Semiconductors

AN10042

ISP176x Linux Programming Guide

Rev. 01 — 7 April 2005 **Application note**

Document information

Info	Content
Keywords	isp1760, isp1761, universal serial bus, usb, peripheral controller, host controller, on-the-go controller, hal
Abstract	This document provides information on the interfaces and data structures required to use the ISP176x Host Controller, OTG Controller and Peripheral Controller driver layers for the Linux operating system. **Remark**: The ISP176x denotes the ISP1760 and ISP1761 Hi-Speed Universal Serial Bus controllers, and any future derivatives.

PHILIPS

Revision history

Rev	Date	Description
01	20050407	First release.

Contact information

For additional information, please visit: http://www.semiconductors.philips.com

For sales office addresses, please send an email to: sales.addresses@www.semiconductors.philips.com

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Remark: The ISP176x denotes the ISP1760 and ISP1761 Hi-Speed Universal Serial Bus Controllers, and any future derivatives.

1. Introduction

The Universal Serial Bus (USB) Host Controller has become an integral part of most embedded systems in recent years. Usually, the Host Controller is based on the PCI card that is targeted for PC-based architecture. Embedded systems that are not equipped with such a controller bus can benefit from the Philips embedded Host Controller.

In addition to the host functionality, some embedded systems need the peripheral functionality. Such embedded systems must contain a USB Host Controller and a USB Peripheral Controller as part of the On-The-Go (OTG) implementation. OTG is a supplement to *Universal Serial Bus Specification Rev. 2.0* that allows access to the USB host and the USB peripheral through a single physical connector. The OTG protocol entails switching between the host and peripheral functionalities.

The ISP1760 is a USB Host Controller. The ISP1761 is a USB OTG Controller—USB host and USB peripheral. The ISP1760 has three host ports and the ISP1761 has two host ports and an OTG port that can be used as either a host or a peripheral.

This document provides information on the interfaces and data structures required to use the ISP176x Host Controller, OTG Controller and Peripheral Controller driver layers for the Linux operating system.

Fig 1 shows the interfacing of the ISP176x blocks to an operating system.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

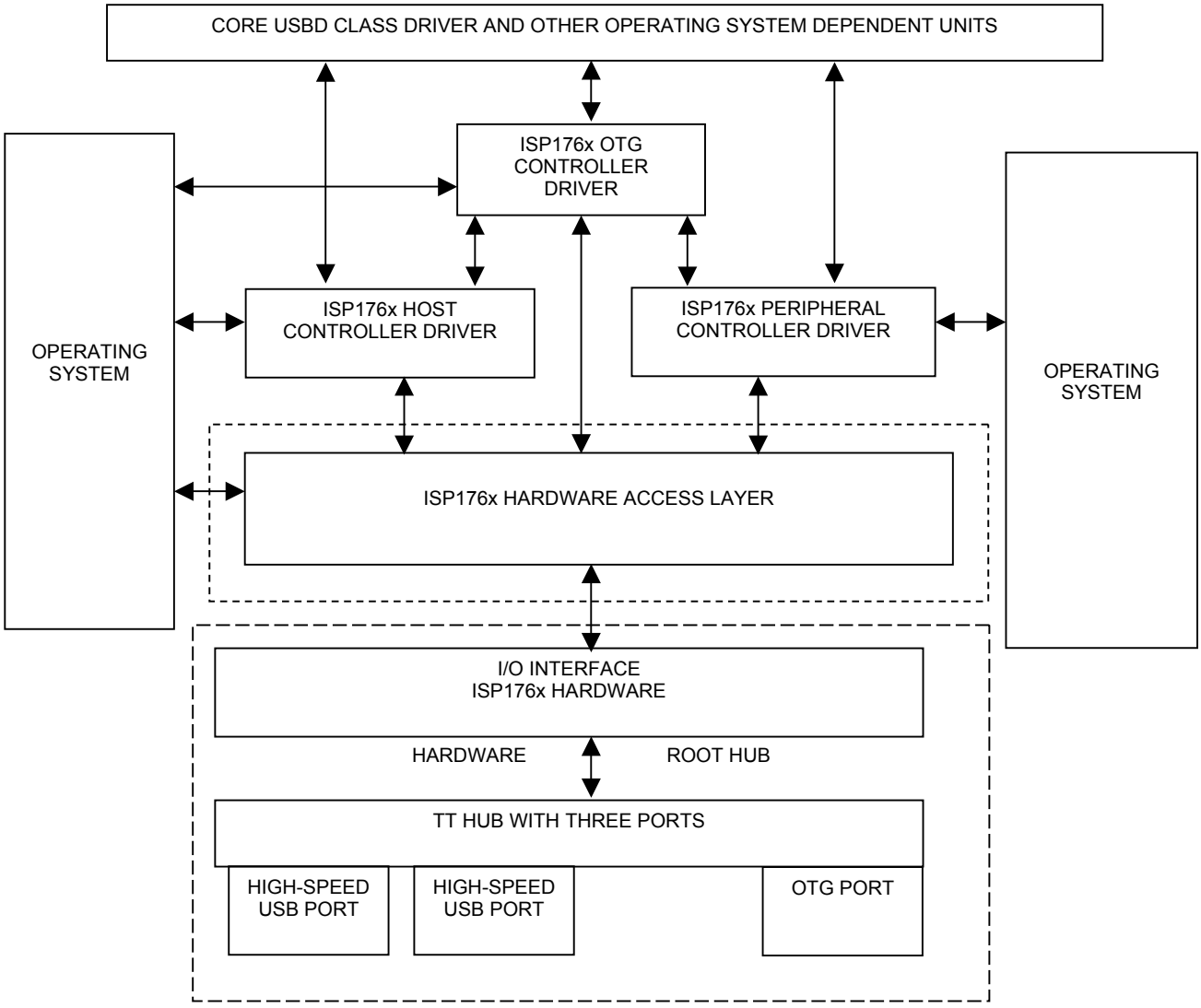


Fig 1. ISP176x system interface.

1.1 ISP176x peripheral hardware

The ISP176x Peripheral Controller includes the ISP1582 and ISP176x peripheral function. Hardware Access Layer can run on the Peripheral Controller hardware.

Remark: In this document, the ISP176x Peripheral Controller includes the ISP1582, unless mentioned otherwise. The ISP1582 hardware can be on any platform with any bus interface that is defined for the ISP176x.

1.2 ISP176x host hardware

The ISP176x provides the host-capability function. The ISP176x contains an OTG state machine running within. If the device connected to the port is dual-role capable, then the device negotiates the role and acts accordingly under the direction of the software.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

1.3 ISP176x Hardware Access Layer

The ISP176x Hardware Access Layer provides functions to access the ISP176x hardware by using the NOR flash interface and OS platform-related functions. This layer depends on the platform and the NOR flash interface hardware. Port this layer, depending on the platform used. This document provides information on interfacing the ISP176x Hardware Access Layer to other drivers.

1.4 ISP176x Host Controller Driver

The ISP176x Host Controller Driver (HCD) transfers data on the USB bus for the USB devices connected to the downstream facing port of the ISP176x. This layer must interface to the Hardware Access Layer to configure the ISP176x hardware and the Host Controller hardware access.

1.5 ISP176x Peripheral Controller Driver

The ISP176x Peripheral Controller Driver transfers data over the USB cable to the connected USB host through the upstream facing port of the ISP176x.

1.6 ISP176x OTG Controller driver

The OTG driver maintains the OTG Finite State Machine (FSM) by accessing and controlling the OTG controller registers in the ISP176x through the Hardware Access Layer.

2. Hardware Access Layer

The Hardware Access Layer provides functions to access the host hardware and the OS platform-related functions. The implementer must port this layer based on the platform.

2.1 Starting the Host Controller

The following steps are required to set the Host Controller into operational mode.

