = 255,  ft4.v2 = 255;
ft4.u3 = 248,  ft4.v3 = 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.

Rules for evaluating rendering speed

The rendering speed of the PlayStation is determined by either the rendering speed of the GPU or the computation speed of the CPU, whichever is slower. The time a DrawSync() operation is blocked can be used to determine whether the CPU or the GPU is faster. If the CPU is slower than the GPU (i.e. the speed bottleneck is in the CPU), DrawSync() will return immediately. If the CPU is faster than the GPU (i.e. if the speed bottleneck is in the GPU), then the amount of time DrawSync() is forced to wait is a measure of the latency through the GPU. The following is an explanation of a few of the factors that determine the rendering speed of the GPU.

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).

Access rules

(1) Basic rules

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

Table 16: Access cycles

Access direction	pixels/cycle	Notes
Write	2	SPRT,TILE,POLY_F3,POLY_F4
	1	Other
Read	1	

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

Table 17 Number of access cycles

	POLY_G4	POLY_F4
Total number of pixels	100x100=10000	100x100=10000
Total cycles	10000	5000

Calculating the speed of rendering texture maps is extremely complex. However, we can first consider a simple texture miss/hit, where the mapping is 1:1.

Table 18 Number of cycles in POLY_FT4

In the case of a 100x100 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

Table 19 Number of cycles in SPRT

In the case of a 100x100 SPRT:

Mode	4-bit	8-bit	16-bit
Total number of reads	10000/4=2500	10000/2=5000	10000
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. These calculations depend on the number of texture pixels that are packed into a single pixel.

(2) Texture enlargement ratio dependencies

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

Table 20 Number of cycles used when reduction is involved

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

Please note that simply halving the area will not halve the rendering time.

Rendering speed will improve when a texture is expanded. This is because the same texture pixels can be used multiple times and fewer texture reads will be needed for rendering an area.

(3) Texture cache dependencies

In the above examples, the texture cache is always missed. However, the GPU has an internal texture cache, and a texture for which there is a cache hit can be used directly without a read operation.

Calculating using the previous example:

Table 21 Texture cache dependencies

In the case of a 100x100 SPRT

Mode	4-bit	8-bit	16-bit
Cache on	5000	5000	5000
Cache off	7500	10000	15000

```
+---------------+------+-------+-------+
```

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

Clipping

The number of rendering cycles will also vary depending 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.

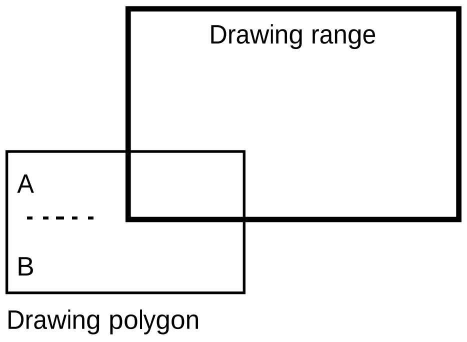


Fig. 21 Clipping

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

When rendering a texture-mapped polygon, the texture pattern in the frame buffer must be read first. Thus, rendering speed will decrease by the number of read cycles required to read in the texture pattern. For this reason the PlayStation is equipped with an internal texture cache which allows texture patterns that are in cache to be used without performing read accesses from the frame buffer.

In general, it is sufficient to prepare data so that it fits into the texture cache. However, a more detailed knowledge of the cache is necessary if the rendering speed needs to be fine-tuned. The following is a detailed description of the texture cache for the benefit of the power-user who wants to fine-tune the use of the texture cache.

(1) Cache blocks

The texture page is handled by dividing it 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 and the texture page is divided into 16 cacheblocks as shown below.

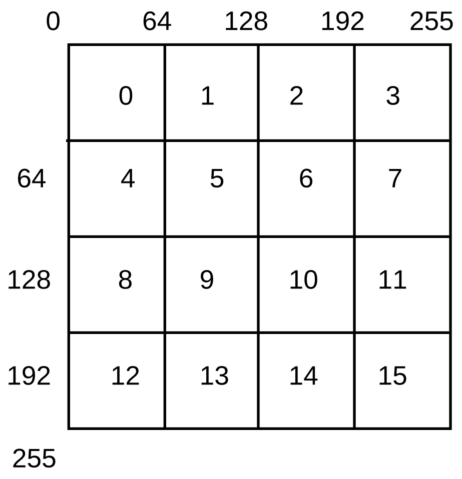


Fig. 22 Cache blocks

(2) Cache entries

Cache blocks can be divided further into smaller 16 x 1 regions (known as cache entries). In 4-bit mode, there are 256 cache entries and they are arranged as shown below.

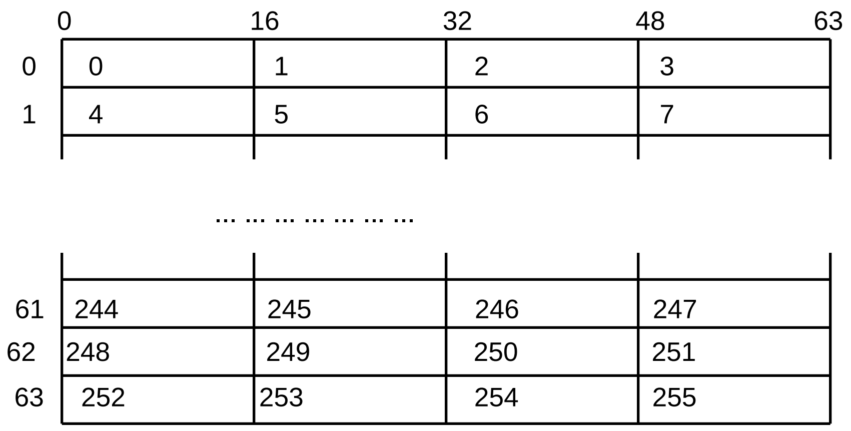


Fig. 23 Cache entries

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.

(3) 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

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.

(4) Mode dependencies

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

Table 22

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

Handling PAL format

Currently, PlayStation applications are designed based on NTSC video signals. A number of changes are necessary in order to output a signal for PAL-format TV receivers.

(1) Differences between NTSC and PAL

The two points below are the major differences between NTSC and PAL.

Table 23 Differences between NTSC and PAL

	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. Conversely, if the NTSC-format display area is used on a PAL TV, an upper or lower section of the screen will appear dark.

(2) 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.

(a) Enable PAL mode using the SetVideoMode() function
(b) Adjust the display starting position
(c) Adjust timing
(d) Adjust display area

Of these, changes in (a) and (b) are required. Changes in (c) and (d) are optional depending on the application.

Enabling PAL mode

For the DTH-2000, 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

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.

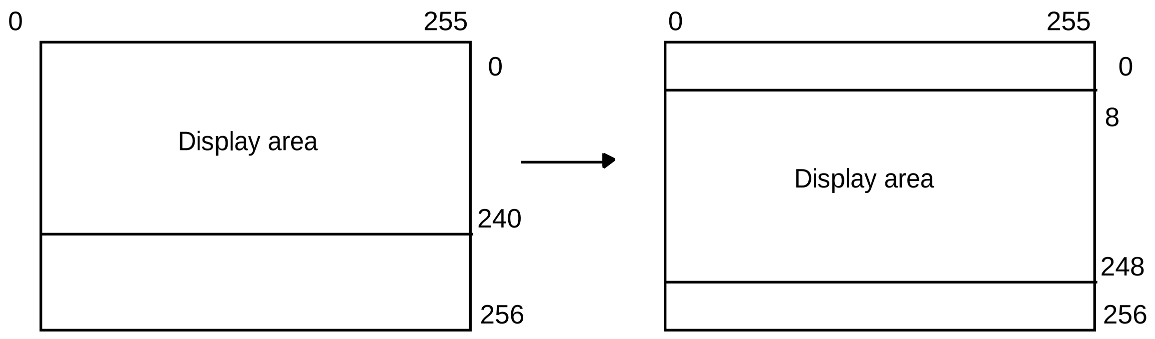


Fig. 24 Display Starting Position

```
        DISPENV         disp;
```

```
disp.screen.x = 0;      /* same as NTSC */
disp.screen.y = 8;      /* (256-240)/2 */
disp.scrborn.w = 256;   /* same as NTSC */
disp.scrbonn.h = 240;   /* same as NTSC */

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 sec, updates will be performed at intervals that are not in synchronization with the vertical sync interval, resulting in flickering on the screen. The user may think of this as a deficiency in the application.

If the time spent in calculation and rendering operations exceeds the rate at which the frame is updated (the frame rate), motion on the screen may appear delayed for a moment. This is referred to as a skipped frame and is a phenomenon that is generally tolerated, but it may still be interpreted by some users as a defect.

Therefore, please take note of the following points related to managing frame rates when developing an application.

(1) Timing for updating double-buffer switching

Switching of double buffers is generally performed by synchronizing the switching with the vertical sync.

In (A), switching of the buffers 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);
}                            }
```

(2) Keeping the frame rate constant

If buffer switching is forcibly synchronized with vertical sync, operations will complete precisely in the vicinity of 1/60 sec. This will result in the frame rate switching frequently between 1/60 sec and 1/30 sec, making objects move in an unnatural manner and can lead to complaints from users.

In these cases, the frame rate can be fixed to 1/30 sec by using VSync(2).

```
while (1) {

.....

DrawSync(0);
VSync(2);    /* force to 1/30 sec */
swap_buffer();
DrawOTag(ot);
}
```

As much as possible, it is desirable to keep the frame rate constant using VSync(n). For example, when the frame rate is 20 frames/sec, it can be fixed using VSync(3). Similarly, when the frame rate is 15 frames/sec it can be fixed using VSync(4).

(3) Using absolute time

Some applications cannot be uniformly converted to the slowest possible frame rate. For example, it would not be efficient to convert an entire program to perform at the slowest possible frame rate if frames are skipped only at specific instants.

Also, forcing buffer switching to stay in sync with the fixed vertical sync necessarily generates idle periods where the CPU/GPU perform no real operations. An application may be able to avoid this problem by not synchronizing with vertical sync when the change between frames is slight or localized.

In these cases, the internal clock of the program should not be based on the number of buffer switches performed. Instead, an absolute counter such as VSync(-1) or RCnt3 should be used.

```
(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);
}                       }
```

If frame counts are performed as shown in (B), the internal counter will not be delayed even if frames might be skipped momentarily due to overflow in calculation or rendering. If this counter is used in updating the position of an object, the object will appear to move naturally even if frames are skipped.

Conversely, there is no need to keep a fixed frame rate if the displacement or scrolling of an object is based on an absolute counter. Thus, this measure can be considered a more thorough solution than forcing a fixed frame rate.

However, there will necessarily be an increased load on the program if updates are consistently made independently from the switching of the double buffers. Thus, the choice of method will have to be determined based on the objective of the application.

(4) Cancelling rendering operations

In interlaced mode, both calculation and rendering operations must be completed within 1/60 sec. For this reason, buffer switching must give vertical sync higher priority than the completion of rendering (DrawSync(0)).

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.

(5) Return value from VSync(1)

If a '1' is passed as a parameter to VSync(), the return value will be the time elapsed since the last time VSync() was called, in units of number of horizontal sync intervals.

However, in this code,

```
VSync(0);              /* wait for V-BLNK */
t = VSync(1);                  /* value of H from last VSync(0) */
```

t should be close to 0. But, this is not the case because

* sound callbacks
* drivers for the controllers

are executed between the above operations.

Chapter 8: Basic Geometry Library

This library uses the GTE as a co-processor to perform high-speed geometry operations in the PlayStation. The Basic Geometry library provides functions that call GTE functionality and useful functions that perform other geometry operations.

Table of Contents

Overview

Depiction of polygons with the PlayStation is not done with a pipeline processing of polygon units which proceed from calculation of geometry to drawing.

Geometry arithmetic → drawing

Rather it proceeds from calculation of geometry, to sorting, to drawing, in which polygons (plural) of one screen must be processed as units. Because of this, one function does not perform everything from the calculation of geometry through to the drawing of one polygon. Rather, one must proceed from calculation of geometry to drawing in stages, by combining separate functions.

Geometry arithmetic → sorting → drawing

Thus, a GTE coprocessor (geometry conversion engine) which performs high-speed geometry calculation is part of the PlayStation. The basic geometry library (libgte) provides the functionality of the GTE and other useful functions in geometry calculation.

Library File

The file name of the basic geometry file is `libgte.lib`. Every program calling services must link with this library.

Header File

The include file of the basic geometry library header file is `libgte.h` It is the same header file as is used by the basic graphics library.

Each type of data structure and macro are defined by this file. Every program calling services must link with this library.

Table 8–1

Contents	File name
library	libgte.lib
header file	libgte.h

Theoretical Geometry Operations Using the Basic Geometry Library

The GTE is activated By using the functions provided by the basic geometry library, and coordinate conversion may be calculated at high speed with the PlayStation. The geometry calculations supported by this library may be roughly divided into coordinate calculation and light source calculation.

Coordinate calculation is calculation from the three dimensional coordinates of the vertices of polygons to find 2 dimensional coordinates on a screen (display screen) and consists of coordinate conversion and or perspective conversion.

Light source calculation is calculation to find the lighting of a polygon on a 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:

- The local coordinate system:

The fixed coordinate system of the object,

- The world coordinate system:

The fixed coordinates of the world in which the object is placed

- The screen coordinate system:

The fixed coordinates of the screen.

The “object” is an object composed of an aggregation of multiple polygons and multiple objects composing one screen. Thus, it is possible that there will be multiple local coordinate systems.

Normally, the vertex data of each polygon are given in the coordinate values of the local coordinate system. To convert these into coordinates on the screen the following coordinate conversion is necessary.

Local coordinate system (Vx,Vy,Vz) $\longrightarrow$ World coordinate system (Wx,Wy,Wz)

World coordinate system (Wx,Wy,Wz) $\longrightarrow$ Screen coordinate system (Sx,Sy,Sz)

$$\begin{bmatrix}Wx\\Wy\\Wz\end{bmatrix} = [WLij] \times \begin{bmatrix}Vx\\Vy\\Vz\end{bmatrix} + \begin{bmatrix}WLx\\WLy\\WLz\end{bmatrix}$$

$$\begin{bmatrix}Sx\\Sy\\Sz\end{bmatrix} = [SWij] \times \begin{bmatrix}Wx\\Wy\\Wz\end{bmatrix} + \begin{bmatrix}SWx\\SWy\\SWz\end{bmatrix}$$

[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:

$$\begin{bmatrix}Sx\\Sy\\Sz\end{bmatrix} = [SWij] \times \left([WLij] \times \begin{bmatrix}Vx\\Vy\\Vz\end{bmatrix} + \begin{bmatrix}WLx\\WLy\\WLz\end{bmatrix} \right) + \begin{bmatrix}SWx\\SWy\\SWz\end{bmatrix}$$

$$= [SWij] \times [WLij] \times \begin{bmatrix}Vx\\Vy\\Vz\end{bmatrix} + \left([SWij] \times \begin{bmatrix}WLx\\WLy\\WLz\end{bmatrix} + \begin{bmatrix}SWx\\SWy\\SWz\end{bmatrix} \right)$$

The synthesized coordinate conversion matrices between coordinate systems and translating vectors are called:

Rotation matrix (RTM)

$$[Rij] = [SWij] \times [WLij]$$

Translating vector (TRV)

$$\begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix} = [SWij] \begin{bmatrix} WLx \\ WLy \\ WLz \end{bmatrix} + \begin{bmatrix} SWx \\ SWy \\ SWz \end{bmatrix}$$

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 $\longrightarrow$ Screenl coordinate system

(Vx,Vy,Vz) (Sx,Sy,Sz)

$$\begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix} = [Rij] \begin{bmatrix} Vx \\ Vy \\ Vz \end{bmatrix} + \begin{bmatrix} TRx \\ TRy \\ TRz \end{bmatrix}$$

....(1)

The constant rotation matrix and the constant translating vector may be set by the SetRotMatrix() and SetTransMatrix() functions of the basic geometry library. By calling this, coordinate conversion with the constant rotation matrix and the constant translating vector can be performed.

The constant rotation matrix and the constant translating vector need not be changed if the location of the coordinate system and position do not change and can be set by these functions:

SetRotMatrix
SetTransMatrix

However, when a different local coordinate system is used for each object, each will need to be separately reset.

Note: The local coordinate system setting method is up to the user.

RotTrans is the function used to find the screen coordinate value calculation (1) from the local coordinate system in libgte.

If this is called, coordinate conversion can be jperformed by the previously set rotation matrix and translating vector.

Using a screen coordinate value found with the RotTrans() function, a parallel projected image of the object may be formed on the screen. In real vision, a distant object needs to be perspective converted so that it appears small.

$$\begin{tikzcd}\text{Screen coordinate system } (Sx, Sy, Sz) \arrow[r, "\text{perspective conversion}"] & \text{Screen coordinate system } (SSx, SSy)\end{tikzcd}$$

$$\begin{bmatrix} SSx \\ SSy \end{bmatrix} = (h/Sz) \begin{bmatrix} Sx \\ Sy \end{bmatrix}$$

h expresses the distance from the eye to the screen.Perspective conversion is done by multiplying the screen coordinate X and Y components with h/Sz.

There is a RotTransPers() function for performing RotTrans() and perspective conversion in the basic geometry library.

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 queing interpolation coefficient p is calculated at the same time.

