PlayStation®2 IOP Library Overview
Release 2.4.3

Device Libraries

Summary Table of Contents

About This Manual

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

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

Changes Since Last Release

Chapter 1: i.LINK Driver Library

- A description on unloading the library has been added to the "Termination Operations" section of "Usage".

Chapter 2: i.LINK Socket Library

- A description on unloading the library has been added to the termination processing information in the "Outline of Procedure for Performing Socket Communication" section of "Library Overview."

Chapter 3: USB Driver Library

- In the "Compatibility of Samples and USB Devices" section of "Library Overview," descriptions of the selfunld and usbspkr sample programs have been added.

- The following sections have been added to "Library Overview".

    "Reducing The Load On The CPU For Isochronous Transfers (Multiple Isochronous Transfers)"

    "Unload Support"

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:
i.LINK Driver Library

Library Overview

The i.LINK driver library is comprised of a resident driver for monitoring the 1394 bus and a group of functions for communicating with the driver. The i.LINK driver library is located on a layer immediately above the LINK layer. The application program can use the library to perform basic operations such as:

- Setting up various parameters for the i.LINK resident driver
- Controlling the bus and retrieving information
- Handling events
- Responding to asynchronous request packets from callback functions
- Sending asynchronous request packets using transaction context
- Accessing the configuration ROM

The functions of the i.LINK driver library are implemented only on the IOP. To use these from a program on the EE, RPC client/servers must be implemented for the required functions.

Figure 1-1

IOP Application Program
i.LINK Driver Library (ilink.irx)
LINK layer
PHY layer
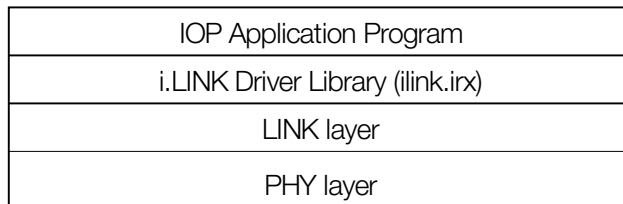
Related Files

The following files are needed to use the i.LINK driver library.

Table 1-1

Category	File
Header file	ilink.h
IOP module	ilink.irx

Reference Documents

- General information on the 1394 standard
  URL: http://www.1394ta.org/
- Overview of the standard
  Structure and overview of the IEEE1394 standards group, Interface, September 2000, CQ publications.
- Windows software development for 1394 evaluation boards
  Special feature: An introduction to IEEE1394 hardware and software, Interface, July 1999, CQ Publications.
- Connecting to PCs (Windows)
  Document name: Plug and Play Design Specification for IEEE 1394 v.1.0c
  URL: http://www.microsoft.com/HWDEV/respec/pnpspecs.htm

Connecting to Hubs/Repeaters/Other Devices

Compatibility between the PlayStation 2 and commercial 1394 hubs/repeaters/1394 cards (including PCs) has not been adequately studied. A message such as "Connecting to a device other than the PlayStation 2 may result in malfunction" must be included in the packaging, manual, etc. of titles that use i.LINK.

The 1394 Bus Protocol Analyzer

The 1394 bus protocol analyzer is a device that performs trace analysis on packets going through the 1394 bus. Certain models can also send packets.

As long as the socket library (described elsewhere) is used and communication is between PlayStations, there is no particular need for an analyzer. However, development of programs that communicate with non-PlayStation 2 devices may require the use of an analyzer.

i.LINK Driver Specifications

Node Capability

The i.LINK driver (ver 2.9) provides transaction-level node capability. In other words, positioned in the following levels:

- Bus manager (BM):   X
- Isochronous resource manager (IRM):   X
- Isochronous sending/receiving: X
- Transactions: O

While not provided by default, cycle master (CM) capabilities are provided by specifying SCE1394CF_NODE_CMC when using sce1394ConfSet().

However, since neither isochronous communications nor IRM are provided, CM capability is meaningless in a network consisting only of PlayStation 2's.

Restrictions

The i.LINK driver (ver 2.9) has the following limitations.

For retries, only single-phase retries are provided. (Hardware specification). The bus cannot be reset via a force-root (changing the root node).

The following steps must be performed to change the root node:

1. Use sce1394SbPhyPacket() to send a PHY configuration packet.
2. Reset the bus using sce1394SbReset().

This is needed because sce1394SbPhyPacket() is currently not implemented.

The Configuration ROM

The following is an example of the default configuration ROM provided by the i.LINK driver.

Figure 1-2

0400 : 0404A55F [**Bus_Info_Block**]

0404 : 31333934 "1394"

0408 : 00FF1122

040C : 08004607 Node_Vendor_Id | Chip_Id_Hi

0410 : 00000001 Chip_Id_Lo

0414 : 000526D2 [**Root_Directory**]

0418 : 03080046 Module_Vendor_Id

041C : 81000007 Textual_Descriptor (Vendor Name)

0420 : 0C0083C0 Node_Capabilities

0424 : 8D000009 Node_Unique_Id leaf offset

0428 : C3000001 Module_Vendor_Id directory offset

042C : 0002141D [**Module_Vendor_Id directory**]

0430 : 171A0000 Model_Id

0434 : 81000008 Textual_Descriptor (Model Name)

0438 : 00039E26 [**Vendor Name**]

043C : 00000000

0440 : 00000000

0444 : 536F6E79 "Sony"

0448 : 00028EDE [**Node_Unique_Id leaf**]

044C : 08004607 Node_Vendor_Id | Chip_Id_Hi

0450 : 00000001 Chip_Id_Lo

0454 : 000545C4 [**Model Name**]

0458 : 00000000

045C : 00000000

0460 : 53435048 "SCPH"

0464 : 2D313030 "-100"

0468 : 30300000 "00"

1. Chip_Id_Lo is set up with an identification number unique to the unit.

2. Model_Id and Model_Name contain an ID and a string associated with a PlayStation 2 model name. The example above is for the SCPH-10000.

Usage

Preparing for Communication

To prepare for communication using the i.LINK driver library, the following steps should be performed.

1. **Initialize the driver library**

    The i.LINK driver library is initialized using sce1394Initialize().

2. **Add Unit_Directory (if necessary)**

    Use sce1394UnitAdd() to add a Unit_Directory to the Root_Directory of the configuration ROM. The CRC of this Unit_Directory is automatically calculated by the library.

3. **Register callback functions (if necessary)**

    Use sce1394TrDataInd() to register callback functions for responding to Read/Write/Lock requests.

4. **Register event handling threads (if necessary)**

    The beginning and end of a bus reset as well as certain types of errors cannot be detected through return values from functions but instead must be detected as events. If these events are to be detected, separate threads specifically for event detection must be prepared.

5. **Connect to the bus**

    Once all preparations have been made, sce1394SbEnable() can be used to send and receive packets. To allow other nodes to immediately recognize that packets can be sent and received, use sce1394SbReset() to issue a bus reset.

Issuing Transactions

In the i.LINK driver library, requests such as Read/Write/Lock are issued using transaction contexts. The procedure for this is roughly as follows.

The important point to note is that the generation number of the bus must be the same as the generation number when the node ID is fetched and when the transaction is issued. This should be considered during development. Internally, sce1394TrRead(), etc., are set up to compare the generation number of the transaction context with the current generation number, and to only issue transactions when these two numbers match.

The steps are outlined below.

1. **Create transaction context**

    Use sce1394TrAlloc() to create a transaction context.

2. **Get generation number**

    Use sce1394SbGenNumber() to get the current bus generation number.

3. **Get node ID of peer**

    The node ID of the peer is obtained, and the retrieval method is a function of the protocol. Examples include directly scanning the configuration ROM for all devices on the bus, having the peer send the information, and so on.

