# PlayStation®2 EE Library Overview
# Release 2.4

## Kernel Libraries

Publication date: October 2001

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

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

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

# Summary Table of Contents

# About This Manual

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

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

## Changes Since Last Release

### Chapter 1: EE Kernel

- In the "Setting Up the Stack Area/Heap Area" section of "Description", an error with _stack in the table has been corrected as follows:

  Incorrect : _stack_size + 0x100(4Kb)

  Correct : _stack_size + 0x1000(4Kb)

### Chapter 2: Standard I/O Service Functions

- In the "Specifying FIles" section of "Library Overview", descriptions of valid devices have been added.

## Related Documentation

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

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

## Typographic Conventions

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

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

## Developer Support

### Sony Computer Entertainment America (SCEA)

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

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

### Sony Computer Entertainment Europe (SCEE)

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

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

# Chapter 1: EE Kernel

# Library Overview

The EE kernel is the operating system running on the EE. The kernel runs in an address space separate from applications and provides system calls for priority-based multi-thread scheduling, inter-thread communications using semaphores, exception handling, C0P0 control, etc.

I/O services for controllers and host files are provided by EE kernel extensions.

## Related Files

The following file is needed to use the EE Kernel API.

**Table 1-1**

| Category | Filename |
|---|---|
| header file | eekernel.h |

# Description

## Threads Overview

Threads serve as the logical unit of programs, as seen from the perspective of parallel processing. The processing performed by a single application program is divided into multiple threads, and these threads are run simultaneously and in parallel. Since there is only one CPU, however, only one thread will be running during any given time slice. Which thread to run is determined by the priority level of the individual thread.

Priority levels are absolute, so if a thread with high priority is running, a thread with lower priority will not be executed until the high-priority thread enters a wait state or its state is changed by an interrupt handler. This is in contrast to the method used in TSS (Time Sharing System), where multiple tasks (or processes) are switched to provide even execution.

Threads are executed in parallel, but adjustments must be made so that resources such as I/O devices and specific work areas in memory can be used exclusively by threads. The EE kernel provides semaphores for synchronization between threads.

A clear distinction is made between threads and interrupt handlers. Threads are executed as part of user programs, but interrupt handlers are executed as system programs (even when user-created).

## Thread States and Operations

Threads managed by the EE kernel have the following seven (or five, depending on how broadly they are categorized) states.

**Table 1-2**

| State | Description |
|---|---|
| RUN | Running state:<br>CPU is executing the thread. |
| READY | Ready for execution:<br>Thread is in stand-by as the CPU is executing another thread. |
| WAIT | Wait state:<br>The thread has put itself in this state until a certain condition is met. |
| SUSPEND | Forced wait state:<br>The thread has been forced into a waiting state by a system call issued by another thread. |
| WAIT-SUSPEND | Double-wait state:<br>The thread was forced into a waiting state by another state while it was in WAIT state. |
| DORMANT | Thread is sleeping:<br>The thread has been generated but has not been activated yet, or the thread has terminated but has not yet been deleted. |
| NON-EXISTENT | Thread is not registered:<br>An imaginary state for a thread that has not been generated or that has already been deleted. |

- **RUN: Running state**

  Indicates that the CPU is running the thread. At any given moment, only one thread will be in this state.

- **READY: Ready for execution**

  The thread meets all conditions for execution but is on stand-by because a thread with higher (or the same) priority is running. If there are multiple threads waiting for execution, these threads form a queue to wait for CPU availability. This is referred to as a Ready Queue.

- **WAIT: Wait state**

- **SUSPEND: Forced wait state**

- **WAIT-SUSPEND: Double-wait state**

  The thread is stopped since it does not meet conditions for execution. WAIT state occurs when execution is stopped due to a system call issued by the thread itself. SUSPEND state occurs when a thread has been forced into a waiting state by another thread. WAIT-SUSPEND state occurs when a thread is forced into a waiting state by another state when it was already in WAIT state.

  When execution is resumed after a waiting state, execution picks up from where it was interrupted, and information representing the execution state of the program (context) such as the program counter and the registers are all restored to their original values.

- **DORMANT: Sleeping state**

  This state occurs when the thread has not been activated yet or after it has finished. This state is enabled right after a thread is created.