Light Source Calculation

The light source calculation model hypothesized by GTE is a parallel light source complete diffusion reflection model. In other words, it does not rely on the position of the point of view but decides lighting on the basis of the attributes of the source of illumination and the attributes of the polygon.

Here we shall explain light source calculation for one vertex of a polygon. There are two attributes of one vertex of a polygon:

- Normal line vector (Nx, Ny, Nz)
- The color of the vertex (R, G, B)

The normal line vector is usually given by the local coordinate system. On the other hand, there are three attributes of the light source:

- Light source vector (direction and intensity) (Lx, Ly, Lz)
- Color of the light source (LR, LG, LB)
- The ambient color (BKr, BKg, BKb)

The light source is a parallel light source, so there is no position information. Because the light source 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) The coordinates of the normal line vector are converted into the world coordinate system.

$$\begin{tikzcd}\text{Normal line vector (local)} \arrow[r] & \text{Normal line vector (world)}\end{tikzcd}$$

$$\begin{bmatrix} NWx \\ NWy \\ NWz \end{bmatrix} = [WLij] \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

2) Take the inner product of the light source vector and the normal line vector (world).

Normal line vector (world) * Light source vector $\longrightarrow$ Light source effect (L)

$$L = \begin{bmatrix} Lx & Ly & Lz \end{bmatrix} \begin{bmatrix} NWx \\ NWy \\ NWz \end{bmatrix} = \begin{bmatrix} Lx & Ly & Lz \end{bmatrix} \begin{bmatrix} WLij \end{bmatrix} \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

3) This inner product and the color of the light source are multiplied for each item. The color effect of the light source (local color) for vertices can be found from this.

Light source effect (L) * Light source color (Lr, Lg,Lb) $\longrightarrow$ Light source color effect (Ll)

$$\begin{bmatrix} LIr \\ LIg \\ LIb \end{bmatrix} = L \begin{bmatrix} Lr \\ Lg \\ Lb \end{bmatrix}$$

)

4) The light source color effects and the ambient colors are totalled to find the color effect of the whole environment.

Light source color effect (Ll) + Ambient color (BK) $\longrightarrow$ 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) The color of a vertex is multiplied by the color effect to find the color of the vertex for display.

$$RR = R \times Ltr$$

$$GG = G \times LTg$$

$$BB = B \times LTb$$

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.

$$\begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} = [Lij] \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

This matrix [Lij] is called the Local Light Matrix (LLM) in GTE.

Thus it can be seen that in the basic geometry library, there is no need to convert the normal line vector given 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 designated by the SetLightMatrix() function.

Further, if there are three light sources, then there are three light source colors so that (3) above is as follows:

$$\begin{bmatrix} LIr \\ LIg \\ LIb \end{bmatrix} = L1 \begin{bmatrix} L1r \\ L1g \\ L1b \end{bmatrix} + L2 \begin{bmatrix} L2r \\ L2g \\ L2b \end{bmatrix} + L3 \begin{bmatrix} L3r \\ L3g \\ L3b \end{bmatrix}$$

$$= \begin{bmatrix} L1r & L2r & L3r \\ L1g & L2g & L3g \\ L1b & L2b & L3b \end{bmatrix} \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix}$$

If this matrix,

$$[LRij] = \begin{bmatrix} L1r & L2r & L3r \\ L1g & L2g & L3g \\ L1b & L2b & L3b \end{bmatrix}$$

is [LRij], then (3) and (4) above for 3 light sources will be as follows

$$\begin{bmatrix} LTr \\ LTg \\ LTb \end{bmatrix} = [LRij] \begin{bmatrix} L1 \\ L2 \\ L3 \end{bmatrix} + \begin{bmatrix} BKr \\ BKg \\ BKb \end{bmatrix}$$

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 SetColorMatriix.

Also, ambient color is called Back Color (BK) and may be designated by the SetBackColor() function.

The above explained procedures (1) (2) (3) (4) and (5) may be summarized as follows based on up to 3 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

$$RR = R \times LTr$$

$$GG = G \times LTg$$

$$BB = B \times LTb$$

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

GTE has a register which maintains the GPU packet code. A function which outputs an RGB value from a light source calculation outputs 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. The GPU packet can not be properly generated if the GPU packet code register is not specified correctly.

When the GPU packet code register is set, the GPU packet code is automatically copied to the upper 8 bits of the input primary color vector of each light source calculation function. You should use the SetRGBcd() function when there is a function that has no input primary color vector.

Functions that copy the GPU packet code are as follows; those marked with * have no primary color input vector.

```
DpqColorLight
DpqColor
DpqColor3
Intpl                   *
NormalColor             *
NormalColor3            *
NormalColorDpq
NormalColorDpq3
NormalColorCol
```

```
NormalColorCol3
ColorDpq
ColorCol
RotColorDpq
RotColorDpq3
RotAverageNclipColorDpq3
RotAverageNclipColorCol3
RotColorMatDpq
ColorMatDpq
ColorMatCol
```

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 decided by sign of the Z component of the normal line screen coordinate system of the polygon.

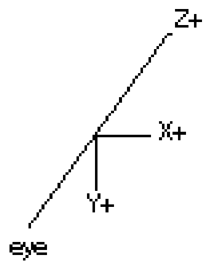
Normal line clipping is effective when there is a closed curved surface such as that of a sphere. This is also effective in reducing the so-called Z sorting problem.

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 2 sides of the polygon.

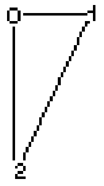
A function calculating the 2-dimensional vector product for normal line clipping, Normal Clip() is provided in the basic geometry library. NormalClip() calculates a value to distinguish between the front and back of triangles from the screen coordinates of 3 vertices. Front and back can be judged by 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 8–1: 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 8–2: Vertex Order

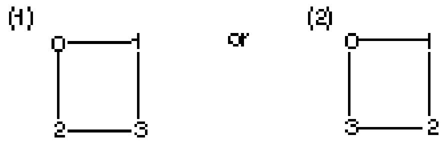


With the following performing the same calculation as NormalClip(), normal line clipping is performed,coordinate calculation is halted and an incorrect sx, sy value is returned. when the vector product is negative or 0. When using these functions, the order of the vertices of a polygon must be modeled so that seen from the front they will rotate clockwise.

```
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 8–3: 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 which makes objects that are at a distance appear hazy.

Depth cueing is accomplished by blending (through interpolation) the original polygon color together with the far color, as a function of the Z-value of the screen coordinate system. If the far color is white, distant objects will appear as if they were slightly obscured by fog. If the far color is black, distant objects will appear as if they were dark.

The method of depth cueing can be broadly divided into two categories:

(1) Method using vertex colors

Appropriate applications:

(a) Non-textured polygons

(b) Textured polygons in cases where one of the following conditions apply:

* If the far color is black or a dark color close to black.

* If the texture is a relatively bright color and if the texture is composed only of colors close to the far color and without any dark points. In this case, the object may not completely blend with the far color.

(2) Method using texture colors

Appropriate applications:

(a) General textured polygons. In particular, polygons that were not included in condition (1)-(b) described above for the method using vertex colors.

Implementation of depth cueing (common operations)

(1) Interpolation coefficients

GTE has a feature for performing efficient depth cueing through non-linear interpolation of the far color. To use this function, it is necessary to set the depth for depth cueing using the functions 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.

In many cases, the far color may be the same as the background color, which means that the rendering of the polygon will be unnecessary if p is 4096. For a given otz value, the otz2p function can be used to obtain roughly the same value of p as the interpolation coefficient generated by GTE. Conversely, the p2otz function can be used to determine otz from p. Please note that 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.

(2) Method for preparing interpolated data (using CLUT or texture)

Data can be prepared for different values of p beforehand or data can be generated using a specific value of p at runtime. DpqColor is a useful function for interpolating colors such as these. The far color should be set beforehand with the SetFarColor function before calling DpqColor.

Depth cueing using vertex colors

In the vertex color method, depth cueing is accomplished by interpolating the polygon vertex colors with the far color as follows.

First, depth must be set using the SetFog* function described above. Next, SetFarColor is used to set the far color. Then the RotTransPers functions are used to obtain the interpolation coefficient, p, for each vertex or each polygon. The value of p is passed to a function such as NormalColorDpq, which selects a vertex screen color that has been interpolated with the far color.

Besides the NormalColorDpq* functions, the DpqColor* functions, *ColorDpq functions and the Intpl function can also be used for this calculation.

Depth cueing using textures

When textures are used, depth cueing is accomplished by interpolating the texture color and the far color. The implementation method is different for each case.

What color to interpolate:

(1) Interpolation using CLUT colors
CLUT interpolation can be performed for textures that use a CLUT.

(2) 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 thetexture 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.

Timing for generating interpolation data

(1) Interpolated data generated beforehand (CLUT or texture)
First, CLUT or texture data is generated 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 reduces the time required for runtime calculation, but, from the perspective of memory utilization, the amount of data that can be stored is less and the resolution of the interpolation coefficient p cannot be set very high.

(2) Run-time generation of interpolated data (CLUT or texture)

In this method, CLUT or texture data is generated at runtime for a specific interpolation coefficient p. This method requires calculation time during execution, but the generation of data for each frame requires only the data for the original colors, so relatively little memory is used. Also, p can have a high resolution.

Changing the rendered texture

(1) 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.

(2) 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 performed easily by rendering using the Gs library.

The following three methods can be used to modify data in the frame buffer:

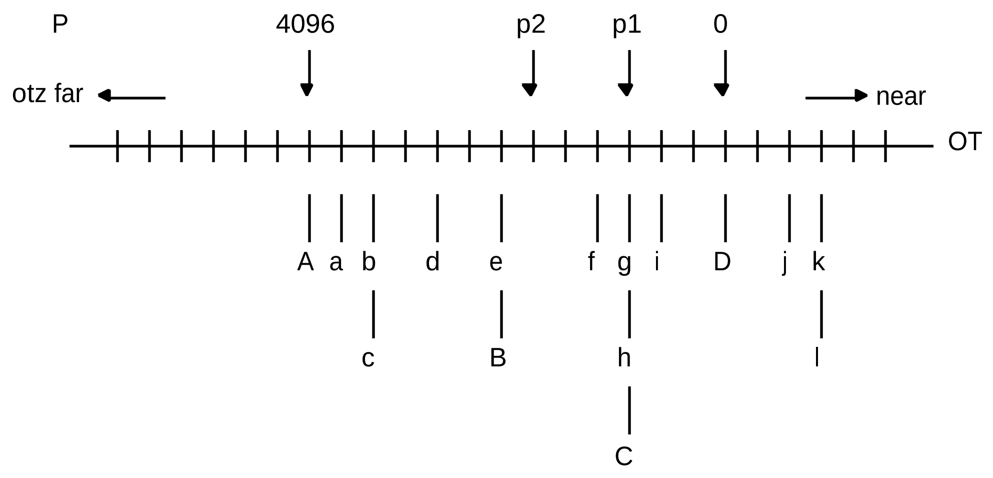
(a) Writing data using DR_LOAD primitives

In this method, data is set up in DR_LOAD primitives and is written to the frame buffer. Large amounts of data can be divided up into multiple DR_LOAD primitives and transferred. In these cases, it is more efficient in terms of speed 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.

The operations below should be performed if multiple occurrences of the interpolation coefficient, p, corresponding to depth, are to be used for a single CLUT or texture in a single frame. DR_LOAD primitives are entered into a 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 OTag using DrawOTag or the Gs library.

There are primitives for writing pixel data directly to a specific region in the frame buffer. For details, please refer to the sections on libgpu and SetDrawLoad().



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 certain value of p and a rendering range from p1 to p2, a DR_LOAD (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, and polygons f - h would be rendered with the data transferred using DR_LOAD B.

(b) Writing data using DR_MOVE primitives
In this method, data for different values of p are written in 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.

(c) Using LoadImage to transfer data from main memory
In this method, the LoadImage function is used to transfer data from main memory to the area to be used in the frame buffer. Note that using LoadImage too often will result in 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.

Back Color, Far Color, BG Color

The following depth-cueing terms are used in the PlayStation.

Back color, BK	Ambient color set by the function SetBackColor().
Far Color, FC	Far colorset by the function SetFarColor().
BG color	The color applied in the background.

When you wish to blend colors into the background with depth cueing, you match the far color and the background color. Please note that back color and BG color differ.

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)
	(RBK, GBK, BBK):	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 * v0
(2)    LC = BK + LCM * LLV
(3)    v2 = v1 * LC
```

Following the calculation of (a') you may also obtain LLV again with each item of LLV squared in the following manner

```
(1)    LLV = LLM * v0
```

```
(2)    LLV = LLV^2 = (L1^2, L2^2, L3^2)
(3)    LC = BK + LCM * LLV
(4)    v2 = v1 * LC
```

If this is done, the lighted portion onscreen 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 numerals are expressed by fixed point numbers. For example each component of the rotational matrix is a (1,3,12) format fixed point number.

(1,3,12) here refer to

Sign : 1 bit

Integer section: 3 bits

Fractional part: 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) 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.

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) ij]	(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 appended to the Rot...() coordinate calculation function family for performing clipping. The coordinate calculation functions are as follows:

```
RotTransPers()
RotTransPers3()
RotTrans()
RotTransPers4()
RotAverage3()
RotAverage4()
RotNclip()
RotNclip3()
RotNclip4()
RotAverageNclip3()
RotAverageNclip4()
RotColorDpq()
RotColorDpq3()
RotAverageNclipColorDpq3()
RotAverageNclipColorCol3()
RotColorMatDpq()
```

These 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 can not 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 8–2: 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)
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

NOTES: `h` is the distance between the viewpoint and the screen.

The following functions return 16-bit flags:

- RotTransPersN
- RotTransPers3N
- RotMeshH

The 16-bit flag bits are as follows:

Table 8–3: 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)

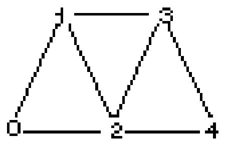
NOTES: `h` is the distance between the viewpoint and the screen.

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.

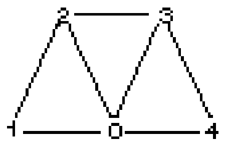
One is called strip mesh and the vertices are arranged in zig-zags as shown below:

Figure 8–5: Strip Mesh



The other is called round mesh and the vertices surround vertex 0 as shown below:

Figure 8–6: Round Mesh



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 previously mentioned 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.

The primitive defined in PMD is different to the drawing primitive handled by libgpu. Together with the processing of the perspective conversion by libgs it is also converted to the drawing primitive.

One primitive group has the following configuration

bit31(MSB)	bit0(LSB)
TYPE	NPACKET
Packet Data #0	
Packet Data #1	
Packet Data #2	
:	

NPACKET : Number of packets

TYPE: Packet type

Figure 6 PACKET Gp configuration

Table 4

bit No.	When 0	When 1
16	Triangle	Quadrangle
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) bit20,21 have no effect on the Packet Data configuration.

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. Because 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 8–4: 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 9: Extended Graphics Library

The Extended Graphics library (libgs) uses the basic graphics library and the basic geometry library to construct a 3-dimensional graphics system. The data handled here is in units larger than object data, background data, and the like.

Table of Contents

Overview

Libgs is an extended graphics library which can integrate the 2D (sprite/BG (background) etc.) and 3D (polygons, etc.) graphics systems structured in libgpu and libgte

Multiple objects, scale information and texture address information are stored in TMD format data in optimum condition for decoding by libgs.

Image resolution, color numbers and color look-up table information are stored in TIM format data, together with pixel data.

As both TMD format and TIM format are standard formats for PlayStation authoring tools, data created by the authoring tools can be used as is by libgs.

Since libgs processes data by object units (a group of polygons), in contrast with the libgpu and libgte libraries which process polygon-level data, 3D programs can be prototyped easily.

Various special effects can be created using the attributes added to the object. However, since the input format handled by libgs is restricted as compared to the program optimized by libgpu/libgte, overhead exists.

As a result in actual game production, part must be created at the libgpu/libgte level. However, as libgs is open architecture and was developed using low-level libraries it can be extended by adding the user’s module units.

Library File

The file name of the extended graphics library is `libgs.lib`. Every program calling services must link with this library.

Header File

The expanded graphics header file is `libgs.h`. Each type of data structure and macro used by libgs is defined in this file. Every program calling services must link with this header file.

Table 9–1

Content	Filename
Library	`libgs.lib`
Header	`libgs.h`

Libgs functions

Following are functions which libgs can perform:

-Hierarchical coordinate system

-Light Source calculation (3 light sources, depth cueing , ambient)

-Automatic division of object/semi-transparent processing

-Viewpoint control

-Z-sort processing

-OT initialization hierarchic compression

-Frame double buffer

-Automatic adjustment of aspect ratio

-2D clipping offset processing

-Sprite/BG/Line

-libgpu/libgte coexistence

Coordinate Systems

GsCOORDINATE2 is the 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 is composed of parameters to describe the coordinate system and a work area for speeding-up coordinate calculation. The parameters which describe the coordinate system are MATRIX type. The MATRIX parameters are parameters for transformation from the parent coordinate system of that coordinate system to that coordinate system. Also, for the size of the coordinate system space, X, Y and Z are all 32 bits.

Coordinate System Initialization

The function GsInitCoordinate2() is used for coordinate system initialization.

However, GsInitCoordinate2() only initializes each parameter of GsCOORDINATE2. Therefore, it may be initialized by other methods (methods of substituting values in members). The parameters become effective from the moment of rewriting, and can be used for later calculations.