1. Reset the Host Controller using the following code and wait for 50 ms to stabilize.

```
HostWriteReg_32Bit(0x030C /*REG_RESETDEVICE*/, 0x00000002UL);
Delay(50ms);
```

2. Set the Host Controller into 32-bit bus mode and 16-bit bus mode as follows:

```
HostWriteReg_32Bit(0x0300/*REG_HWMODECTRL*/,0x00000000); for 16-bit mode
HostWriteReg_32Bit(0x0300/*REG_HWMODECTRL*/,0x00000100); for 32-bit
Mode. The HC is in 32-bit Bus mode at power-on reset.
```

3. You can now write to and read from the Scratch register to verify that the ISP176x I/O operation is working properly.

```
HostWriteReg_32Bit(0x0308 /*REG_HCSCRATCH */, uData2Write);
/*Read Back the Scratch Values */
HostReadReg_32Bit(0x0308 /*REG_HCSCRATCH */, uData2Write);
if(uData2Write!=uData2Read)
{ }
```

4. Write appropriate values to the following registers to finally set the Host Controller into operational mode.

```
HostWriteReg_32Bit(0x0134 /*REG_ISOPTD_SKIP */,0xFFFFFFFF);
HostWriteReg_32Bit(0x0144 /* REG_INTPTD_SKIP */,0xFFFFFFFF);
HostWriteReg_32Bit(0x0154 /* REG_ATLPTD_SKIP */,0xFFFFFFFF);

HostWriteReg_32Bit(0x0130 /* REG_ISOPTD_DONE */,~0xFFFFFFFF);
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
HostWriteReg_32Bit(0x0140 /* REG_INTPTD_DONE */,~0xFFFFFFFF);
HostWriteReg_32Bit(0x0150 /* REG_ATLPTD_DONE */,~0xFFFFFFFF);

HostWriteReg_32Bit(0x0028 /* REG_USBINTR */,0x0);
HostWriteReg_32Bit(0x031C /* REG_INTIRQMASKOR */,0x0);
HostWriteReg_32Bit(0x0328 /* REG_INTIRQMASKAND */,0x0);
HostWriteReg_32Bit(0x0320 /* REG_ATLIRQMASKOR */,0x0);
HostWriteReg_32Bit(0x032C /* REG_ATLIRQMASKAND */,0x0);
HostWriteReg_32Bit(0x0318 /* REG_ISOIRQMASKOR */,0x0);
HostWriteReg_32Bit(0x0324 /* REG_ISOIRQMASKAND */,0x0);
HostWriteReg_32Bit(0x0030 /* REG_CTLDSSEGMENT */,0);
/* Enable the Port, before enable, reset and power up and enable the port */
HostWriteReg_32Bit(0x0064 /* REG_PORTSC1 */,0x100);  /*Reset */
Delay(20ms);/* Port Reset delay */

HostWriteReg_32Bit(0x0064 /* REG_PORTSC1 */,0x1000); /*Power */
HostWriteReg_32Bit(0x0064 /* REG_PORTSC1 */,0x04);   /*Enable Port */

HostWriteReg_32Bit(0x0314 /* REG_HCINTRENBL */,0x000001B1); /*Enable int for ATL
PTD done,INTL PTD done, SOF and HC Suspend */

HostReadReg_32Bit(0x0300 /* REG_HWMODECTRL */, &uData2Read);
HostWriteReg_32Bit(0x0300/*REG_HWMODECTRL*/,(uData2Read|0x61));

/*Enable the Global Int,Dreq Polarity */
HostWriteReg_32Bit(0x0060 /* REG_CONFIGFLAG */,1);
```

The Host Controller will now start to send SOFs on the bus and SOF interrupts will start appearing on the interrupt pin of the Host Controller. For details on these registers and their values, refer to the ISP1760 and ISP1761 data sheets.

2.2 Module management interface

This interfaces to the operating system. It is called when the ISP176x HCD is loaded to or unloaded from the kernel.

The following functional interface is based on PCI x86 platform or Accelent Linux platform. The functional interface can, however, be modified, depending on the operating system.

2.3 isp176x_hal_module_init

This function initializes the ISP176x hardware access driver module. The Linux kernel module manager calls this function.

```
int __init isp176x_hal_module_init (void)
```

Parameters: None.

Return value:

0: The ISP176x hardware access driver kernel module is successfully completed.

<0: The ISP176x kernel module initialization has failed.

2.3.1 isp176x_hal_module_cleanup

This function deinitializes the ISP176x hardware access driver module. The Linux kernel module manager calls this function during unloading of this module.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
void __exit isp1761_hal_module_cleanup (void)
```

Parameters: None.

Return value: None.

2.4 ISP176x Controller Driver interface

This interfaces to the ISP176x controller drivers: host, peripheral, and OTG. It includes the following interfaces:

- Driver registration interface
- Resource Management interface
- I/O access interface
- Kernel tracing interface.

These interfaces are explained in the following sections.

2.5 Driver registration interface

2.5.1 isp176x_register_driver

This function registers driver access functions to the ISP176x Hardware Access Layer driver.

```
int   isp176x_register_driver(struct isp1761_driver *drv)
```

Parameters:

drv: Pointer to the ISP176x driver data structure (struct isp176x_driver). The structure has the following elements.

```
  struct isp176x_driver {
  struct list_head node;
  char       *name;
  unsigned long index;
  int (*probe)(struct isp176x_dev *dev);
  void (*remove)(struct isp176x_dev *dev);
  void (*suspend)(struct isp176x_dev *dev);
  void (*resume)(struct isp176x_dev *dev);
} isp_176x_driver_t;
```

node: Linked list node. Managed by the ISP176x Hardware Access Layer.

name: Name of the driver registering to the Hardware Access Layer.

index: Driver type. Its values are given in Table 1:.

Table 1: Driver type

Value	Description
ISP176x_HC	USB Host Controller device
ISP176x_DC	USB Peripheral Controller device
ISP176x_OTG	USB OTG Controller device

probe: This probe function is called by the Hardware Access Layer when it finds the hardware of the type specified by the index. The input parameters to this function are the ISP176x device data structure (struct isp176x_dev).

remove: This is a removal function. The Hardware Access Layer calls this function when it finds that the hardware is unavailable or inactive. The input

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

parameters to this function are the ISP176x device data structure (struct isp176x_dev).

suspend: This function is called by the Hardware Access Layer when it finds that the hardware must be suspended. The input parameters to this function are the ISP176x device data structure (struct isp176x_dev). This function interface is applicable only when power management is enabled.

resume: This function is called by the Hardware Access Layer when it finds that the hardware must be resumed from the suspended state. The input parameters to this function are the ISP176x device data structure (struct isp176x_dev). This function interface is applicable only when power management is enabled.

Return value:

0 Successful registration of the driver with the Hardware Access Layer.

< 0 OTG driver registration has failed.

2.5.2 isp176x_unregister_driver

This function deregisters controller drivers from the ISP176x Hardware Access Layer.

```
void  isp176x_unregister_driver(struct isp176x_driver *drv)
```

Parameters:

drv: Pointer to the controller driver registration data structure.

Return value: None.

2.6 Resource management interface

These functions are usually required for the bus, such as PCI, to acquire the memory region and ports from the bus.

2.6.1 isp176x_check_io_region

This function checks whether the I/O port region is free. The I/O region that will be checked is specified in the input parameter data structure.

```
int   isp176x_check_io_region(struct isp176x_dev *dev)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value:

0 I/O region of the specified device is free.

< 0 I/O region of the specified device is already in use.

2.6.2 isp176x_request_io_region

This function allocates I/O port resources to the specified controller device.

```
struct resource* isp176x_request_io_region(struct isp176x_dev *dev)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value:

NULL: Resource allocation has failed.

Others: Resource allocation was successful (pointer to struct resource defined by the Linux kernel).

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

2.6.3 isp176x_release_io_region

This function deallocates I/O port resources of the specified controller device.

```
void  isp176x_release_io_region(struct isp176x_dev  *dev)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value: None.

2.6.4 isp176x_request_irq

This function registers an Interrupt Service Routine (ISR) to the interrupt line. The interrupt line is specified in the device data structure elements.

```
int isp176x_request_irq(
void (*handler)(struct isp176x_dev dev*, void *isr_data),
   struct isp176x_dev *dev, void *isr_data);
```

Parameters:

handler: This function is called whenever the Hardware Access Layer receives an interrupt on the device interrupt line. The input parameters to this function are the ISP176x device data structure (dev) and the controller driver ISR data (isr_data).

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

isr_data: Pointer to the controller data identifier. This is an input parameter when the ISR is called.

Return value:

0 ISR registration is successful.

<0 ISR registration has failed.

2.6.5 isp176x_free_irq

This function frees the ISR from the interrupt line of the device.

```
void  isp176x_free_irq(struct isp176x_dev  *dev,
void *isr_data)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

isr_data: Pointer to the controller data (identifier).

Return value: None.

2.7 I/O access Interface

The functions described in the following sections access the ISP176x hardware and are not exported to host, peripheral or OTG stack.

2.7.1 isp176x_reg_read32

This function reads the 32-bit ISP176x register.