- **NON-EXISTENT: Unregistered state**

  An imaginary state that occurs before a thread is created or after it is deleted. A thread goes into DORMANT state once it is created, then enters READY state when it is activated and put in the Ready Queue. The Ready Queue is ordered by priority levels, with threads having identical priority levels being queued in the order they entered READY state.

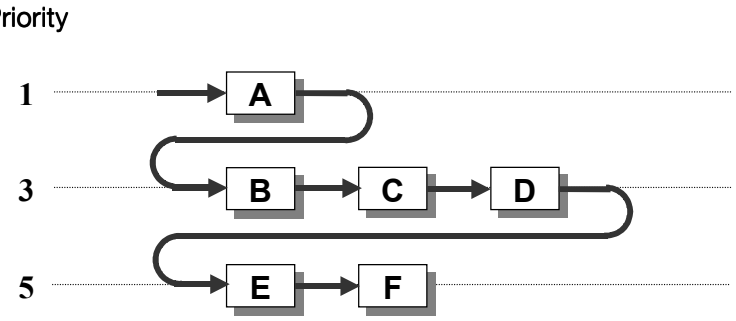
  Execution is performed for the thread at the head of the Ready Queue, with the thread being placed in RUN state. The thread in RUN state enters WAIT state when it waits for a semaphore or an event flag. When the wait conditions are met, the thread goes back to READY state.

## Thread Scheduling (Priority Levels)

Threads are managed in the Ready Queue by priority levels and execution is scheduled based on the Ready Queue.

In the following example, the Ready Queue contains a thread A with priority level 1, threads B, C, D with priority level 3, and threads E, F with priority level 5.

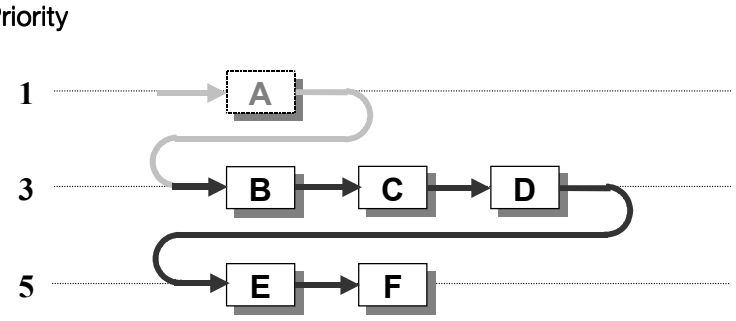
**Figure 1-1**




The thread with the highest priority in the Ready Queue is thread A, so thread A is executed (the thread being executed is considered to still be a part of the Ready Queue). While thread A is running, no other threads will be executed.

When the thread A calls SleepThread() and enters WAIT state, the Ready Queue will be as shown below and thread B will be executed.

**Figure 1-2**




If an interrupt is generated in this state, control proceeds to the interrupt handler and scheduling is performed again when control returns from the interrupt handler. However, execution privilege will not be taken from thread B just because an interrupt takes place. Execution of thread B will resume after the interrupt handler is finished unless the interrupt handler moves thread A to an executable state or lowers the priority of thread B.

If thread A is put in an executable state by the interrupt handler, thread A will be executable when the interrupt handler is finished, and thread B will move from RUN state to READY state. This is referred to as "thread B being preempted". In this case, the position of thread B in the Ready Queue remains unchanged, so if thread A is removed from the Ready Queue, thread B will be executed.

Thread priority levels are set to 1 when a user program is started and can be changed to 1-127 using ChangeThreadPriority() from the kernel API. The ordering of threads having the same priority level can be changed by rotating the Ready Queue using RotateReadyQueue(). For example, if RotateReadyQueue() is used for priority level 3, the leading thread will be moved to the end and the ordering will be C, D, B.