4. **Set up transaction context**

   The generation number and the node ID of the peer are set up in the transaction context created in (1). Transfer speed and block size can also be specified, as required.

5. **Issue transaction**

   A transaction is issued using a function such as sce1394TrRead(). This will block until the communication operation completes, but other threads can be executed since internally WaitEventFlag() is used.

6. **Perform post-processing after the transaction completes**

   Based on the return value of sce1394Read(), etc., the following operations should be performed.

   a. 0 or greater: communication was successful. sce1394TrRead(), etc. can be called repeatedly. However, if a partition size is specified, and an error occurred midway through the transaction, the returned size can be different from the request. If this happens, information about the error can be obtained using sce1394TrStatus().

   b. SCE1394ERR_RESET_DETECTED: A bus reset caused the bus generation number to change, making the bus information in the context obsolete. The steps beginning with (2) must be repeated.

   c. SCE1394ERR_FAILED_RESPONSE: A response-code other than resp_complete, ack_data_err, ack_type_err, etc. was returned by the peer. The response-code can be determined with sce1394TrStatus() (ack_xxx_err will be converted to the corresponding response-code).

   d. SCE1394ERR_ACK_MISSING: An ack packet could not be received.

   e. SCE1394ERR_RETRY_LIMIT: The maximum number of retries for ack_busy was exceeded.

   f. SCE1394ERR_TIMEOUT: A response packet was not returned within the maximum allotted time.

   g. SCE1394ERR_REQUEST_DISABLED: sce1394SbEnable() has not been called.

7. **Free the transaction context**

   When the transaction context is no longer needed, it can be freed using sce1394TrFree().

Termination Operations

The following procedure should be performed once all communications have been completed, in order to terminate use of the i.LINK driver.

1. Use sce1394SbDisable(1) to disable packet transmission/reception and to stop responding to external requests. By setting the argument to 1, a bus reset will also be issued.

2. Use sce1394Destroy() to reset the driver and free the resources allocated by sce1394Initialize().

3. To unload the library, run the unload module after freeing all allocated resources.

Retries

The retry count for ack_busy is set by default to 0 (no retries).

This value can be changed by writing the BUSY_TIMEOUT register of the CSR. The CSR is written by issuing a Read/Write transaction even if it is for reading/writing the CSR of the local node. Also, the BUSY_TIMEOUT register will be cleared if the bus is reset.

Procedure for Changing Thread Priorities

The thread priorities used by the i.Link are High and Low. The default values are set so that High = 28 and Low = 34.

To set the thread priorities to values other than the defaults shown above, a third argument can be specified when ilink.irx is loaded using sceSifLoadModule(). The following is an example where High = 20 and Low = 28.

```
char* mes = "thpri=20,28";
    sceSifLoadModule( "host0:/usr/local/sce/iop/modules/ilink.irx",
    strlen(mes)+1, mes );
```

Connecting to PCs

The following is a description of issues that have come up when connecting to PCs running Windows 98 Second Edition(SE).

The Configuration ROM

In the default configuration ROM provided by the i.LINK driver, Windows 98 SE will not be able to recognize the PlayStation 2 as a 1394 device. To allow recognition from Windows, a Unit_Directory with the following entries must be added below the Root_Directory.

Table 1-2

key (hex)	value
Unit_Spec_Id (0x12)	0x080046(*)
Unit_Sw_Version (0x13)	0xFFFFFF(*)

Windows will determine a driver to use from the values of Unit_Spec_Id and Unit_Sw_Version.

The above values can be used to provide a simple test connection between PlayStation 2 and a PC. However, for commercial programs that involve connecting to Windows, appropriate values should be used.
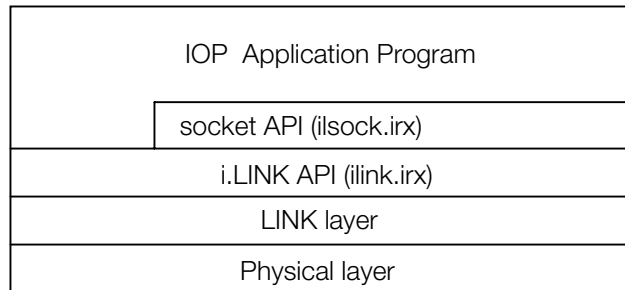
Chapter 2:
i.LINK Socket Library

Library Overview

The socket library, which is positioned at a higher layer than the i.LINK driver, implements socket communication using i.LINK. Using this API, an application program can perform communication with packets without having to be concerned with complex processing such as:

- Changing the node number due to a bus reset
- Transaction processing

Figure 2-1

```
┌─────────────────────────────────────────────┐
│          IOP  Application Program            │
│   ┌─────────────────────────────────────┐    │
│   │       socket API (ilsock.irx)       │    │
├───┴─────────────────────────────────────┴────┤
│          i.LINK API (ilink.irx)               │
├──────────────────────────────────────────────┤
│               LINK layer                      │
├──────────────────────────────────────────────┤
│              Physical layer                   │
└──────────────────────────────────────────────┘
```

Related Files

The following files are required to use the socket library.

Table 2-1

Category	File
Header file	ilink.h
	ilsock.h
	ilsocksf.h
IOP module	ilink.irx
	ilsock.irx

Outline of Procedure for Performing Socket Communication

The general procedure for using the socket driver to perform communication processing is as follows.

1. **Initialize the library**

   Use sceILsockInit() to initialize the library. At this time, specify the maximum number of sockets to be created.

2. **Create the sockets**

   Use sceILsockOpen() to create the required number of sockets. Since a descriptor representing each socket is returned, those descriptors are used to specify the sockets in subsequent processing.

3. **Set an address**

   Use sceILsockBind() to set address information (combination of node ID and port number) for a socket. This information is appended as data representing the sender to packets that are sent from that socket.

4. **Set the target**

    Use sceILsockConnect() to set address information for the target. This processing can be omitted.

5. **Send/receive data**

    Use sceILsockSend() or sceILsockRecv() to send data to or receive data from the target that was previously specified. The functions sceILsockSendTo() and sceILsockRecvFrom() can also be used to send data to and receive data from a target that is specified on each invocation.

6. **Termination processing**

    The sceILsockReset() function frees memory areas and event flags that were used by drivers such as the socket table driver, and were allocated by the sceILsockInit() function.

    The sceILsockClose() function frees memory areas for sockets such as the transaction context, that were allocated by sceILsockOpen().

    To unload the library, unload the module after freeing resources with sceILsockReset() and sceILsockClose(). At the same time, if there are resources that were allocated with a function such as sce1394TrAlloc() in a higher level application program, a function such as sce1394TrFree() must be used separately to free them in advance.

Precautions

- The socket library does not guarantee packet arrival. Application programs should be written on the assumption that there may be cases when a packet does not reach its target.

- Descriptor operations are not re-entrant. If more than one thread shares the same socket, exclusive control must be performed.

- Data receive is a blocking operation. If no received data is available when a function is called, the function will not return until data is received.

Protocol

The following is a description of the internal protocol used by the i.LINK socket library.

Structure of the Configuration ROM

Root_Directory

A single Unit_Directory containing the following entries must be the only item in the Root_Directory of the configuration ROM.