```
__u32 isp176x_reg_read32(struct isp176x_dev *dev, __u16 reg)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

reg: Register index of the ISP176x device.

Return value:

32-bit register content.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

2.7.2 isp176x_read16

This function reads the 16-bit ISP176x data.

```
__u16 isp1761_reg_read16(struct isp176x_dev *dev)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value:

16-bit data content.

2.7.3 isp176x_reg_write32

This function writes to the 32-bit ISP176x register.

```
void  isp176x_reg_write16(struct isp176x_dev *dev, __u32 reg, __u16 data)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

reg: Register index of the ISP176x device.

data: Data to be written to the ISP176x.

Return value: None.

2.7.4 isp176x_reg_write16

This function writes to the 16-bit ISP176x register.

```
void  isp176x_reg_write16(struct isp176x_dev *dev, __u16 reg, __u16 data)
```

Parameters:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

reg: Register index of the ISP176x device.

data: Data to be written to the ISP176x.

Return value: None.

2.7.5 isp176x_read16

This function reads the memory from the ISP176x in the 16-bit format. The following sample code is for access in PIO mode.

```
__u16 isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value:

16-bit data content.

2.7.6 isp176x_read32

This function reads the memory from the ISP176x in the 16-bit format. The following sample code is for access in PIO mode.

```
__u32 isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value:

32-bit data content.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

2.7.7 isp176x_write16

This function writes to the memory from the ISP176x in the 16-bit format. The following sample code is for access in PIO mode.

```
__u16 isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value: None.

2.7.8 isp176x_write32

This function writes to the memory from the ISP176x in the 16-bit format. The following sample code is for access in PIO mode.

```
__u16 isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev).

Return value: None.

2.7.9 isp176x_Memory_read16_DMA

This function writes to the memory from the ISP176x in 16-bit format DMA mode.

```
BOOL isp176x_reg_read16(struct isp1761_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev) is filled with the memory descriptor buffer supplied.

Return value:

Operation successful = TRUE.

Operation failed = FALSE.

2.7.10 isp176x_Memory _read32_DMA

This function writes to the memory from the ISP176x in 16-bit format DMA mode.

```
BOOL isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev) is filled with the memory descriptor buffer supplied.

Return value:

Operation successful = TRUE.

Operation failed = FALSE.

2.7.11 isp176x_Memory _write16_DMA

This function writes to the memory from the ISP176x in 16-bit format DMA mode.

```
BOOL isp176x_reg_read16(struct isp176x_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev) filled with the memory descriptor buffer supplied.

Return value:

Operation successful = TRUE.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Operation failed = FALSE.

2.7.12 isp176x_Memory _write16_DMA

This function writes to the memory from the ISP176x in 32-bit format DMA mode.

```
BOOL isp176x_reg_read16(struct isp1761_dev *dev)
```

Parameter:

dev: Pointer to the ISP176x device data structure (struct isp176x_dev) filled with the memory descriptor buffer supplied.

Return value:

Operation successful = TRUE.

Operation failed = FALSE.

2.8 Kernel tracing interface

2.8.1 func_debug

This is a macro interface to print information at function level. Controller drivers call this function to print function entry traces.

```
void  func_debug(args)
```

Parameters:

args: The arguments of printk.

Return value: None.

2.8.2 detail_debug

This is a macro interface to print information at detailed-level trace. Controller drivers call this function to print function entry traces, as well as detailed-level information trace.

```
void  detail_debug(args)
```

Parameters:

The arguments of printk.

Return value: None.

2.9 Common structures

2.9.1 struct isp176x_dev

```
  struct isp176x_dev {
  struct  isp176x_driver *driver;
  void       *driver_data;
  unsigned char index;
  unsigned int  irq;
 void (*handler)(struct isp176x_dev *, void *);
  void       *isr_data;
  unsigned long int_reg;
  unsigned long alt_int_reg;
  struct resource  *io_res;
  unsigned long io_base;
  unsigned long io_len;
  unsigned long io_data;
  unsigned long io_cmd;
  unsigned short chip_id;
  char       name[80];
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
 void *           dma_buff
 int              dma_chan
   int        active;
}isp176x_dev_t ;
```

Parameters:

driver: Pointer to the driver data structure.

driver_data: Driver private data.

index: Controller type. The values are defined in the preceding code.

irq: Interrupt line allocated for this controller.

handler: Interrupt Service Routine of this handler used by the ISP176x Hardware Access Layer.

isr_data: Interrupt Service Routine parameter.

int_req: Interrupt register of the controller.

alt_int_reg: Alternate Interrupt register for the Host Controller.

io_res: Pointer to the I/O resources structure defined by the Linux kernel.

io_base: I/O access start base.

io_len: I/O access total length.

io_data: Data register I/O port.

io_cmd: Command register I/O port.

chip_id: Controller chip ID.

dma_buff: Buffer for the DMA transfer.

dma_chan: DMA channel number resource.

name: Peripheral Controller name.

active: Whether the device is active. 1—active; 0—inactive.

3. Host Controller interface

The HCD (HCD) transfers data to the connected USB devices and manages the Root Hub ports. The OTG driver controls activities on the OTG port by using the HCD port control interface.

3.1 Module management

This interfaces to the operating system and is called when the ISP176x HCD is loaded to or unloaded from the kernel.

The following functional interface is based on the PCI x86 platform or the Accelent Linux platform, and can be modified, depending on the operating system.

3.1.1 phci_module_init

This function initializes the ISP176x hardware access driver module. The Linux kernel module manager calls this function.

```
int __init phci_module_init (void)
```

Parameters: None.

Return value:

0 The ISP176x hardware access driver kernel module is successfully completed.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

< 0 The ISP176x kernel module initialization has failed.

3.1.2 phci_module_cleanup

This function deinitializes the ISP176x hardware access driver module. The Linux kernel module manager calls this function during the unloading of this module.

