PlayStation®2 IOP Library Overview
Release 2.4

Kernel Libraries

© 2001 Sony Computer Entertainment Inc.

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.

The *PlayStation®2 IOP Library Overview - Kernel Libraries* manual is supplied pursuant to and subject to the terms of the Sony Computer Entertainment PlayStation® license agreements.

The *PlayStation®2 IOP Library Overview - Kernel Libraries* manual is intended for distribution to and use by only Sony Computer Entertainment licensed Developers and Publishers in accordance with the PlayStation® license agreements.

Unauthorized reproduction, distribution, lending, rental or disclosure to any third party, in whole or in part, of this book is expressly prohibited by law and by the terms of the Sony Computer Entertainment PlayStation® license agreements.

Ownership of the physical property of the book is retained by and reserved by Sony Computer Entertainment. Alteration to or deletion, in whole or in part, of the book, its presentation, or its contents is prohibited.

The information in the *PlayStation®2 IOP Library Overview - Kernel Libraries* manual is subject to change without notice. The content of this book is Confidential Information of Sony Computer Entertainment.

and PlayStation are registered trademarks of Sony Computer Entertainment Inc. All other trademarks are property of their respective owners and/or their licensors.

Summary Table of Contents

About This Manual

This manual is the 2.3 release of the *PlayStation®2 IOP Library Overview - Kernel Libraries.*

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

Changes Since Last Release

Chapter 1: IOP Programming

- The "Types of Modules" section has been deleted from "Overview".
- A new “Lifetime of Modules (Resident / Non-resident)” section has been added in "Overview".
- In the "Entry Routine" section of "Module Loading and Starting", the skeleton description of the program with resident modules has been changed.
- In the "Stop and Unload Procedures" section of "Stopping and Unloading Modules", the skeleton description of the program with unloadable resident modules has been changed.
- In the "Self-stop and Self-unload" section of "Stopping and Unloading Modules", the skeleton description of the program with self-unloadable resident modules has been changed.
- The description of " Resident Library Organization" in "Resident Libraries" has been changed.
- Descriptions of reservation entry and exit process have been added to the item of "Entry Table Structure and Library Entry Definition File" of "5. Resident Library".
- A new “Skeleton” section was added to "Resident Libraries".
- In the "Entry Table Structure" section of " Reference: Linking Between Modules ", the description of reservation entry has been deleted.

Related Documentation

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

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

Typographic Conventions

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

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

Developer Support

Sony Computer Entertainment America (SCEA)

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

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

Sony Computer Entertainment Europe (SCEE)

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

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

Chapter 1: IOP Programming

Overview

Modules

More than one program can be loaded into IOP memory.

Programs are stored as relocatable object files in the ROM or on a CD/DVD-ROM and are loaded into available memory areas as required, then relocated to the load address and executed.

These IOP object files always have a ".irx" file extension, and are also referred to as IRX files. A program that has been loaded into memory from an IRX file and relocated is known as a program module, or simply a module. Every module has a single entry routine (usually the start function).

When a module is loaded, its entry routine is called and the return value determines whether the module will remain resident in memory.

Lifetime of Modules (Resident / Non-resident)

During the time that they are loaded in memory, modules can be classified into different types.

One type is a module whose entry routine is executed only once; after its purpose has been fulfilled, it is deleted from memory. This type of module is referred to as a "non-resident module."

Another type of module, the resident module, remains in memory even after its entry routine has completed execution. This type does its processing when there is a request from another module or an interrupt.

Modules containing entries that are called as subroutines from other modules are referred to as "resident libraries."

Some resident modules can be deleted from memory when they are no longer needed; these are referred to as "unloadable resident modules."

Program Loader

The program loader handles the loading and relocating of program modules from CD/DVD-ROMs, ROM, etc, and manages modules in memory. The program loader consists of a lower-layer module manager and a higher-layer file loader.

The module manager does the following:

- Examines IRX-formatted object data provided from the file loader, and relocates the object.
- Performs linking between loaded modules.
- Registers and deletes entry tables of resident-library functions.

The file loader does the following:

- Loads and starts object files from disk.
- Registers and deletes modules.

Module Structure

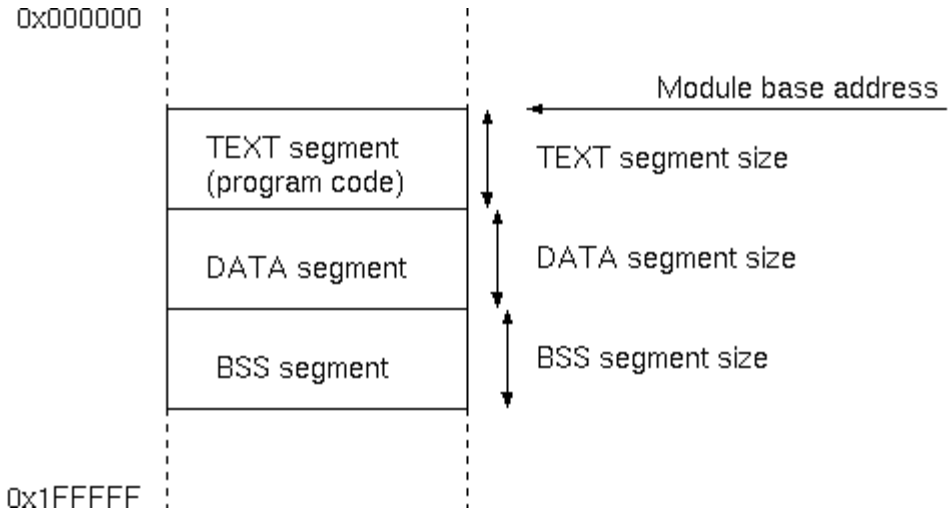
Location of Segments

A single module in IOP memory comprises the following three segments, each located in sequence, as shown in the figure below. Each has a size that is a multiple of 16 bytes.

Table 1-1

Segment	Contents
TEXT segment	Program code (required) Includes instruction code and data structures used for linking between modules.
DATA segment	Initial data values (may be omitted)
BSS segment	Uninitialized data area (may be omitted)

Figure 1-1



Module ID

Modules loaded into memory are assigned an ID number for identification purposes. This ID number is referred to as the "module ID."

Module Names and Module Versions

Names and versions can be assigned to modules. To do so, include the following declaration in the module source program.

```
ModuleInfo Module = {"Module_name", version number};
```

The global variable name Module is reserved for the purposes of indicating the module name and version. The structure of ModuleInfo is as follows, and is defined in kernel.h.

```
typedef struct _moduleinfo {
      const char          *name;
      const unsigned short version;
```

```
} ModuleInfo;
```

The name field specifies the module name. We suggest that the characters used for the module name be limited to alphabetic, numeric, "_", and "/".

The module's major version number is specified in the upper 8 bits of the version field and its minor version number is specified in the lower 8 bits. Both the major and minor numbers must be 1 or greater. Zero should not be used.

Some of the main uses for module names and versions are as follows:

- So that the boot loader can detect replacement modules at system boot-up.
- So that programs (modules) can acquire module IDs of modules with known names.
- To ascertain the load addresses of modules during program debugging.
- So that the debugger can confirm that booted modules are identical to the object files on disk.

Note that the IOP program loader has no problem loading and executing programs that lack module names and version. Also note that it is allowable to have more than one module with the same module name and version present in memory.

Module Loading and Starting

As directed during IOP system boot or by LoadStartModule() API, etc, the program loader loads the IOP object file and calls the module's entry routine. The value returned by the entry routine determines whether or not that module will remain resident in memory or be deleted.

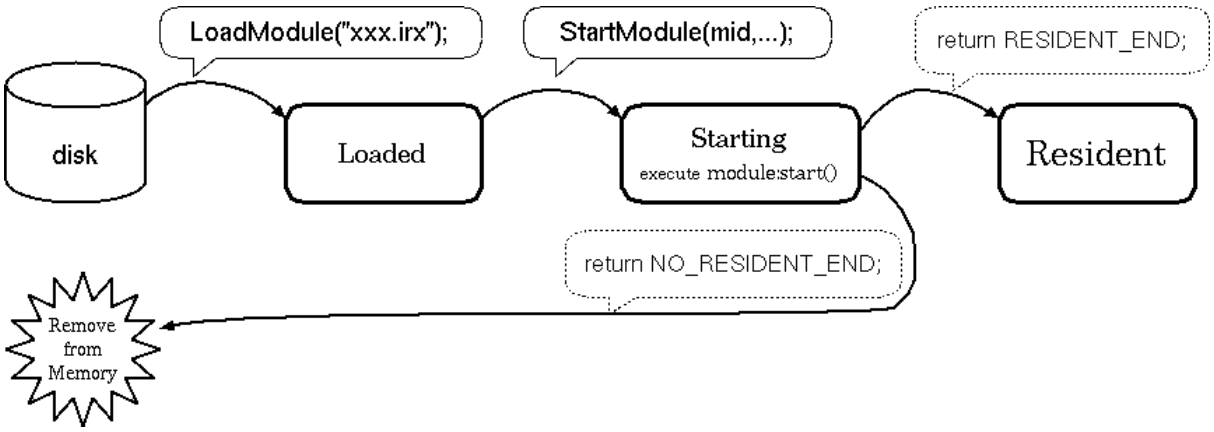
Load and Start-up Procedure

Loading a module and calling its entry routine involves the following steps:

1. Search for object data with an IOP object file format; if necessary, allocate memory, and after reading in the object, relocate it.
2. Zero-out the module's BSS segment.
3. Examine the module's TEXT segment and link if any resident-library functions are being called.
4. Allocate a module ID to the module.
5. Call the module's entry routine.
6. Determine whether the module is resident or not based on the value returned by the entry routine and perform processing accordingly.

A module undergoes the state transitions illustrated below in the course of the above procedure.

Figure 1-2



Entry Routine

A module's entry routine must be based on the following prototype:

```
int start(intc argc, char *argv[]);
```

The program loader activates a thread allocated for executing the entry routine; that thread calls the entry routine through the following procedure:

1. Push the argv[] array and argument strings onto the stack.
2. Set the module's global pointer value to the GP register ($28).
3. Issue a subroutine call of the entry routine.
4. (Module's entry routine executes.)
5. Based on the lower 2 bits of the value returned by the entry routine, determine whether the module is resident or non-resident.
6. Thread complete.