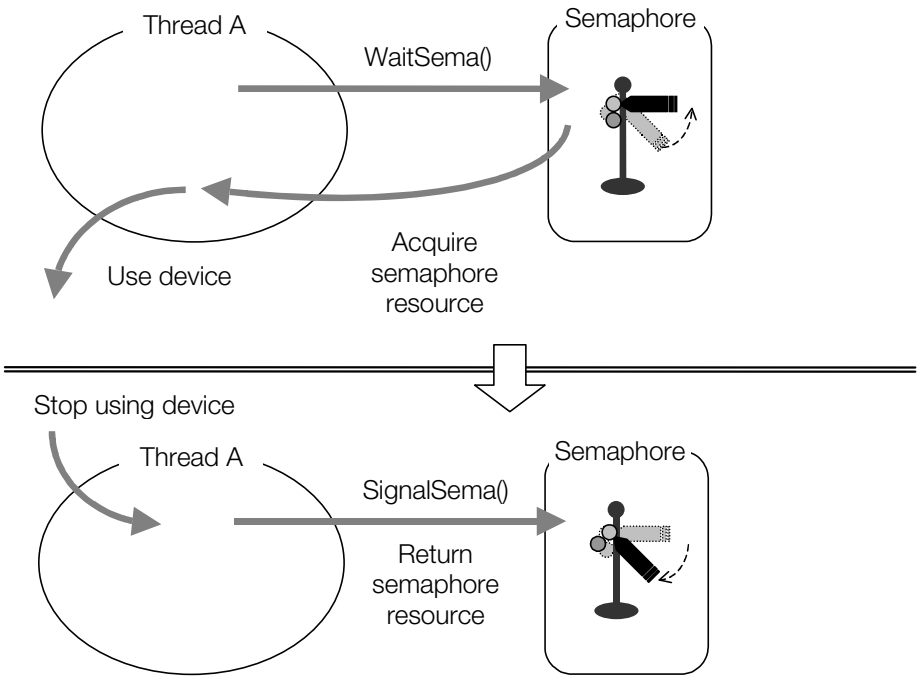
When a thread waiting to be executed is executed, the thread will be placed at the end of the Ready Queue for that priority level.

## Semaphores

When multiple threads are using the same device or buffer areas, the semaphore (signaling) feature allows other threads to be notified of the usage status in order to provide synchronization.

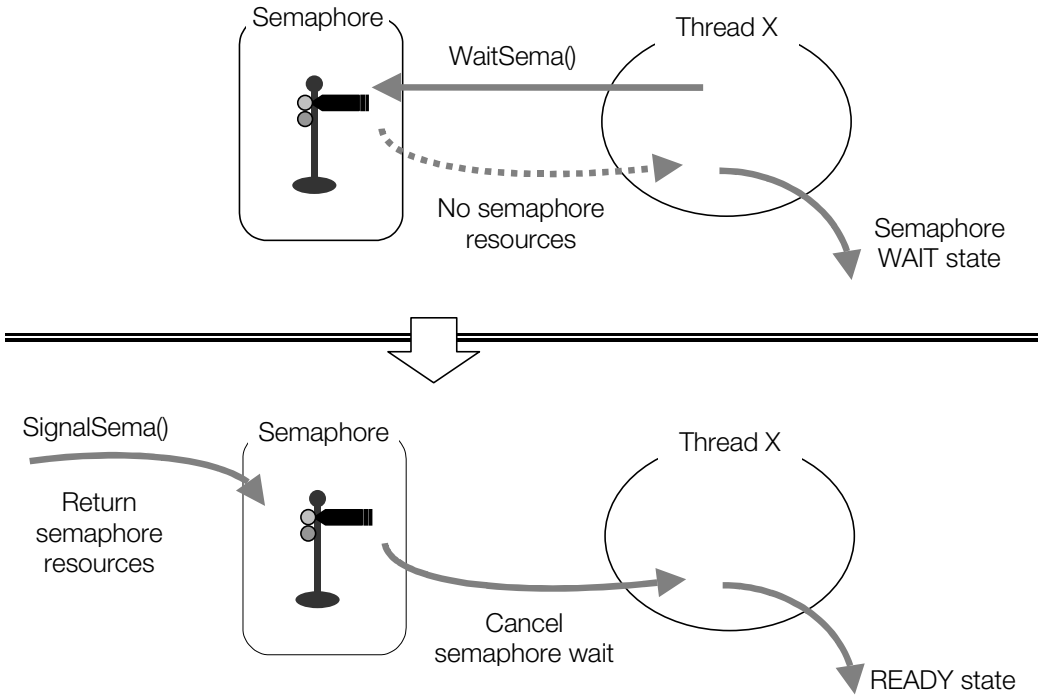
Threads that want to use a device, etc. must acquire the semaphore resource by executing WaitSema(), i.e., usage privilege is obtained by setting an in-use flag. Once the thread is done using the resource, SignalSema() is executed and the semaphore resource is returned to the system.