Order of Rotation/Parallel Shift

Rotation and parallel shift of the parameters set in the coordinate system are executed in the order Rotation_ Parallel Shift.

Also, the order of rotation when the rotation matrix is created by the function RotMatrix(), and this is set in GsCOORDINATE2, is [Z → Y → X].

Clearing Flags

When requesting local world matrix from the hierarchical coordinate system, speeding-up has been designed in the form of setting 1 at the flags of the GsCOORDINATE2 members in the already calculated coordinate system and storing the results in member workm in the same way.

However, even if the parameters of GsCOORDINATE2 have been rewritten, recalculation cannot be performed unless the flags are 0-cleared. This means that the contents of workm have already been used. So, when parameters are rewritten, always remember to 0-clear the flags of the rewritten GsCOORDINATE2.

If parent coordinates are modified, this is transmitted to the child coordinates, so there is no need to clear the child coordinate flag.

Examples of Coordinate System Setting

Examples of parallel shift and rotation are presented below.

Example 1: Parallel Shift

GsCOORDINATE2 sample_coord;   /*coordinate system which sets parallel shift*/

```
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 which sets rotation*/
       SVECTOR rot; /*rotation angle set (x,y,z)*/
       MATRIX tmp1; /*rotation matrix requested*/
       RotMatrix(&rot.&tmp1);     /*when RotMatrix() is used to transform
from          rotation vector to rotation matrix, the order         of
rotation is zyx*/
       sample_coord.coord=tmp1
```

Objects

The three types of 3D object handlers in LIBGS are GsDOBJ2, GsDOBJ3, and GsDOBJ5. Programmers use this handler to manipulate objects. In this section, we explain the basic object handler GsDOBJ2. (

Use GsDOBJ2 member coord2 and attribute to manipulate objects.

coord2 is a pointer to the coordinate system GsCOORDINATE2. The location of an object may be controlled by setting the parameters of the GsCOORDINATE2 to which this points.

attribute is for setting object attributes. There are a variety of attributes, from general attributes such as display/non-display to special effects such as shifting of the light source calculation method.

Object Initialization

To handle an object with GsDOBJ2, the read-in TMD data and the handler must be linked. The GsLinkObject4() function is used to do this.

GsLinkObject4() sets which object of the TMD data is to be linked with GsDOBJ2.

Object Location Control

To control the location of an object, set the GsCOORDINATE2 parameters to those specified by the GsDOBJ2 member coord2.

For example, in the case of the GsCOORDINATE2 super being WORLD, its coordinate system is equivalent to the world coordinate system. At this time, the location parameters set become the world coordinate system location. The rotation parameter takes the origin of the world coordinate system as its center.

For example when object one points to coordinate one or object 2 points to coordinate 2, in order to make object 1 the parent of object 2, the super of coordinate 2 is set as coordinate 1.

Object Movement (Hierarchical Structurization)

Different objects may be linked by defining a hierarchy in the coordinate system in which GsDOBJ2 member coord2 is specified.

When defined in this way, the movement of object 2 links with object 1. The movement of object 1 does not link with object 2.

Location is set at coordinate 1 when moving object 1 and object 2. When moving only object 2, it is set at coordinate 2.

Table 9–2: Hierarchical Structurization

GSDOBJ2		COORDINATE2
Object1	¾¾¾→	Coordinate1
		↑
Object2	¾¾¾→	Coordinate2

Object Attribute Control

Operating the GsDOBJ2 attribute member bit has a variety of effects objects.

Material Attenuation

This sets the relationship between normal vector inclination and brightness attenuation when performing light source calculations. This parameter is used to change the look of an object.

Values that may be specified are 0_3, and the maximum attenuation is 3. As the attenuation value becomes higher, the time taken for calculation becomes longer.

This parameter may be ignored in cases when material lighting has no effect in light source calculation.

Material attenuation is not supported in the current version.

Lighting Mode

This determines the method of light source calculation.

The function GsSetLightMode() sets the default method of calculation, so this bit should be set only when you wish to control the light source calculation method object by object.

Light Source Calculation Off

This is the bit for forcibly cancelling light source calculation.

If this bit is set, processing is speeded up because it can be cancelled, even for TMD light source calculations.

Near Clipping

If this bit is set, in cases where the polygon end point is very close to the viewpoint (distance between viewpoint and polygon < (distance between viewpoint and screen) /2), a polygon that has overflowed during perspective transformation will not be simply clipped, but can be forcibly displayed, even if its shape is distorted.

Back Clipping

A polygon has a front and back determined by the order of its vertices. In the case of a convex object, it is not necessary to display the back face, so a back-facing polygon will be clipped. However, if this bit is set, a back-facing polygon can be displayed.

The current version does not support back clipping.

Semitransparency Rate

In addition to the normal semitransparency mode, there are three other semitransparency modes: 100% addition and substraction, and 25% addition. These modes are controlled.

Table 9–3: Semitransparency Rates

Value	Background	Primitive
0	0.5	0.5
1	1.0	1.0
2	1.0	-1.0
3	1.0	0.25

In order to render a texture-mapped polygon semitransparent, you must set the most significant bit (STP bit) of the texture color field (the CLUT field when in texture pattern or index color).

Also, semitransparent processing is possible in pixel units by the STP bit.

Display Control

This controls whether an object will be displayed or not displayed.

When it is not displayed it is excluded from calculation, so the load is lighter.

Automatic Division

This operation separates an object into its component polygons at the time of execution. Select the number of divisions from among 2x2, 4x4, 8x8,16x16, 32x32 or 64x64.

You can use this operation to eliminate the problems accompanying a perspective transformation, such as texture distortion and Near clipping. You must take care that memory use and processing speed are not adversely impacted as the number of divisions increase.

GsSortObject4() and GsSortObject5() are functions that create packets capable of automatic division. When using automatic division, you must pass the scratch pad address, used as a working argument, in the last argument of the packet creation function.

Viewpoint

In 3D graphics, the image projected in a window is a projection on a screen set in front of the viewing point, so you must set the viewing point and the screen in order to project an image in a window.

Viewpoint Setting

Viewpoint setting is executed by substituting values in the GsRVIEW or GsVIEW2 structure members and calling the GsSetRefView2() and GsSetView2() functions.

The difference between GsRVIEW2 and GsVIEW2 is in the viewpoint setting method.

GsRVIEW2 sets viewpoint by setting the coordinates of the viewpoint and a reference point. GsVIEW2 sets 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

This sets the screen in front of the viewpoint.

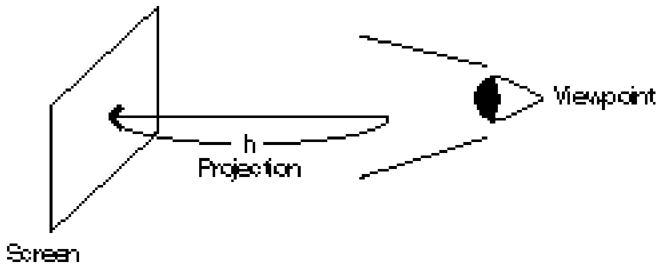
The distance between the viewpoint and the screen is called projection (h). Projection is set by the GsSetProjection() function.

The vertical and horizontal of the screen are equal to the resolution of the window. For example, if the resolution is 640/480, the horizontal of the screen is 640 and the vertical is 480.

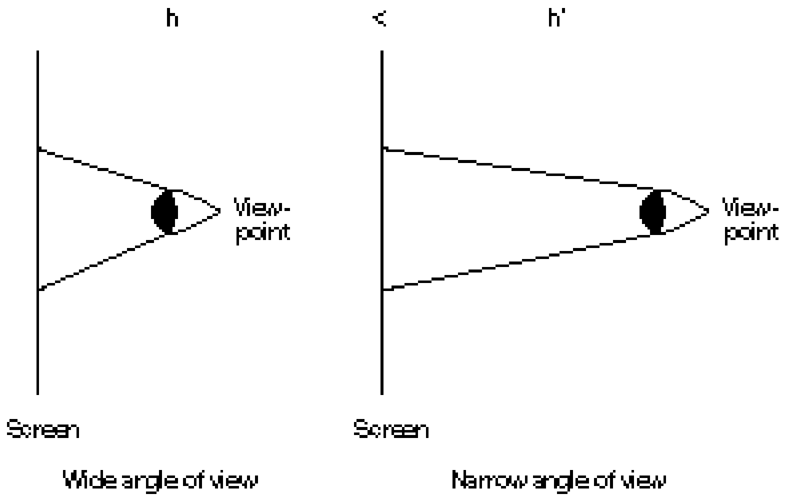
When the window resolution is not normal dot, that is to say when the window resolution aspect ratio is not 4 to 3, the vertical is adjusted. For example, in the case of 640/240 dot, the vertical of an object is displayed by reducing it by 1/2. In appearance, this is the same as a normal dot aspect ratio.

To use the libgs 3-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 9–1: 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 emphasised.



Light Sources

A maximum of 3 parallel light sources may be set with libgs. A parallel light source is a light source in which the brightness 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.

Light Source Setting

The light source is set in the system by setting the GsF_Light structure parameter and executing the GsSetFlatLight() function.

Also, since it is possible to set up to 3 parallel light sources, the GsSetFlatLight() argument ID is set to 0-2.

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.

Ambient light setting is performed as follows. . Ambient light may be set for every RGB. 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);

The reason there are 3 parameters is because RGB can independently set the ambient light. In general the image becomes warm when the ambient light is increased and cool when it is 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.

With libgs, depth cueing may be executed for all polygons that are not textured.

In order to execute depth cueing, call GsSetLightMode(1) or GsSetLightMode(3) and specify the depth cueing light source calculation.

Then set GsFOGPARM rfc,gfc and bfc as background colors and set to libgs by GsSetFogParam().

When background colors are made whitish, this is the FOG effect.

Also depth-cueing is possible for texture-mapped polygons with this method only when the color is black. This is effective when you are 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 to each other.

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.

Execute GsSetLightMode(2) or GsSetLightMode(3) to execute material lighting.

Control of attenuation is executed by setting the material attenuation bit of the 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)

Z-sort is adopted as a method of hidden-surface removal for the PlayStation, and, in order to speed up the performance of Z-sort, the concept of an ordering table (OT) has been introduced. Hereafter, Z refers to a coordinate value on the vertical direction axis in the window; in other words, the distance between the screen and the polygon.

An ordering table is a kind of Z ruler applied to memory. Each graduation of the ruler may hold any number of polygons.

Sorting is executed according to the Z value of a polygon by placing the polygon at the graduation equal to that 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 it.

GsOT

libgs OT are handled by a structure called GsOT. A pointer (member org) to an actual OT, and parameters that indicate the attributes of that OT, are stored in this structure.

In libgs, 14 stages, from the 1st power of 2 to the 14th power of 2, may be set on member length as Z graduation resolutions.

GsOT_TAG

GsOT_TAG indicates 1 graduation of the ruler, and an actual OT is defined as an array of GsOT_TAG.

For example, if its length is 4, an actual OT is an array of the 4th power of 2, that is to say 16 OT_TAG.

OT Initialization

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 OT is initialized, the polygons are unlinked, and a re-sort is possible. That is to say, it is always necessary to initialize when executing a sort.

Multiple OTs

libgs allows multiple OTs. Multiple OTs may be sorted by the GsSortOt() function. At this time, the value referred to as the representative value Z of an OT is the GsOT member point.

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.

OT Compression

Sort speed will be increased by using OT. However, OT consumes a considerable amount of memory.

There is a method of reducing OT resolution by shortening length in order to restrict the amount of memory consumption. However, sort resolution is also reduced, and a polygon flicker phenomenon (Z-sort problem) will readily occur due to error in Z relationships.

Therefore, there is a method of using an offset as a method of reducing OT memory consumption without reducing resolution. This may be used when it is known that the Z values of the polygons sorted are greater than a certain value. If this value is delivered to GsClearOt() as the offset, memory consumption will be reduced, since OT does not store the part up to offset in memory.

Z-Sort Problem

A problem will always occur when using Z-sort to perform hidden-surface removal. This problem is that the polygons will flicker due to errors in priority order.

This is caused by the fact that the Zs referenced when determining priority order are referenced 1 per polygon. With libgs, the polygon center of gravity Z is referred to when sorting. However, the phenomenon is liable to occur with a polygon of particularly long depth since it is sorted with only one average Z value, despite the Z value differing greatly across the polygon.

There is a method of dividing polygons into small parts as a method of resolving the Z-sort problem. However, with this method, the number of polygons is increased.

As another countermeasure, there is a 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 that OT is being drawn. For this reason, you must prepare 2 OTs when drawing with a background, and use the OT not being used for drawing for sorting. This is the OT double buffer.

Frame Double Buffer

The PlayStation has a 2-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 base line 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 GsDefDispBuffer().

Switching of this defined double buffer is done by GsSwapDispBuff().

GsGetActiveBuff() can be used to get whichever double buffer is currently being drawn.

Double buffer expression

Double buffer may be achieved by altering the display location of the frame buffer. The upper left point of the display area (starting point) does not necessarily have to be in the uper left point of the frame buffer.

Drawing that goes to a frame buffer must have an offset attached. You may choose from two methods of offsetting with libgs.

One method is to put the offset at the libgte level. If you choose this method, the double buffer offset is added at the stage where the packet calculation is being made. The other method is to place the offset at the libgpu level. If you choose this method, the offset is added at the stage where a frame buffer not attached to the packet is drawn. The 3rd argument of GsInitGraph is used to choose the offset method.

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

During interlace, a double resolution of 480 can be specified as the vertical resolution. In this case, double-buffering is automatically executed between even number addresses and odd number addresses of scanning lines. So, it is necessary to designate the same buffers as the GsDefDispBuff() arguments.

When vertical resolution is specified as 240 during interlace, it is necessary to set different buffers, as you do in non-interlace.

Clipping

In libgs, clipping is divided into 2-dimensional clipping and 3-dimensional clipping.

2-dimensional clipping is clipping after transforming the screen coordinate system. 3-dimensional clipping is clipping according to the distance from the viewpoint.

2-Dimensional Clipping

The rendering chip has a function which designates any rectangle on 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.

3-Dimensional Clipping

In libgs, 3-dimensional clipping is performed at the application level. Accordingly, library level 3-dimensional clipping is not supported for other than default values. The 3-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 CLIPWhen 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 occurs when polygons approach the viewpoint without limitation, such as in the road surface of a racing game, and the polygons themselves become extremely large due to their nearness to the viewpoint. When clipped by polygon units, large holes appear in the road surface close to the viewpoint, and this makes for extremely difficult viewing. As a solution to this problem libgs supports the automatic division of polygons.

When an object is approaching near clipping can be performed smoothly by the setting of the authomatic division attribute. However since the load from automatic division is heavy, please restrict its use to 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:

(1) Preset packet buffer

(2) 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 the automatic division is set to ON in the GsDOBJ5 attribute, the packet is created in the run-time packet buffer, it 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().

Because the buffer size is displayed in one frame at the same time, it 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.

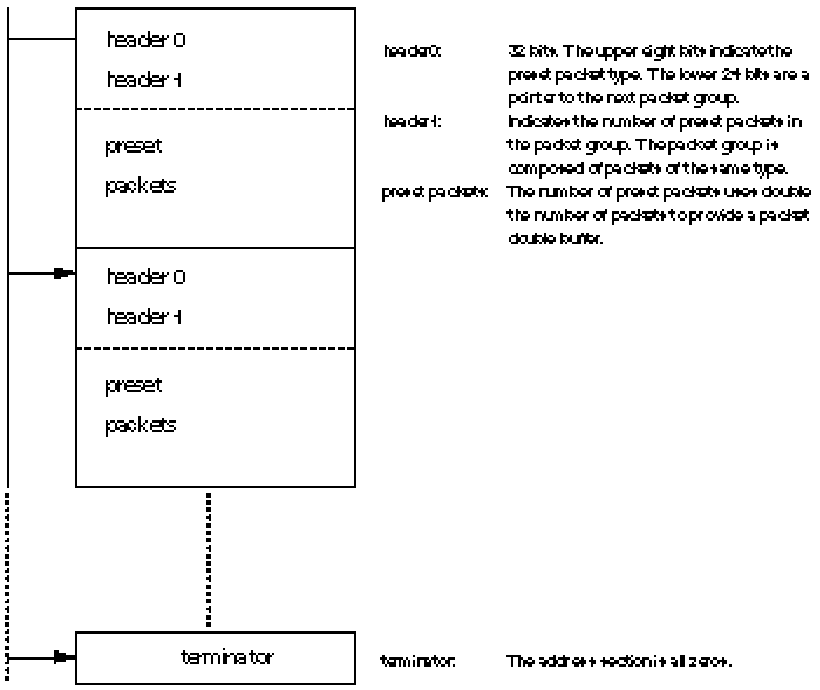
PMD format is an exclusivepreset packet modeling format;GsSortObject5() is the packet creation function for preset package. GsSortObject() is used in creating a preset package.

PMD format incorporates both modeling data and preset packets.

The packet is a collection of structures (primitives) such as libgpu.h POLY_FT4. The primitve 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 9–3: 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 the explanation of authoring tools for details on using this command.

If either of the GsSortObject4() or GsSortObject5() packet creation functions use the sorted TMD data, the speed will be enhanced.