Module programmers must arrange for entry routines to be run in the following environment:

- Stack size for execution threads: 2 KB.
- Execution thread priority: MODULE_INIT_PRIORITY (=8)
- argc: same as general C-language main function arguments (one or more).
- argv: same as general C-language main function arguments

Because only one thread is allocated for executing entry routines, as a general rule, it is not possible to execute entry routines from different modules in parallel. So it is not necessary to take into account conflicts with entry routines from other modules with the entry routine of an ordinary module. In rare circumstances, one entry routine may call LoadStartModule() or a related API to start up another module. When this happens, the execution of entry routines is nested, and as a result, the usable stack size is reduced.

The lower 2 bits of the entry routine's returned value must always be as described below. Other bits can be freely used by the programmer.

Table 1-2

Value	Meaning
NO_RESIDENT_END	Remove module from memory (non-resident module)
RESIDENT_END	Make module resident in memory (resident module)
REMOVABLE_RESIDENT_END	Make module resident in memory (unloadable resident module)

Program skeleton for a non-resident module:

```
#include <kernel.h>

int start(int argc, char *argv[])
{
       /* perform operations by the program */
       return NO_RESIDENT_END;
}
```

Program skeleton for a resident module:

```
#include <kernel.h>

int start(int argc, char *argv[])
{
/* perform required initialization processing after module is
   made resident */
if( initialization was successful )
       return RESIDENT_END;
else
       return NO_RESIDENT_END;
}
```

Stopping and Unloading Modules

Resident modules whose entry routines return REMOVABLE_RESIDENT_END are recognized by the program loader as unloadable resident modules, and can be deleted when they are no longer resident using the StopModule() and UnloadModule() APIs.

Stop and Unload Procedures

The program loader allocates the memory area that will store the module itself when the resident module is loaded. When the module is unloaded, it frees that memory area. If there are any other resources that were allocated in the course of the module's execution (memory, threads, event flags, semaphores, etc), because these fall outside the program loader's control, they must be freed by the resident module itself.

As such, unloading a module is a two-step process. First, a stopping process is performed by the module that frees resources. Then the module manager deletes the resident module from memory.

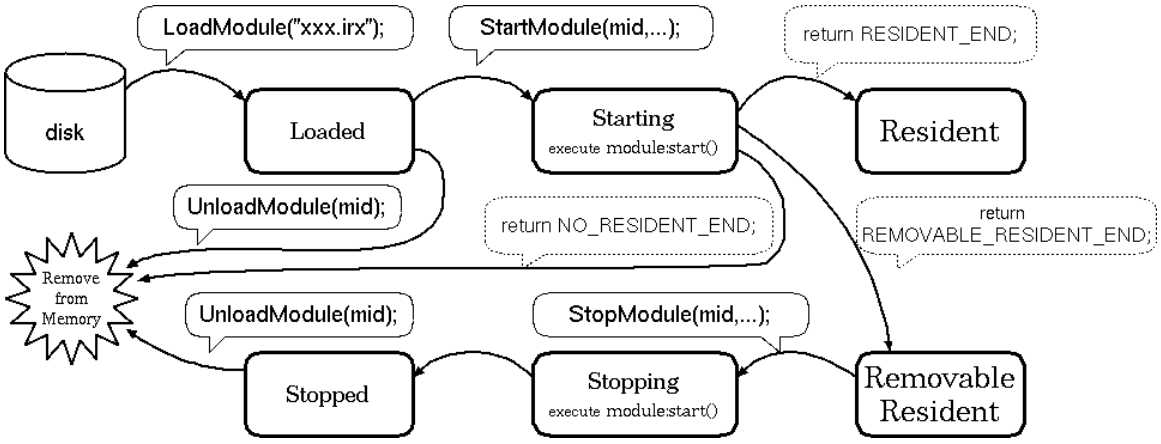
A resident module is stopped by calling the StopModule() API. When executing StopModule(), the program loader activates a thread allocated for executing the entry routine, then calls the entry routine with the following procedure:

1. Push the argv[] array and argument strings on the stack. argv[0] must always be set to the string "other". Also, to indicate that this is a stop process being called, the argument argc must be set to a negative value (the absolute value is the number of argument strings passed by argv[]).
2. Set the module's global pointer value to the GP register ($28).
3. Issue a subroutine call to the entry routine.
4. (Module's entry routine executes. Since the argument argc is negative, free resources, etc.)
5. If the lower 2 bits of the value returned by the entry routine are NO_RESIDENT_END, then the stop process succeeded, and the module is considered to be in the stopped state. If the return value is REMOVABLE_RESIDENT_END, the stop process failed, and the module is considered to still be in an operating state.
6. Thread complete.

Resident modules are deleted using the UnloadModule() API. UnloadModule() makes sure that the target module is in a stopped state, then deletes the module's registration. Memory that had been automatically allocated for the target module at load time is now freed.

The module undergoes the state transitions illustrated below in the course of the above procedure.

Figure 1-3



Program skeleton for an unloadable resident module:

```
#include <kernel.h>

int module_start(int argc, char *argv[]) {
     /* perform required initialization processing after module
       is made resident */

     if( initialization was successful )
            return REMOVABLE_RESIDENT_END;
     else {
            /* If necessary, cancel initialization processing */
            return NO_RESIDENT_END;
     }
}

int module_stop(int argc, char *argv[]) {
     /* Before deleting module, perform required termination
processing */
     if( termination processing was successful )
            return NO_RESIDENT_END;
     else {
            /* If possible, cancel termination processing and return
```

```
                * to a state in which module can stay resident */
                return REMOVABLE_RESIDENT_END;
        }
 }

 int start(int argc, char *argv[]) {
        if( argc >= 0 )    return module_start(argc, argv);
        else               return module_stop(-argc, argv);
}
```

Self-stop and Self-unload

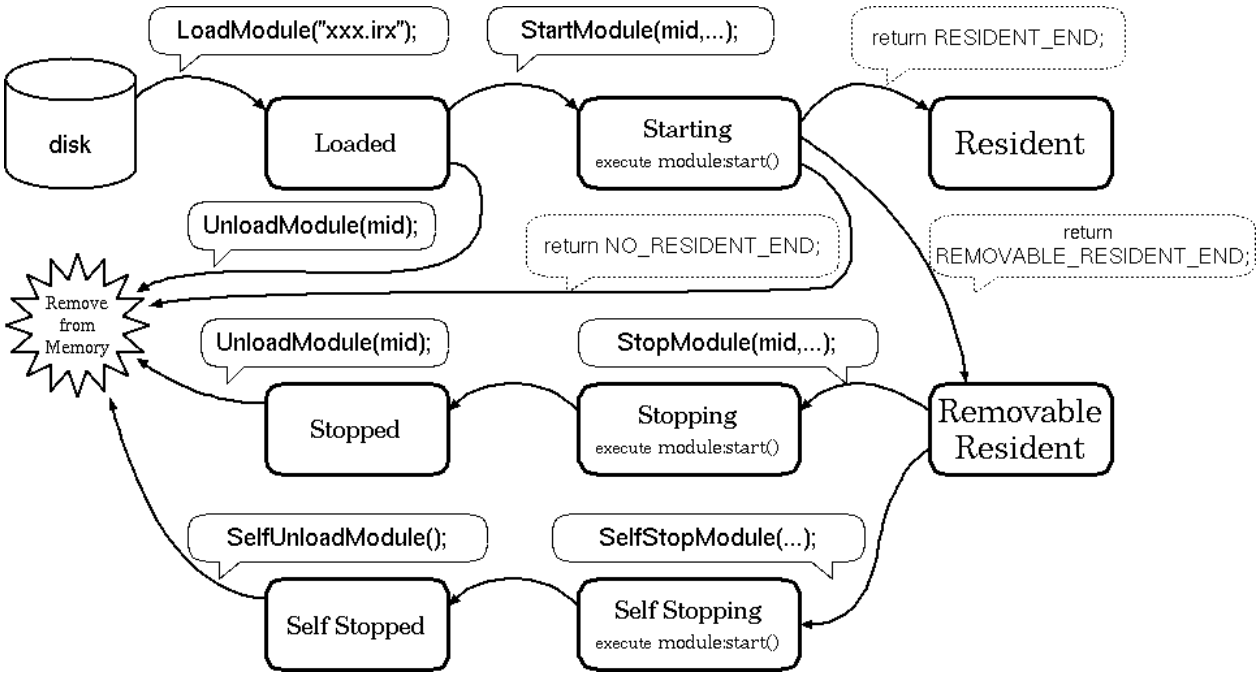
Modules cannot call StopModule() and UnloadModule() themselves. For a module to unload itself, use SelfStopModule() and SelfUnloadModule(), and take the following points into consideration:

- Create a thread to execute SelfStopModule() and SelfUnloadModule(), which will be used to self-unload the module.
- Do not delete the above thread when the entry routine is performing the stop process.
- Consider the possibility that the thread might be deleted by other modules.

When executing SelfStopModule(), the module's entry routine is called in a similar manner to StopModule(). The difference is that with SelfStopModule(), argv[0] is assigned the character string "self". Accordingly, the entry routine must determine the number of threads to be deleted during stop processing based on the contents of argv[0].

The module undergoes the state transitions illustrated below, including the self-unloading case.