**Figure 1-3**




If another thread is using the device, etc. when WaitSema() is executed, the semaphore resource will not be available, causing the thread to go into WAIT state and enter a semaphore waiting queue. When the thread using the device, etc. executes SignalSema() and returns the semaphore resource to the system after it is done, the returned semaphore resource will be given to the thread at the head of the semaphore waiting queue. The thread will return to READY state and will be executed according to the priority-based scheduling, thus allowing use of the device, etc.

**Figure 1-4**




The number of semaphore resources can be specified when a semaphore is created. Thus, in addition to the exclusion controls described above, semaphores can also be used to indicate the effective data count in data buffers shared by multiple threads. When a thread supplies data that is stored in a shared buffer, SignalSema() is executed to increase the effective data count. When a thread retrieving the data tries to take data from the shared buffer, WaitSema() is executed. If the semaphore resource can be retrieved, the data in the shared buffer is processed. If the resource cannot be retrieved, the thread is put in WAIT state and waits for data to be provided.

## Interrupt Handling

The EE includes two interrupt controllers: INTC and DMAC:

**Table 1-3**

| Interrupt Controller | Interrupt Trigger |
|---|---|
| INTC | GS |
| INTC | SBUS |
| INTC | V-blank start |
| INTC | V-blank end |
| INTC | VIF0 |
| INTC | VIF1 |
| INTC | VU0 |
| INTC | VU1 |
| INTC | IPU |
| INTC | Timer 0 |
| INTC | Timer 1 |
| DMAC | DMA channel ch-0 (VIF0,DIR:to,GP:A) |
| DMAC | DMA channel ch-1 (VIF1,DIR:both,GP:C) |
| DMAC | DMA channel ch-2 (GIF,DIR:to,GP:C) |
| DMAC | DMA channel ch-3 (IPU,DIR:from,GP:C) |
| DMAC | DMA channel ch-4 (IPU,DIR:to,GP:C) |
| DMAC | DMA channel ch-8 (SPR,DIR:from,GP:C) |
| DMAC | DMA channel ch-9 (SPR,DIR:to,GP:C) |

The EE kernel allows interrupt handlers to be registered for individual interrupt triggers. If an application is to perform an operation when it receives an interrupt, the operation should be prepared as a function and the function should be registered as an interrupt handler. Multiple interrupt handlers can be registered for a single interrupt trigger. In such cases, the order in which the interrupt handlers are called can be specified.

Before a DMAC interrupt calls a registered handler, the D_STAT.CIS that corresponds to the channel that generated the interrupt is cleared.

Sample code demonstrating the use of interrupts is provided in sce/ee/sample/mpeg/ezmpeg and sce/ee/sample/ipu/ezcube3.

## Alarm Feature Overview

The EE kernel provides 64 timers that work off of Hsyncs. SetAlarm() can be used to specify an interrupt handler for the timer and the time at which the interrupt handler is called. When the specified time elapses, the interrupt handler is called.

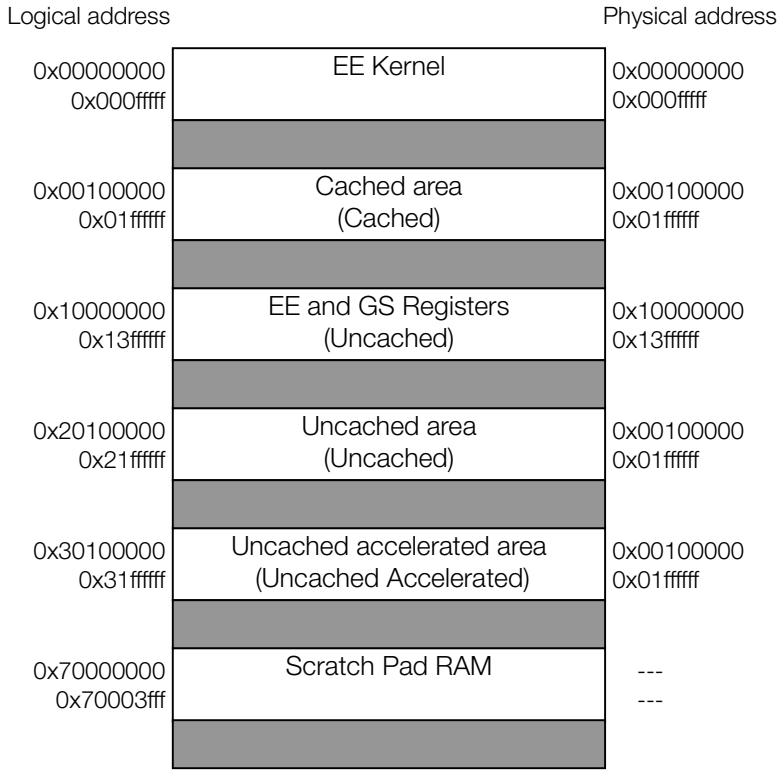
Calling ReleaseAlarm() will release the specified interrupt handler.