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 is combination of 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 the explanation of authoring tools 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 9–4: 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 SortObject2 and less efficient than SortObject5

Table 9–5: Packet Creation Function Comparison Chart 2

	Presort	Preset	Preshad	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

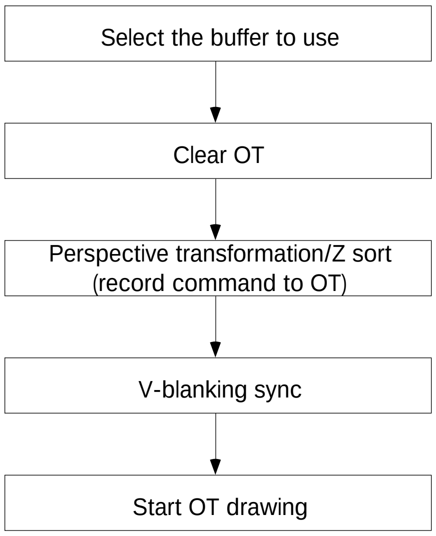
Call the GsDrawOt() function to begin drawing. The drawing area is swapped each time GsSwapDispBuff() is called. Drawing occurs in the background so sufficient time must be allowed to complete the operation.

In drawing process images from the previous two frames that remain in the drawing area are cleared. Call GsSortClear() to register the packet to the OT before clearing the screen. The cleared screen color may be specified 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 9–4: 3-Dimensional Processing Flowchart



Scratch Pad Usage Volume

In the Gs library the Scratch Pad address can be passed by argument by GsSortObject4, GsSortObject4J, GsSortObject5 and GsSortObject5J. The scratch pad is used in polygon division to improve speed. The Gs library explains how much and in what condition the scratch pad area is used.

Scratch Pad Consumption Status

The scratch pad consumption condition uses the following functions and attribute:

Table 5 State of scratch pad consumption

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 6 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

The Gs library supports mip-map. Mip-map is the switching of the texture of a textured rectangular polygon included in the TMD data according to the size of the polygon. By using this it becomes easier to hit the texture cache and the drawing time is shortened.

Usage Method

The GsSortObject4J() lower functions which support mip-map are as follows:

Table 7 mip-map lower function group

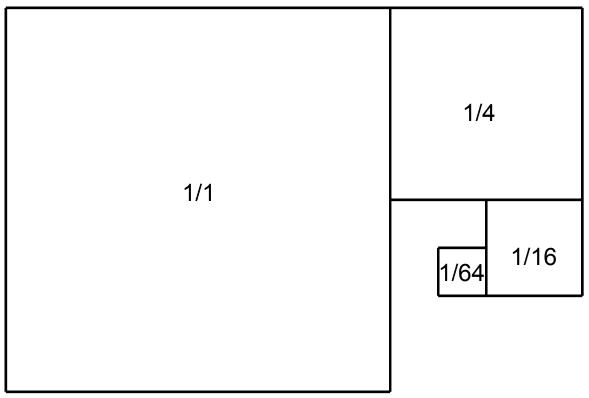
GsTMDfastTF4LM	Flat textured quadrangle	(light source calculation)
GsTMDfastTF4LFGM	Flat textured quadrangle	(light source calculation+FOG)
GsTMDfastTF4NLM	Flat textured quadrangle	(no light source calculation)
GsTMDfastTNF4M	Flat textured quadrangle	(no light source calculation)
GsTMDfastTG4LM	Gouraud textured quadrangle	(light source calculation)
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-map please locate the texture as follows:

Fig.: Texture location

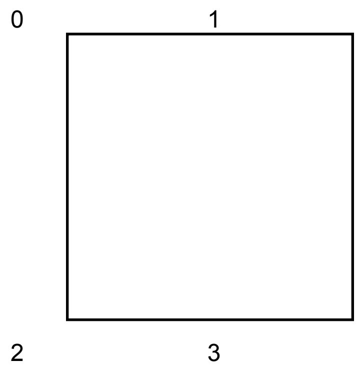


The texture size is in four stages: 1, 1/4. 1/16 and 1/64. The texture being used can be calculated by using the external product value. The above four textures must be within the same tpage.

Polygon Vertex

The polygon vertex order must be as follows:

Chart: Polygon vertex order



Chapter 10: CD/Streaming Library

The CD/Streaming Library is composed of CD-ROM and CD Streaming libraries.

The CD-ROM library provides a service for controlling the PlayStation built-in CD-ROM drive.

The Streaming Library is a group of functions for continuous reading of realtime data such as movies or sounds or vertex data stored on high-capacity media.

This chapter describes the CD-ROM system and overviews the related libraries.

Table of Contents

CD-ROM Library Overview

The CD-ROM library provides a service for control of the PlayStation CD-ROM. For an overview of the Streaming library, see the Streaming Library Overview, page 10-34.

The following are the services that the CD-ROM library provides. An overview of each service will be given later.

- CD-ROM controll
- CD sound control
- Other

Library Files

The name of the CD-ROM header file is `libcd.lib`. Every program calling CD-ROM services must link with this.

Header Files

The header file of the CD-ROM library is `libcd.h`. All of the CD-ROM related data structures and macros are defined by this file. Every program calling CD-ROM services must include this.

Table 10–1

Content	Filename
Library	libcd.lib
Header	libcd.h

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 10–2: 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.

Transfer Rate

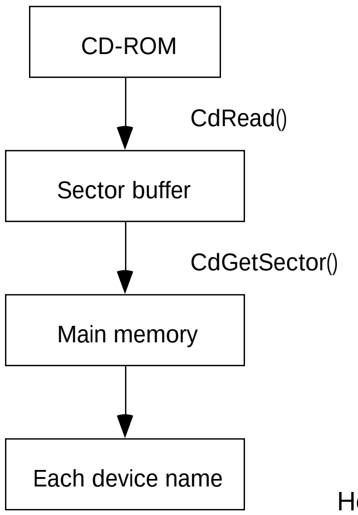
A CD-ROM can rotate the disk at either normal speed or double speed. Normal speed has the same RPMs as an ordinary CD player, and double speed is twice as fast. The faster the disk rotation the faster the disk transfer speed.

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 10–1: Process of CD-ROM Transfer



However, this is an examplle 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) --> ATV0 --> SPU (L)
CD (R) --> ATV1 --> SPU (R)
```

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(com, param, result)
unsigned char  com;     /* command code */
unsigned char  *param; /* command argument set address */
```

unsigned 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;
unsigned 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 folllows:

Table 10–3

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
CdlBackword	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)
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 (param)

A primitive command needs a list of arguments called parameters, as shown below. Command arguments are as follows:

Table 10–4: 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	unsigned char *	Set basic mode
CdlGetTD	unsigned 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 {
unsigned char minute;      /* sector location(min)*/
unsigned char second;      /* sector location(sec)*/
unsigned char sector;      /* sector location(sector)*/
unsigned 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 {
unsigned char file; /* play file ID */
unsigned char chan; /* play channel ID */
unsigned short pad;
} CdlFILTER;
```

Command return values (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 10–5: 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							
CdlBackword	Status							
CdlReadN	Status							
CdlStanby	Status							
CdlStop	Status							
CdlPause	Status							
CdlMute	Status							
CdlDemute	Status							
CdlSetfilter	Status							
CdlSetmode	Status							
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 10–6: 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	0x01	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 preceeding 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 preceeding 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 10–7: 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/CdlBackword

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 the CdlGetlocL command.

Table 10–8: Mode Settings of CdlSetmode

Symbol	Code	Details		
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

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 10–9: CdlGetlockL Parameters

Result byte no.	Details
0	Minute (BCD)
1	Second (BCD)
2	Sector (BCD)
3	Mode (see CdlSetMode command)
4	File number (see CdlSetFilter command)
5	Channel number (see CdlSetFilter command)

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 10–10

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.

Result	Contents
0	Status
1	Inital 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.

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 commands (asynchronous commands).

On the other hand, those which wait for the completion of processing before returning are called blocking commands (synchronous commands).

Non-blocking commands continue processing in the background even after a CdControl() function returns. During this time the host system can continue processing in parallel.

The actual completion of non-blocking command processing uses the CdSync() function or the callback function which will be described later.

The return value (result) of the function 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.

The following example shows a function which blocks all commands until completion.

```
CdControlB(u_char com, u_char *param, u_char *result)
/* command issue */
if (CdControl(com. param, result) == 0)
return(0);
/* blocks until command completion */
if (CdSync(0, result) == CdlComplete)
return(1);
else
```

```
return(0);
}
```

Command Execution Status

Primitive commands have the following processing status.

Table 10–11: 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(mode, result)
unsigned char mode;        /* mode 0: blocking;  1:non-blocking  */
unsigned 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 10–12: CdSync() Mode Argumentvalues 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, (unsigned char *)pos, 0);
CdSync(0, result);
```

Example b

```
CdControl(CdlSeekL, (unsigned 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 CdlNoIntr. Therefore the following codes produce the same result.

Example a

```
CdControl(CdlSeekP, (unsigned char *)pos, 0);
CdControl(CdlPlay, 0, result);
```

Example a

```
CdControl(CdlSeekP, (unsigned 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, (unsigned char *)pos, 0);
CdControl(CdlPlay, 0, result);

/* Not Blocked During Seek */
CdControl(CdlPlay, (unsigned 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.

```
callback(unsigned char intr, unsigned char *result)
unsigned char intr;      /* execution status at that point in time  */
unsigned 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, (unsigned char *)&pos, 0);
}

/* the following function is called when the command ends */
void callback(unsignedchar intr, unsignedchar *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 as 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 10–13: 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(mode, result)
unsigned char mode;     /* Mode 0: blocking; 1:non-blocking  */
unsigned char *result;  /* Most recent command return value */
```

The CdReady() function returns the following sector buffer status.

Table 10–14: 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 retransferred 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(loc, buf, nsec)
CdlLOC *loc;        /* target position */
unsigned long *buf; /* read buffer  */
int n;              /* number of sectors */
{
        unsigned char  param[4];
        /* set double speed mode */
        param[0] = CdlModeSpeed;
        CdControl(CdlSetmode, 0, 0);
        /* issue retry command */
        CdControl(CdlReadN, unsigned char *loc, 0);
        /* transfer to main memory as soon data is ready */
        while (n--) {
                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(sectors, buf, mode)
int sectors;            /* number of sectors read */
unsigned long *buf;     /* main memory address */
unsigned 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.

```
CdReadB(loc, buf, nsector)
CdlLOC *loc;        /* target position */
unsigned long *buf; /* memory address */
int nsector;        /* number of sectors read */
{
       int cnt;
       unsigned char param[4];
       /* set double speed mode */
       param[0] = CdlModeSpeed;
       CdControl(CdlSetmode, 0, 0);
       /* set target position */
       CdControl(CdlSetloc, (unsigned 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);
}
```

Streaming

The process of continuously reading data from CD-ROM and transferring it to main memory is called streaming. 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.

Refer to other sections where streaming has been described in detail as the St*() function.

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.

Table 10–15: 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.

Table 10–16: Example Multichannel Interleave

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	.	.
A 0	A 1	A 2	A 3	A 4	A 5	A 6	A 7	A 0	A 1	A 2	A 3	A 4	A 5	A 6	A 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 multichannel 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 through the 9660 file system. Addressing via the file system 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 through the 9660 file system. 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, (unsigned char *)&fp.pos, 0);
CdRead((fp.size+2047)/2048, sectbuf, CdlModeSpee);

CdSearchFile () returns the following CdlFILE structure members.

typedef struct {
CdlLOC pos;          /* file position */
unsigned 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 10–17: 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.

Event Service

At initialization, a default callback function is registered for each callback. These distribute the events shown below.

Table 10–18

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 10–19

Called function	Sync detect	Callback	Details
CdControl	CdSync	CdSyncCallback	Issue command
———	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 beany 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
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 were hung up.

Generally, FF and FR commands cannot be performed when data is being read. If these commands are used in an environnent 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.

(1) Record dummy data on the outer three minutes (the last three minutes of data). Do not use the dummy data.

(2) 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.

(3) 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).

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.

(1) Skipped sectors

In double-speed mode, data is read from a CD-ROM at 300 sectors/sec. Therefore, one sector will be skipped if the host system does not finish processing the read operation for the previous sector within 1/300 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 be used beforehand to set CdlModeSize1 (the mode for reading the sector header) as well.

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;
......
```

(2) 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.

(3) Deleting callbacks

When a callback completes it should be cleared quickly.

```
CdReadyCallback(callback);
/* Operation corresponding to CdRead() */
CdReadyCallback(0);
```

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.

(4) 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.

(5) Other

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.

(1) 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 );
```

(2) Detecting a swapped CD

To see whether the CD has been replaced, the following two tests should be performed: (A) determine whether the cover has been opened; and (B) determine the spindle rotation. Either test can be performed using the CdlNop command.

```
CdControlB( CdlNop, 0, result ); /* char result[ 8 ]; */
```

(A) 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.

(B) Use the CdlNop command and wait for bit 1 of result [0] (0x02) to change to 1.

(3) 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 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 ]; */
```

(4) Checking for PlayStation disc

A logical access command (CdlReadS/N, CdlSeekL, etc.) 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.

(5) 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 problemswhich 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 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)
```

libcd message reference

The error messages from libcd are described below. The levels here correspond to the modes in CdSetDebug().

Table 20 Error levels

level	output conditions
0	always output
1	output if debug level is 1
2	output if debug level is 2

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])
Level:	0
Parameter:	[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
Parameter:	[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
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 not exist 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.

Library Files

The streaming library file name is “libcd.lib”. All programs that call services must be linked to this file.

Header Files

The streaming library header file is “libcd.h”. All types of data structures and macros are defined in this file. The programs called by each service must include this file.

Table 10–1

Contents	File Name
Library	libcd.lib
Header	libcd.h

Streaming

Streaming performs realtime processing of data created for playing in realtime, such as playing video or 3D vertex animation.

Processing which continuously reads CD-ROM sectors implements processing that makes full use of the CD-ROM transfer rate.

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 bv head access in this method, XA audio and streaming cannot be used at the same time. Refer to StGetBackLoc andd 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.

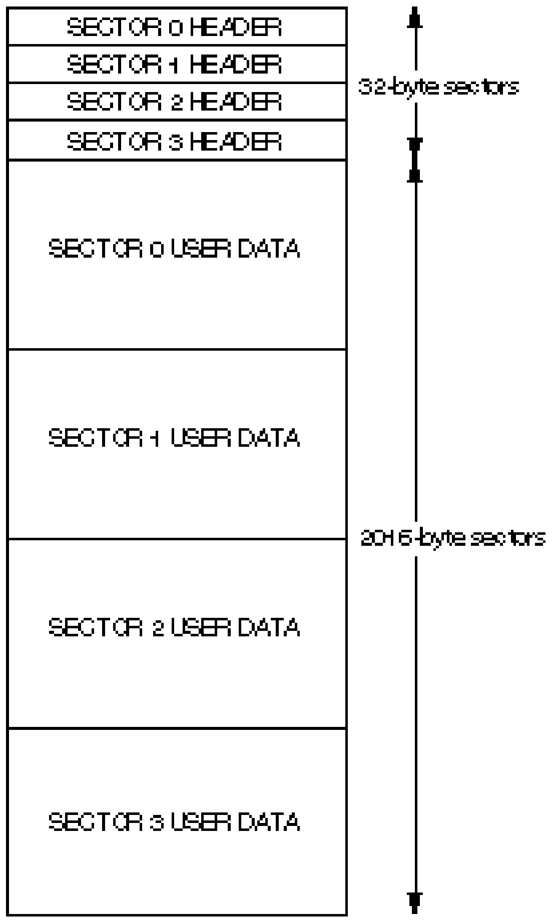
Ring Buffer Format

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.

Figure 10–1: Ring Buffer Size 4 Example



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. This function is called automatically by interrupts from the CD-ROM , and usually does not need to be executed.

However, this function 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.

Interrupt Functions Used

The streaming library uses the following two interrupt functions.

Table 10–2

Function name	Details
CdDataCallback	Sector data transfer completion callback
CdReadyCallback	Sector data ready callback

CD-ROM Drive Control

See the service provided by libcd for CD-ROM drive control. For details, see the chapter for the CD-ROM library.

Chapter 11: Controller/ Peripherals Library

The Controller/Peripherals libaries are primarily comprised of the following libraries:

- Controller library (libetc)
- Gun library (libgun)
- Multi-tap libary (libtap)

The Controller library (libetc) controls callbacks for performing low-level interrupt processing and controller-related functions. At present, functions relating to the controller are also included in this chapter.

The Gun library (libgun) is provided to detect the position of the gun connected to the PlayStation and pointed towards the television screen.

The Multi-tap library (libtap) describes the communication services between multiple controllers and memory cards and the PlayStation provided by connecting a Multi-tap to the PlayStation.

Some ancillary Controller/Peripheral functions are also provided in the Kernel library (libapi)

For a complete description of all related Controller/Peripherals structures and functions, please refer to *the Run-time Library Reference*.

Table of Contents

Controller Library Overview

The Controller 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 File

The file name of the Controller library file is `libetc.lib`. All programs that call services must be linked to this file.

Header File

The ETC library header file is `libetc.h`. All types of data structures and macros are defined in this file. The programs called by each service must include this file.