Figure 1-4



Program skeleton for a self-unloadable resident module:

```
#include <kernel.h>
/* Self-deletable resident modules must include the following
   threads to delete themselves.*/

int special_thread() {
      :
      :
      SelfStopModule(args, argp, &result);
      SelfUnloadModule(*); /* unload & exitDeleteThread */
      / * never returns here */
}

int module_start(int argc, char *argv[]) {
      /* perform required initialization processing after module is
      made resident
      * Also create thread here to execute
      *SelfStopModule()/SelfUnloadModule() */
      if( initialization was successful )
            return REMOVABLE_RESIDENT_END;
      else {
            /* If necessary, cancel initialization processing */
            return NO_RESIDENT_END;
      }
}

int module_stop(int argc, char *argv[]) {
      if( strcmp(argv[0],"other") == 0 ) {
            /* StopModule() being called by another module.
             * Before deleting module, perform required termination
             * processing. */
      } else if( strcmp(argv[0],"self") == 0 ) {
            /* SelfStopModule() called by this module.
            * Before deleting module, perform required termination
            * processing.
            * However, only the thread which executes
            * SelfStopModule()/SelfUnloadModule() is not deleted.
            */
      }
      if( termination processing was successful )
            return NO_RESIDENT_END;
      else {
            /* If possible, cancel termination processing and return
             * to a state in which module can stay resident */
             return REMOVABLE_RESIDENT_END;
      }
 }

 int start(int argc, char *argv[]) {
      if( argc >= 0 )    return module_start(argc, argv);
      else               return module_stop(-argc, argv);
 }
```

Resident Libraries

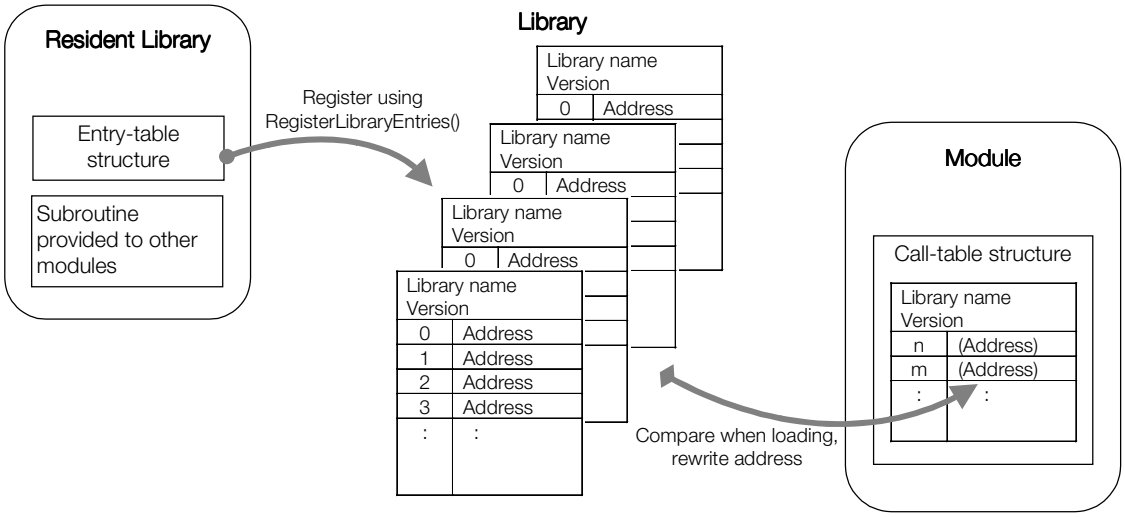
A resident module that makes some of the groups of functions it contains available to other modules is a resident library.

Resident Library Organization

A resident library contains a structure known as an entry-table structure that collects together subroutine entries provided to other modules. Normally during the time the module is resident, RegisterLibraryEntries() is used to register its entry-table structure. When stopping, ReleaseLibraryEntries() is used to delete the registration of the entry-table structure.

Conversely, modules that use a resident library will contain a call-table structure with the indexes of the subroutines the module wants to use. When that module is loaded, the call-table is compared with the entry table structure of the registered resident library and the entry addresses of the relevant subroutines are rewritten.

Figure 1-5



Unloadable resident modules can be used as resident libraries. In such cases, processes relating to the above entry-table structure must be executed when it is necessary to stop the unloadable resident module.

Entry Table Structure and Library-Entry Definition File

The entry-table structure can be created using the ioplibgen utility, which is based on the library-entry definition file.

The library-entry definition file is a text file which contains the following four kinds of statements.

Comment

A line beginning with "#" is a comment.

Library Name Declaration

Library names are 8 characters or less, and are declared as follows:

```
Libname libname
```

Library Version Declaration

Given as a decimal number with the major version (8 bits) and minor version (8 bits) separated by a "." in the form "**Version** *MM.mm*". The major and minor versions should be numbers between 1 and 255; zero is reserved.

Entry Declaration

Using the following formats, library-entry function names can be defined in multiple ways.

Entry/*level entry_symbol[entry_internal_symbol]*

Entry *entry_symbol[entry_internal_symbol]*

Entry-*level* indicates the *entry_symbol's* public level using a one-digit number. By indicating the public level using ioplibgen's -l option, only *entry_symbols* with that public level or lower will be output to the entry table file. Omitting the *level* is interpreted as a public level of 0.

Reserved Entries

Functions registered in the entry-table structure are identified by their order in the table (index) at execution time. The correspondence between function name and table index in principle shouldn't matter, and be left to the programmer. However, for the convenience of managing system services, the first four entries have fixed roles, regardless of function names, as shown below.

Table 1-3

Index	Function
0	Reserved for initializing library (details to be defined)
1	Reserved for re-initializing library (details to be defined)
2	Library termination
3	Reserved (details to be defined)

If the relevant functions exist for these four entries, they will be registered. If the functions do not exist, functions must be registered that don't do anything but simply return.

Termination Processing

The library termination processing function has the following prototype.

```
void  lib_terminate(int mode);
```

The reason for calling the termination processing function is specified by the mode argument.

Table 1-4

mode	Cause
0	System shutdown (reboot preparation)
1	Module deletion (currently unimplemented, reserved)

The termination processing function for reboot preparation is normally called after the system is placed in an interrupt-inhibited state. However, since there may also be resident libraries that cannot execute termination processing from an interrupt-inhibited state, the SetRebootTimeLibraryHandlingMode() function can be used in advance to set when to call the termination processing function for each resident library. The timing can be set in the following three ways.

- Call after the system is placed in an interrupt-inhibited state (default)
- Call before the system is placed in an interrupt-inhibited state
- Call once each before and after the system is placed in an interrupt-inhibited state

Sample library-entry definition file

```
# First, declare the library name. We'll use "mylib" for this
   example.
Libname mylib
# version declaration
Version 1.1
# === Declare each entry in resident library ===
# By convention, the first four entries are reserved for the system's
   use.
# Number 1 is reserved for the library initialization process
  (details not defined).
# Number 2 is reserved for the library reinitialization process
  (details not defined).
# Number 3, if present, is used for the entry termination process
# Number 4 is reserved
Entry -
Entry -
Entry -
Entry -
Entry AllocMemory
Entry ReAllocMemory
# Externally called function names and actual function names can
   differ as shown below
# External name     Internal name
Entry FreeMemory    mylib_free_memory
```

Call-table Structure

The call-table structure, required by the program that will use the resident library, is normally created automatically using the ioplibld utility.

Skeleton

The library entry definition file (= entry table structure) and program skeleton are shown below as reference information for creating a resident library. Program skeletons are shown for a normal resident library and for an unloadable resident library, respectively.

Skeleton for a library entry definition file:

```
Libname mylib
Version 1.1
Entry -
Entry -
Entry -
Entry -
Entry   libentry1
Entry   libentry2
```

Program skeleton for a resident library:

```
#include <kernel.h>

int start(int argc, char *argv[])
{
      extern libhead mylib_entry;

/* perform required initialization processing after the module is
made resident */
 if( initialization failed ) {
```

```
        /* If necessary, cancel initialization processing */
                    return NO_RESIDENT_END;
        }

        /* register own entry table in system */
        if( RegisterLibraryEntries(&mylib_entry) == KE_OK )
                    return RESIDENT_END;
        else {
                    /* If necessary, cancel initialization processing */
                    return NO_RESIDENT_END;
        }
        }

        int libentry1()
        {
          /* .... */
        }

        int libentry2()
        {
          /* .... */
        }
```

Program skeleton for an unloadable resident library:

```
#include <kernel.h>

int module_start(int argc, char *argv[]) {
            extern libhead mylib_entry;

/* perform required initialization processing after the module is made
resident */
if( initialization failed ) {
            /* If necessary, cancel initialization processing */
            return NO_RESIDENT_END;
}

/* register own entry table in system */
if( RegisterLibraryEntries(&mylib_entry) == KE_OK )
            return RESIDENT_END;
else {
            /* If necessary, cancel initialization processing */
            return NO_RESIDENT_END;
      }
}

int module_stop(int argc, char *argv[]) {
/* remove own entry table from system */
ReleaseLibraryEntries(&mylib_entry);

/* Before deleting module, perform required termination processing. */
if( termination processing was successful )
            return NO_RESIDENT_END;
else {
      /* If possible, cancel termination processing and return
       * to a state in which module can stay resident */
       * If necessary, re-register own entry table
       */
            return REMOVABLE_RESIDENT_END;
```

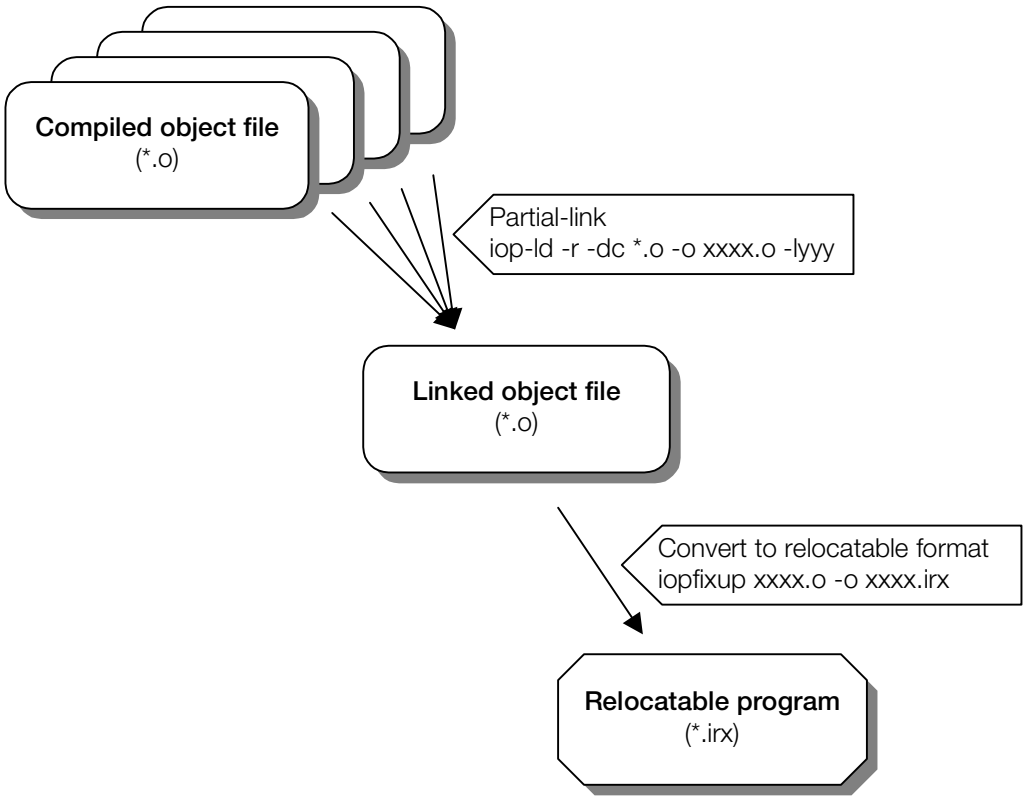
```
}
}

int start(int argc, char *argv[]) {
if( argc >= 0 )    return module_start(argc, argv);
else               return module_stop(-argc, argv);
}

int libentry1()
{
/* .... */
}

int libentry2()
{
/* .... */
}
```

Compiling

Compiling Relocatable Programs

To create a relocatable program, in the last step of the traditional compile-link procedure, instead of fixing the program's addresses, everything is collected into a single object using the linker's partial-link function, and using the iopfixup utility, it is then converted to the IOP relocatable-execution format.