## Memory Map

The memory map for the EE kernel is shown below.

The range 0x00000000 - 0x000fffff is the space used by the EE kernel and cannot be accessed by the application program.

**Figure 1-5**




### EE Kernel

The range 0x00000000 - 0x000fffff is the space used by the EE kernel and cannot be accessed by application programs.

### Uncached

- Reads do not go through the cache.
- Writes go through the write back buffer.
  (Continuous stores of 128 bytes or less are done all at once)

### Uncached Accelerated

- Reads goes through the uncached accelerated buffer
  (128 bytes read all at once)
- Writes go through the write back buffer
  (Continuous stores of 128 bytes or less are done all at once)

Uncached accelerated provides higher speeds for accesses to contiguous memory without affecting the cache.

Uncached accelerated is better than Uncached for accesses to contiguous memory, but will be slower for random accesses to memory.

## Setting Up the Stack Area/Heap Area

A linker script file (.cmd) sets up the memory map for an applicationprogram, e.g., start address (entry point), stack area, heap area. Memory is allocated in a start-up routine (crt0.s) based on the settings in the linker script file.

In the sample programs, etc., the linker script file /usr/local/sce/ee/lib/app.cmd is used. The following are the main items that are set up.

- Information about the stack area (_stack / _stack_size)
- Information about the heap area (_heap_size)
- Entry point (0x00100000)
- Information about the location and structure of each section
- End of the user program (end, _end)

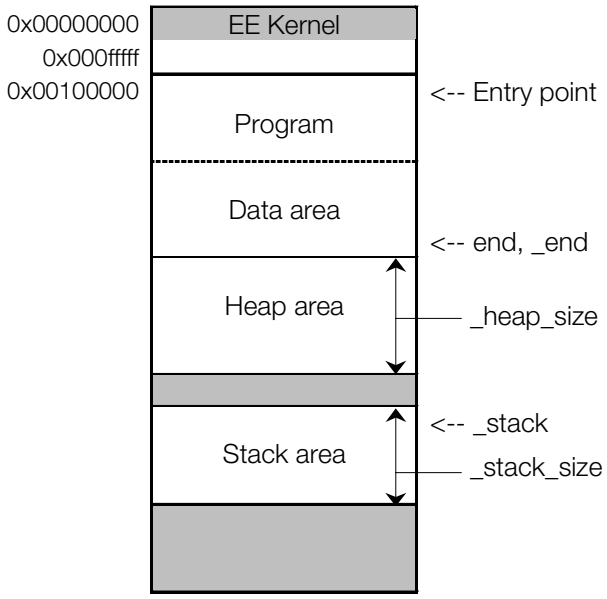
The stack area/heap area is determined by _stack, _stack_size, _end, _heap_size in the following manner.

**Table 1-4**

| Symbol | Description |
|---|---|
| _stack | Starting address of stack area<br>Specifying -1 indicates an address where _stack_size + 0x1000(4Kb) is subtracted from the final address in memory. The final 4 Kbytes are used as a system debugging work area. This area will be present only if -1 is specified. |
| _stack_size | Size of stack area |
| _end | End address of the program area/address directly before the heap area |
| _heap_size | Size of the heap area |

The stack pointer goes from the last address of the stack area and moves toward the start.

**Figure 1-6**




/usr/local/sce/ee/lib/app.cmd can be copied to a local directory and customized. If this is done, set up an LCFILE for the Makefile in the local directory so that the updated app.cmd will be linked.

### Sample Settings

```
_stack_size = 0x00200000;
_stack = 0x1d00000;
_heap_size = 0x00080000;
```

In this case, the stack area is 0x01d00000 - 0x01effffff, and the heap area is (_end) - (_end+0x7ffff).