Table 2-2

Key(hex)	Value
Unit_Spec_ID(0x12)	0x080046
Unit_Sw_Version(0x13)	0xFC0000
Unit_Location(0x95)	(offset to leaf)

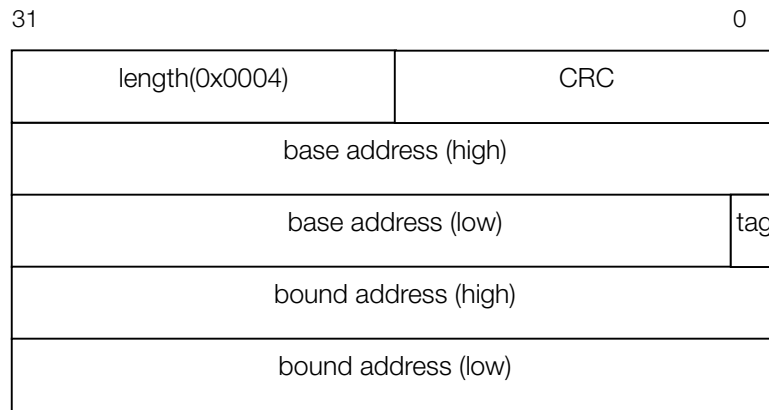The Unit_Spec_ID entry must be located at the start of the Unit_Directory. The Unit_Sw_Version entry must follow immediately after the Unit_Spec_ID. Only one Unit_Location entry is permitted.

When searching these entries, location and sequence must be ignored. Also, if more than one Unit_Location entry is detected, the second and subsequent entries must be ignored.

Unit_Location Leaf

Unit_Location leaf specifies the incoming FIFO area for the node. The data structure is shown below.

Figure 2-2

31		0
length(0x0004)	CRC	
base address (high)		
base address (low)		tag
bound address (high)		
bound address (low)		

base address is the starting address of the incoming FIFO area and must be 4-byte aligned. bound address is the largest address in the incoming FIFO area + 1.

The size of the incoming FIFO area must be at least equal to the maximum packet size (512 bytes).

The upper 16 bits of the address must be all zeros. tag is always set to 0x01.

When reading, the tag value must be ignored.

Transactions

All datagrams are transferred as "async write request for data block" transactions for the incoming FIFO area.

Broadcasting is performed by unicasting to all associated nodes.

If an incoming request is received where the destination offset of the packet does not match the base address of the incoming FIFO area, the request must be ignored or an address error must be returned.

If a request is received where the packet destination EUI-64 does not match the Bus_Info_Block EUI-64 of the local node or the broadcast address (0xffffffff:ffffffff), the request must be ignored.

A retry should not be performed in response to a transaction_timeout. In such cases, it is still possible that the packet has reached its target. Thus, starting a retry can result in packets getting overlapped. Because of this restriction, the socket library cannot guarantee packet arrival.

Retries may be performed in response to ack_busy and ack_data_err.

With release 2.4 and later, no error processing is required when ack_missing occurs since ilsock.irx will automatically resend the packet and perform a redundancy check when ack_missing is detected.

destination ID

The 1394 destination ID, which becomes the datagram target, is determined automatically when the socket driver scans the configuration ROM. The resulting mapping information may be cached.
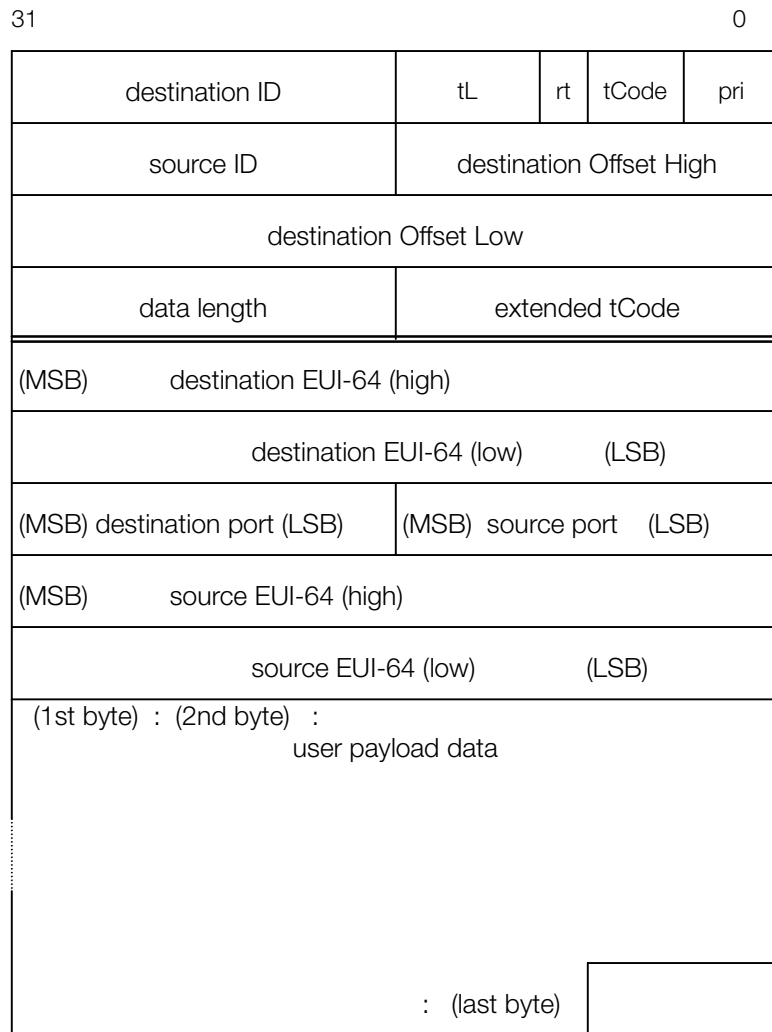
Even if the Bus_Info_Block EUI-64 and destination EUI-64 match, nodes that do not have the required Unit_Directory must be treated as if they do not exist.

packet format

The format of the "async write request for data block packet" used to actually transfer datagrams is shown below.

Figure 2-3

(transmit first)

(transmit last)

Unit_Sw_Version and Packet Format for Release 2.4 and Later

Unit_Sw_Version

We discovered that the value of Unit_Sw_Version: 0x1A0000, which was used by ilsock.irx for release 2.3.x and earlier, is not correct. The correct value should be 0xFC0000, which is used in release 2.4 and later.

When 0x1A0000 was used, a communication problem developed related to the ilink standard. Therefore, do not use ilsock.irx from release 2.3.x and earlier. Since ilsock.irx is based on a specification which only permits communication with devices having the same Unit_Sw_Version, a game that had been previously created under release 2.3.x and earlier cannot communicate with one created under release 2.4 and later.

Unit_Sw_Version

With release 2.4 and later, the sending side will automatically resend a packet when an ack_missing occurs. To make sure that the receiving side does not get redundant packets when packets are resent, the first four bytes of the user payload data of release 2.3.x and earlier are used to send a "packet number". When a packet is resent, the same packet number as was in the original send data is used. Therefore, redundant reception of a packet is avoided by referencing this number. As a result, the user payload data in release 2.4 and later is four bytes smaller than the user payload data in release 2.3.x and earlier.

Chapter 3:
USB Driver Library

Library Overview

USB Specification Compatibility

Compatible with USB 1.1 Specification

Related Files

The following files are related to the USB driver. To install the USB driver package independently, copy these files to the /usr/local/sce directory of the development computer.

Table 3-1

Category	File name	Copy destination
Library file	usbd.ilb	/usr/local/sce/iop/gcc/mipsel-scei-elf1/lib/
Header files	usb.h usbd.h	/usr/local/sce/iop/gcc/mipsel-scei-elf1/include/
Module	fileusbd.irx	/usr/local/sce/iop/modules/

Library Configuration

Software that handles the USB generally has the following kind of hierarchical structure.