Figure 1-6



Sample compilation procedure for relocatable programs

The procedure for compiling relocatable programs is shown in the following example:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ iop-ld -r -dc -o xxxx.o xxx1.o xxx2.o -lyyy
$ iopfixup -o xxxx.irx xxxx.o
```

This can be simplified to:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ iop-gcc -o xyz.irx xxx1.o xxx2.o –lyyy
```

This can be simplified further, all the way down to a single command:

```
$ iop-gcc -o xyz.irx xxx1.c xxx2.c -lyyy
```

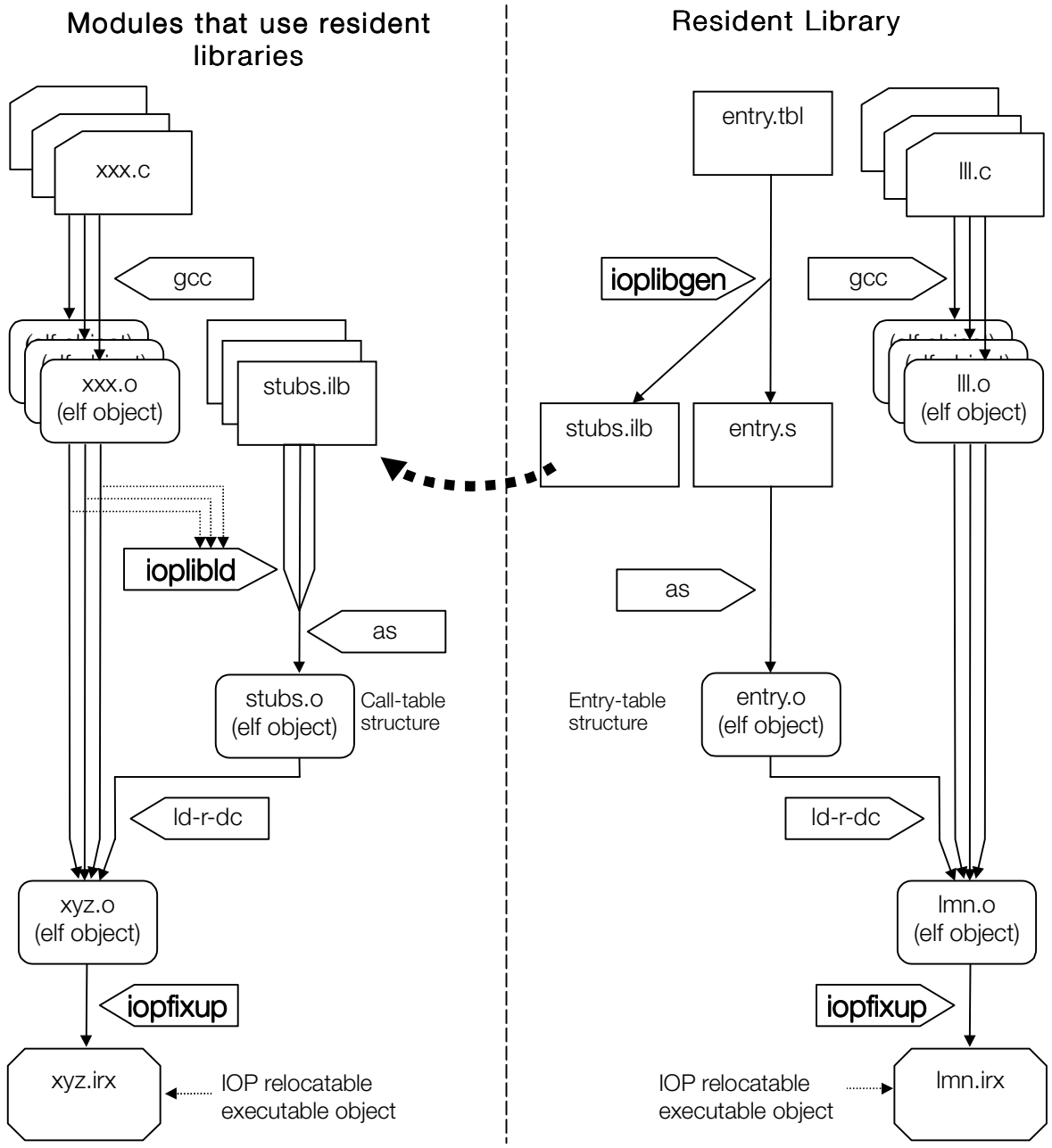
Compiling Resident Libraries

A resident library is a module that provides program code that other modules can call as subroutines.

An entry-table structure is required so that other modules can discover what subroutines the resident library makes available. Also, there needs to be a call-table structure that shows which subroutines of which modules are using the subroutines provided by the resident library. The ioplibgen and ioplibld utilities are used for creating and linking to these structures.

The procedure for compiling and linking modules that use resident libraries is as follows.

Figure 1-7



Sample compilation procedure for resident-library modules:

```
$ ioplibgen -d stubs.ilb -e entry.s entry.tbl
$ iop-as -o entry.o entry.s
$ iop-gcc -c lll1.c
$ iop-gcc -c lll2.c
$ iop-ld -r -dc -o lmn.o lll1.o lll2.o entry.o
$ iopfixup -o lnm.irx lnm.o
```

Sample compilation procedure for modules that use resident libraries:

a. If not linking to static libraries:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ ioplibld -s stub.s xxx1.o xxx2.o : stubs.ilb
$ io-as -o stubs.o stubs.s
$ iop-ld -r -dc -o xyz.o xxx1.o xxx2.o stubs.o
$ iopfixup -o xyz.irx xyz.o
```

Which can be simplified to:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ iop-gcc -o xyz.irx xxx1.o xxx2.o -ilb=stubs.ilb
```

b. If linking to static libraries:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ ioplibld -s stub.s xxxall.o xxx1.o xxx2.o -lsss : stubs.ilb
$ io-as -o stubs.o stubs.s
$ iop-ld -r -dc -o xyz.o xxx1.o xxx2.o stubs.o -lsss
$ iopfixup -o xyz.irx xyz.o
```

Which can be simplified to:

```
$ iop-gcc -c xxx1.c
$ iop-gcc -c xxx2.c
$ iop-gcc -o xyz.irx xxx1.o xxx2.o -lsss -ilb=stubs.ilb
```

Also note that the IOP kernel itself consists of several resident-library modules. So in that sense, all application programs are created as modules that use resident libraries.

Reference: Linking Between Modules

When loading a module, the IOP program loader checks to see whether that module calls any already-loaded modules (resident libraries), and links to other modules if necessary. The linking function acts on the entry-table structure in the resident library being called, and on the call-table structure in the module making the call.

Entry-table Structure

The set of functions the resident library provides to other modules is recorded in an entry-table structure as described below, and using the program loader's RegisterLibraryEntries() API, are registered with the system when the modules are activated according to the modules themselves.

There is no limit on the number of entry-table structures that a single module can register. In other words, a single module can register multiple libraries.

Entry-table structure format

Entry-table structures follow this format:

```
        .text
        .set noreorder
        .globl  mylib_entry
mylib_entry:
        .word  0x41c00000          /* magic number of entry table */
```

```
        .word  0                  /* reserved */
        .short  0x0101            /* version */
        .short  0x0000            /* flags */
        .ascii  "mylib\0\0\0"     /* library name (max 8 characters) */
        .align  2

        .word        MylibEntry1  /* address of function 1 */
        .word        MylibEntry2  /* address of function 2 */
        .word        MylibEntry3  /* address of function 3 */
        .word        MylibEntry4  /* address of function 4 */
        :
        :
        .word  0           /* table end mark */
```

Because of the reserved entries (mentioned below), you must register at least 4 entry functions. The 'flags' field is currently used by the IOP system to manage entry tables. Always set this to zero.

Call-table Structure

The call-table structure represents the functions, provided by a resident library, that a given module is using. The format is similar to that of an entry-table structure, except that only the functions needed by the module appear in the table.

Immediately after loading the module, the IOP program loader overwrites any "jump" commands that appear in the call-table structure and performs linking between modules.

Call-table structure format

```
        .text
        .set noreorder
        .globl  mylib_stub
mylib_stub:
        .word  0x41e00000   /* magic number of call table */
        .word  0                  /* reserved */
        .short  0x0101            /* version */
        .short  0x0000            /* flags */
        .ascii  "mylib\0\0\0"     /* library name (max 8 characters) */
        .align  2

        .globl          MylibEntry1
MylibEntry1:
        j      $31
        .half   0          /* entry table index */
        .half   0x2400     /* index magic : 0x2400XXXX == addiu $0,$0,XXXX
         */