```
void __exit phci_module_cleanup(void)
```

Parameters: None.

Return value: None.

3.2 ISP176x host management service

This interfaces to the ISP176x HCDs. It has the following types of interfaces:

- Host Controller basics
- Host Controller routines
- Memory management interface
- Root hub and internal hub management
- Data transfer interface.

The following sections explain each of these interfaces in detail.

3.2.1 Host controller basics

Fig 2 shows a portion of the ISP176x Host Controller conceptual block diagram.

The Host Controller basics consists of the following:

- Host—Hi-Speed USB host
- Peripheral—Hi-Speed USB peripheral
- Hi-Speed USB hub with Transaction Translator (TT)[1]
- USB OTG.

1. Patent pending: TT under host.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

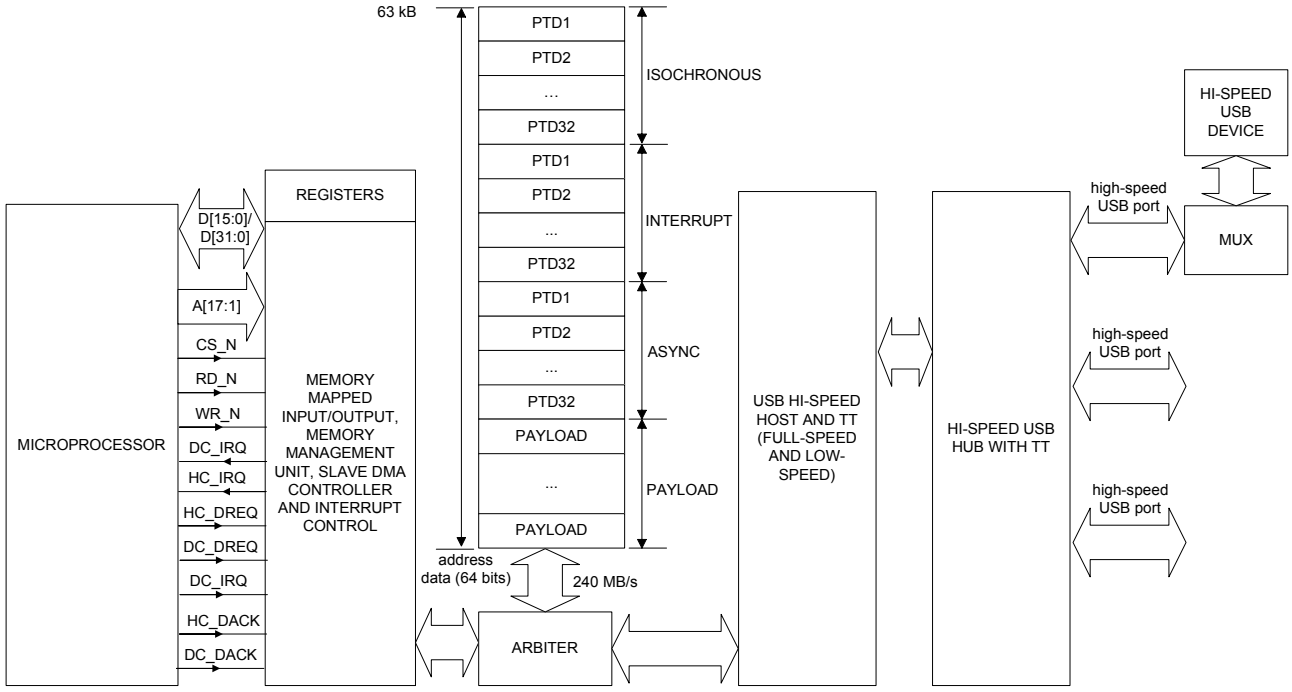


Fig 2. Conceptual block diagram of the Host Controller.

3.2.1.1 Host—Hi-Speed USB host

The transfers that arrive from the USB core driver in the form of the USB Request Block (URB) with the user buffer are scheduled over the USB bus in a round-robin method. The standard EHCI driver can be modified, without changing the code.

The main reason for not changing the EHCI driver code is to use the horizontal and vertical traversal rules set by the EHCI driver. To transfer the user buffer, EHCI uses the hardware schedule list that is traversed by the hardware and scheduled over the USB bus.

The ISP176x uses the software driven interrupt-based scheduler that pulls the TD out from the EHCI scheduler, schedules it, and completes the required schedule transaction. The transaction scheduler also removes the transactions from the memory once completed so that new transactions can be scheduled.

You can also use a wrapper around the TD; and map, link and traverse the enhanced Philips Transfer Descriptor[2] (PTD) used by the ISP176x.

Scheduling transfer over the ISP176x means scheduling the EHCI transfer over the shared memory. Memory schedules are organized by the software and filled in the shared memory of the ISP176x for the hardware to execute.

The Host Controller memory is divided into two parts: data payload area and header area. The data payload contains the data to be transferred. The header contains the PTD.

The ISP176x traverses the memory header using the linear method. This method of scheduling is of fixed priority, and achieves a more simpler and predicted traversal of the schedule for the Host Controller. A PTD is dynamically added and removed from the endpoint list by using the Skip bit to inform the Host Controller not to access the required transfer. The driver uses the Done bit to check that the transfer is complete.

2. Patent pending: PTD protocol.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

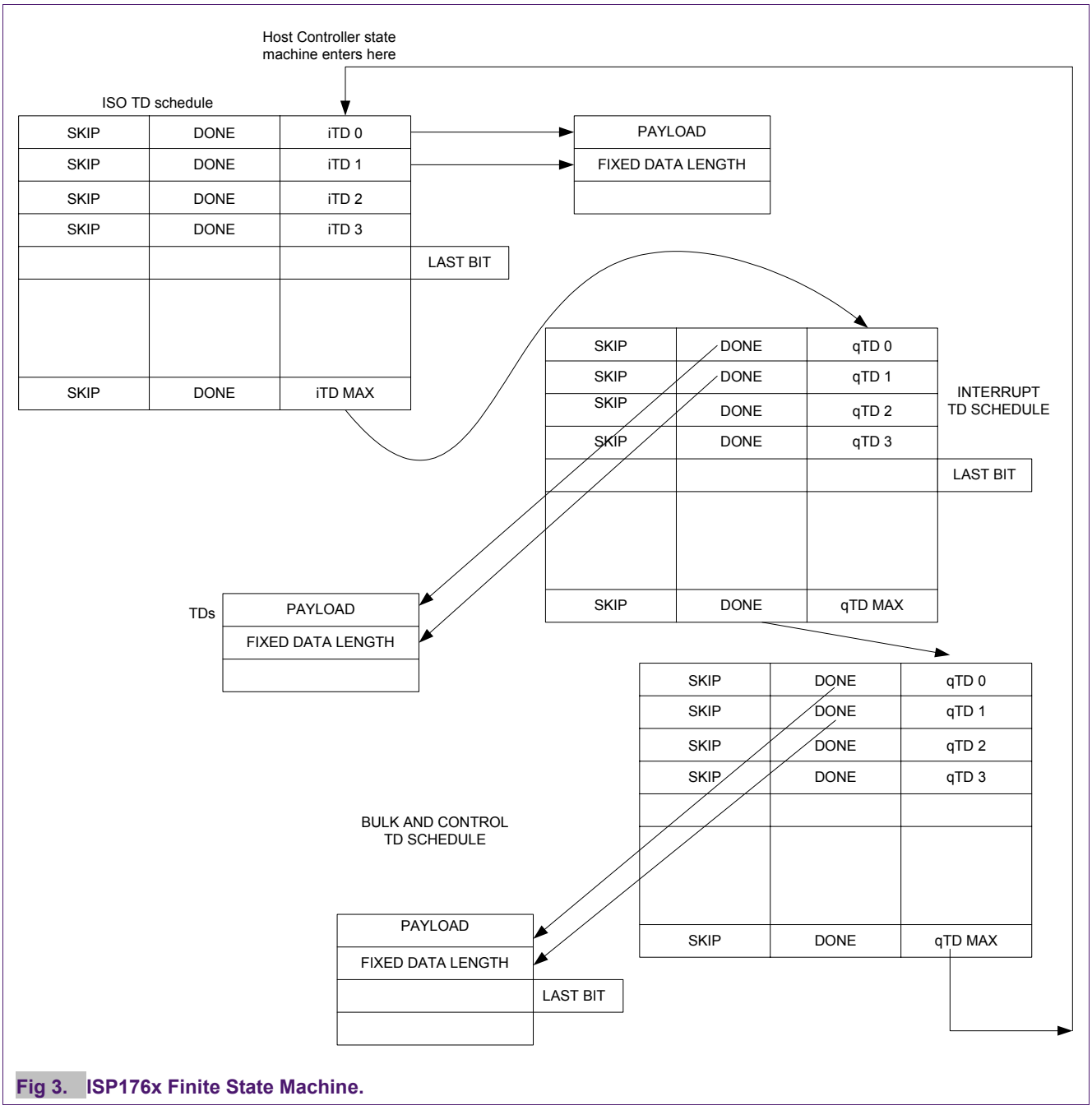


Fig 3. ISP176x Finite State Machine.

The ISP176x controller executes transactions for devices by using a simple and shared-memory schedule. This memory schedule is an extension to the EHCI memory schedule.

EHCI data structures are optimized for the bus master operation that is managed by the hardware state machine. New data enhanced PTD structures are the translations of the EHCI data structures that are optimized for the ISP176x. This is because the ISP176x is a slave Host Controller and has no bus master capability. The EHCI data structures are designed to provide the maximum flexibility required by USB, minimize memory traffic, and hardware and software complexity.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

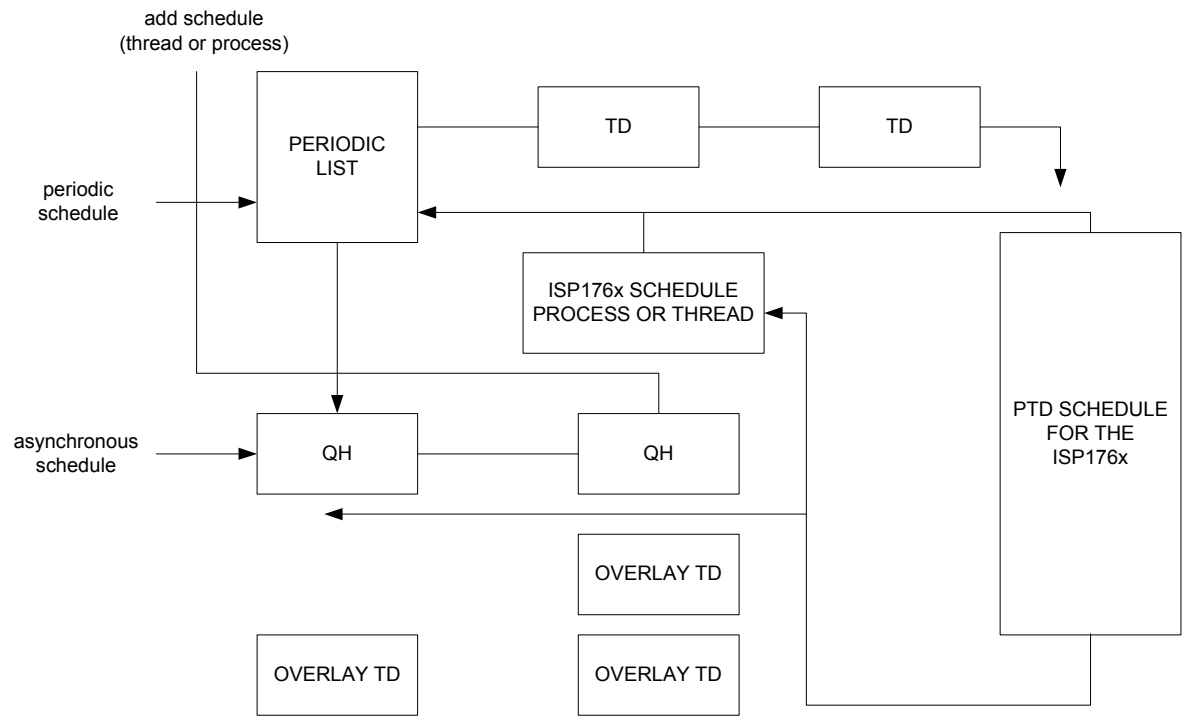


Fig 4. Scheduling a PTD.

The ISP176x controller executes transactions for devices by using a simple and shared memory schedule. The schedule consists of data structures that are organized into three lists:

- ISO: Isochronous transfer schedule list
- INTL: Interrupt transfer list
- ATL: Asynchronous transfer list for the control and bulk transfers.

The system software maintains two schedules for the Host Controller: periodic and asynchronous. The root of the periodic schedule is the PERIODICLISTBASE register, which is the physical memory base address of the periodic frame list. The periodic frame list is an array memory pointer. The objects referenced from the frame list must be valid schedule data structures. An asynchronous list base is also a common list of queue head (endpoint) that is served in a schedule. This endpoint data structure is linked to the EHCI transfer descriptor that is the valid schedule (queue TD).