## Cache Management Overview

The EE data cache is a write-back cache. This means operations are required that explicitly write back to memory. The EE kernel provides a system call for writing back data in the data cache, as shown below.

**Table 1-5**

| System Call | Description |
|---|---|
| FlushCache() / iFlushCache() | Write back the entire data cache |
| SyncDCache() / iSyncDCache() | Write back a specified area |

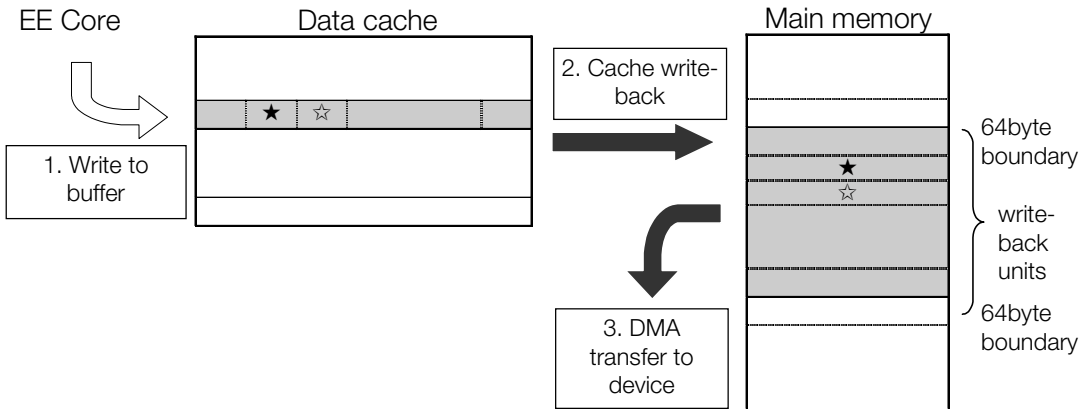
### DMA Transfers and Cache Management

When a buffer variable is set up in a cached region and DMA transfer is performed, the data cache must be written back. The write-back timing depends on the direction of the DMA transfer, as shown below.

DMA transfers from memory to peripherals:

1. Data is written to buffer variable (written to cache)
2. Cache is written back to memory
3. DMA transfer is performed

**Figure 1-7**

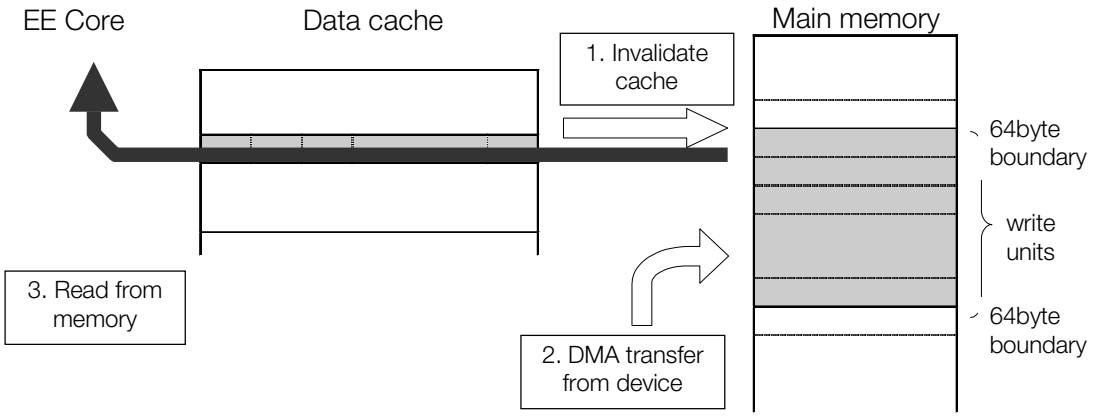



If a write-back is not performed at 2, then the data transferred to the peripheral at 3 would be data that was previously in memory rather than the data written to the buffer at 1.

DMA transfers from peripherals to memory:

1. Write back cache (invalidate cache)
2. Perform DMA transfer
3. Read data from memory

**Figure 1-8**




If the cache is not invalidated at 1, 3 would end up reading old data in the cache rather than the data that was transferred from the peripheral at 2.

If buffer variable read/write operations are performed through an uncached area, the cache write-back operations described above are not needed.