        .globl       MylibEntry2
MylibEntry2:
        j      $31
        .half   2          /* entry table index */
        .half   0x2400     /* index magic */
        :
        :
        :
        .word   0, 0 /* table end mark */
```

Call-table structures must contain at least one called function entry.

The 'flags' field is currently used by the IOP system to manage call-tables. Always set this field to zero.

The ilb File Format

When creating a resident library, an ilb file must also be created. This is a data file that stores the library name and version, the name of all the functions in the library and their corresponding table indexes.

When linking a module that uses the library, this file is used to get information for generating the call-table structure.

These files, which have ".ilb" file extensions, are text files with a fixed column format as shown below:

#IOP-ILB# *insert any text here*
L lib*rary name*
V *0xHHHH*
F *0xHHHH*
E *ddd entryname*
E *ddd entryname*
E *ddd entryname*

The first line, beginning with #IOP-ILB#, is followed by lines of text that declare the format for ilb-data.

The line beginning "L" is always the second line, and gives the library name, which can be up to 8 characters, beginning from the third column on the line.

The line beginning "V" is always the third line, and gives a four-digit hexadecimal number beginning from the fifth column of the line.

The line beginning "F" is always the fourth line, and gives zero expressed as a four-digit hexadecimal number beginning from the fifth column of the line.

From the fifth line on, lines begin with "E", and show a three-digit decimal number beginning from the third column of the line, and the entry name beginning from the line's seventh column until the end of line.

It is OK to combine several ilb-data formats in a single file, in which case #IOP-ILB# serves as the delimiter.

Checking Compatibility When Linking

As mentioned above, the entry-table structure and call-table structure both have fields for library names and versions.

The library name is 8 bytes; if the name itself is fewer than 8 bytes, it should be padded with null characters. The upper 8 bits of the version field represent the library's major version number; the lower 8 bits the minor.

In general, the version field can be set to any value in the program, with the following stipulations:

Major version numbers must be set so that if function table ordering and function specifications are compatible, the numbers must be the same.

Minor version numbers are incremented to reflect bug-fixes and additional functionality for backwards compatibility.

Major and minor version numbers both must be 1 or greater. Zero should not be used.

When the module is loaded into IOP memory, and the call-table structure it contains is to be linked, the registered entry-table structure is checked to make sure that the library name field and major version field match; if they do, linking proceeds.

(Note: As of April 2000, the program loader performs this operation, but because it has been deemed preferable not to link when the minor-version number in the entry-table structure is smaller than that in the call-table structure, we plan to change this accordingly.)

When the entry-table structure is registered with the system, even if the library name and major version match those of the registered library, if a higher minor-version exists, then registration will fail.

Reference: IOP Object Format

The IOP object format is essentially the MIPS R3000 ELF relocatable file format, with certain additions, changes, and restrictions.

IOP Module Information Section

The IOP module information section is a new addition to the IOP object format. The IOP module information section includes a structure with variable-length data, as shown below. After the IOP program loader loads a module, it sets information in this structure needed to execute the module.