The ISP176x has a maximum of 32 ISO, 32 INTL and 32 ATL TDs. These TDs are used as a channel to transfer data from the shared memory to the USB bus. These channels are allocated and deallocated on receiving the transfer (URB) from the core USB driver.

On allocating a channel, a queue-TD is converted into a PTD while scheduling the transfer to the ISP176x shared memory, along with the payload. A single transfer can allocate multiple channels.

Multiple transfers are scheduled to the shared memory for various endpoints by traversing the next link pointer provided by the EHCI data structure, until it reaches the Terminate bit in each microframe. If the schedule is enabled, then the Host Controller executes the ISO schedule, followed by the INTL schedule, and then the ATL schedule.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

The EHCI periodic and asynchronous lists are traversed by the software according to the EHCI traversal rule and are executed from the asynchronous schedule only after it encounters the end of the periodic schedule. The Host Controller traverses the ISO, INTL and ATL schedules. It fetches the element and begins traversing the graph of linked schedule data structures.

The last bit identifies the end of the schedule for each transfer type, indicating that the remaining channels are empty. Once this transition is made, the Host Controller executes from the next transfer descriptor in the schedule, till the end of the microframe.

The completion of a transfer is indicated to the software by the interrupt, which can be grouped under the various PTDs by using the AND or OR registers that are available for each schedule type—ISO, INTL and ATL. These registers are simple logic registers to decide the individual and group PTDs that can interrupt the CPU for a schedule, when the logical condition of the done bit is true in the shared memory, which completes an interrupt.

There are four types of interrupts in the ISP176x: ISO, INTL, ATL and SOF. The latency can be programmed in multiples of μSOF (125 μs).

A static TD schedules inside the ISP176x shared memory, allowing the next_TD mechanism that will enable the HCD to schedule multiple TDs of single endpoint type and reduce the interrupt to the CPU. This is achieved by a small modification to the traversal rule as shown in Fig 5.

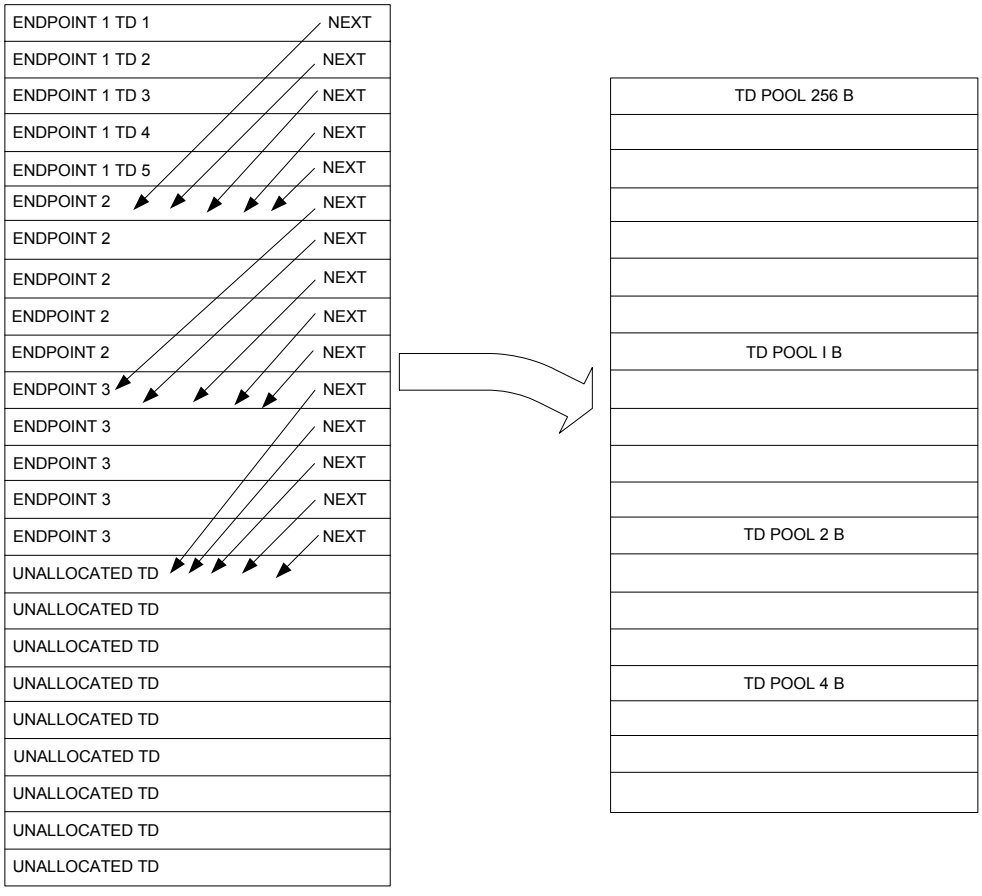
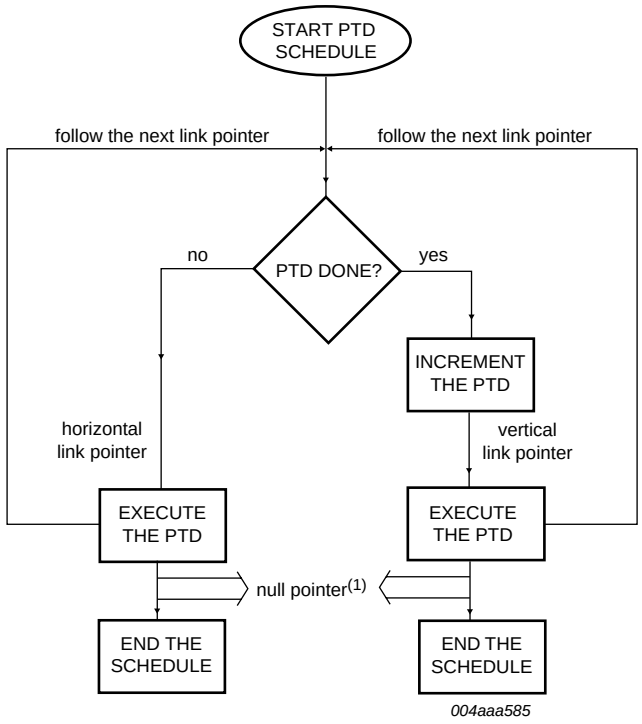


Fig 5. Endpoint traversal.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

The next_TD traversal rules for the hardware ISP176x are:

1. Start ATL header traversal.
2. If the current TD is active and not done, perform the transaction.
3. Follow the next link pointer.
4. If TD is not active and done, jump to the next TD.
5. If the next link pointer is NULL, it means the end of the traversal.



(1) The NULL pointer terminates. Goes to the next link.

Fig 6. NextPTD traversal rule.

3.2.1.2 Peripheral—Hi-Speed USB peripheral

The ISP176x Hi-Speed USB peripheral has the following features.

- Complete Hi-Speed USB peripheral.
- Separate endpoint memory.
- 1 control, 7 IN and 7 OUT endpoint that can be configured to bulk, interrupt and isochronous.
- Maximum endpoint memory is 1 kB.
- Command-data architecture for the data transfer.
- DMA transfer: 16-bit or 32-bit.
- Interrupt for every validated endpoint.

3.2.1.3 Hi-Speed USB hub with the TT

A Hi-Speed USB hub is always connected to the default root-hub port 0, which is an internal port. This hub is Hi-Speed USB root-hub, including the TT. After power-on reset, the Host Controller initializes the root hub and polls, until a new connection is found. The internal hub in enumerated and the polling on the internal root-hub is stopped according

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

to the USB 2.0 specification. On enumeration, it releases the three high-speed USB ports. You can connect the high-speed, full-speed and low-speed devices to this port. Port 0 of the Hi-Speed USB hub is gated through the multiplex for OTG purposes.

3.2.1.4 USB OTG

OTG enables the ISP176x to switch between host and peripheral. For details, see Section 4.

3.2.2 Host Controller routines

The TDs described in the following sections provide the basic routines to initialize and manage the Host Controller.

3.2.2.1 Phci_hcd_start

This function is used to initialize the Host Controller and set it in operation mode. It is recommended that you acquire a spin-lock after initialization so that no other function can preempt the process.