Figure 3-1: Hierarchical Structure
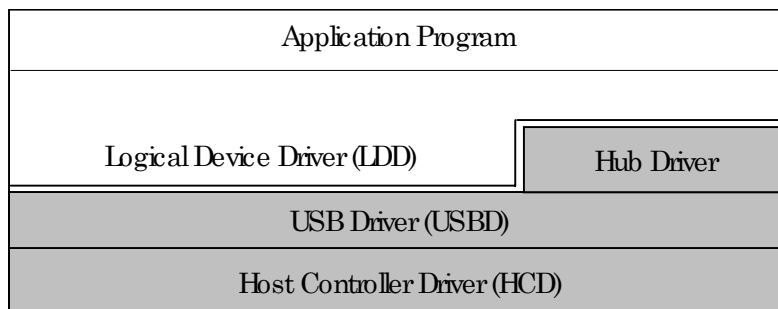
The usbd.irx module, which is provided as an IOP resident library, has three of the functions within the hierarchy mentioned above (HCD, USBD, and Hub Driver) and presides over processing that does not depend on the communication destination equipment. (In the following explanation, these three functions are collectively referred to as USBD.) To handle USB equipment such as a mouse or microphone, an LDD for performing class-dependent processing must be created on the user side.

Reference Materials

You must understand the USB specifications before creating an LDD. For general information related to the USB specification, refer to the following website: http://www.usb.org/developers/docs.html

Compatibility of Samples and USB Devices

Samples that use USBD include the following:

1. **sce/iop/sample/usb/usbmouse and sce/ee/sample/usb/usbmouse**
   Sample mouse drivers.

2. **sce/iop/sample/usb/usbkeybd and sce/ee/sample/usb/usbkeybd**
   Sample keyboard drivers.

3. **sce/iop/sample/usb/usbdesc**
   Sample program for dumping the static descriptor of a USB device.

4. **sce/iop/sample/usb/selfunld**
   Sample LDD that can unload itself.

   It is based on the sample mouse program, however, the RPC server has been removed.

   Use this program as an example of an LDD that does not perform RPCs, such as a modem driver.

5. **sce/iop/sample/usb/usbspkr**
   Example of isochronous transfers.

   Although it allows a WAV file to be played through a USB speaker, the AUDIO class has not been rigorously implemented.

   Use this program only as an example of the isochronous transfer.

Although drivers 1 to 4 have been implemented according to the USB specification, they will not operate properly with some USB devices. This is because some USB devices that are on the market have not undergone compatibility testing by a third-party organization.

USB Bus Protocol Analyzer

A commercial USB bus protocol analyzer may be used to trace and analyze the USB bus protocol.

When a sample driver is used as is, it is usually not necessary to use an analyzer. However, after the driver is newly created (and after the sample is modified), the new driver may malfunction. In these cases, the protocol analyzer should be used.

Reducing The Load On The CPU For Isochronous Transfers (Multiple Isochronous Transfers)

"Multiple isochronous transfers" can be performed beginning with Release 2.4.3 using the sceUsbdMultiIsochronousTransfer function.

Previously (with the sceUsbdTransferPipe function), performing an isochronous transfer for a single pipe applied a 50% load on the R3000.

When a "multiple isochronous transfer" is used, that load can be significantly reduced.

The reason that a large load was applied by a conventional isochronous transfer was that an interrupt was generated every 1 msec. With a "multiple isochronous transfer," up to eight packets can be grouped together to reduce the number of interrupts caused by the transfers.

As a rough estimate, when eight packets are grouped together and transferred, the transfers can be done using only about 1/6 of the CPU load compared with doing the transfers using the sceUsbdTransferPipe function.

Unload Support

Unloading has been supported since Release 2.4.

If you want to use the USBD Unload function, all registered LDDs must support unloading.

The reason is that the USBD forgets information about registered LDDs when performing an unload.

USBD unloading should be performed according to the following procedure.

1. Unload all LDDs.
2. Unload the USBD.

Reloading should be performed according to the following procedure.

1. Load & Start the USBD.
2. Load & Start the LDDs.

Notes on Preparing an LDD and Theory of Operation

LDD (Logical Device Driver)

The LDD is a USB device driver provided for each model or type of device.

The LDD always implements the following three external functions. These functions are called by the USBD when an event occurs.

- Probe function (This function is called if no LDD is associated with the device after sceUsbdRegisterLdd() is called, or when a new device is inserted on the USB bus.)
- Attach function (called when the Probe function's return value is "This device is managed by this LDD").
- Detach function (Called after a device on the USB bus has been detached.)

Principles of LDD Operation

The figure below shows operating principles that emphasize the relationship between the USBD and LDD.

The arrows in the figure indicate function calls.

The functions are called in the numerical order shown.

Figure 3-2: Principles of LDD operation

USBD

LDD

| sceUsbdRegisterLDD function | | start function |
| LDD registration | 1. Register the LDD in the USBD | Initialize the LDD |

2. Device is inserted on the USB bus!

probe function
Decide whether or not the inserted device is managed by this LDD.

3. Decision result

4. From decision result in 3, determine that the LDD at the right is the LDD of the inserted device.

attach function
Begin execution of transfer

5. Communication preparation

sceUsbdSetPrivateData, sceUsbdOpenPipe, etc.

6. Transfer request

callback function
Process after completion of transfer command

sceUsbdTransferPipe function

7. Transfer complete

Repeat 6. and 7. from this point onwards

detach function
Completion processing

8. Device was detached from the USB bus!

LDD Control

The USBD controls the LDD with an LDD control structure (sceUsbdLddOps).

Figure 3-3: LDD control
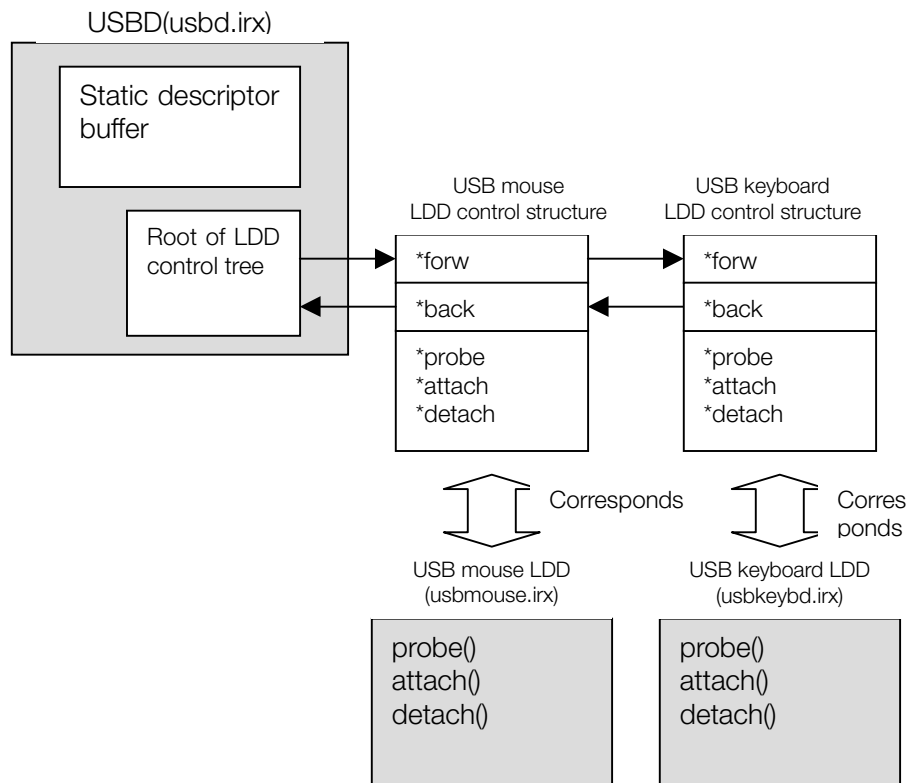
Figure 3-3 shows the state after two LDDs are registered with sceUsbdResisterLdd(). The LDD control structure corresponds one-to-one with the LDD.