Table 11–1

Content	Filename
Library	libetc.lib
Header	libetc.h

Callbacks

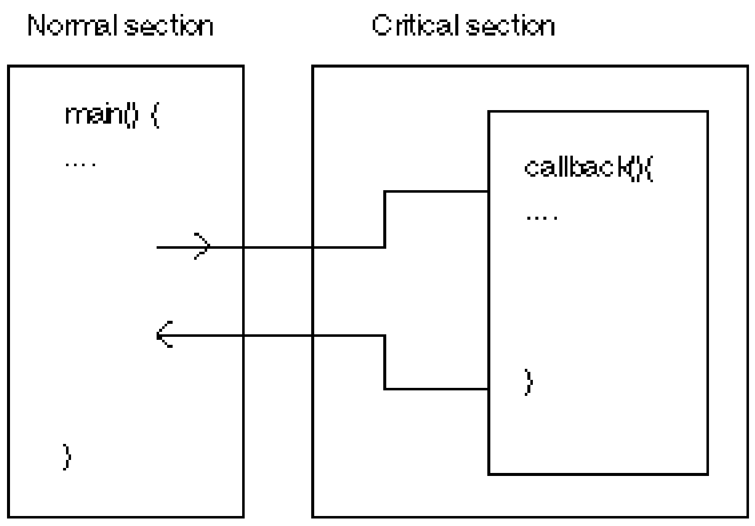
Many of the functions which perform such asynchronous processes as graphics drawing, transferring to the sound buffer, and loading from the CD-ROM, may execute in parallel in the background. These functions are called non-blocking functions. Processing the actual termination of the non-blocking function may be defined beforehand in the callback function.

The callback function is called when the corresponding non-blocking function completes. 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 function returns, the program returns to the point where the callback began, and normal processing resumes.

A dedicated local stack is used inside the callback function so that control can return to the original state after the callback function returns. All interrupts are prohibited within the callback function. (Critical section)

Figure 11–1: Callback Context



Callback types

Callback currently supports the following items. Please refer to the documentation for each library for details.

Callback types

Function Name	Corresponding non-block functions
VSyncCalllback	
DrawSyncCallback	DrawOTag()/Load()/StoreImage()
DecDCTinCallback	DecDCTin
DecDCToutCallback	DecDCTout
CdSyncCallback	CdConroll
CdReadyCallback	CdConroll
CdDataCallback	

Callback Initialization

When using callbacks, it is necessary to initialize creation of the local stack in advance. That is, programs must always call the initialization function ResetCallback() before using callback. However, as the initialization functions of most libraries already include a call to ResetCallback(), it is not necessary for an application to call this function explicitly.

A list of initialization functions that currently call ResetCallback() automatically is listed below.

All callback pointers are initialized to NULL(0) immediately following the calling of ResetCallback().

Table 11–2: Initialization Function that Calls 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

Callback Termination

The callback facility 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

Function pointers created by a callback are called callback pointers. The callback interface is a low-level interface, so a single callback is limited to recording one pointer at a time. Setting a new callback discards the previous pointer. For this reason, the mechanism for creating multiple pointers in a single callback is the responsibility of the application program.

```
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);
}
```

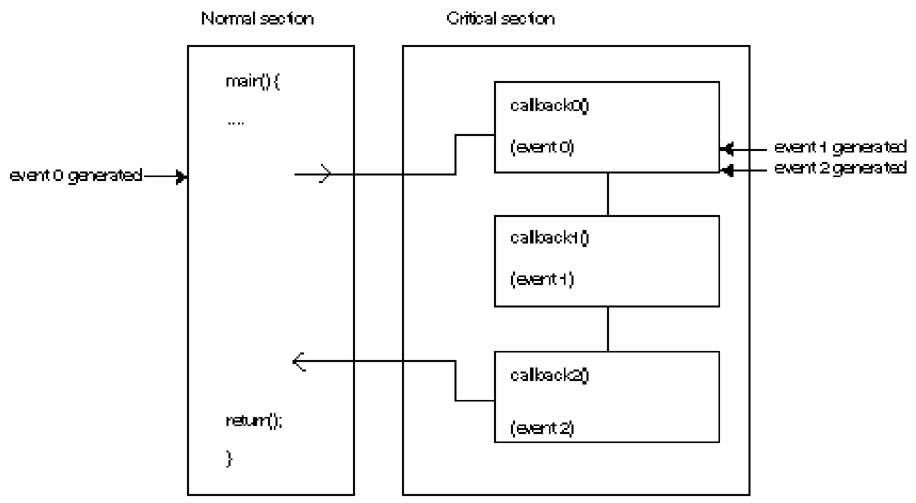
Multiple Callbacks

The callback context uses one local stack referenced by all current callbacks. This means that 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.

Example: Multiple Callback Processing

Figure 11–2



Child Processes and Callbacks

Part of the callback code and data are linked in memory with the application. The callback code and data use identical copies of a library, not shared with companion processes, so a shared library cannot be implemented. Whenever a parent process transfers control to a child process, the callback environment must be inherited.

The current Exec Interface does not inherit the callback environment automatically, so this operation must be performed by the application when jumping to a program loaded from CD-ROM or transferring control to a child process.

The process of environment inheritance is outlined below.

- The parent process
  1. Closes all events (CloseEvent)
  2. Momentarily halts the callback (StopCallback)
  3. Performs a global jump to the child process
- The child process
  1. Initializes the callback (ResetCallback)
  2. Initializes the new library (CdInit, ResetGraph, etc.)
  3. Establishes the new application callback
  4. Opens the new event (OpenEvent)

You must return control to the parent process from the child process.

Default Callbacks and Events

The default callback functions of each corresponding callback are registered when a library is initialized.

For example, when CdSyncCallback() is initialized (this function 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. Use caution when set.

However, since the default callback is cancelled and the event transmission is halted when a new callback is created, please be careful.

Controller

The standard controller must be initialized by the PadInit() function before it can be opened..

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 displayed by controller A, and the lower 16 bits by controller B.

See libetc.h for a description of controller button assignments.

The PadInit/PadRead interface is controlled by the standard controller. To use other controller interfaces, you must use the controller driver directly.

Video mode

SetVideoMode() function is provided in the library for declaring the present video signal mode. Although the NTSC mode video signal enviironment 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

The following issues relating to applications programming are further described below:

(1) VSync callbacks
(2) The stack pointer and operations related to Exec processing
(3) 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.

VSyncCallback() is used if a single callback needs to call multiple functions in a specific sequence.

Example:

```
/* Main program*/
main(void) {
void callback(void);
...
VSyncCallback(callback);
..
}

/* Callback table */
void (*func)() = {
func0,        /* Callback to be called first */
func1,        /* Callback to be called second */
func2,        /* Callback to be called third */
0,
};

/* Parent callback program */
void callback(void)
{
int    i;
for (i = 0; func[i]; i++)
(*func[i])();
}
```

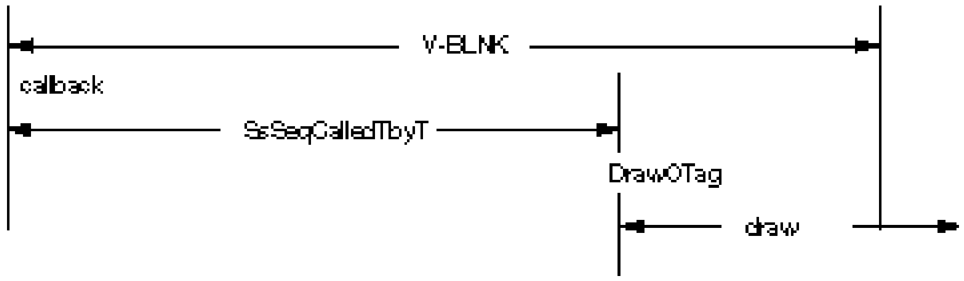
Timing of VSync interrupts

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()
{
SsSeqCalletTbyT();
DrawOTag(ot);
DrawSync();
}
```

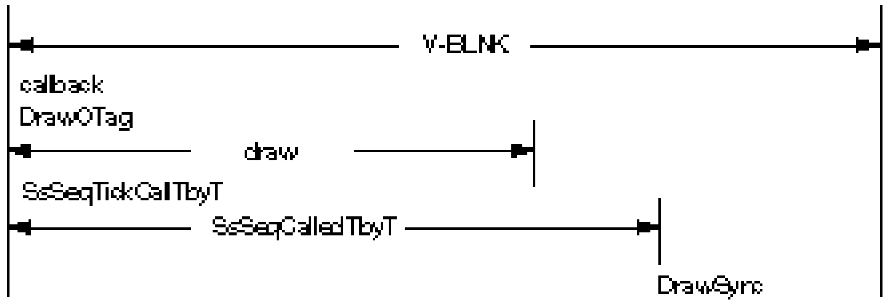
Figure 6 Timing for VSync interrupts



```
In the following example, rendering can be performed in parallel with sound
driver execution.
callback()
{
DrawOTag(ot);
SsSeqCalletTbyT();
DrawSync();
}
```

Note that since DrawOTag() is a non-blocking function, it should return immediately.

Figure 6 Timing with VSync interrupts



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 and 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

With applications that make use of 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 the 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.

Callback initialization and completion

Callback functions are functions that are automatically called when the corresponding non-blocking function completes. In other words, when a non-blocking function completes, an interrupt is generated causing the program to jump to the function address that was saved as the callback.

The initialization function ResetCallback() must always be called when a callback function is used. A callback may be suspended by calling StopCallback(). A callback that has been suspended by StopCallback() can be reactivated by calling ResetCallback(). StopCallback() and ResetCallback() should always be used during process switching.

Child processes and callbacks

Data, together with a section of code for the callback, are linked in memory with the application. If shared libraries are not used, the code and data for the callback will not be shared between processes, and identical copies of the information will be used for each process.

For this reason, the callback environment will need to be switched when transferring control from the parent process to a child process.

In this case,

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();  /* do not suspend callbacks */
Exec(&child_program);      /* activate child process */
/* ResetCallback(); /* do not reset callbacks */
.....
}

Child:
main() {
/* ResetCallback(); /* do not reset callbacks */
.....
/* StopCallback();  /* do not 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.

Gun Library Overview

Library File

The gun library file name is libgun.lib. The dedicated driver libgun.lib must be linked when using the gun.

Header File

The gun library does not have its own header file.

Table 1

Contents	File Name
Library	libgun.lib

Button Data

Following are the bufA, bufB buffer structures defined in the InitGun (char *bufA, char*lenA, char*bufB, long lenB, char *buf0, char *buf1, long len)function.

Table 2

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

```
            76543210
First byte      ----S---
Second byte      #%------
```

S: Start button
#: Trigger of gun
%: Button

Location data in the horizontal/vertical direction on the screen

Following are the buf0, buf1 buffer structures defined in the InitGUN (char*bufA, char*ltnA, char*bufB, long lenB, char*buf), char*buf1, long len)function

The maximum number of receive data is 20.

Table 3

Byte	
0	Not used
1	Number of available horizontal/vertical direction counter value
2,3	Horizontal direction counter value 0
4,5	Vertical direction counter value 0
6,7	Horizontal direction counter value 1
8,9	Vertical direction counter value 1
10,11	Horizontal direction counter value 2
12,13	Vertical direction counter value 2
14,15	Horizontal direction counter value 3
16,17	Vertical direction counter value 3
18,19	Horizontal direction counter value 4
20,21	Vertical direction counter value 4
22,23	Horizontal direction counter value 5
24,25	Vertical direction counter value 5
26,27	Horizontal direction counter value 6
28,29	Vertical direction counter value 6
30,31	Horizontal direction counter value 7
32,33	Vertical direction counter value 7
34,35	Horizontal direction counter value 8
36,37	Vertical direction counter value 8
38,39	Horizontal direction counter value 9

40,41	Vertical direction counter value 9
42,43	Horizontal direction counter value 10
44,45	Vertical direction counter value 10
46,47	Horizontal direction counter value 11
48,49	Vertical direction counter value 11
50,51	Horizontal direction counter value 12
52,53	Vertical direction counter value 12
54,55	Horizontal direction counter value 13
56,57	Vertical direction counter value 13
58,59	Horizontal direction counter value 14
60,61	Vertical direction counter value 14
62,63	Horizontal direction counter value 15
64,65	Vertical direction counter value 15
66,67	Horizontal direction counter value 16
68,69	Vertical direction counter value 16
70,71	Horizontal direction counter value 17
72,73	Vertical direction counter value 17
74,75	Horizontal direction counter value 18
76,77	Vertical direction counter value 18
78,79	Horizontal direction counter value 19
80,81	Vertical direction counter value 19

(The counter value is given as a half word.)

Memory Card

When using the memory card, set 0 for the InitCARD(0) argument.

```
InitGun (bufA, lenA, bufB, lenB, buf1, buf2, len);
InitCARD(0);
StartCARD();
_bu_init();
StartGUN();
ChangeClearPAD(0);
```

Multi-tap Library Overview

It is possible to use up to 4 controllers and memory cards cards for 1 port with a Multi-tap.

Library File

The Multi-tap library file name is libtap.lib. The dedicated driver, libtap.lib, must be linked to the use the Multi-tap.

Header File

The Multi-tap library does not have its own header file.

Table 1

Content	Filename
Library	libtap.lib

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 2 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 3 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'.

```
 InitPAD(bufA, lenA, bufB, lenB);
  InitCARD(0);
  StartCARD();
_bu_init();
  StartPAD();
  ChangeClearPAD(0);
```

Chapter 12: Link Cable Library

The Link Cable library (libcomb) provides a service for connecting PlayStation units via a cable for linked operation. This is applicable to many games, especially combat games.

Run-time Library Overview

Table of Contents

Overview

The link cable library provides the function of communication by connecting the extension SIO connectors of two PlayStations using the link cable.

Communication is performed by the read() function and the write() function. Both functions support the asynchronous mode, in which events occur when processing is complete, and the 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. The communication rate can be changed with the _comb_control() function.

Library Files

The file name of the link cable library is `libcomb.lib`. Every program calling services must link with this library.

Header Files

The link cable library has no unique header file. The kernel library header file `kernel.h` and the standard header file `sys/file.h` are required.

Table 12–1

Contents	File name
Header	libcomb.h
Library	libcomb.lib

Alterations to Contents

The following alterations have been made to the library 3.5 for the library 3.6 version:

(1) Asynchronous write is supported. Characters are sent one at a time by the DSR interrupt from the other station..

(2) A time out control function for synchronous input/output has been provided. A callback function which is called by _comb_control() during the synchronous read() and synchronous write() loop can be set.

(3) An input request cancellation has been added for use during the processing of a receiving-related errors which occur during asynchronous input. _comb_control (2,3,0) is called before the error event delivery.

(4) In order to reduce the rate of receiving-related error occurence, CTS and DTR are set to OFF during serial controller-related interrupt handler operation and the other station’s transmission is stopped.

(5) The default value of the unit-number of characters for receiving has been changed from 8 to 1.

(6) Without performing communication an information exchange function by the control line only has been added to _comb_control.

(7) The control line transition has been described clearly.

(8) The termination conditions for synchronous input/output have been described clearly.

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 provides input/output functions in accordance with the standard C language.

Table 2: Link cable driver

Item	Contents
Device name	sio
Block size	1 byte Asynchronous writing must be carried out every 8 bytes.
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 3

Cause descriptor	Event type	Contents
HwSIO	EvSpIOEW	Completion of asynchronous writing
	EvSpIOER	Completion of asynchronous reading
	EvSpERROR	Error occurrence related to receiving
	EvSpTIMOUT	Timeout in synchronous reading/writing

Installing and dismounting the driver

The link cable driver is installed with the AddCOMB() function, and dismounted with the DelCOMB() function. Opening the sio device without installing the driver will cause an error.

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 the function which is called during this loop. The callback function is not registered in the default state.

The func function can be made into a callback function by means of _comb_control (4,0,func). This function must meet the following specifications. Also, the callback function registration can be cancelled by _comb_control(4,0,NULL).

Syntax	long func (long spec, unsigned long count)	
Argument	spec	1:during synchronous read 2: during synchronous write

	count	Current value of internal counter
Return Value	Returns 0 when the wait loop is timed out and returns 1 when the wait continues	

Termination Conditions for Synchronous Input/Output

Synchronous read terminates when the specified number of characters can be received. However, 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.

Synchronous write terminates when the specified number of characters can be transmitted. However, it can also terminate when the wait callback function returns a prescribed value. In addition to this, changing the other station DTR to 0 also causes termination. These interruption conditions become effective when the Open Mode is specified as O_NBLOCK.

Interrupt and read/write functions

Within each function of the link cable library the code for accessing the common resource serial controller is executed in interrupt-prohibited status, in other words by the critical section.

Transition to the interrupt-prohibited status is executed by a method which differs from the usual EnterCriticalSection(), calling during event handler is possible and also the overhead is smaller.

However, this method is not perfect with respect to CPU register control protection in a multi-thread environment. Although there is absolutely no problem when the existing PlayStation library is used by single thread, when the original thread control is being carried out, the library driver’s operation cannot be guaranteed.

Unit-number of characters for receiving

This parameter is effective only in asynchronous read.

PlayStation is equipped with an 8 bit receiving buffer and the receiving interrupt can be set to occur each time1,2,4 or 8 bytes are received when using in the asynchronous read package. This is called the unit-number of characters for receiving. It is recommended that the number of received units be set to 1 in the default in order to prevent overrun errors. In contrast to this, since the transmission buffer is only 1 byte, the serial controller will operate only in 1 byte units in asynchronous write mode.

Error processing

When detecting each overrun parity frame receiving error during asynchronous input/output, the driver will cancel the asynchronous read request by calling the next function, and will successively deliver EvSpERROR events. This function clears the serial controller error flag:

```
_comb_control(2,3,0);          /*Cancel asynchronous read request */
```

BIOS

The link cable BIOS provides precise driver control and a function which obtains information which cannot be covered by the standard C language functions. The interface function is _comb_control(). BIOS will work without installing the driver.

Following is the explanation of the _comb_control() function.

Syntax   long _comb_control(cmd, arg, param)

unsigned long cmd;

```
unsigned long arg;
unsigned long param;
```

Arguments	cmd	command
	arg	subcommand
	param	argument

Table 4: Command Summary

cmd	arg	Function
0	0	Returns the serial controller status (*1).
0	1	Returns the control line status (*2).
0	2	Returns the communication mode (*3).
0	3	Returns the communication rate by bps.
0	4	Returns the "unit-number of characters for receiving".
0	5	Returns the number of the remaining asynchronous input/output data (bytes) being processed. Asynchronous writing will be applied if the value of param is 0. Asynchronous reading will be applied if the valueof param is 1.
1	0	System reserved
1	1	Sets the value of param as the control line status (*2).
1	2	Sets the value of param as the communication mode (*3).
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 by making the control line to an improbable status for transmission processing (DTR==0, RTS==1), the other station is informed of a special status. When param is 0, DTR and RTS are made 0.
3.	1	If (DSR==0 && 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 5: 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 6: : Control line status

bit	Contents
31-2	Undefined
1	1: RTS is ON
0	1: DTR is ON

Table 7: 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 which drives the link cable connector is a serial controller which supports asynchronous communication (following referred to simply as serial controller).

Buffer

The serial controller is equipped with a 1 byte transmission buffer and an 8 byte receiving buffer.

Control Line

The serial controller is equipped with two sets of control lines: DTR/DSR and RTS/CTS. Receiving operations cannot be performed when neither DSR and RTS are in the ON status. Also, DSR can be made the level trigger interrupt source.

Table 8: Control Line

Transmission Name	Receiving Name	Receiving Function	Transmission Interrupt
DTR	DSR	Receiving functions automatically halt when OFF	Level trigger
RTS	CTS	Receiving functions automatically halt when OFF	None

Communication Specifications

Communication specifications can be selected from the settings shown below:

Table: Communication Specifications

Item	Setting Value
Character Length	5,6,7,8 bits
Stop Bit	0,1,1.5,2 bits
Parity Check	None, Even, Odd
Communication Rate	1~2073600bps (2073600 divisor only)

Default Settings

The initial settings carried out in the link cable driver installation are as follows:

Table 10: Default Settings

Items	Values
Character length	8 bits
Stop bit	1 bit
Parity check	Disabled
Communication rate	9600 bps

Control Line transition

The driver operates DTR (DSR for receiving) and RTS (CTS for receiving) as follows:

Table 11: Control Line transition

Driver Operation	DTR	RTS
Power On (No other station or other station power supply off	1	1
(Other station present, driver not initialized)	0	0
Driver Initialization		
AddCOMB():	0	0
Synchronous Write		
open (:sio”, O_RDONLY);	-	-
write(...);	-	-
write complete	-	-

close();		-	-
Synchronous Read			
open("sio",O_WRONLY/NOWAIT)		-	-
write (...);	-	-	
DSR interrupt occurrence		0	0 (status saved)
DSR interrupt completion			(status returned)
write completion		-	-
close();		-	-
Asynchronous Read			
open("sio", O_RDONLY/O_NOWAIT);	-	-	
read(...);		1	1
Received interrupt occurrence		0	0 (status saved)
Received interrupt completion		0	0 (status returned)
Read completion		0	0
close();		-	-

In the control line operation by the above-mentioned driver DTR and RTS always have the same value. When each is given a different status, the other station can be notified of a specific status (for example, waiting to charge into combat mode). _comb_control(3,0,val) and _comb_control(3,1,0) provide notification functions based on this principle.

Programming Hints

Detection of the other station's connection (1)

Through the insertion of the driver provided by this library synchronous and asynchronous input/output operation the DTR and RTS control line pairs are always present. By using _comb_control() to alter the status of one of the control lines, preparations for carrying out transmission can be transferred to the other station. In _comb_control() the function which sets the RTS to 1 and the DTR to 0 along with the function which detects its status are provided so the above method can be easily used.

With this method there is no simple way to find out whether the other station has recognized this side's status or not. In addition to having prepared an overall time out time frame, while monitoring the other station's control line status, it is necessary to use a time-saving based detection algorithm package which periodically returns to the default status (DTR==0 && RTS==0).

Detection of the other station's sonnection (2)

An easier method for detecting the other station's connection involves issuing a read request and then waiting until both control lines of the other station are 1. When these conditions have been met, specified data can be written. It must not be forgotten that even when all control lines are 1 it is possible that the other station is not connected.

Background Receiving by the Ring Buffer

The read function can be executed in the 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 background receiving can be easily carried out by operating the receiving buffer pointer provided in the read function.

Since the write function can also be carried out in the critical section, it is possible to package processing such as the retransmitted request issuance with the above-mentioned 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 receiving error such as overrun can occur and this is generally attributable to a decrease in efficiency. Also, in principle it is unavoidable that the efficiency of asynchronous communication is lower when compared to synchronous communication.

From the above four combinations (synchronous read/write and asynchronous read/write), asynchronous write puts the heaviest load on the CPU.

Lightest Overhead Transmission

The data which should be transmitted is divided into 8 character length packets, scattered suitably within the code and transmitted by the synchronous write. This transmission format has the lightest load on the CPU. Of course, since each write function should conclude within a definite time period, the maximum waiting time is set by the time out callback function for all write functions. When the time limit is reached the transmission will be 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.

Chapter 13: Extended Sound Library

The Extended Sound library is created with the dedicated PlayStation Sound Artist Tool. It is a service that converts sound data so it may be used by the PlayStation.

Table of Contents

Overview

The sound Library (libsnd) is composed of the following groups of functions.

(1) VAB data access functions

Functions for accessing sound source data (VAB data)

(2) Functions for initial setting and termination functions

Required when activating and terminating the sound system

(3) SEQ access functions

Functions for handling music score data (SEQ data)

(4) Individual sound setting functions

Functions for the production of single sound sound effects, rather than musical sound effects

(5) Common attribute setting functions

Functions for executing the necessary settings for using the extended sound library and for common attribute settings of each SPU voice

(6) Sound Utility Function

Functions for changing the attribute table in VAB data to run-time and for applying the effect after KeyOn to the allocated voice.

Library Files

The file name of the extended sound library is `libsnd.lib`. Every program calling services must link with this library.

Header File

The header file for the sound library is `libsnd.h`. Every type of data structure and macro used by libsnd is defined in this file. Every program calling sound services must include this file.

Table 13–1

Contents	File name
Library	libsnd.lib
Header	libsnd.h

Score Data

In the extended sound library, music data is defined as SEQ data format and SEP data format.

SEQ Data Format

SEQ is a format 1 SMF (Standard Midi File) converted for use with the PlayStation. The MIDI data structure track/chunk data is merged with the time order in SEQ format.

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 13–1: 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 change note off status to note on status and velocity 0. SEQ also supports the following status data used by MIDI.

- Note on
- Note off
- Program change
- Pitch bend

The list below is for control change:

- 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)

NOTES: 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 3.5/Reference* for details of access-related functions.

The SEP data format is illustrated below.

Figure 13–2: SEP data format

Sound ID (4 bytes)
Version number (4 bytes)
SEQ number (4 bytes)
Quarter-note resolution (2 bytes)
Tempo (3 bytes)
Time (2 bytes)
Data size (4 bytes)
Data
SEQ end (3 bytes)
SEQ number (2 bytes)
Quarter-note resolution (2 bytes)
Tempo (3 bytes)
Rhythm (2 bytes)
Data size (4 bytes)
Data
SEQ end (3 bytes)

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.

- The tone numbers range from 0 to 15. To change attributes of all of the tones, specify 16.
- The hardware specifications state that coefficients, such as for reverb depth, must be set as a group using the SPU. Consequently, it is not possible to set the reverb type or depth, feedback amount, and the like 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 to real-time. The ‘x’ in the right column indicates that these attributes are not currently supported.

Table 13–2: Data1~Data3 Contents

ATTRIBUTE (CC06)	Data1 (CC99)	Data2 (CC98)	Data3
Priority	Tone Number	0	0~15
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

(**) Reverb Type (Refer to Sound Delicatessen DSP)

Table 13–3: Data3 Mode Type

Number	Mode
0	Off
1	Vibrate
2	Portamento
3	1&2 (Portamento and Vibrate on)
4	Reverb

Table 13–4: 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 Change to Set Repeat 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 13–5: Looping Using Control Change

ATTRIBUTE	Data1 (CC99)	Data2 (CC06)
Loop1(start)	20	0~127 (***)
Loop2(end)	30	

(***) For continuous looping, set 127(0x7f).

Note:

Depending on the sequence, when setting a repeat loop to the same Delta Time it is possible that the order will shift when it is modified to SMF even if it was input in regular sequence. Since this can cause the data to become invalid, please do not set repeat loops to the same Delta Time.

Also, values become valid from the KeyOn immediately after the Data Entry is read in VAB attribute data settings

Marking Function Using Control Change

NRPN data may also be used to implement a function for marking places within 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 13–6: Marking via Control Change

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).

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 data of 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. It is a sound processing format that is handled as a single file at runtime.

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 multitimbral (multisampling) functions.

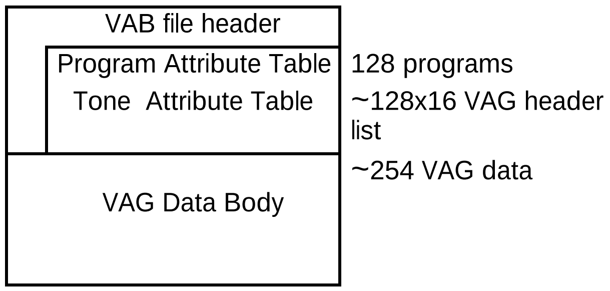
Each VAB file may contain up to 128 programs. Each of these programs can contain up to 16 tone lists. Also, each VAB file can contain up to 254 VAG files.

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.

Since multiple tone lists can reference the same waveform, a sound method can also have different waveforms.

A VAB format file is organized as follows:

Figure 13–3: VAB format and VAB header



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.

a. Tone attributes

```
struct VabHdr {
    long form;                  /*always VABp*/
    long ver;                   /*format version number*/
    long id;                    /*bank ID*/
    unsigned long fsize;        /*file size*/

    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 mpan;         /*master pan*/
    short reserved0:            /*system reserved*/
    unsigned char reserved1;    /*system reserved*/
    unsigned char reserved2;    /*system reserved*/

    long reserved3;             /*system reserved*/
};
```

b. Program attributes

```
struct ProgramSlot {
    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*/
    short attr;            /*program attribute*/
    char reserved0;      /*system reserved*/
    long reserved1;      /*system reserved*/
    long reserved2;      /*system reserved*/
};
```

c. Tone Attribute

```
struct Vagatr {
    unsigned char prior;     /*tone priority*/
    unsigned char mode;      /*tone mode*/
    unsigned char vol;       /*tone volume*/
    unsigned char pan;       /*tone pan*/

    unsigned char center;    /*center note (0~127*/
    unsigned char shift;     /*pitch correction (0~99.cent(Note) 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(Note)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*/