```
static int phci_hcd_start(phci_hcd *hcd)
```

Before the controller is set into operational mode, the following three operations are performed to set the Host Controller in reset mode and to enable interrupts.

1. Reset the device. phci_hcd_reset(hcd); see Section 3.2.2.2.
2. Enable the interrupt. phci_hcd_enable_interrupts(hcd) Section 3.2.2.3.
3. Initialize map buffers. phci_hcd_init_map_buffers(hcd) Section 3.2.2.4.

The HCD sets the last PTD bits of all the schedule types by writing 00000001h to HC_ISO_PTD_LASTPTD_REG (138h), HC_INT_PTD_LASTPTD_REG (148h) and HC_ATL_PTD_LASTPTD_REG (158h), indicating that the first TD is the last TD.

```
/*set last maps, for iso its only 1, else 32 tds bitmap*/
isp1762_reg_write32(hcd->dev, hcd->regs.atltdlastmap,0x80000000);
isp1762_reg_write32(hcd->dev, hcd->regs.inttdlastmap, 0x80000000);
isp1762_reg_write32(hcd->dev, hcd->regs.isotdlastmap, 0x00000001);
```

The HCD allocates a new dummy QH that is always linked to the asynchronous base address, according to the standard procedure. Initialize this QH and link it to the base address. The QH must halt the schedule to reclaim the bandwidth.

```
qh = phci_hcd_qh_alloc(hcd);
hcd->async = qh;
hcd->async->qh_next.qh = 0;
hcd->async->hw_next = QH_NEXT (hcd->async->qh_dma);
hcd->async->hw_info1 = cpu_to_le32 (QH_HEAD);
hcd->async->hw_token = cpu_to_le32 (QTD_STS_HALT);
hcd->async->hw_qtd_next = EHCI_LIST_END;
hcd->async->qh_state = QH_STATE_LINKED;
```

ISO transfers must not be active. The Host Controller data structure keeps track of the frame number. Initialize the frame number to –1 and the periodic schedule to 0 to indicate that ISO transfers are nonactive.

```
/*iso transfers are not active*/
hcd->next_uframe = -1;
hcd->periodic_sched = 0;
```

Initialize periodic list base addresses and periodic list heads with the appropriate values, depending on your program. Set the HCD state in the HCD structure to ready but do not process anything yet. Initialize the timer for the root hub polling. Poll until the device is

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

connected to the internal root hub using the appropriate method, depending on the USB core and the operating system. Now start enumerating the root hub, which is a hub that is controlled through register PORTSC1. This register also shows the status of the port.

Complete the Host Controller start routine by performing the following:

```
/*set the state of the host to ready */
hcd->state = USB_STATE_READY;
```

This completes the Host Controller initialization. The SOF is now on the internal root hub port. This allows to detect the port status change with a connection status change, allowing the internal hub to enumerate.

3.2.2.2 phci_hcd_reset(phci_hcd hcd)

The process start with the Host Controller reset: by writing to the RESET_HC bit in the HC_RESET_REG (030Ch) and waiting for 250 ms to complete the Host Controller reset. The Host Controller reset is indicated by setting bit RESET_HC in the Command register.

3.2.2.3 phci_hcd_enable_interrupts(phci_hcd hcd)

The ISP176x has four types of group interrupts and mechanisms:

- ATL asynchronous transfer group complete interrupt
- INTL interrupt periodic transfer group complete interrupt
- ITL isochronous periodic transfer group complete interrupt
- SOF start-of-frame group complete interrupt. Process all the schedules—ATL, ITL and INTL.

An interrupt indicates a transfer completion. To control an interrupt and relate to the individual transfer, the Host Controller provides registers—DONE_MAP, SKIP_MAP, INTERRUPT_OR and INTERRUPT_AND—to indicate the status of the transfer. These registers are hardwired to the address. The maximum transfer that can be scheduled to the ISP176x is fixed as 32 transfers per memory bank. These registers are per transfer type, except for the SOF interrupt that uses the respective registers while processing the type bank for the respective memory block.

This routine enables the required interrupt by programming HC_INTERRUPT_REG (310h).

This routine also programs HC_INT_THRESHOLD_REG (340h) that decides the maximum latency, the pulse width of the interrupt, the count in number of clocks, and the latency in number of µSOFs.

3.2.2.4 phci_hcd_init_map_buffers(phci_hcd hcd)

Map buffers are used for transfer management to transfer data between the EHCI TD and the embedded Host Controller specific PTD. To globally manage the transfer, map from TD to PTD and maintain the status of active channels.

This structure maintains the global position in the ISP176x buffer.

```
typedef struct td_ptd_map_buff {
   u8 buffer_type;  /*Buffertype: BUFF_TYPE_ATL/INTL/ISTL*/
   u8 active_ptds;  /* number of active td's in the buffer */
   u8 total_ptds;/* num of td's in the buffer (active + removed + skip) */
   u8   max_ptds;/* Maximum number of ptd's(32) this buffer can withstand */
   u32    active_ptd_bitmap; /* Active PTD's bitmap */
     u32    pending_ptd_bitmap;/* skip PTD's bitmap */
     td_ptd_map_t   map_list[TD_PTD_MAX_BUFF_TDS];/* td_ptd_map list */
}
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

This structure maintains the individual channel position of the current transfer:

```
typedef struct td_ptd_map {
   u32      state;      /* ACTIVE, NEW, TO_BE_REMOVED */
   u8       datatoggle; /*preserve the data toggle ATL/ISTL transfers*/
   //u16    total_bytes;  /* Number of bytes for this PTD &header) */
   u32      ptd_bitmap; /* Bitmap of this ptd in HC headers */
   u32      ptd_header_addr;/* headers address of  this td */
   u32      ptd_data_addr; /*data addr of this td to write in and read from*/
   /*this is address is actual RAM address not the
CPU address* RAM address = (CPU ADDRESS-0x400) >> 3 * */
   u32      ptd_ram_data_addr;
   u8    lasttd;       /*last td , complete the transfer*/
   struct ehci_qh *qh;       /* Queue head */
   struct ehci_qtd *qtd;        /* qtds for this endpoint */
   struct ehci_itd *itd;              /*itd pointer*/
   u32    grouptdmap;             /* complete with error, then process all the tds
              in the groupmap */
} td_ptd_map_t;
```

These buffers are initialized when starting the Host Controller.