Example of USBD and LDD Operation

The state of Figure 3-3 shows an example of operation after a keyboard device is inserted on the USB bus.

In this example, the auto-acquire function of the report descriptor is OFF (option reportd=0).

1. The state shown in Figure 3-3, is an example of a keyboard device inserted on the USB bus.

2. USBD assigns an address to the device.

3. USBD gets the device's static descriptors (devices, configuration, interface, endpoint). The obtained static descriptors are stored in the "static descriptor buffer."

4. If the acquisition of the static descriptors terminates normally, the USBD traverses the LDD control tree from the head.

5. In the case shown in Figure 3-3, the initial LDD control structure is for a USB mouse, so reference the *probe pointer in order to call the USB mouse LDD's probe function.

6. The USB mouse LDD's probe function references the "static descriptor buffer" with the sceUsbdScanStaticDescriptor function. It then decides whether or not it should manage the device. Here, the inserted device is the keyboard, so the probe function's return value is, "This device is not managed by this LDD."

7. For a USBD that receives a return value stating that "This device is not managed by this LDD," it further traverses the LDD control tree in order to reference the next-linked LDD control structure.

8. The next LDD control structure is the control structure of the USB keyboard LDD. Consequently, the USBD calls the USB keyboard LDD's probe function.

9. The USB keyboard LDD's probe function uses the sceUsbdScanStaticDescriptor() to reference the "static descriptor buffer," and then decides whether or not it should manage the device. Here, the inserted device is a keyboard, so the probe function's return value is "This device is managed by this LDD."

10. A USBD that received the return value "This LDD manages this device." references the LDD control structure's *attach pointer in order to call the USB keyboard LDD's attach function.

11. The USB keyboard LDD's attach function reserves one private datum for this device. One private datum per device is required. (For example, when three USB keyboards are present, three private data are required.) If it can be reserved, use sceUsbdSetPrivateData() to associate it with this device.

12. The USB keyboard LDD's attach function uses sceUsbdScanStaticDescriptor() to reference the "static descriptor buffer," in order to gather the information (packet size, etc.) required for communication. The collected information is then stored as private data.

13. The USB keyboard LDD's attach function opens a communications pipe with either sceUsbdOpenPipe() or sceUsbdOpenPipeAligned().

14. The USB keyboard LDD's attach function performs the required communications using sceUsbdTransferPipe().

15. The keyboard inserted in 1. is removed from the USB bus.

16. In order to call the detach function for the LDD that manages this device, the USBD references the *detach pointer of the LDD control structure for the removed device.

17. The LDD's detach function returns after performing end processing (releasing private data).

18. The USBD closes the device's communications pipe.

Note Regarding LDD Preparation

The LDD's USB event functions (probe, attach, detach) and callback function (called when transfer ends) are called using a method such as the one shown in Figure 3-4.

Figure 3-4: How the USBD calls the LDD function
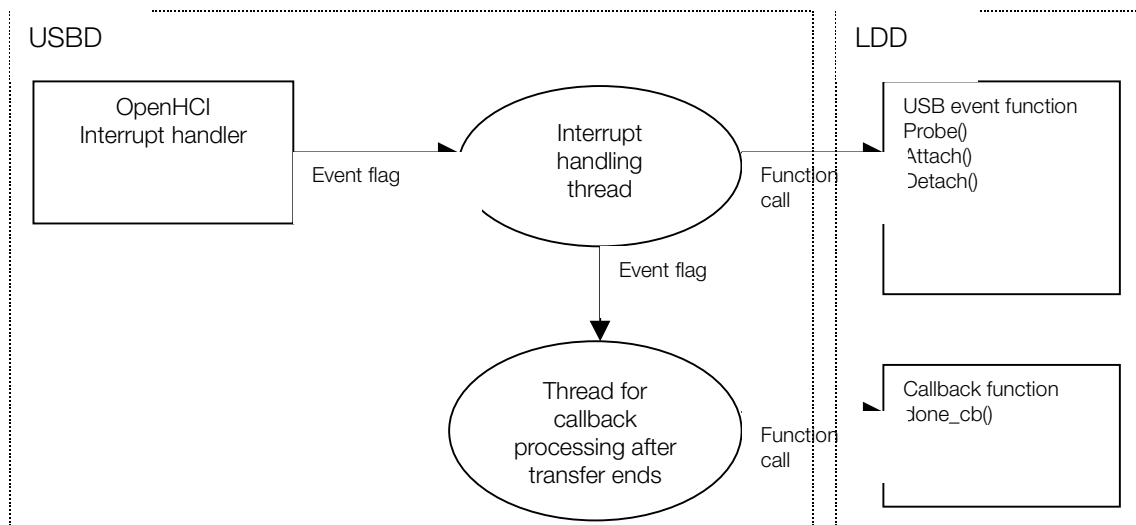
Regarding the "interrupt handling thread" and the "thread for callback processing after the transfer ends" of Figure 3-4, only one of each is generated (regardless of the number of devices on the bus).

As a result, problems occur when the following types of wait processing, etc., are performed using the USB event function or callback function:

1. Time delay resulting from DelayThread(), etc.
2. Waiting for a semaphore, event flag, or message from another thread
3. Waits (while (var == 0) ;) for the specific variable to change. (This processing has problems not limited to the USBD.)
4. Moreover, although it is not wait processing, there is heavy processing such that the processing time is on the order of the previously mentioned 1, 2, 3.

When this kind of wait processing is performed, all of the USBD's USB events and callback calls stop without returning to the thread(s) in the USBD, until the wait state is cancelled.

Although a restart is performed after a cancel, in the case of a device with a time limit, data are lost, etc. But, even without a time limit, problems can occur such as the momentary freezing of the mouse pointer, etc.

When using the USBD, finish processing in the USB event function and the callback function quickly. Wait processing is not allowed.

If wait processing is necessary, use the following method:

1. Create a thread for wait processing and for issuing the next transfer request.
2. Restrict the callback function to such processing as event flag setting, and quickly call and return to the original state.
3. The thread of 1. waits for 2. After the necessary wait processing has been performed, the next transfer request is transmitted to the USBD.

Note: Lower the priority of the thread of 1. below that of the USBD thread.

Restrictions and Precautions

Interrupts

The various entry functions of the USBD cannot be called from an interrupt context.

Precautions When the Packet Size is 63 or 64 Bytes

For an Interrupt, Control, or Bulk transfer, although a data transfer request of up to 4K bytes can be specified, the actual transfer is performed by splitting the data into the packet size (the splitting is performed by the hardware). To avoid problems that are caused due to hardware difficulties when the packet size is 63 or 64 bytes, the USBD sets the maximum packet size to 62 bytes when all of the following conditions are satisfied.

- When performing Interrupt(OUT), Control(IN/OUT), or Bulk(OUT) transfers
- The packet size is 63 or 64 bytes (MaxPacketSize is 64)
- When performing transfers using the Data pipe opened by Control pipe or sceUsbdOpenPipe()

When transferring a large amount of data under the above conditions, performance will decrease somewhat. If the transfer speed must be increased or if 64-byte length transfers are required, use sceUsbdOpenPipeAligned().

Precautions When Performing Isochronous Transfers

When performing isochronous transfers, the transfer may fail if the bus right for the memory bus is blocked for a period of at least 300 cycles on the IOP. The error that is returned when this data transfer fails will be BufferOverrun for a data IN transfer (host <= dev) and BufferUnderrun for a data OUT transfer (host => dev). When performing isochronous transfers, make sure the following kinds of processing are not being performed by the IOP.