```
/* A section of type SHT_SCE_IOPMOD contains the following
structure.  */
typedef struct _Elf32_IopMod {
       Elf32_Word   moduleinfo;
       Elf32_Word   entry;
       Elf32_Word   gp_value;
       Elf32_Word   text_size;
       Elf32_Word   data_size;
       Elf32_Word   bss_size;
       Elf32_Half   moduleversion;
       char         modulename[1];      /* null terminate */
} Elf32_IopMod;
```

Here, "entry" contains the offset from the start of the TEXT segment of the program's activation entry address.

"gp_value" contains the offset from the start of the TEXT segment of the program's GP register value.

"text_size", "data_size", and "bss_size" give the sizes of the TEXT, DATA, and BSS segments, respectively.

When the Module variable is present in the ModuleInfo structure in the program's DATA segment, it will contain the offset from the start of the TEXT segment of the Module variable in moduleInfo. "moduleversion" will contain a copy of Module.version, and modulename[] will contain a copy of Module.name. If the Module variable is not present, moduleinfo will contain 0xffffffff, and moduleversion and modulename[0] will both contain zero.

The section name of the IOP module information section is ".iopmod"; each field in the section header will contain values as shown below.

sh_type	= SHT_SCE_IOPMOD(=0x70000080)
sh_offset	= offset from start of section data file
sh_size	= sizeof(Elf32_IopMod)+strlen(Elf32_IopMod.modulename)
sh_addralign	= 4
all others	= 0

Also note that a .reginfo section is not required in an IOP object file and should not be included.

Data Layout in the File

Although there are no particular restrictions on the way data is organized in a standard ELF relocatable file, the IOP object file format does require that data be arranged in the following order.

1. ELF header
2. Program header table
3. IOP module information section data
4. TEXT and DATA segment data
5. Data for other sections
6. Section header table
7. Relocation table data for TEXT and DATA segments
8. Other section data

Data Layout in Memory

IOP modules must follow the layout described in "Segment Layout" under "Module Structure".

That is, the TEXT, DATA, and BSS segments must appear consecutively and in that order, with align = 0x10, in other words, aligned on 16-byte boundaries.

In Unix, read-only data can be included in a TEXT segment, such as a .rodata section, this is not permitted in an IOP object file.

ELF Header Details

Contents of the ELF header must be as follows:

e_ident	== 'ELF', ELFCLASS32, ELFDATA2LSB, EV_CURRENT
e_type	== ET_SCE_IOPRELEXEC (==0xFF80)
e_machine	== EM_MIPS
e_version	== EV_CURRENT
e_flags	== anything (ignored)
e_entry	== offset from the start of the TEXT segment of the program's activation entry address (equal to Elf32_IopMod.entry)
e_ehsize	== sizeof(Elf32_Ehdr)
e_phoff	== location of program header table in the file (equal to sizeof(Elf32_Ehdr))
e_phentsize	== sizeof(Elf32_Phdr)
e_phnum	== 2
e_shoff	== location of section header table in the file
e_shentsize	== sizeof(Elf32_Shdr)
e_shnum	== number of sections in the file
e_shstrndx	== section index of .shstrtab section

Program Header Details

Program header tables must always consist of two elements and be laid out in the following order.

IOP Module Header

p_type	== PT_SCE_IOPMOD(== 0x70000080)
p_flags	== PF_R
p_offset	== location of IOP module information section data in the file
p_filesz	== size of IOP module information section data sizeof(Elf32_IopMod)+strlen(Elf32_IopMod.modulename)
p_vaddr,p_memsz	== 0

p_align	== 4

Segment Header

p_type	== PT_LOAD
p_flags	== PF_R+PF_W+PF_X
p_offset	= location of TEXT segment data in the file
p_vaddr	== 0
p_filesz	== size of load data in the file (TEXT segment size + DATA segment size)
p_memsz	== data size after loading into memory (TEXT segment size + DATA segment size + BSS segment size)
p_align	== 0x10

Reserved Symbols

SCE's IOP object file generator iopfixup automatically generates the following reserved symbols.

_ftext	Starting address of TEXT segment
_etext, etext	Address following the last byte of TEXT segment
_fdata	Starting address of DATA segment
_edata, edata	Address following the last byte of DATA segment
_fbss	Starting address of BSS segment
_end, end	Address following the last byte of BSS segment
_gp	Initial value of gp register

The following relationships exist between the symbols shown above.

0 <= (_fdata - _etext) < 0x10

0 <= (_fbss - _edata) < 0x10

To guarantee program source compatibility, we recommend that the linker or IOP object file generator should generate the symbols shown above.

Symbol Table Entries

If the value held by the st_value field of a table entry of the symbol table is an address, it will be a program offset (offset from the beginning of the TEXT segment), not a section offset.

Also, SHN_RADDR(==0xff1f) is entered in the st_shndx field for the etext, _etext, edata, _edata, _fbss, end, _end, and _gp entries among the reserved symbols shown above.

Otherwise, the entries are the same as for a standard R3000 ELF relocatable file.

Relocation Table Entries

The program offset (offset from the beginning of the TEXT segment) is stored for the r_offset field of a table entry in the relocation table, and STN_UNDEF is always stored for the r_sym field (==ELF_32_R_SYM(r_info)).

When converting from an ELF relocatable file to an IOP object file, after the value of the symbol indicated by the r_sym field is converted to a program offset, the relocation field of the relevant section data must be appropriately supplied.

Only the following six relocation types are allowed.

R_MIPS_NONE
R_MIPS_16
R_MIPS_32
R_MIPS_26
R_MIPS_HI16
R_MIPS_LO16

Also, corresponding R_MIPS_HI16 and R_MIPS_LO16 types must be stored consecutively in the table as a pair, as defined in the original MIPS R3000 specification. (Although the gcc extension allows multiple R_MIPS_LO16 types to exist for a single R_MIPS_HI16 type, this is not allowed here.)

Otherwise, the entries are the same as for a standard R3000 ELF relocatable file.

Debugging Information

No rules related to debugging information are defined for the IOP object format. Also, the IOP program loader ignores sections other than those defined in this document.

Therefore, compiler vendors can include sections that contain unique debugging information in an IOP object file.

However, since SCE's utilities assume that debugging information that is generated by the SCE-provided iop-gcc (which is almost identical to the standard R3000 gcc) is in a section named ".mdebug" with section type SHT_MIPS_DEBUG, this section name and section type cannot be used when generating debugging information that is incompatible with the MIPS gcc.

Chapter 2: IOP Kernel API

Library Overview

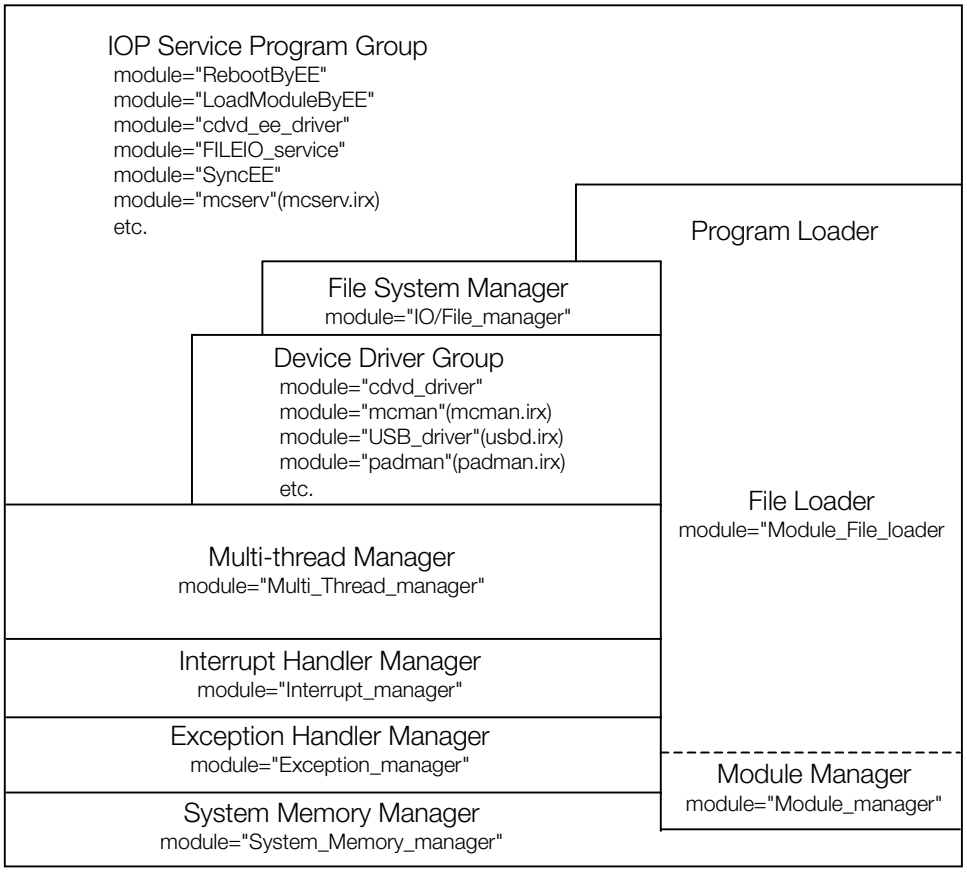
The IOP Kernel is an operating system that runs on the IOP. It offers multithread scheduling functions suitable for priority-based, simple, realtime processing. The kernel also provides inter-thread synchronization / communication functions using semaphores, event flags and message boxes. Other system calls provide interrupt handler management, memory pool management, timer management and V-blank synchronization.

Although the IOP Kernel manages IOP programs as modules in memory, it also supports a "resident library" function, which makes some modules available as subroutines to other modules.

Software Hierarchy

The software that is resident in IOP memory has the following hierarchical structure.

Figure 2-1



IOP Service Program Group

IOP service programs accomplish the work that the IOP is supposed to perform as an I/O processor by calling the various managers, drivers, and kernels located in lower layers. The IOP service programs accept EE requests from the device drivers that are handling communication with the EE, call appropriate I/O device drivers to execute I/O operations based on those requests, and return the results to the EE.

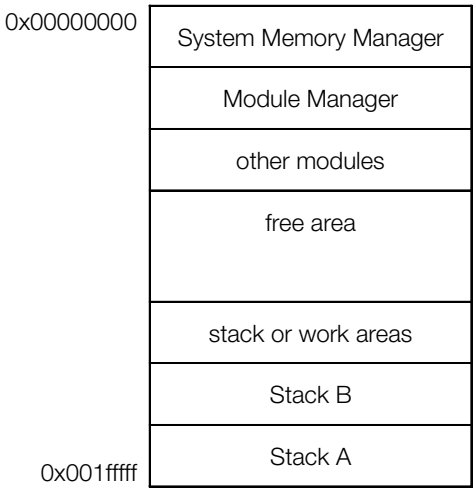
System Memory Manager

The System Memory Manager manages the entire memory area and provides the following functions to software in the higher layers.

a. Obtain total memory size
b. Allocate memory area (starting from the beginning of memory)
c. Allocate memory area (starting from the end of memory)
d. Release memory area

Memory is allocated in units of 256 bytes. The memory usage image is as follows.

Figure 2-2



Normally, only the Multithread Manager and Program Loader reserve memory by directly calling the System Memory Manager, and programs in the layers above the device drivers use the Multithread Manager's memory allocation service.

Program Loader

The Program Loader loads program modules from CD-ROM, DVD-ROM, ROM, or other sources and manages the modules in memory. It consists of the Module Manager, which deals with lower layers, and the file loader, which deals with higher layers.

The Module Manager has the following functions:

- Manage program modules in memory
- Manage resident libraries (including link processing during loading)
- Locate relocatable modules in memory

The file loader calls the Module Manager and File System Manager to implement the following functions:

- Load program module files
- Compare or rewrite versions of program module files and modules in memory

Exception Handler Manager

The Exception Handler Manager consists of a jump table management routine for the various exception handlers of the CPU (INT, IBE, DBE, Sys, Bp, ...) and an integrated exception handler that analyzes the exceptions and passes control to the various handlers. The Exception Handler Manager, which is not called from normal applications, is mainly used from the debugging system and the Multithread Manager.

Interrupt Handler Manager

The Interrupt Handler Manager consists of the interrupt handler registration and management routines and an integrated interrupt handler that analyzes the interrupt causes and passes control to the various handlers. The IOP has 20 interrupt causes. You can set whether to enable or disable interrupts according to per-cause mask registers and a master mask register, and functions are provided for manipulating these masks. When an interrupt occurs, the Interrupt Handler Manager references the interrupt status registers and mask registers to analyze the interrupt cause and calls the appropriate registered interrupt handler.