    short adsr1;             /*ADSR1*/
    short adsr2;             /*ADSR2*/
    short prog;              /*parent program*/
    short vag;               /*waveform (VAG) used*/
    short reserved[4];       /*system reserved*/
};
```

Function Execution Sequence

When using sound library functions, execute the functions in the following order. For details, see sample program.

(1) Initialization

Initialize with the SsInit() function. Use the SsSetTableSize() function to maintain the SEQ attribute data area.

(2) Tick mode setting

Set tick mode with the SsSetTickMode() function.

(3) Opening data

- When using VAB data: execute SsVabOpenHead() → SsVabTransBody(),SsVabTransCompleted()
- When using SEQ data execute SsSeqOpen()
- When using SEP data execute SsSepOpen().

(4) Starting the sound system

Start the sound system by executing the SsStart() function.

(5) Required processing

Set main volume. Execute required processing.

(6). Closing data

- When using VAB data execute SsVabClose()
- When using SEQ data execute SsSeqClose()
- When using SEP data execute SsSepClose()

(7) Halting the sound system

Halt the sound system by executing the SsEnd() function.

(8) Terminating the sound system

End the sound system by executing the SsQuit() function.

Frequently Asked Questions on libsnd

This FAQ is a collection of frequently asked questions about sound, their answers and hints.

Q-1: Why is there noise in CD-DA/XA playback?

A-1-1: Is the converted data correct?

Assume 16-bit straight PCM data with the sound tool. Make sure that the data is not AIFF data. After AIFF data is converted, noise is inserted into the beginning and end of the data as header and footer sound information.

SoundDesignerII 2.5 sampling data format is straight PCM data, so it may be used as is.

A-1-2: Does the volume drop when you seek after pausing and then resuming playback?

If a CD is halted in the middle of play, a drop in volume may be caused by the production of clipping noise when seeking. When you pause the game, the CD also pauses; issue a CD command after fadeout.

A-1-3: Are there a lot of high area components in the XA data?

Noise is generated in XA data because sound is compressed to 1/4th. The noise is obvious particularly when there are a lot of high area components. You should apply a lowpass filter as part of preprocessing.

Q-2: Why is SPU voice sound production not correct?

A-2-1: Are there more than 24 sounds generated?

SPU is able to generate 24 voices at the same time. Use of the voices changes according to the ADSR value of those voices using waveform data rather than actual KeyOn/KeyOFF requests. An example of this is that, when you extend the Release Rate, sound continues a little after the KeyOff request is transmitted; it continues in exclusive possession of the voice.

Even if the same sound continues when a KeyOn request is issued, it is in exclusive possession of the next voices, and when the voice is insufficient, it completely erases other sounds.

A-2-2: Are there overlapping interrupts?

See the answer to the next question about SEQ playback for methods of dealing with this problem.

Q-3: Why is the tempo uneven when a CD is reading during SEQ playback?

A-3-1: In run-time Library Ver. 3.0 (and previous libraries),
when you use the Extended Sound library to read CD SEQ file playback status, the SEQ file tempo varies. This phenomenon is confirmed.

To prevent this variation in tempo, perform the processing shown below.

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()set within the funciton 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.

Q-4: I don’t understand the monaural/stereo setting.

A-4-1: The CD(DA/XA) stereo/monoaural settings are made in the libcd function CdMix(); SPU (SEQ,SE) stereo/monaural settings are made in the libsnd functions SsSetMono() and SsSetStereo().

When you use the functions above, control via the software is possible, but in the PlayStation itself, there is a hardware stereo/monaural switch.. If you use this switch, the software stereo setting is sufficient.

When the cable is connected only to the left channel of the PlayStation, right channel sound also comes through the left channel. That is, in this case, monaural sound comes through the left channel. When the cable is connected to both channels, the sound is divided between the two channels and stereo sound is produced.

Chapter 14: Basic Sound Library

The Basic Sound library directly controls the PlayStation sound play processor (SPU). Individual functions are provided for operations such as transferring data other than music data to the sound buffer.

Table of Contents

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), although individual functions for operations such as transferring other types of data in addition to music (for example, texture data, etc.) to the sound buffer are provided.

This library functions only as a library for the SPU, so it does not have time control functions. These functions need to be controlled by a separately provided kernel library such as libapi.

Library File

The basic sound library file name is `libspu.lib`. All programs that call services must be linked to this file.

Include Header

The basic sound library header file name is `libspu.h`. All types of data structures and macros are defined in this file. Programs which call all functions must include this file.

Table 14–1

Content	Filename
Library	libspu.lib
Header	libspu.h

Voice Audio Source Control Function

The functions which can be controlled in the basic sound library are indicated below. The following attributes may be set individually for 24 ADPCM audio sources (hereafter referred to simply as ‘voice’).

- Sound volume (can set L/R independently)
- Sound interval
- Address of waveform data in sound buffer
- Envelope (ADSR)
- Loop point

These attributes, as well as key on/key off, are set for each voice. Key on/key off can also be set independently for each of 24 voices.

These attributes may be changed while key on is in effect and sound is being generated. So, 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 Control Function

The SPU has one noise generator. This noise generator may be set and used for each voice instead of sound buffer waveform data. 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.

LFO Control Functions in Intervals

By using adjoining voices, the SPU can produce an LFO (Low Frequency Oscillator) effect in an interval.

This may be expressed by the equation below. For LFO control, be aware that two voices are used to generate one tone.

NewPitch(n) = (1 + V(n-1)) * Pitch(n)

Table 14–2: 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 Control Function

Reverb is provided using various types of templates. These templates have many variable parameters, and effects may be varied by adjusting these parameters.

Reverb uses the sound buffer as its work area with the offset (starting) address varying according to each individual parameter. Since this is also prepared for use as a template, the area before the offset (starting) address may be used as an actual waveform data area.

Reverb is not just settable to on and off for all the data at once; individual voices may also be set. Reverb may also be applied to CD input and external digital input (to be described later).

Function for Optional Data Transfer to Sound Buffer

Use the methods below when transferring waveform data from main memory to the sound buffer (=Write) or when transferring waveform data from the sound buffer to the main memory (=Read).

- I/O transfer (Write only)
- DMA transfer (Write/Read)

DMA transfer transfers asynchronously using the DMA controller, so the CPU is able to do other processing during the transfer.

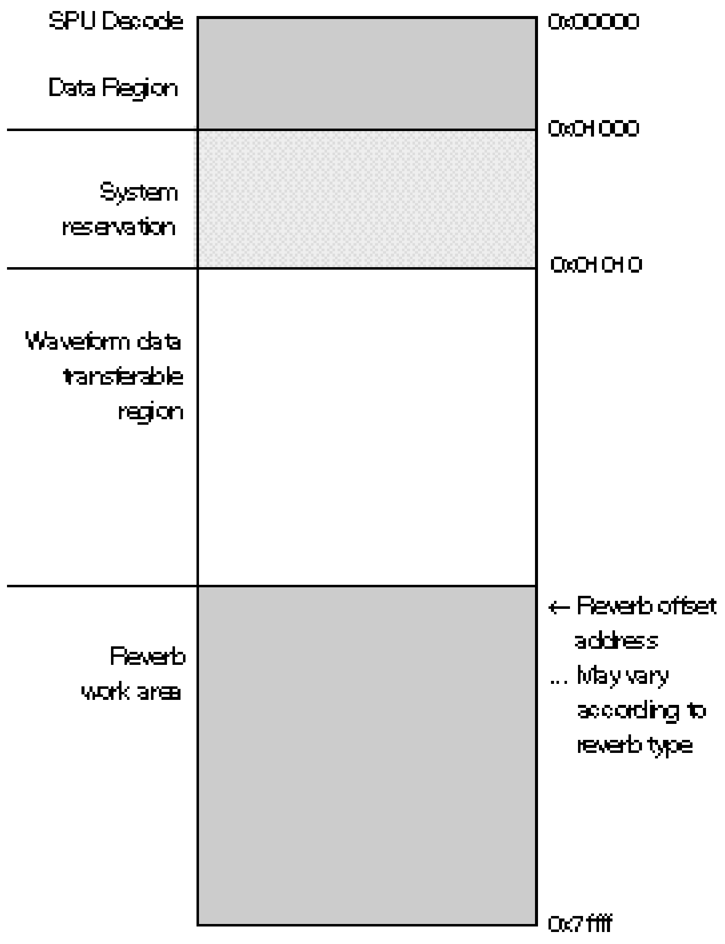
I/O transfer uses the CPU, so other processing cannot be performed during the transfer. You must select DMA transfer if you are transferring data while continuing playback.

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 loop area by malloc() or a similar function. In other words, it cannot be the address of a stack region of a variable declared within a function (auto variable).

Active memory management is not performed in the sound buffer. So, data transfer should avoid the areas listed below. If the areas below are not used, data transferred there 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 (described later).

Figure 14–1: Sound Buffer Memory Layout



Interrupt Request Function 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 done by generating a hardware interrupt (interrupt request) when access is made to a specific address. It is also possible to specify a function to be called in response to this interrupt request.

Sound Buffer Memory Management

The sound buffer memory management function provides a library with very limited functions. It manages only the table containing occupied memory and reports only that information. Doing transfers to sound buffer regions using this information makes possible simple sound buffer memory management.

Function for 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.

Transferring Data Decoded by SPU to Main Memory

The SPU writes to the sound buffers starting 0x1000 bytes (0x800 short int) area 16 bits at a time (= 1 short int) at each clock (44.1 kHz) pulse. Data is written after CD input volume processing sound data and after Voice 1 and Voice 3 envelope processing sound data. The individual sound buffers are each 0x400 bytes (0x200 short int) long and divided into the first half (0x200 bytes [0x100 short int])and the second half (0x200 bytes [0x100 short]). By deciding which buffer region to write to, it is possible to write a maximum of 100 samples (100 / 44100 = 0.0022 ... seconds) of data at one time.

SPU Streaming Library

Overview

The "SPU streaming library" provides the playback functionality of large-sized wave forms, which will overflow the sound buffer of PlayStation(R), with SPU.

SPU originally plays back only the wave form data contained in the sound buffer, and that is to say, it cannot play back the wave form data which is larger than the sound buffer.

Therefore, by transferring the next play-backed data to a designated area in the sound buffer continuously during the playback by SPU, the wave form data which is larger than the sound buffer can be played back.

Basic Operations

With the SPU streaming library, the use of the voice contained by SPU performs its own playback. The playback performed by the voice itself is the same processing as the ordinary sound generation (Key on).

In the main memory the wave form data used by the SPU streaming library is placed. (a part of the data is enough when starting the SPU streaming processing) The wave form data used in the SPU streaming library is:

"The VB file which includes only one VAG data."

In other words,

"The VAG file which doesn't include the header."

In the sound buffer the area is allocated for each voice used by the SPU streaming library. The SPU streaming library allows SPU to play back the wave form data contained in the main memory, transferring it to the allocated area continuously.

The area for each voice used by the SPU streaming library in the sound buffer is called a "stream buffer."This buffer is necessary for each voice used by the SPU streaming library. When the SPU streaming library uses more than one voice, the size of every stream buffer must be the same.

Also, a series of the processing, that the wave form data contained in the main memory is transferred to the stream buffer continuously and is played back by SPU, is called a "stream".

The SPU streaming library can handle 24 streams and 24 stream buffers at the maximum, and all of the 24 voices can be the voices used by SPU streaming library.

Wave Form Data Processing

The SPU streaming library can treat the larger wave form data (VBdata) than the sound buffer, and all parts of the wave form data don't have to exist on the main memory at the time of starting the stream processing.

With the SPU streaming library, at any points in the processing, the wave form data at least half as large as the stream buffer is necessary for each stream. When the transfer of the processed wave form data is requested, by specifying the head address and the attributes of the necessary part of the wave form data, the SPU streaming library is informed of the continuation of the stream processing .

The wave form data is used being rewritten partly at the time of transferring.

Four States in the SPU Streaming Library

There are 4 states in the stream processing by the SPU streaming library.

* Idle

Streams are not processed in this state. In this state the transfer to the stream buffer is not performed. Thus, the SPU streaming library puts no load on the PlayStation(R) system.

Also, all the processing by the SPU streaming library can be completed in this state.

* Preparation

Prior to the stream processing, for all the streams, in order to eliminate the time-lag of the actual sound generation the wave form data half as large as the stream buffer must be transferred to the stream buffer. This states indicates the transfer and the end of the transfer. The end of the transfer can be detected by the "preparation finished callback function".

* Transfer

In this state the sound generation is actually performed in the designated stream after the preparation. Even in this state the request for the above "preparation" for the other streams is accepted, but the state doesn't change to the "preparation". The preparation is performed in the "transfer state".

In this state the half of the stream buffer is processed, and the processing can be detected by the "transfer finished callback function" explained later.

* Termination

The termination is designated for all the streams, and the processing is going to the "idle state". In this state the transfer is completed. Also, any requests for the next "preparation" or "transfer" are not accepted. Once the state becomes "idle", the request for the "preparation" is accepted.

---> Idle --> Preparation --> Transfer --> Termination --

Figure 14–2 Four States and their transitional states

Callback functions

The SPU streaming library provides 3 types of the callback functions for the stream processing.

Each callback function is called with the same timing in the multiple streams. The requested stream can be recognized by the argument of the callback function.

  * Preparation finished callback function

This is the function called when the transfer is completed in the "preparation" described above.

  * Transfer finished callback function

This is the function called when the transfer of the wave form data half as large as the stream buffer is completed. In this function the attributes for the next transfer are set. However, this function is not called at the completion of the transfer in the "preparation".

  * Stream finished callback function

This is the function called when the playback of the termination-designated stream is completed.

The attributes for the next transfer in each stream must be processed by the "transfer finished callback function". Also, if the start of the stream follows immediately after the preparation, the attributes for the next transfer must be set in the "preparation finished callback function".

Without these callback functions, the processing continues. But since the arguments for the next transfer in each stream must be set every time, the "transfer finished callback function" must be called without fail. At the same time, the arguments for the next transfer must be set in the callback function.

```
  [Idle]          |
--------------------  + <== SpuStTransfer (SPU_ST_PREPARE,)
                  |
 [Preparation]      +----> Preparation finished callback function
                  |
--------------------  + <== SpuStTransfer (SPU_ST_PLAY,)
                  | (Transfer to the stream buffer)
                  +----> Transfer finished callback function
                  |
                  |
  [Transfer]        | (Transfer to the stream buffer)
                  +----> Transfer finished callback function
                  |
                 -----
                 -----
                  |
                                 |(Transfer to the stream buffer)
  Stream A ==>     +----> Transfer finished callback function
  is completed.    |
                  |
                  |(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       ==> +----> Transfer finished callback function
  is completed.       |
 ------------------   |
                    |
  [Termination]       |
                    |
-------------------  +----> Stream finished callback function
  [Idle]              |
```

Figure 14–3 Four callback functions and transitional states

Stream Processing

Stream Preparation and Start

The preparation for each stream is always performed for the former part of the stream buffer, and the wave form data is transferred to that part. Therefore, if the preparation for the stream is requested in the "idle state", it is processed promptly. On the other hand, if another preparation is requested in the "transfer state", the preparation must be waited until the processing is transferred to the former part of the stream buffer in the currently processed streams. Consequently, the transfer is carried out with the other stream processing when the former part of the stream buffer is processed.

Each stream is always started by the transfer to the latter part of the stream buffer because the transfer to the former part has been completed. Therefore, if the start of the stream is requested in the "preparation state", it is processed promptly. On the other hand, if the start of another stream is requested in the "transfer state", the start must be waited until the processing is transferred to the latter part of the stream buffer in the currently processed streams. Consequently, the transfer is carried out with the other stream processing when the latter part of the stream buffer is processed.

Attributes for the Next Transfer

The attributes for the next transfer in each stream are specified in the "transfer finished callback function". The necessary attributes are as follows.

* The head address of the wave form data area (half as large as the stream buffer) for the next transfer

* If the stream is completed in the next transfer,

  1. specify the "termination" in "status".

  2. specify the size of the last-transferred wave form data.

     (The size must be a half of the stream buffer or less.)

Stream Termination

The "termination" of each stream is specified by setting the "termination" for the attribute "status" and the size of the last-transferred wave form data (half as large as the stream buffer or less) when setting the next-transferred attributes. The stream is terminated when the playback of the stream specified in this setting is completed.

Key on/Key off

With the SPU streaming library, only the sound generation (Key on) is carried out automatically. The sound generation (Key on) is performed when the start of the streaming is processed. The sound cancellation (Key off) requires to be processed by the program.

However, if the sound cancellation (Key off) is carried out for the voice where the stream is processed, the state of the sound library may be unstable. Be sure to carry out the sound cancellation after the stream termination processing.

Actual Flow of Stream Processing

The simple flow of the processing by the SPU streaming library is as follows. As for the details on each function, refer to the explanation for each.

Initialization

The SPU streaming library initialization is performed by SpuStInit().

```
SpuStEnv *stenv;

stenv = SpuStInit (0);
```

The SpuStEnv structure returned by SpuStInit() is shown below.

```
==================================================================

typedef struct {

    char status;            /* stream status */

    char pad1;              /* padding */

    char pad2;              /* padding */

    char pad3;              /* padding */

    long last_size;         /* the size of last transferring

                          (last_size <= (size / 2)) */

    unsigned long buf_addr;    /* The start address of stream

                               buffer */

    unsigned long data_addr;    /* The start address of SPU

                               streaming data in main memory */

} SpuStVoiceAttr;


typedef struct {

    long size;              /* The size of stream buffer */
    SpuStVoiceAttr voice [24];
} SpuStEnv;
======================================================================
```

The stream is processed by specifying the attributes for this structure.

Attribute Initialization

"size, low_priority" in the SpuStEnv structure is the common attribute in all streams. The size is the same in all streams. The size of the stream buffer is specified here.

Example:

```
        stenv->size = 0x8000;
```

Set SPU_ON when lowering the priority level when processing SPU in low_priority. The fixed value (where the priority level is not lowered) is SPU_OFF.

Example:

```
        stenv->low_priority = SPU_ON;
```

The attributes which must be initialized for each stream are:

The head 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;
```

The head address of the wave form data transferred in the "preparation" on the main memory

= 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

Each callback function is set as necessary. All the callback functions in the SPU streaming library take the following syntax.

```
SpuStCallbackProc

callback_proc (unsigned long voice_bit, long c_status)
```

When the callback function is actually called, the value of the voice numbers to be processed in each callback function is given to the argument voice_bit by the bitOR of SPU_0CH to SPU_23CH. The state in which the callback function is called is given to the argument c_status. The program analyzes the arguments voice_bit and c_status, and processes them appropriately.

At least the "transfer finished callback function" must be called in order to process the stream. With this callback function, the head address of the next-transferred wave form data is specified, and the argument of the "termination" is specified to terminate the stream.

Voice Setting

The attributes for each voice where the stream is processed are set. For the "head address of the wave form data" in the voice attributes, the same value as the head 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

In order to eliminate the time-lag of the actual sound generation of the stream, as the preparation for starting the stream, the wave form data half as large as the stream buffer is transferred to the stream buffer beforehand. The preparation for the stream is carried out by SpuStTransfer(), and SPU_ST_PREPARE is specified as the first argument in the function.

The voices used for the stream are set for the second argument of SpuStTransfer() by the bitOR of SPU_CH to SPU_23CH. The attribute initialization must be performed for the designated voices.

Example:

```
SpuStTransfer (SPU_ST_PREPARE, (SPU_0CH | SPU_1CH));
```

When the transfer in the preparation for the stream corresponding to the requested voices is completed, the "preparation finished callback function" is called.

Prior to starting the stream following the preparation, the attributes for the next transfer must be set. If the start of the stream follows immediately after the preparation, the arguments for the next transfer must be set in the "preparation finished callback function".

As the attributes for the next transfer, the head address of the wave form data area (half as large as the stream buffer) is set.

Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
```

In the main memory, in each setting the area where the wave form data is placed doesn't have to continue from the previously processed area. Any area can be specified.

Start of the Stream

When the preparation is completed for each stream, the stream can be started. SPU_ST_PLAY is specified as the first argument of SpuStTransfer() as done in the preparation.

The voices used for the stream are set by the bitOR of SPU_0CH to SPU_23CH for the second argument of SpuStTransfer(). This value must be the same as the value specified in the preparation.

Example:

```
SpuStTransfer (SPU_ST_PLAY, (SPU_0CH | SPU_1CH));
```

As soon as the stream is started, the sound generation (Key on) is performed.

When one transfer is completed in all the streams, the "transfer finished callback function" is called.

In the "transfer finished callback function" the attributes for the next transfer are set.

As the attribute for the next transfer, the head address of the next-transferred wave form data area (half as large as the stream buffer) is specified.

Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
```

In the main memory, in each setting the area where the wave form data is placed doesn't have to continue from the previously processed area. Any area can be specified.

When more than one stream is processed, the "transfer finished callback function" is called without fail at the end of the transfer.

Stream Termination

To terminate the stream, SPU_ST_STOP is set for voice[].status in the SpuStEnv structure when specifying the attributes for the next transfer in the "transfer finished callback function" processing. At this time, the size of the last-transferred wave form data (half as large as the stream buffer or less) is set for voice[].last_size.

The stream is terminated after transferring the wave form data area represented by voice [].data_addr when SPU_ST_STOP is specified.

Example:

```
stenv->voice [n].data_addr += (0x8000 / 2);
stenv->voice [n].status = SPU_ST_STOP;
stenv->voice [n].last_size = 0x4000;
```

The "stream finished callback function" is called at the completion of the stream playback. (precisely before the start of the next transfer if the other streams are processed at this time.)

Completion

The completion of the SPU streaming library is performed by SpuStQuit().

```
SpuStQuit ();
```

Prior to calling this function, the termination processing must be completed for all the streams, and the state must be idle, that is to say, the sate after the "stream finished callback function" is called.

Frequently asked questions on libspu

This FAQ is a collection of frequently asked questions from the BBS about the Basic Sound Library, answers to these questions, and hints.

The information in this FAQ relates to content newly added to the library and included in the upgrade to 3.1. Make sure you keep your version of the library up-to-date.

Reverb setup

Q-1: With libsnd you can set 8 types of reverb mode. Is there a function you can use to find the size of the reverb work area used by the SPU sound buffer for each mode?

A-1: A function to find the reverb size of the work area is not supported.

The size of the reverb work area in the sound buffer is uniquely determined by the mode decision. This area is allocated and held without dynamic variation.

The table below shows the amounts of work area consumed. These are given also under SpuSetReverbModeParam in the Basic Sound Library document "function.txt".

Table: Volume Occupied By Reverb Mode In Sound Buffer

attr.mode	mode	hexadecimal	decimal
SPU_REV_MODE_OFF	off	0/80 (*)	0/128 (*)
SPU_REV_MODE_ROOM	room	26c0	9920
SPU_REV_MODE_STUDIO_A	studio (small)	1f40	8000
SPU_REV_MODE_STUDIO_B	studio (med)	4840	18496
SPU_REV_MODE_STUDIO_C	studio (big)	6fe0	28640
SPU_REV_MODE_HALL	hall	ade0	44512
SPU_REV_MODE_SPACE	space echo	f6c0	63168
SPU_REV_MODE_ECHO	echo	18040	98368
SPU_REV_MODE_DELAY	delay	18040	98368
SPU_REV_MODE_PIPE	half echo	3c00	15360

(*)If SpuReserveReverbWorkArea (SPU_ON) is used for address setting, it takes 128 bytes even if the mode is off. If SpuReserveReverbWorkArea (SPU_OFF) is used, it takes 0 bytes.