- DMA inhibit, spanning at least 300 cycles
- Cache flush
- Processing of at least 300 cycles within a DMAC interrupt routine

Restrictions during Interrupt Transfers

The Endpoint descriptor has a bInterval member, which indicates the transfer period.

bInterval can take on values from 1 ms to 255 ms, however, when an Interrupt Transfer is performed, the period will actually be one of the following six values according to the hardware specification (OpenHCI specification).

Table 3-2

bInterval Value	Actual Transfer Period
0 to 1 ms	1 ms
2 to 3 ms	2 ms
4 to 7 ms	4 ms
8 to 15 ms	8 ms
16 to 31 ms	16 ms
32 to 255 ms	32 ms

USBD Startup Options

The usbd.irx startup options are shown below. XXX and YYY are all decimal numbers. If an illegal value is specified for a startup option, operation is not guaranteed.

Table 3-3

Option	Function	Default value
dev=XXX	Upper limit for number of devices	32
ed=XXX	Upper limit for number of endpoints	64
gtd=XXX	Upper limit for number of General Transfer Descriptors	128
itd=XXX	Upper limit for number of Isochronous Transfer Descriptor	128
ioreq=XXX	Upper limit for number of I/O transfer requests	256
conf=XXX	Number of static descriptor bytes for for each device	512
hub=XXX	Number of HUB devices other than Root hub	8
port=XXX	Maximum number of ports for each HUB	8
thpri=XXX, YYY	Priority value of the 2 threads created by USBD	30,36
reportd=XXX	Switch to automatically obtain the report descriptor	0

dev=XXX

Specifies the upper limit for the number of devices to be handled simultaneously by USBD. The root hub device is also included in this number of devices. One device area is consumed if a port exits even if the device is not connected. For example, the number of required devices when no Hub is used at all is three (root hub and two ports). A value exceeding 128 cannot be specified.

ed=XXX

Specifies the upper limit for the number of endpoints to be handled simultaneously by USBD. The control endpoint is also included in this number of endpoints.

gtd=XXX

Specifies the upper limit for the number of Transfer Descriptors used for non-isochronous transfers. One control transfer with no data transfer consumes two gtds. A control transfer that includes a data transfer consumes three gtds. One interrupt transfer or bulk transfer usually consumes one gtd.

itd=XXX

Specifies the upper limit for the number of Transfer Descriptors used for isochronous transfers. One isochronous transfer usually consumes one itd.

ioreq=XXX

Specifies the maximum number of transfers that can be requested simultaneously by sceUsbdTransferPipe(). Requests can overlap without waiting for the end of previously requested transfers up until the number of requests specified here.

conf=XXX

Specifies the total number of static descriptor bytes for each device.

hub=XXX

Specifies the maximum number of HUB devices, excluding root hub.

port=XXX

Specifies the maximum number of ports that each HUB can have.

thpri=XXX,YYY

Specifies the priorities of the USBD-created threads.

XXX is the priority of the thread for interrupt handling, and YYY is the priority of the thread for callback processing.

Note: Set the XXX priority higher than that of YYY (i.e. numerically lower).

reportd=XXX

Switches the function ON and OFF that automatically obtains the report descriptor.

The function is OFF when XXX=0 (default) and ON when XXX=1.

When the function is turned ON, the report descriptor is obtained before USBD calls the probe function. Consequently, the report descriptor can also be accessed within the probe function.

The report descriptor can be accessed using the sceUsbdGetReportDescriptor function.

* The storage area for a report descriptor is maintained separately as a static descriptor. This increases the amount of memory that can be used.

Chapter 4:
USB Module Autoloader

Overview

The USB module autoloader is a module that automatically loads USB device drivers according to a driver database file. USB device drivers must support autoloading.

The purpose of autoloading is to conserve IOP memory.

Modems, for example, require a driver for each type of equipment because 3rd party modem drivers do not conform to the standard specification. If autoloading is used, IOP memory is conserved because the autoloader can load only the driver of the type of equipment that has been installed.

The USB module autoloader is located between the USB driver (usbd.irx) and the various types of USB device drivers (such as usbmouse.irx).

Figure 4-1: Configuration Diagram

IOP Application			
usbmouse.irx	usbkeybd.irx	etc...	
usbmload.irx			
usbd.irx			
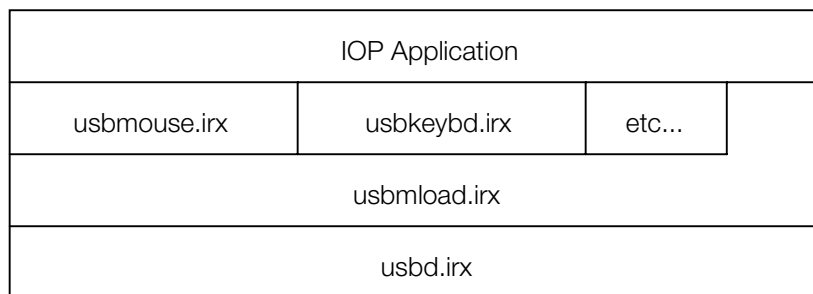
Changes Associated With Unload Support

Unloading is supported beginning with Release 2.3.4.

- When usbmload.irx is unloaded, it unloads the previously loaded LDD.
- The LDD must also support unloading. Although a conventional LDD can also be used, it cannot be unloaded (see the IOP kernel overview).
- Since functions to be registered with sceUsbmlRegisterLoadFunc() will no longer be compatible, they must be rewritten (see the function reference). In addition, an IRX module that contains a function that will be registered must also support unloading.

Sample Programs

The following sample programs support usbmload.

- **sce/iop/sample/usb/activate/***
  This is the simplest sample.
  Use it when you want to quickly run usbmload.

- **sce/iop/sample/usb/usbloadf/***
  This sample uses all usbmload functions.
  It also shows how to create a function that will be registered with sceUsbmlRegisterLoadFunc().

- **sce/iop/sample/usb/usbmouse/***
  This is a sample LDD for a mouse that supports usbmload and unloading from an external source.

- **sce/iop/sample/usb/usbkeybd/***
  This is a sample LDD for a keyboard that supports usbmload but does not support Unload.

- **sce/iop/sample/usb/selfunld/***

  This is a sample LDD that can unload itself.

  Although it was created based on the mouse sample, the RPC server has been removed.

  It is intended as a sample LDD for a device that does not require an RPC such as a modem driver.

LDDs that can unload themselves should be limited to those that do not use RPC. This is because the current RPC specification does not take into account the fact that the RPC server is dynamically unloaded (if the client executes SifCallRpc after the server is unloaded, no error will occur but processing will freeze).

To implement this in spite of the difficulty, consider a method that also implements an RPC server on the EE, reports that the LDD will unload itself, and performs the removal after receiving permission from the EE. Note, however, that this is rather cumbersome.

We plan to investigate a solution that will also include enhancements to RPC.

Principles of Autoloader Operation

The USB module autoloader is implemented as one Logical Device Driver (LDD).

The usb driver (usbd.irx) manages LDDs with a list structure. However, usbmload.irx is always located at the end of the list. No matter how many LDDs are registered or deleted, usbmload.irx will be located last.