Multithread Manager

The Multithread Manager manages CPU resources, memory resources, and interrupt resources and provides the following functions to software in the higher levels.

a. Thread management functions
b. Thread scheduling functions
c. Inter-thread synchronization functions
d. Memory management functions
e. Interrupt management functions

Device Driver Group

Device drivers, which are modules that use Multithread Manager functions for handling various I/O devices, consist of the following three parts:

- Interrupt handler (when necessary)
- Device control thread (zero or more)
- Entry function group

The entry function group, which is called from higher level programs, performs processing for passing I/O requests to the device control threads or for directly controlling the devices. The entry function group itself is executed in the context of the calling thread.

File System Manager

The File System Manager, provides device-independent file access APIs such as open(), close(), read(), and write(), and registers and manages the entry function groups of the device drivers.

Related Files

The following files are required to use the IOP Kernel API.

Table 2-1

Category	Filename
Header file	kernel.h

Multithread Management

Thread Overview

From the perspective of parallel processing, a thread is a logical unit of a program. An application program's processing can be divided into multiple threads, which can be run in parallel, simultaneously. Although the threads are said to be running in parallel simultaneously, since there is only one CPU, only one thread is actually being executed at a time, when viewed within a given short interval. The thread that is to be executed is determined according to the state and priority of each thread. If a high-priority thread is being executed, a thread having a lower priority will not be executed as long as the high-priority thread does not enter a wait state or the state is not changed within the interrupt handler. This differs significantly from a Time Sharing System (TSS), which attempts to execute multiple tasks (or processes) equally while switching tasks.

Access to resources such as I/O devices or specific work areas in memory, must be coordinated between threads so that these resources can be exclusively used. The IOP Kernel provides semaphores, event flags, and message boxes as inter-thread synchronization and communication functions.

Threads and interrupt handlers are clearly different. They are executed based on different system states, with threads executed as part of a user program, and interrupt handlers executed as part of the kernel. With respect to this, several points must be carefully noted when writing programs.

Thread State and Operation

Threads that are managed by the IOP Kernel have the following seven (five when broadly viewed) states.

Table 2-2

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

- RUN: Execution state

  State in which the CPU is currently executing that thread. At a given instant of time, only one thread can be in this state.

- READY: Executable state

  State in which the preparations for executing the thread are completed, but the thread is queued because a thread having a higher (or the same) priority than that thread is being executed. If there are multiple threads in READY state, a queue is created, and those threads wait in the queue for the CPU to be free. This queue is called a ready queue.

- WAIT: Wait state, SUSPEND: Forced wait state, WAIT-SUSPEND: Double wait state

  These are execution pending states in which execution is stopped because the conditions that enable that thread to be executed are not satisfied. WAIT is a state in which execution is stopped by a system call invoked by the thread itself, SUSPEND is a state in which execution was forcibly stopped by another thread, and WAIT-SUSPEND is a state in which execution of a thread that is already in a wait (WAIT) state is further stopped by another thread.

  Execution is restarted from a wait state from the location where execution was suspended, and information that represents the program execution state such as program counters or registers (this is called the context) is restored directly.

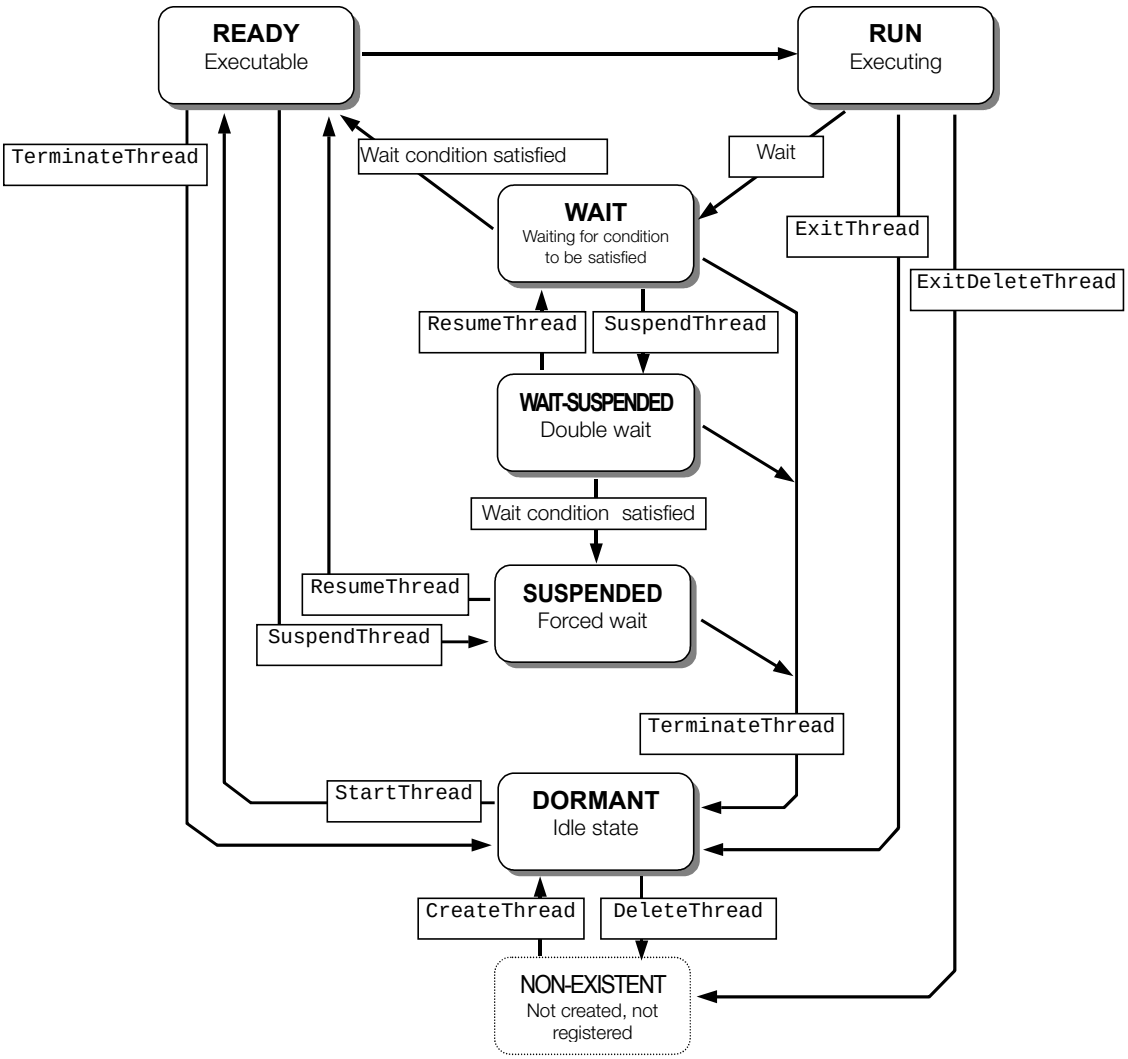
- DORMANT: Inactive state

  State in which a thread has not yet been started or has ended. A thread is in this state when it is first created.

- NON-EXISTENT: Unregistered state

  This is a virtual state before a thread is created, or after it is deleted.

Figure 2-3



A created thread is first placed in DORMANT state. When the thread starts up, it transitions to READY state and is entered in the ready queue (details are described later). The thread at the beginning of the ready queue will be executed and switched to RUN state. A thread in RUN state is switched to WAIT state by causing it to sleep or by causing it to wait for a condition such as a semaphore. When the wait condition is satisfied, the thread returns to READY state.

Thread Scheduling

The regulation or control that determines the order in which multiple threads that are in READY state are executed (i.e. switched to RUN state) is called scheduling. Scheduling performed by the IOP Kernel is described below.

Priority

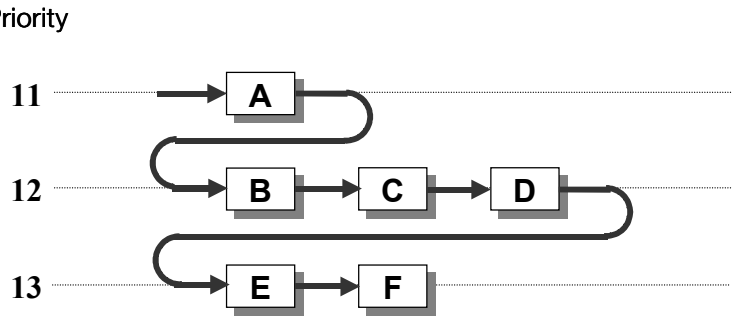
Each thread has a priority, and threads that are in READY state are scheduled and executed according to the priority of each thread. Basically, an executing thread never relinquishes the CPU to (is never removed from RUN state due to) a thread having the same or lower priority. The range of priorities is 1 to 126, where smaller numeric values represent higher priorities. Some priorities have been reserved by the system. The range of priorities that can be used by user programs is limited to USER_HIGHEST_PRIORITY (=9) to USER_LOWEST_PRIORITY (=123).

Ready Queue

Threads in RUN and READY states are queued in the ready queue according to the priority with which they are executed. The ready queue order is in descending order of the priority that each thread has, with threads having the same priority arranged in the order in which they switched to READY state.

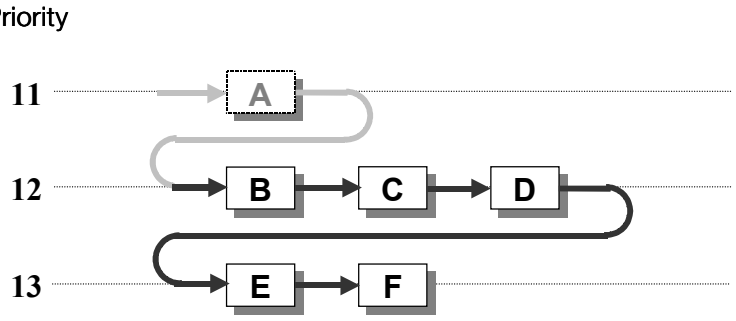
For example, assume that thread A with priority 11, threads B, C, and D with priority 12, and threads E and F with priority 13 reside in the ready queue as follows.

Figure 2-4



Since the thread having the highest priority in the ready queue is thread A, thread A is executed. By tacit consent, execution will not switch to another thread while thread A is being executed. If thread A enters WAIT state, the ready queue will be as shown in the following figure, and thread B will be executed.

Figure 2-5



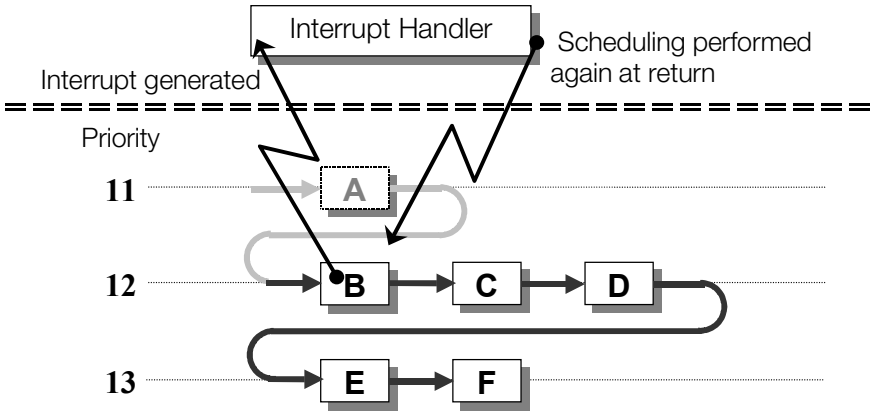
In actual programming, each thread should be switched to WAIT state when a process pauses or when a process must wait for an external process.

Interrupts and Scheduling

When an interrupt is generated, control is switched to the interrupt handler (provided it is not in an interrupt disabled state), and at the time when control returns from the interrupt handler, scheduling is performed again according to the state and priority of each thread.

As long as no processing is performed for switching thread A to READY state or for lowering the priority of thread B within the interrupt handler, thread B will be executed after control returns from the interrupt handler. If thread A is switched to READY state within the interrupt handler, when scheduling is performed again after control returns, thread A will be switched to RUN state and thread B will be switched from RUN to READY state (thread B is said to be preempted). Even in this case, since the position of thread B within the ready queue does not change, thread B will be executed again if thread A is removed from the ready queue.

Figure 2-6



In actual programming, it is common to queue a process that should be performed when an interrupt is generated, by placing the thread in WAIT state, then switch that thread to READY state within the interrupt handler before control returns.

Changing a Priority

The priority of a thread can be changed using ChangeThreadPriority(). A thread can change its own priority and can change the priority of another thread. Also, RotateThreadReadyQueue(), which rotates the ready queue, can be used to change the execution order of threads having the same priority. For example, if threads having priority 12 are rotated using RotateReadyQueue(), the leading thread will be switched to the end and the order will become C, D, B.

Also, in a thread-independent part (within an interrupt handler), iChangeThreadPriority() can be used to change the priority of a thread and iRotateThreadReadyQueue() can be used to rotate the ready queue.

When a thread in a wait state is switched to READY state, that thread is entered at the end of the ready queue at the corresponding priority.

Synchronization Between Threads

To prevent access contention for an I/O device between threads or to synchronize processing, the IOP Kernel provides several synchronization functions.

Putting a Thread to Sleep or Waking Up a Thread

When a thread does not urgently require the CPU such as when a process is paused or is waiting for an external device, SleepThread() can be called to switch that thread to WAIT state. This is referred to as "putting the thread to sleep."

A sleeping thread can be returned to READY state using WakeupThread() or iWakeupThread() from another thread or from a thread-independent part. This is referred to as "waking up the thread."

Event Flags

Event flags can be used to perform more flexible synchronization processing. This is useful in a program that wakes up numerous threads at one time. Event flags are used as follows.

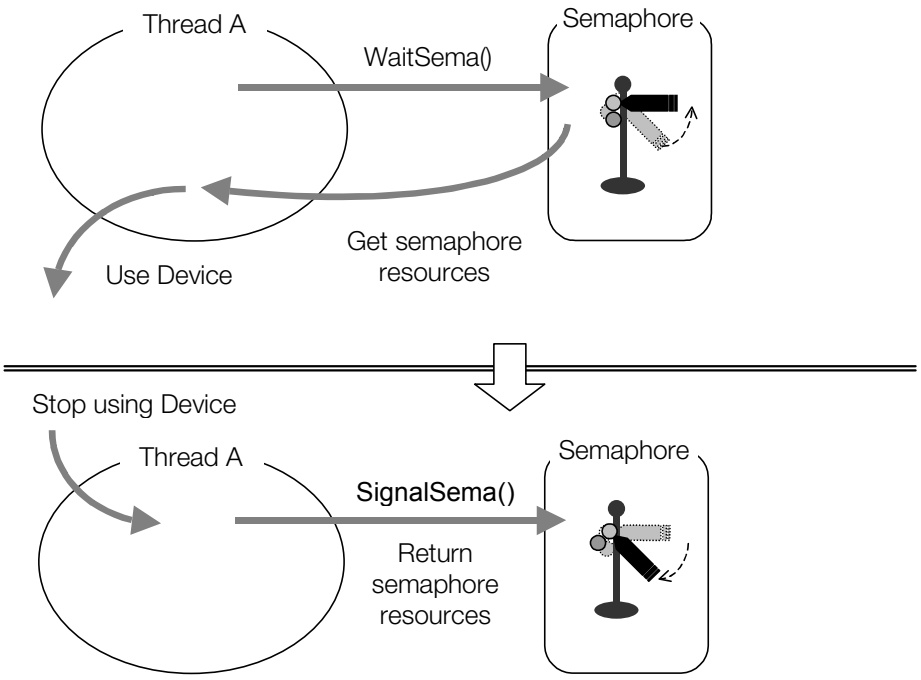
First, create an event flag and set its initial value. A thread that is to wait for a condition to be satisfied is switched to WAIT state by calling WaitEventFlag() and specifying a cancellation condition. When another thread or interrupt handler changes the flag value using SetEventFlag() or iSetEventFlag(), the thread among those waiting for the event flag for which the latest flag value matches the cancellation condition, will be returned to READY state.

Semaphores