Related functions: SpuReserveReverbWorkArea, SpuSetReverbModeParam, SpuSetReverb

Q-2: What is the relationship between the work area, whose size varies according to the reverb mode setting, and the sound buffer memory management mechanisms ('SpuMalloc', etc.)?

A-2 The work areas are managed via the following algorithms.

1. Cases in which reverb work area has been reserved with SpuReserveReverbWorkArea (SpuOn)

SpuMalloc/SpuMallocWithStartAddr

Depending on the mode, you can allocate an area of size (0x7ffff - work area size), starting from address 0x01000.

2. Cases in which a work area has not been reserved with SpuReserveReverbWorkArea (SpuOff)

SpuMalloc/SpuMallocWithStartAddr

Area can be allocated in the entire sound buffer area, addresses 0x01000 to 0x7ffff.

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 (SpuOn) 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 oninformation from the sound buffer memory management mechanisms, and if possible reserves the area at that time. If the area cannot be allocated, SpuSetReverbModeParam returns withoutreserving the area.**4. If you execute SpuMalloc/SpuMallocWithStartAddr when there is no reverb work area reserved by SpuReserveReverbWorkArea,**

and afterwards attempt to reserve the reverb work areaagain 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.

Related functions: SpuReserveReverbWorkArea, SpuIsReverbWorkAreaReserved, SpuSetReverbModeParam, SpuSetReverb, SpuMalloc, SpuMallocWithStartAddr, SpuSetTransferStartAddr

Q-3: Why doesn't reverb take effect evenafter I've turned reverb on and set the reverb parameters?

A-3: If you change themode when setting reverb, noise may be generated for an instant due to indefinite values remaining in the reverb work area.

In order to avoid this as much as possible, the reverb depth value is set internally at 0. Since SPU_REV_MODE_OFF is set as the reverb mode initial value, even at the first setting if the Depth in the SpuSetReverbModeParam() has been previously set to anything other than 0, the reverb Depth will be set at 0 by means of a new SpuSetReverbModeParam() call. Therefore, for the Depth settings in SpuSetReverbModeParam() please call the SpuSetReverbModeParam() (mode specification mask is not set) again after the reverb mode setting without fail or set the Depth attributes in SpuSetReverbDepth().

Also, even executing the mode settings in SpuSetReverbModeParam() and SpuSetReverbModeParam immediately after the execution of the mode settings in SpuSetReverbModeParam, noise will inevitably be generated due to the inherent nature of reverb. This can be avoided by setting the SPU_REV_MODE_CLEAR_WA in the attr.mode argument to bit OR. In this way the reverb work area of the range corresponding to the mode which was set can be forcibly cleared. This process applies the DMA transfer, but since the processing is being done synchronously, no other processes (drawing, including pronunciation) can be performed during the processing and according to the type of reverb it will be necessary to wait for a short time.

Also, when setting the reverb mode in optional timing, the domain required in the designated reverb mode can be cleared by using SpuClearReverbWorkArea.

Until the reverb is actually required:

-Do not set the Depth for SpuSetReverbModeParam() or SpuSetReverbDepth()

or

It will be necessary to forcibly clear the reverb work area in either SpuSetReverbModeParam() or SpuClearReverbWorkArea().

Summarizing from the above, the points where care is required are:

If you are going to use reverb, then set the mode well in advance, not just before use. When you set the mode, reverb depth will go to 0.

The order in which you do reverb setup should be either

a. SpuSetReverbModeParam (specifying Mode/Feedback/Delay)
   SpuSetReverbModeParam (specifying Depth)

or,

b. SpuSetReverbModeParam (specifying Mode/Feedback/Delay) SpuSetReverbDepth (specifying Depth)

In the case of a. above, the value of the member of the member of the structure passed as an argument of SpuSetReverbModeParam must be changed. In the case of b. above, the structure used by SpuSetReverbModeParam may be use as is.

To avoid noise when you need to change reverb mode during the game, avoid doing “set mode ♦ set reverb depth” immediately beforehand.

If you really need to do this setting immediately beforehand, forcibly clear the work area first, using SpuSetReverbModeParam or SpuClearReverbWorkArea.

Note also that if you end up allocating the entire sound buffer area for waveform data it will be impossible to allocate a reverb work area, and therefore impossible to use reverb, since SpuSetReverbModeParam will end without setting reverb mode, as described in "Q/A2 :3.". To avoidbeing unable to use reverb, keep track of the size of your reverb work area and the size of waveform data. Set up the reverb modes and build the waveform data so that the work area and waveform data required will fit into 512 KB-0x1010 bytes.

Related functions: SpuSetReverbModeParam, SpuSetReverb, SpuClearReverbWorkArea, SpuReserveReverbWorkArea, SpuIsReverbWorkAreaReserved, SpuSetTransferStartAddr, SpuMalloc, SpuMallocWithStartAddr

Q-4: Why doesn't reverb take effect even though I've set up the reverb parameters and turned reverb on as described above in Q/A-3?

A-4: Because in the SoundDelicatessen (DTL-S710) Mac artist tool, the individual tones were not given a reverb attribute.

Display the ADSR window with Program menu >> Tone menu >> ADSR ... , and place an [↔] in that window in the 'reverb' check box in 'play mode'. This will set the reverb attribute in the selected Tone mode (bit 3 of mode = 1, i.e., mode = 4), so that reverb will take effect.

Note, however, that because reverb goes into effect universally on a Tone for which you checked [↔], you must consider reverb depth, overall atmosphere, and such when checking [x] in the 'reverb' check box in 'play mode'.

Q-5: With programsthat set reverb on, running the program again sometimes generates noise. What about this?

A-5: 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 (SpuOff SPU_OFF);
```

Extended Sound Library

```
#include <libsnd.h>

SsUtReverbOff();
SsUtSetReverbType (0);
SsUtSetReverbDepth (0, 0);
```

If you do not do this, you may get noise sometimes on the run that follows.

Related functions: SpuSetReverbModeParam, SpuSetReverb, SsUtReverbOff, SsUtSetReverbType, SsUtSetReverbDepth

Q-6: The reverb work area can be reserved using SpuReserveReverbWorkArea() in libspu, but what happens in libsnd?

A-6: The reverb work area is not reserved in the libsnd default (same as the libspu default). Since the sound buffer memory control in libsnd uses libspu functions, the SpuReserveReverbWorkArea() can be used in the same ways as with libspu.

Related functions:

SpuReserveReverbWorkArea, SpuIsReverbWorkAreaReserved

CD Volume

Q-7: There is a sound volume setting function CdMix in libcd. Is this used to set the CD/DA and/or CD-ROM/XA sound volume?

A-7: In CD output, when the CD decoder interprets data read from the CD such as CD/DA or CD-ROM/XA (ADPCM) as music data, it splits that data into left and right and sends it to the SPU, where it is input to the SPU section called Serial A and mixed with sound data output by the SPU, and then output from the audio output.

The Basic Sound Library (libspu) and Extended Sound Library (libsnd) provide functions to manipulate the sound volume of this music data from the CD.

The functions and their usage are given below.

Basic Sound Library (libspu)

```
#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 */
attr.cd.mix = SPU_ON;

/* set attributes */
SpuSetCommonAttr (&attr);
```

Extended Sound Library (libsnd)

```
#include <libsnd.h>

/* CD input ON */
SsSetSerialAttr (SS_SERIAL_A, SS_MIX, SS_SON);

/* set volume to mid-range */
SsSetSerialVol (SS_SERIAL_A, 0x40, 0x40);
```

You cancontrol the volume of CD-ROM music data via either of the setups above.

Related functions: SpuSetCommonAttr, SsSetSerialAttr, SsSetSerialVol

Q-8: When I execute SpuInit (or SsInit) after executing CdInit, I lose the CD sound that I had up to then by executing CdInit alone. What about this?

A-8: Both SpuInit and SsInit set the CD volume to 0. This is because, as their names indicate, they are for sound system initialization.

Therefore, to use music data from the CD, youmust use SpuSetCommonAttr (in the Extended Sound Library, SsSetSerialAttr, and SsSetSerialVol) as described in Q/A-7.

Related functions: SpuSetCommonAttr, SsSetSerialAttr, SsSetSerialVol

Diagnosing the key status of an individual voice

Q-9: When I'm using 24 voices with libspu, is there a wayI can find out whether a particular voice is currently keyed on?

A-9: You canconfirm the key status for a particular voice or for all voices by using SpuGetKeyStatus and SpuGetAllKeysStatus.

Related functions: SpuGetKeyStatus, SpuGetAllKeysStatus

Voice data attributes

Q-10: I want to force looping on the waveform of a voice that I've keyed on with SpuSetKey. But it won't loop for me even though I set SPU_VOICE_LSAX in mask in the SpuVoiceAttr structure and call SpuSetVoiceAttr specifying the address in the sound buffer in loop_addr.

A-10: The only waveform data for which you can modify looping with SPU_VOICE_LSAX is waveform data that was set up for looping at the time it was created. If looping wasn't set up when it was created, setting SPU_VOICE_LSAX and calling SpuSetVoiceAttr will have no effect. If you want to do looping, you must set up the looping when you create that waveform data.

Related functions: SpuSetVoiceAttr, SpuSetKey

Waveform transfer

Q-11: If I transfer data to the sound buffer with an I/O transfer, do I need to call SpuIsTransferCompleted to check for completion?

A-11: You do not need to use SpuIsTransferCompleted to check whether an I/O transfer has completed, because execution will return from SpuWrite when processing is complete.

If you call SpuIsTransferCompleted when the transfer mode is I/O transfer, SpuIsTransferCompleted will return 1 regardless of its argument.

Related functions: SpuIsTransferCompleted, SpuWrite, SpuRead, SpuSetTransferMode

Q-12 : Why doesn't execution return from SpuIsTransferCompleted if I call it to confirm completion after executing a DMA transfer?

A-12: The overall operation of the current library is such that you must execute either _96_remove() or CdInit() ahead of all other processing. If you do not, execution may not return, especially for SpuIsTransferCompleted (SPU_TRANSFER_WAIT).

Related functions: SpuIsTransferCompleted, SpuWrite, SpuRead, SpuSetTransferMode, _96_remove

Q-13: In libspu the transfer mode can be switched between I/O transfer and DMA transfer, but what happens in libsnd?

A-13: In libsnd the transfer mode defaults to DMA transfer mode. Also in libsnd the transfer mode can be changed using SpuSetTransferMode(). SpuSetTransferMode() should be called before SsVabTransBody()/SsVabTransBodyPartly().

Related functions:

SpuSetTransferMode, SsVabTransBody, SsVabTransBodyPartly

Q-14: In libspu a transfer completion callback function can be used, but can this function also be used in libsnd?

A-14: The transfer completion callback function capability currently provided in libspu cannot be used in libsnd. When using the transfer completion callback function in libspu and also before calling the wave pattern transfer function in (SsVabTransBody()/SsVabTransBodyPartly()) in libsnd make sure to set the transfer completion callback function to NULL.

```
(void)  SpuSetTransferCallback  ((SpuTransferCallbackProc) NULL);
            :
            SsVabTransBody (. . .);
```

Related functions:

SuSetTransferCallback, SsVabTransBody, SsVabTransBodyPartly

Basic Sound Library and Extended Sound Library Common Uses

Following are listed items which the present basic sound library (libspu) and extended sound library (libsnd) have in common.

Initialization

Since SsInit() internally calls SpuInit() in libsnd, it is not necessary to call SpuInit() when using SsInit()

Sequence Data

libspu has no function to handle sequence data such as SEQ. Therefore, it is necessary to use libsnd without fail when handling SEQ/SEP sequence data.

When creating an individual driver to analyze and generate sound sequence data using libspu it will be necessary to use the route counter and event processing functions provided in the kernel library (libapi) 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 libspu supports the data transfer to the sound buffer which does not obstruct VAB, when VAB has been fed to the sound buffer there is no function which will reflect that attribute data to the voice.

Therefore, the transfer uses SsVabOpen(), SsVabOpenHead(), SsVabTransBody() or SsVabTransBodyPartly() to transfer the wave pattern, to acquire header information, to select information with SsUtGetVabHdr(), SsUtGetProgAtr(), or SsUtGetVagAtr(), to find out what location in the sound buffer the wave pattern data has been transferred to and to set the voice attributes used in libspu with SpuSetVoiceAttr().

At this time, in terms of memory regulation, since libsnd also uses SpuMalloc() it can coexist with the SpuMalloc() called by the user (described later).

Furthermore, the default in libsnd is the DMA transfer mode. Even in libsnd the transfer mode can be changed using SpuSetTransferMode(). SpuSetTransferMode() should be called before SsVabTransBody()/SsVabTransBodyPartly().

As for the transfer completion decision, in libspu the transfer completion callback function can be used, but at present this transfer completion callback function capability cannot be used in libsnd. When using the transfer completion callback function in libspu and also before calling the wave pattern transfer function (SsVabTransBody()/SsVabTransBodyPartly()) in libsnd, the transfer completion callback function must be set to NULL. (Ref. FAQ Q/A-14)

Sound buffer memory control and reverb

In libsnd since SsInit() internally calls SpuInitMalloc(), there is no need to call SpuInitMalloc() when using SsInit.

However, that control block is taken in libsnd internal and the size is (32+1). That is to say the number of domains which can be simultaneously maintained by SpuMalloc() and SpuMallocWithStartAddr() is 32.

By making this SpuFree(), when the maximum maintained domain value is 32, the above maintain function can be called as many times as desired.

Furthermore, since every time VAB is opened SpuMalloc() is called once within the library, and every time it is closed internally SpuFree() is called once within the library, at a certain point in the above environment the user’s maintainable maximum number of domains will be (32-(at that point the frequency of VAB openings)).

If a frequency greater than this is desired for SpuMalloc()/SpuMallocWithStartAddr(), by immediately calling SpuInitMalloc() after SsInit() the memory control will function within the domain provided to SpuInitMalloc() by the control table. It will then be impossible to use the domain set in the library.

When replacing this control table it is necessary to provide the

SPU_MALLOC_RECSIZ x ((Frequency of VAB openings + frequency of SPUMalloc() calls by user) +1)

size domain to SpuInitMalloc().

Libsnd reverb has been provided using almost the same functions as in libspu. Therefore, SpuReserveReverbWorkArea() in libspu can be used in the same way.

Please refer to the libspu and libsnd function reference for these functions.

Chapter 15: Serial Input/Output Library

Connects the PlayStation to the PC and provides standard input/output functions between the two. Performs debug support during the outputting of debug information to the PC.

Table of Contents

Overview

The Serial input/output library connects the PlayStation communications connector and the PC RS232C port with the DTL-H3050 communications cable and provides standard input/output functions.

Library File

The standard input/output library file name is [libsio.lib].
Programs which access each service must link this without fail.

Driver and BIOS

The serial input/output library consists of the serial input/output driver and the serial input/output BIOS.

Serial input/output driver

The serial input/output driver provides standard input/output using the C-language standard procedures.

Inclusion and deletion

Inclusion of the serial input/output driver is performed in AddSIO( ). Likewise, deletion is performed in DelSIO( ).

BIOS

The serial input/output BIOS provides low-level driver control and information acquisition functions which cannot be covered by the C-language standard function specifications. The interface function is _sio_control( ). Also, BIOS will operate even when the serial input/output driver is not included.

```
Syntax:
long_sio_control (cmd,arg,param)
unsigned long cmd;
unsigned long arg;
unsigned long param;

Arguments:
cmd    command
arg    subcommand
param  argument
```

Table 15-1: Command Summary

cmd	arg	Function
0	0	Returns driver status (*1)
	1	Returns control line status (*2)
	2	Returns communications mode (*3)
	3	Returns communications speed (bps units)
	4	Returns received unit number of characters
1	0	System reservation
	1	Sets param value as control line status (*2)
	2	Sets param value as communications mode (*3)
	3	Sets param value as communications speed (bps units)
	4	Sets param value as received unit number of characters
2	0	Resets driver
	1	Clears driver status error-related bits

Table 15-2: Driver Status

bit	Contents
31-10	Undecided
9	1: interrupt on
8	1: CTS is on
7	1: DSR is on
6	Undecided
5	1: frame error occurs
4	1: overrun error occurs
3	1: parity error occurs
2	1: no communications data
1	1: able to read communications data
0	1: able to write communications data

Table 15-3: Control Line Status

bit	contents
31-2	undecided
1	1: RTS is on
0	1: DTR is on

Table 15-4: Communications Mode

bit	Contents	
31-8	undecided	
7,6	stop bit length	
	01: 1	
	10: 1.5	
	11: 2	
5	parity 2	1: odd 0: even
4	parity 1	1: exists
3,2	character length	
	00: 5 bit	
	01: 6	
	10: 7	
	11: 8	
1	Always 1	
0	Always 0	

Communications specifications

Communications specifications can be selected from the following settings:

Table 15-5: Communications Specifications

Item	Setting Value
Character length	5,6,7,8 bits
Stop bit	0,1,1.5,2 bits
Parity check	None, even, odd
Communications Speed	1-2073600 bps (2073600 approx. number only)

Default settings

The initialization which is performed in the serial input/output driver inclusion process is indicated below:

Table 15-6: Default Setting

Item	Setting Value
Character length	8 bit
Stop bit	1 bit
Parity check	None