### Cache Management and Alignment

Write-backs from cache to memory are performed in 64-byte units (the size of the cache line). When other data is present in the same 64-byte boundary as the data to be written back, this data will be written back as well. This can lead to unexpected behavior, so make sure 64-byte alignment is performed on the data.

## File Operations Overview

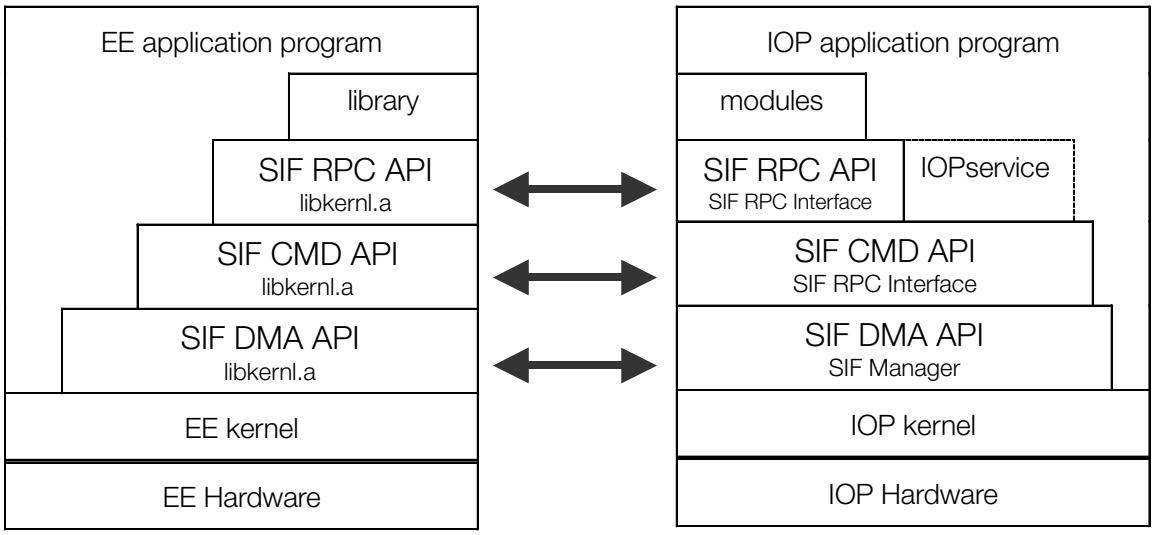
I/O services for files on the CD/DVD-ROM and files on development computers are implemented using the SIF RPC Interface, which is an extension to the EE kernel. For more information, please see the "I/O services" document.

## IOP Control Overview

The kernel provides a function to reboot the IOP and a function to wait for start-up after rebooting. This provides an API so the IOP can be controlled from an EE application, When the IOP is rebooted, it can swap, load and run a default module from ROM.

Also, the three-level API shown in the figure is provided to support operations where the EE and the IOP are working together.

**Figure 1-9**




For more information, see the "SIF system" document.

### Virtual Expansion of Scratch Pad

A section of main memory is mapped to a logical address right after the scratch pad (0x70000000 - 0x70003fff) so that the scratch pad area appears larger. However, DMA transfer cannot be performed on the extended area.

**Table 1-6**

| System Call | Description |
|---|---|
| ExpandScratchPad() | Virtual expansion of scratch pad |

## Notes

### Reserved Timers

Of the four timers in the EE, TIMER2 and TIMER3 are reserved by the EE kernel. These should not be used by applications.

### Saving Registers

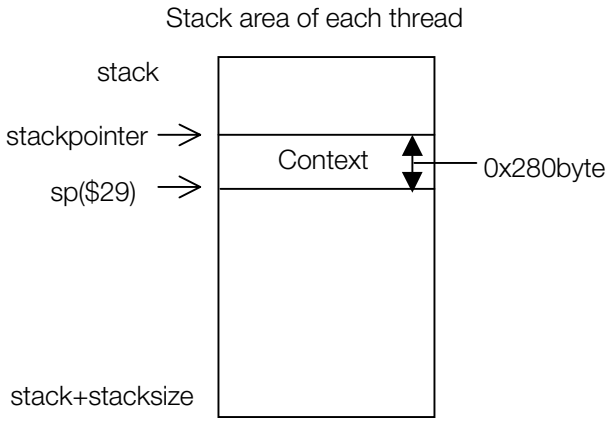
When interrupts take place or threads are switched, registers are saved by the kernel as shown below:

**Table 1-7**

| Registers | Interrupt | Thread-switching |
|---|---|---|
| $1 - $25 | Saved | Saved |
| $28 - $31 | Saved | Saved |
| HI/LO/SA | Saved | Saved |
| $f0 - $f31 | Not saved | Saved |

When a thread is switched, the context of that thread is saved on the thread stack. Then, the context of the switched thread is restored and execution continued.