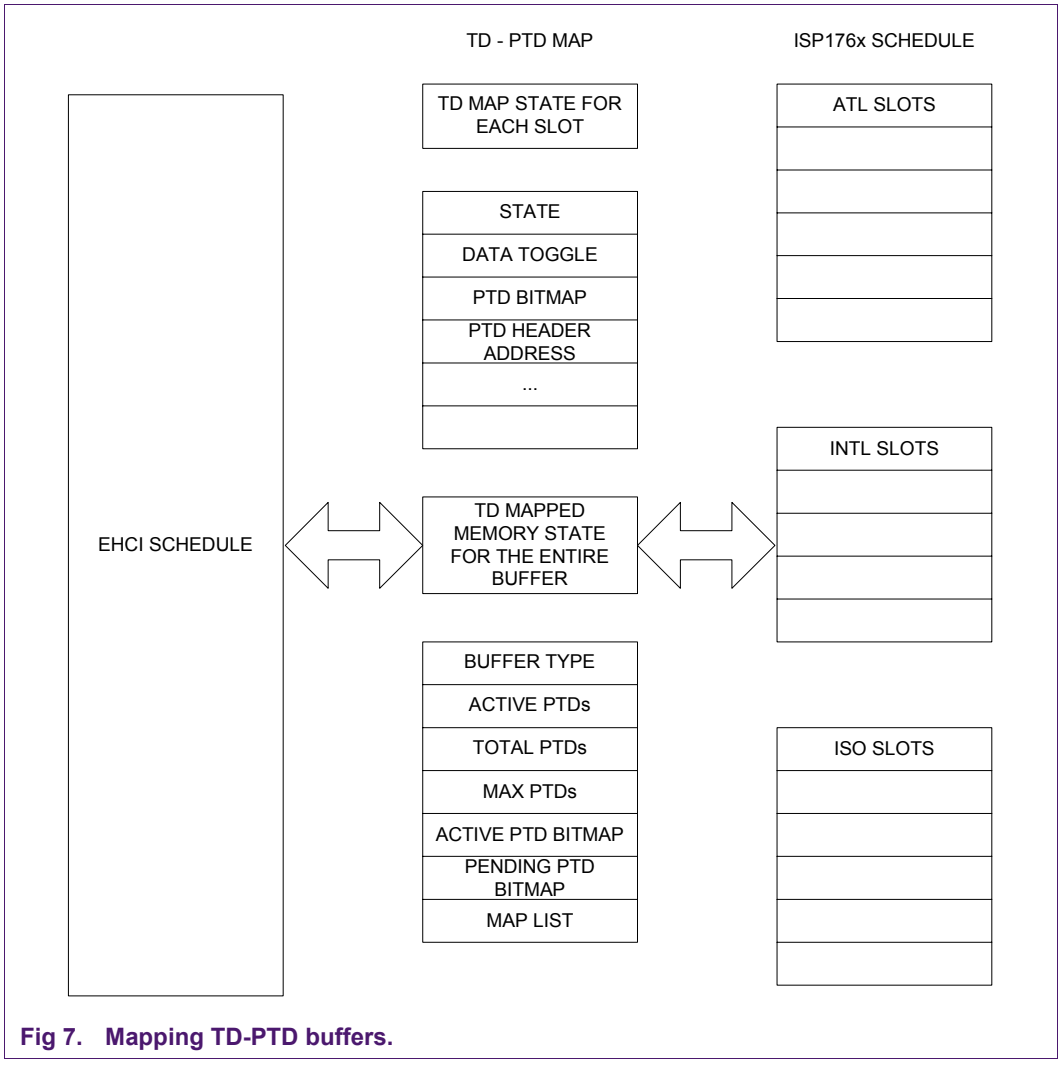


Fig 7. Mapping TD-PTD buffers.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

3.2.2.5 phci_hcd_start_controller(phci_hcd hcd)

This routine starts the Host Controller by setting the RUN/STOP bit in the USBCMD register to RUN and waiting for the bit to set, indicating that the Host Controller is started.

CONFIGFLAG indicates the hardware to set the Host Controller in EHCI mode. Bit 0 informs the Host Controller to set the default port routing to the EHCI host.

3.2.2.6 Int phci_suspend

This routine is called when all the ports are set to suspend state to put the controller into suspend state specified in the parameter.

```
Int phci_suspend (struct phci_hcd *dev, u32 state)
```

Parameters:

dev: Pointer to the ISP176x device structure (struct phci_hcd).

state: Integer value indicating suspend state.

Returns:

1: Success.

0: Failure.

3.2.2.7 Int phci_resume

This function is called before any device is set into resume state.

```
Int phci_resume (struct phci_hcd *dev, u32 state)
```

Parameters:

dev: Pointer to the ISP176x device structure (struct phci_hcd).

state: Integer value indicating resume state.

Return value:

1: Success.

0: Failure.

3.2.2.8 phci_stop

This routine sets the controller into stop state.

```
void phci_stop (struct phci_hcd  *dev)
```

Parameters:

dev: Pointer to the ISP176x device structure (struct phci_hcd).

Return value: void.

3.2.2.9 phci_get_frame_number

This routine is used to schedule transfers in the memory region.

```
int phci_get_frame_number(struct phci_hcd *dev)
```

Parameters:

dev: Pointer to the ISP176x device.

Return value:

Current frame number.

3.2.2.10 phci_hub_control

This routine is used for root hub operations.

```
int phci_hub_control(struct phci_hcd *dev, u16 ReqType,
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
u16 wValue, u16 wIndex, u16 wLength
char *buff)
```

Parameters:

dev: Pointer to the ISP176x device structure (struct phci_hcd).

ReqType: Type of request sent to the hub. For example: clear feature, get descriptor and set feature.

wValue: Type of operation under request. For example, the clear port feature may have any one of these types: port enable, port suspend, port reset, and so on.

wIndex: Port number for the request type to port.

wLength: Length of data sent or received.

buff: Pointer to the buffer to send or receive data of wLength.

3.2.2.11 phci_irq

Interrupt handler for interrupts—SOF, ITL and ATL—are responsible for ATL, INTL and ISO transfers.

```
Int phci_irq (struct phci_hcd *dev, struct pt_regs *regs)
```

Parameters:

dev: Pointer to the ISP176x device structure (struct phci_hcd).

regs: Pointer to the register.

phci_hcd *hcd: Pointer to the ISP176x Host Controller data structure (struct phci_hcd *hcd). The structure has the following elements:

```
/*Host Controller Parameters sample */

typedef struct _phci_hcd {

   spinlock_t  lock; /* async schedule support */
   struct ehci_qh  *async;
 struct ehci_qh *reclaim;
   intreclaim_ready : 1, async_idle : 1;

   /* Periodic schedule support */

   unsigned periodic_size; /* Periodic list size */
   u32      *periodic;   /* hw periodic table */
   dma_addr_t     periodic_dma;
   unsigned    i_thresh; /* uframes HC might cache */
   union ehci_shadow  *pshadow; /*mirror hw periodic table*/
   int      next_uframe; /*scan periodic,start here*/
   int              periodic_sched;/* periodic activity count */
   struct usb_bus     *self; /* hcd is-a bus */
   const char     *product_desc;  /* product/vendor string */
   const char     *description;   /* "ehci-hcd" etc */
   struct usb_device     *otgdev;    /*otg deice, with address 2*/
   struct timer_list  rh_timer; /* drives root hub */
   struct list_head   dev_list;    /* devices on this bus */
   struct list_head       urb_list;  /*iso testing*/

   /*Hardware info/state*/
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
    struct resource    *io_res;
    struct _phci_driver   *driver;/* hw-specific hooks */
    struct isp176x_dev     *dev;
    int       irq;  /* irq allocated */
    struct device      *controller; /* handle to hardware */
    int          state;/*state of the host controller*/
    Unsigned long   reset_done[EHCI_MAX_ROOT_PORTS];

    ehci_regs   regs;
    struct _isp176x_qha *qha;
    struct _isp176x_qhint *qhint;
    struct _isp176x_isoptd *isotd;

    /*called from the irq routine*/

    /*void (*tasklet)(struct _phci_hcd *hcd);*/
    struct tasklet_struct tasklet;
    void (*worker_function) (struct _phci_hcd* hcd);
    struct _periodic_list periodic_list[PTD_PERIODIC_SIZE];
}phci_hcd,*pphci_hcd;
```

3.2.3 Memory management interface

The ISP176x has 64 kB of memory on-chip that must be mapped on the CPU address. The memory is shared between the CPU and the Host Controller to manage transaction over the USB bus. The memory is 2 x double word aligned and managed using bit enable to perform the 32-bit or 16-bit operation.

Memory is divided into two parts: one for the header and another for the payload. The header contains the valid transfer descriptors that must be transferred on the bus. The payload contains the data to be transferred to or received from the bus. The number of TDs is limited to 32 for each type of transaction.