Figure 4-2: Sample LDD List including the Autoloader
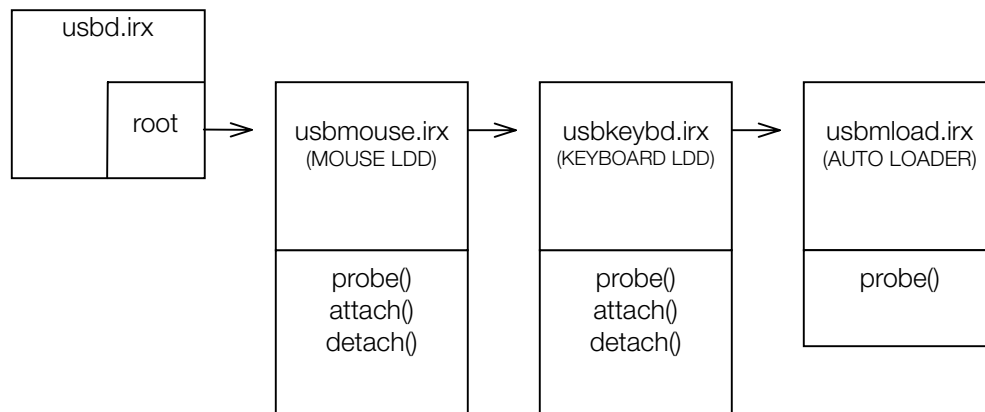
A USB device driver (LDD) has the external functions probe, attach, and detach.

- probe function: Decides whether or not the device driver itself is responsible for the inserted device
- attach function: Starts device execution (communication)
- detach function: Called to perform termination processing when the device is removed

When a USB device is inserted in the PlayStation 2, usbd.irx calls the probe function of each USB device driver in the order that the device drivers are listed. The LDD for which the probe function is called decides whether or not the LDD itself is responsible for the inserted device and returns a value.

If the return value of the probe function has the meaning "I am responsible for that device," usbd.irx calls the attach function of the same LDD. The LDD for which the attach function was called starts communication with the device.

When a USB Device That Corresponds to an LDD is Inserted in the PlayStation 2

Assume, for example, that a USB keyboard is inserted in the PlayStation 2 under the conditions shown in Figure 4-2.

usbd.irx calls the LDD probe functions in the order that the LDDs are listed.

When the usbkeybd.irx probe function is called, usbd.irx learns that usbkeybd.irx is responsible for the inserted device and calls the usbkeybd.irx attach function.

Then, communication with the USB keyboard begins.

When a USB Device That Corresponds to an LDD That Has Not Been Loaded is Inserted in the PlayStation 2

Assume, for example, that a USB modem is inserted in the PlayStation 2 under the conditions shown in Figure 4-2.

usbd.irx calls the LDD probe functions in the order that the LDDs are listed. However, since no LDD corresponding to the USB modem has been loaded, the final usbmload.irx probe function is called.

usbmload.irx, for which the probe function was called, references the driver database file (which will be described later) and loads the appropriate driver.

Mechanism for Loading a Driver

usbmload.irx creates a "driver loading thread" so that USBD operation will not be disturbed. This thread loads the driver.

Figure 4-3: Mechanism for Loading a Driver
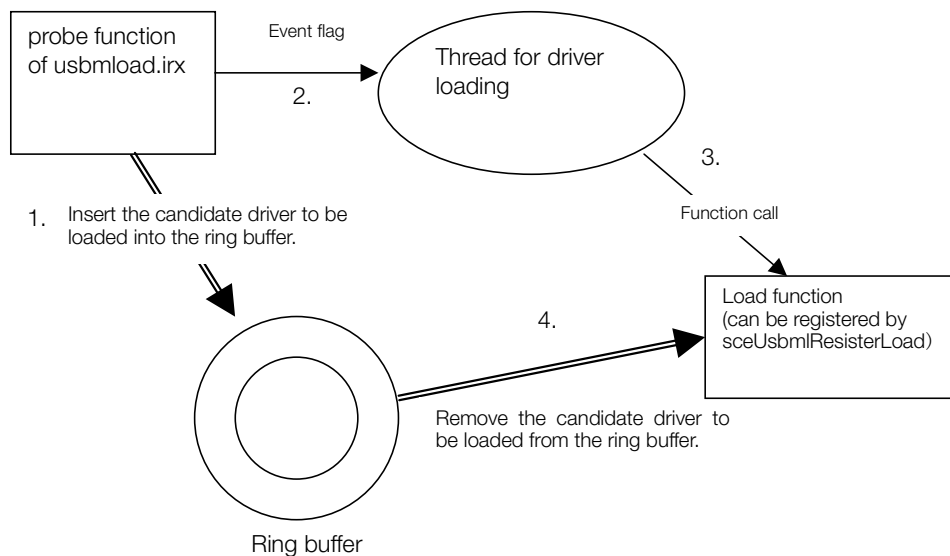
The usbmload.irx probe function references information in the driver database file (which will be described later) and enters information about the candidate driver (USBDEV_t structure) in a ring buffer (1). It also raises the event flag and immediately returns (2).

The "driver loading thread" that was launched according to the flag calls the "load function" (3).

The "load function" fetches the information about the candidate driver from the ring buffer and loads the driver (4).

As soon as that driver is loaded, the probe function of that driver is called. If the return value of the probe function has the meaning "I am responsible for that device," that driver is made resident. Otherwise, the driver is deleted from memory.

If the driver was not made resident, the next candidate is fetched from the ring buffer and loaded again.

Driver Database File

The driver database file is read at the following times.

- When the driver database file path is specified as an argument for usbmload.irx
- When the sceUsbmlLoadConffile function is called

The driver database file contains descriptions of the following parameters for each driver.

Table 4-1

Parameter Name	Value
DeviceName	Driver name (string). Represents beginning of driver settings. SJIS code can be used only for this keyword.
Use	0: Cannot be used; 1: Can be used (usbmload.irx reads information only if Use=1)
Vendor	Device descriptor vendor ID. Can be omitted when wildcard (*) is used.
Product	Device descriptor product ID. Can be omitted when wildcard (*) is used.
Release	USB-Spec Release number. 0x0110 indicates USB-Spec1.1. Can be omitted when wildcard (*) is used.
Class	Interface descriptor class code. Can be omitted when wildcard (*) is used.
SubClass	Interface descriptor subclass code. Can be omitted when wildcard (*) is used.
Protocol	Interface descriptor protocol code. Can be omitted when wildcard (*) is used.
Category	Category (ASCII code string). Required for Activate (which is described later). Case-sensitive.
DriverPath	Driver file path.
DriverArg	Arguments passed to driver. Up to eight can be specified. Actually, "lmode=AUTOLOAD" is added to specified arguments. If omitted, only "lmode=AUTOLOAD" is passed. The total length of all arguments including NULL terminators (also including "lmode=AUTOLOAD") must not exceed 256 characters.

Parameters with arbitrary names can be added.

Usbmload.irx ignores those parameters. No error will occur.

One line must not contain more than 256 half-width characters including the newline code.

A line that begins with "#" is treated as a comment and ignored.

Specify the word "end" at the end of the file as an end-of-file indicator.

DeviceName Naming Conventions

Use the following naming convention for DeviceName.

manufacturer-name model-number [device-type]

The model-number may be nickname+model-number.

Examples:

SCEI yyyy modem SCPH-xxxxx [MODEM]
zzzz, Inc. ABC-10000 [TA]

Category Usage Rules

Use the Category parameter according to the following rules.

- **When SCE standards are followed**

    For a driver that is used by SCE's INET library, only use an SCE-defined Category.

    Currently, the following three categories have been defined.

Table 4-2

Category	Target Device
Mouse	Mouse
Keyboard	Keyboard
Modem	Modem, TA, or mobile phone connection cable

If any other category is required, inquire via the developer support web site.

- **When using a licensee-proprietary Category**

    To use a licensee-proprietary Category, you can define the Category freely as long as it starts with "LOCAL_".

Sample Driver Database File

An entire driver database file is shown below.