**Figure 10**




The general-purpose registers $26/$27 are reserved by the EE kernel. These cannot be used by applications.

The VU0 register can be accessed through co-processor instructions but the EE kernel does not manage this register.

## Interrupt Handler Descriptions

User-created interrupt processing routines (callback functions) can be registered for various interrupts. These callback functions must be programmed with an awareness of the conditions under which they will be executed. In many cases, callback functions will be executed as interrupt handlers, but in some cases they may be executed as threads, depending on the library functions and system calls used to register the callbacks.

Interrupt handlers and threads are executed under different system conditions. When programming interrupt handlers, the following points must be noted.

### API Differences

In interrupt handlers, there is no notion of the handler's "own thread". Thus, interrupt handlers in principle cannot call system calls that put it in a waiting state or system calls that implicitly specify its own thread.

Also, due to the implementation, system calls called from threads and system calls called from interrupt handlers are sometimes handled differently within the kernel.

In the EE-Kernel, the number of system calls that can be called from an interrupt handler is restricted, and entry names are provided that are distinct from those of the system calls called from threads. As a rule, system calls called from interrupt handlers begin with the lower-case letter "i". For example, the WakeupThread system call used to wake up a thread will be called as iWakeupThread when called from an interrupt handler.

### Stack Differences

Unlike standard threads, an interrupt handler has a dedicated stack. The stack size is 4 KB. If a larger stack needs to be used, the user program should allocate memory.

### Exit Operations

Call ExitHandler() when an interrupt handler finishes and exits back to normal operations. ExitHandler() writes all the instructions saved in the EE's write-back buffer.

## Interrupt Management with DI / EI

Since the EE kernel implements the disabling and enabling of interrupts with lightweight operations, information about whether interrupts are disabled or not is not included in the thread context. Thus, problems can occur if threads are switched when interrupts are disabled or when nested DIs/EIs are used.

Frequent use of DI/EI in applications should be avoided and should only be used when atomically accessing register groups mapped in memory. Semaphores should be used for providing exclusivity in software-based resources such as shared buffers, etc.

# Chapter 2:
# Standard I/O Service Functions

# Library Overview

I/O services for files on the CD(DVD)-ROM and files on development computers are implemented as extensions to the EE Kernel using SIF RPC.

In addition to standard APIs such as open/close, read/write, seek, ioctl, the library provides a simplified printf function and a function to reset SIF RPC binding information.

Refer to the Hard Disk Library (for EE) or the PlayStation File System (for EE) documentation for details on the hard disk drive I/O services.

## Related Files

The header file shown below is required to use the I/O services API.

**Table 2-1**

| Category | Filename |
|---|---|
| Header file | sifdev.h |

## Specifying Files

Files are specified by the filename that was used when the file was opened. The filename is a string conforming to the format shown below with a maximum length of 1024 bytes.

Device name: path name

The device name indicates a device on which the file exists. The following is a list of currently valid device names.

**Table 2-2**

| Device name | Device |
|---|---|
| host0 | Development computer disk.<br>Operations performed via dsedb/dsefilesv |
| host | (same as above) |
| host1 | Development computer disk.<br>Operations performed via dsidb/dsifilesv |
| cdrom0 | CD/DVD-ROM |
| hdd0 | Hard disk drive |
| pfs0 | PlayStation File System |

The path name is a string indicating the location of the file on the device. Path names have the following restrictions based on the device.

**Table 2-3**

| Device | Path name restrictions |
|---|---|
| host0/host1 | String based on Linux filename restrictions |
| cdrom0 | String based on ISO-9660 restrictions |

Examples of filenames are shown below.

- **host0:/usr/local/sce/iop/modules/fileio.irx**

(/usr/local/sce/iop/modules/fileio.irx on the host machine)

- **cdrom0:\\SYSTEM.CNF;1**
  (SYSTEM.CNF from the top-level directory on the CD/DVD-ROM)