When multiple threads are to use the same device or buffer area, a semaphore is provided to report the usage conditions of the device or buffer area to other threads for synchronization.

A thread that wants to use the device or buffer executes WaitSema() to get the semaphore resource. That is, it gets a usage right by raising the in-use flag before using the resource. When it is finished using the resource, it executes SignalSema() to return the semaphore resource to the system.

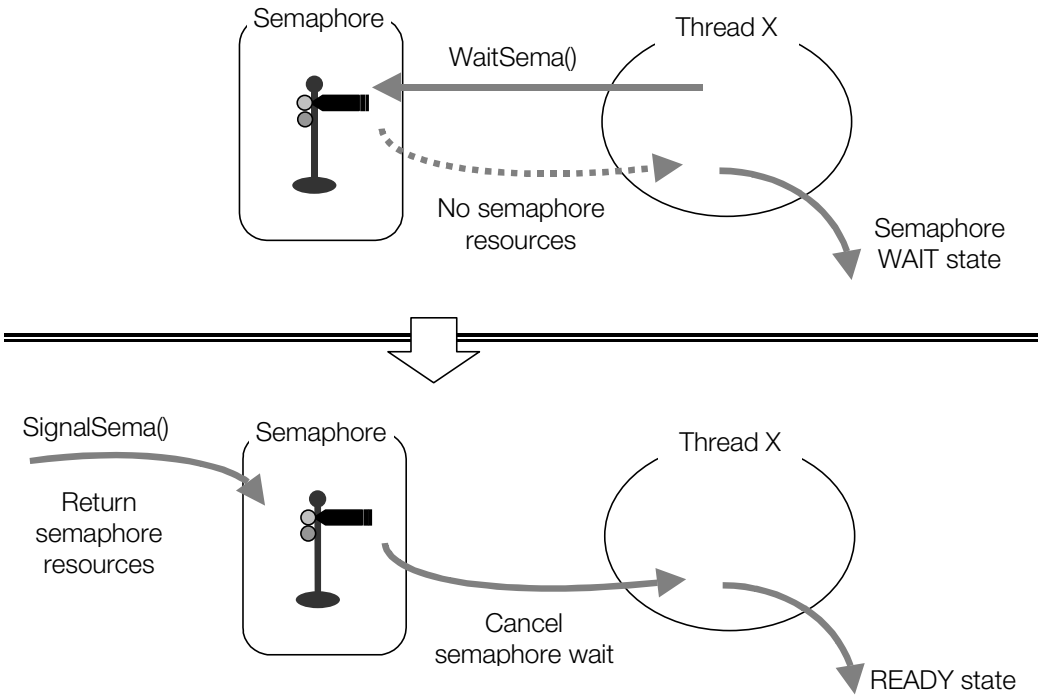
Figure 2-7



If another thread is using the device or buffer area when a thread executes WaitSema(), that thread cannot acquire the semaphore resource.

This causes the thread to switch to WAIT state and enter the semaphore queue. When the other thread is finished using the device or buffer area and returns the semaphore resource by executing SignalSema(), the returned semaphore resource is given to the leading thread in the semaphore queue, and that thread is returned to READY state and executed according to priority-based scheduling so that it can use the device or buffer area.

Figure 2-8



Since the number of semaphore resources can be specified when the semaphore is created, in addition to the kind of exclusive control described above, the semaphore can also be used to represent the number of valid data in a shared buffer to which multiple threads supply data. When a data supply-side thread has stored data in the shared buffer, it executes SignalSema() to indicate that the number of valid data has increased. When a data consumption-side thread intends to fetch data from the shared buffer, it executes WaitSema(), and if it can acquire the semaphore resource, it processes the data in the shared buffer. If it cannot acquire the semaphore resource, it enters WAIT state and waits until the resource becomes available.

Message Boxes

A message box, which is a service that supports the exchange of data between threads, can be thought of as a combination of a semaphore and ring buffer. Message boxes are used as follows.

First, CreateMbx() is used to create one or more message boxes in advance. A thread that will wait for a message specifies a message box number when invoking ReceiveMbx() and enters WAIT state. When another thread or thread-independent part uses SendMbx() or iSendMbx to specify a message box number and send a message, the thread that is waiting to use that message box is switched to READY state.

Other Services

Memory Allocation Service Overview

The Multithread Manager, which manages memory areas as a heap, provides a service that allocates memory blocks according to requests from application programs.

Two heap formats are used. In one heap format, the memory block size is fixed. In the other heap format, the block size can be specified each time memory is allocated. Also, two types of APIs are provided. With one, when there is no free space and memory cannot be allocated, processing will wait (switch to WAIT state) until free space is available. With the other, control will return as an error.

Module Management Overview

Individual programs that have been loaded into IOP memory are called modules. Modules correspond in a one-to-one fashion with relocatable object files (IRX files) that are stored in ROM or on CD or DVD-ROM.

Modules are divided into two types depending on the extent of time they exist in memory. One type is a non-resident module. When this module executes once, it completes its task and is deleted from memory. The other type is a resident module. This module is held in memory and performs its processing in response to requests from interrupts or other modules. When a module is loaded, its start entry is called and a decision is made as to whether to leave that module resident in memory or delete it. This decision depends on the value returned by the start entry.

A resident module can be created so it can be unloaded from memory when not needed. Resident modules can also be created that can unload themselves.

In addition, a module that provides subroutines to other modules is called a resident library. When it is started, a resident library registers in the system, the entry addresses of subroutines provided to other modules. On the other hand, a module that wants to use a subroutine of a resident library has a call table structure that is a collection of indexes of subroutines the module wants to use. When that module is loaded, it is linked with the registered library.

Refer to the document "IOP Programming" for more information on module structure, and starting and unloading operations.

Interrupt Management Overview

The Interrupt Handler Manager analyzes the causes of interrupt controller or DMA interrupts. It has functions for registering multiple interrupt handlers and can control threads from within an interrupt handler by waking up threads or setting event flags.

Timer Management Overview

Hardware Timer Management Functions

The IOP has the following six timer counters.

Table 2-3

Bit	Signal source	Gate signal	Prescaler
16	sysclock or pixel	H-blank	none
16	sysclock or H-blank	V-blank	none
16	sysclock	none	1/1, 1/8
32	sysclock or H-blank	V-blank	none
32	sysclock	none	1/1, 1/8, 1/16, 1/256
32	sysclock	none	1/1, 1/8, 1/16, 1/256

With an IOP timer counter, any of three types of input signals (pixel clock, H-blank, or V-blank) can be counted, and all timer counters can count system clock pulses.

Although the controller library uses the first two of the timer counters (16-bit pixel clock and H-blank) shown in the above table, these timer counters should not be directly used from an application.

The system clock, pixel clock, H-blank, and V-blank periods (frequencies) are shown below.

system clock

36.864 MHz

pixel clock

13.5 MHz (The pixel clock that is supplied to the IOP is always fixed at 13.5 MHz, regardless of the actual screen mode.)

H-blank

NTSC 15.73426573 KHz (858 pixel clock)
PAL 15.625 KHz (864 pixel clock)

V-blank

NTSC interlaced	59.94 Hz (262.5 H-blank)
NTSC non-interlaced	59.82 Hz (263 H-blank)
PAL interlaced	50.00 Hz (312.5 H-blank)
PAL non-interlaced	49.76 Hz (314 H-blank)

Timer counters are controlled with a gate signal. The following five control patterns can be used.

- Ignore gate signal
- Count only while gate signal is ON
- When gate signal goes ON, counter is cleared and counting is started
- When gate signal goes ON, counter is cleared, and when gate signal goes OFF, counting is started
- When gate signal goes ON, counting is started

With a timer counter, counting can be performed by dividing the input signal into 1/8, 1/16, or 1/256 of its original period.

Hardware timer management functions are provided for using these timer counters.

An application program uses service functions according to the following procedure.

1. Use the AllocHardTimer() function to allocate a timer having the required function.
   The timer is in an unused state at this time.
2. When necessary, use the SetTimerHandler() and SetOverflowHandler() functions to register interrupt handlers.
3. Call SetupHardTimer() to set the operating mode of the timer.
4. Call StartHardTimer() to start the timer. The state when the timer is started is called "in-use" state.
5. Call StopHardTimer() to stop the timer. The StartHardTimer() function can be called again later to start the timer.
6. If the timer is no longer needed, call FreeHardTimer() to return the timer after it has been stopped.

Software Timer Management Functions

Service functions for allocating and freeing hardware timers, setting the operating mode, registering interrupt handlers, starting and stopping timer counting, and reading the counter value are provided as an API for the IOP's six hardware timers.

In addition to the hardware timer control functions of the IOP, a function is provided for stopping the calling thread for a specified interval. An alarm function is also provided that calls a specific function (alarm handler) when a specified interval has elapsed. Although the interval is specified in microseconds or system clock pulse units, when implemented, the function may be delayed on the order of 100 to 200 microseconds.

Two of the hardware timers are used by the controller library.

Vblank Management Overview

To support V-blank synchronization processing, an API is provided for suspending the calling thread until the beginning or end of the V-blank interval. A function is also provided for calling a specific function (Vblank handler) at the beginning or end of the V-blank interval.

Precautions

Thread and Thread-independent Parts

In the Multithread Manager environment, the system state while a thread part is being executed and the system state while a non-thread part is being executed differ. A programmer must be aware of these differences in system state when writing a program for creating a non-thread part, that is, an interrupt handler.

A feature of a thread-independent part such as an interrupt handler or timer handler is that specifying the thread that was executing immediately before entering the thread-independent part is meaningless, and the concept of "own thread" or calling thread does not exist. Also, since the thread that is currently executing cannot be specified, thread dispatching cannot occur. Even if dispatching is required, it is delayed until the thread-independent part is exited (this is called a delayed dispatch state).

A service call that enters a wait state or a service call that implicitly specifies the calling thread theoretically cannot be issued from a thread-independent part. Also, depending on the implementation conditions of the Multithread Manager, there are cases in which the processing within the Multithread Manager will differ for a service call issued from a thread-independent part and a service call issued from a thread part even if the function is the same. Therefore, a service call that can be issued from a thread-independent part is provided with an entry name that differs from that of a service call that is issued from a thread part. That is, a service call that has the lowercase letter "i" appended to the beginning of its name is the only kind of service call that can be issued from a thread-independent part. For example, when the WakeupThread()

service call that wakes up a thread is issued from a thread-independent part, it is issued as iWakeupThread(). A service call that was issued based on an inapproriate system state will return an error.