"XXXX" represents a four-digit hexadecimal number.

```
######################################################
#                                                    #
#              USB module auto loader                #
#                   Config file                      #
#                                                    #
######################################################

DeviceName "Standard USB keyboard"
     Use      1
     Class    03
     SubClass 01
     Protocol 01
     Category Keyboard
     DriverPath   "/system/usb/usbkeybd.irx"

DeviceName "Standard USB Mouse"
```

```
                              Use       1
                              Class     03
                              SubClass 01
                              Protocol 02
                              Category Mouse
                              DriverPath    "/system/usb/usbmouse.irx"

                    DeviceName "MODEM1"
                              Use       0
                              Vendor    0xXXXX
                              Product   0xXXXX
                              Category Modem
                              DriverPath    "/system/usb/foom1.irx"
                              DriverArg     "dial=/system/conf/inet/foom1.irx"

                    DeviceName "MODEM2"
                              Use       0
                              Vendor    0xXXXX
                              Product   0xXXXX
                              Category Modem
                              DriverPath    "/system/usb/foom2.irx"
                              DriverArg     "dial=/system/conf/inet/foom2.irx"

                    DeviceName "MODEM3"
                              Use       1
                              Vendor    0xXXXX
                              Product   0xXXXX
                              Category Modem
                              DriverPath    "/system/usb/foom3.irx"
                              DriverArg     "dial=/system/conf/inet/foom3.irx"

                    end
```

Using usbmload.irx

usbmload.irx Syntax

Syntax

usbmload.irx [conffile=driver-database-path]
[rbsize=8-256] [debug=0|1|2]

Options

Table 4-3

Option Name	Contents
conffile	Specifies the driver database path. If this option is not specified, the file must be loaded by calling sceUsbmlLoadConffile().
rbsize	Specifies the size of the ring buffer for storing autoload candidates. The default is 32. The maximum and minimum are 256 and 8. The ring buffer requires 4 bytes per element.

Option Name	Contents
debug	Specifies the debug output level for the printf statement. 0: Title and fatal errors are output (default). 1: In addition to 0, certain operating conditions are output. 2: In addition to 1, the driver database file contents are output.

Using usbmload.irx

The entire processing flow when usbmload.irx is used is shown below.

1. Start up the EE application.
2. Reboot the IOP.
3. Load and start usbd.irx.
4. Load and start usbmload.irx (if conffile was specified, the driver database file is loaded here).
5. If conffile was not specified, use sceUsbmlLoadConffile() to load it.
6. Use sceUsbmlActivateCategory() to tell usbmload.irx the USB driver Category to be used (this is referred to as "activating" the driver).
7. Use sceUsbmlEnable() to enable autoloading.
8. At this time, driver autoloading will begin for USB devices that have already been flagged.
9. The state transitions to standby state.
10. If a USB device having an activated Category is inserted, the driver will be autoloaded enabling the device to be used.

Steps 3. and 4. above are specified as follows (this is done on the EE).

```
char option[] = "conffile=cdrom0:\\USBDRIVE.CNF\0debug=1";

sceSifLoadModule("cdrom0:\\USBD.IRX", 0, "");
sceSifLoadModule("cdrom0:\\USBMLOAD.IRX", sizeof(option), option);

/* IOP applications for which drivers are to be activated must be
loaded here */
```

Specifications of USB Drivers That Support the Autoloader

A USB driver that supports the autoloader must operate as follows, according to the argument specification.

- When "lmode=AUTOLOAD" is specified for the argument, whether or not the driver will be made resident is determined according to the result of the probe function.
- When the "lmode" argument isn't specified, the driver will always be resident (it is called with this method when autoloading isn't used).
- An LDD that can unload itself will be unloaded only when "selfunload=ON" is specified for the argument. By default, the self-unload function is turned off.
- To prevent duplicates, the module name specified in the ModuleInfo structure should be set to Manufacturer name + LDD name.
  Example: SCE_MOUSE_DRIVER

The following function that had been mandatory up to now was made optional beginning with Release 2.3.4. It need not be implemented.

- When "lmode=TESTLOAD" is specified for the argument, the driver will not be made resident regardless of the result of the probe function. Instead, the probe function result can be returned in the return value.

Sample Program

To satisfy the above specifications, the USB driver must be written as follows (the principles of operation are described later).

For information about external unloads and self-unloads, refer to the IOP kernel specification and samples.

```
static sceUsbdLddOps usbdriver_ops = {
        NULL, NULL,
        "usbdriver",
        usbdriver_probe,
        usbdriver_attach,
        usbdriver_detach,
};

static int resident_flag;

#define NORMAL_MODE    0
#define AUTOLOAD_MODE 1
#define TESTLOAD_MODE 2
static int load_mode;

int start(int argc, char* argv[]){
        int r;
        struct ThreadParam param;
        int th;
        int i,j;

        load_mode = NORMAL_MODE;

        for(i=0; i<argc; i++) {
          for(j=0; argv[i][j]!='\0'; j++) {
            if (argv[i][j] == '=') { argv[i][j]='\0'; j++; break; }
          }
          if (strcmp(argv[i],"lmode") != 0) { continue; }

          if (strcmp(argv[i]+j,"AUTOLOAD") == 0)
            { load_mode = AUTOLOAD_MODE; break; }
          if (strcmp(argv[i]+j,"TESTLOAD") == 0)
            { load_mode = TESTLOAD_MODE; break; }
        }
        switch(load_mode)
          {
          case NORMAL_MODE:
            resident_flag = 1; /* Always make resident (normal use) */
            break;
          case AUTOLOAD_MODE:
            printf("usbdriver : AUTOLOAD MODE\n");
            resident_flag = 0; /* Make resident only when probe
function raised flag */
            break;
          case TESTLOAD_MODE:
            printf("usbdriver : TESTLOAD MODE\n");
            resident_flag = 0; /* Not made resident regardless of flag
*/
            break;
```

```
                    }

                if(0 != (r = sceUsbdRegisterLdd(&usbdriver_ops))) {
                  /* probe function called internally by sceUsbdRegisterLdd
function */
                  printf("usbdriver: sceUsbdRegisterLdd -> 0x%x\n", r);
                }

                if (load_mode == TESTLOAD_MODE) {
                  /* Since mode is TESTLOAD, not made resident regardless of
resident_flag */
                  sceUsbdUnregisterLdd(&usbdriver_ops);
                    /* When mode is TESTLOAD, result of probe function is
returned as TESTLOAD_OK or TESTLOAD_NG */
                  if (resident_flag == 1) {
                    return (NO_RESIDENT_END | TESTLOAD_OK);
                  } else {
                    return (NO_RESIDENT_END | TESTLOAD_NG);
                  }
                }

                if ( ! resident_flag ) {
                  /* When resident_flag == 0, not made resident */
                  sceUsbdUnregisterLdd(&usbdriver_ops);
                  return NO_RESIDENT_END;
                }

                  (Other processing)

                return RESIDENT_END;  /* Make resident */
        }

        static int usbdriver_probe(int dev_id){

                (Decision processing)

                if (this USB driver is responsible for inserted device) {
                resident_flag = 1;   /* Set resident decision flag ON */
                if (load_mode == TESTLOAD_MODE)
                  { return(0); }  /*  */
                else
                    { return(1); }
                } else {
                  return 0;
                }

        }

        static int usbdriver_attach(int dev_id){

                (Transfer start processing)

        }

        static int usbdriver_detach(int dev_id){

                (Disconnect processing)

        }
```

Principles of Operation

An LDD that supports the autoloader is loaded and made resident according to the principles shown in Figure 4 (when lmode=AUTOLOAD).

The "load function" in the same figure is a function that was registered by sceUsbmlRegisterLoadFunc() or a function that is built into usbmload.irx by default.

Compare this with the program listing in the previous section.

Figure 4-4: Operation of an LDD That Supports usbmload.irx