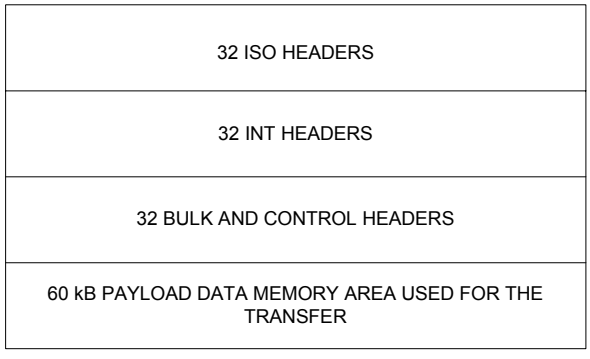


Fig 8. ISP176x memory.

To avoid contention on the Host Controller memory for ownership, the PTD and payload memory shown in Table 2: is created for mutual exclusion.

Table 2: PTD and payload memory structure

Memory map	CPU address	Memory address
Registers	0000h to 0400h	0000h to 007Fh
ISO	0400h to 07FFh	0000h to 007Fh

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Memory map	CPU address	Memory address
INTL	0800h to 0BFFh	0080h to 00FFh
ATL	0C00h to 0FFFh	0100h to 017Fh
Payload	8000h to FFFFh	0180h to 1FFFh

Skip Map: Each TD header corresponds to the status bit in this register. This bit is set by the HCD to indicate to the hardware to skip the TD.

Done Map: Each TD header corresponds to the status bit in this register. The Host Controller indicates to the driver that the transfer is completed and will be removed from the Host Controller shared memory.

For each transfer, the memory is divided into blocks for static allocation. You can also write a dynamic allocable memory manager.

The static allocation assigns appropriate blocks when requested by the HCD. The HCD fills up the block using either the PIO or DMA method.

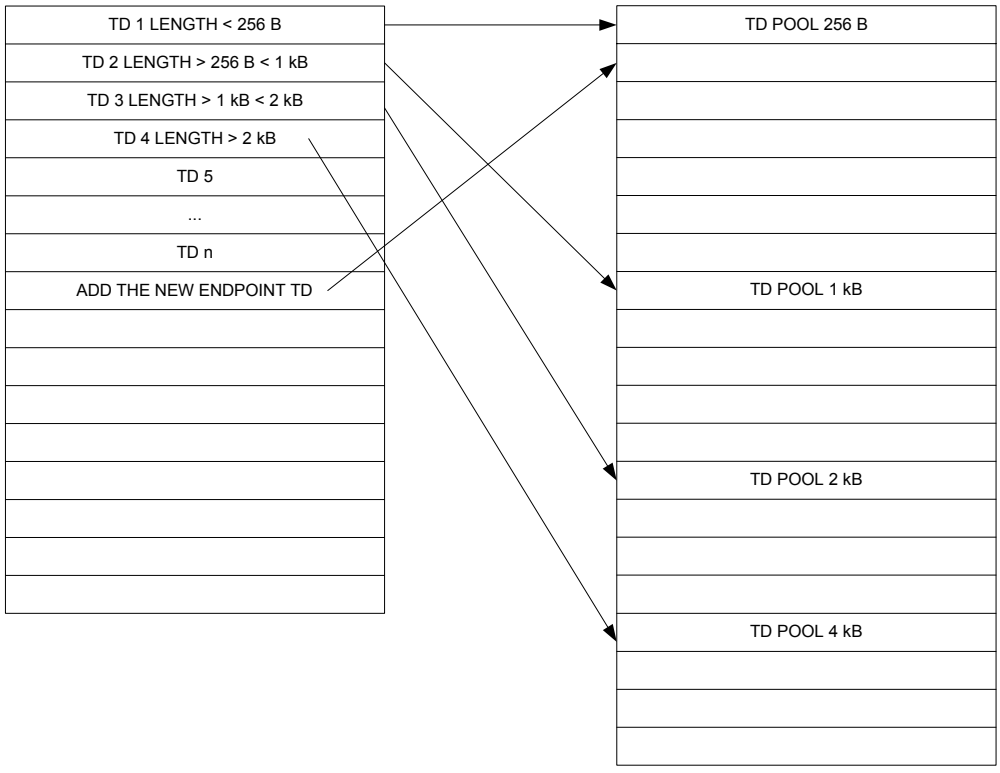


Fig 9. ISP176x memory management.

3.2.3.1 phci_hcd_mem_init

This routine initializes the available memory in various block sizes indicated and predetermined, and preserves the physical address of the blocks in the memory structure.

```
typedef struct isp176x_mem_addr {
          void  *phy_addr;  /* Physical address of the memory */
          void     *virt_addr;   /* after ioremap() function call */
          __u8    usage;
          __u32 blk_size;    /*block size*/
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

```
            u8  blk_num;    /* number of the block*/
            u8  used;       /*used/free*/
}isp176x_mem_addr_t;
```

3.2.3.2 phci_hcd_mem_alloc

```
static void phci_hcd_mem_alloc(u32 size, struct isp176x_mem_addr *memptr, u32 flag)
```

Input:

u32 size: Size of the memory required.

struct isp176x_mem_addr *memptr: The structure to be filled.

u32 flag: Used for dynamic memory allocation.

Return value: void.

3.2.3.3 phci_hcd_mem_free

This function frees the memory based on allocation.

```
static void phci_hcd_mem_alloc(u32 size, struct isp176x_mem_addr *memptr, u32 flag)
```

Input:

u32 size: Size of the memory required.

struct isp176x_mem_addr *memptr: The structure to be filled.

u32 flag: Used for dynamic memory allocation.

Return value: void.

Once a transfer is completed, the buffer will be freed and can be used for the next transfer.

3.2.4 Root hub and internal hub management

The ISP176x Host Controller of a USB bus is required to implement the root hub. The operational register space contains port registers needed to manage the internal root hub of a port.

The Host Controller traverses the EHCI schedules and encounters activities that result in the Host Controller executing USB transactions. These transactions are transmitted through enabled root ports to the attached downstream USB devices.

The port registers provide system software with the control and status information required to manipulate the port according to *Universal Serial Bus Specification Rev. 2.0*. The supported features include device detect, device connect, device disconnect, device reset, port-power manipulation, and port-power management.

System software must provide an abstraction to the USB system software stack to allow root hub ports to be manipulated by the system as if they were ports on an external hub.

The root hub can be managed as an on-chip hub. This needs creating pseudo descriptors and enumerating the hub with the hub driver of the system because a usual hub is enumerating and powering the ports.

The bus address, hub address and the port number in DW1 must be set to 0 for the high-speed port. The root port is a high-speed port.

On the enumeration of the root hub, the ISP176x will detect a connection of the internal hub on port 1, the only internal port. This detection is passed to the hub and the USB core driver. The hub driver returns appropriate URB, enumerating the internal hub that has three ports.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

The transfer URB then links the transfer schedules in the periodic and asynchronous schedules. These schedules are traversed and scheduled by the Host Controller software.

On enumeration of the internal hub, the polling on the root hub can be canceled because this internal hub is always connected to the USB bus and cannot be disconnected. The ports on the internal hub are high-speed USB ports, capable of low-speed and full-speed transfers using the TT. The default transfer is high-speed.

3.2.4.1 phci_rh_call_control

When a control transfer arrives from the hub driver, this routine performs all pseudo operations and maintains the internal hub.

```
static int phci_rh_call_control (phci_hcd *hcd, struct urb *urb);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.4.2 phci_rh_status_urb

The hub driver polls the root hub every 256 ms by using a timer to find the status change. This routine manages transfer checks for the status change and reports to the hub driver.

```
static int phci_rh_status_urb (phci_hcd *hcd, struct urb *urb) ;
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.4.3 phci_rh_urb_enqueue

Submit the URB to the root hub, depending on the URB request. It goes to either the control pipe or the interrupt pipe.

```
static int phci_rh_urb_enqueue (phci_hcd *hcd, struct urb *urb);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.4.4 phci_rh_status_dequeue

Unlink the URB for the root hub.

```
static void phci_rh_status_dequeue(phci_hcd *hcd, struct urb *urb);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Return:

int: Completion status.

3.2.4.5 phci_rh_report_status

This is the root hub polling timer callback function. It checks whether the Host Controller is running and reports the root hub status. This function suspends the polling of the root hub once an internal hub for the ISP176x is detected.

```
static void phci_rh_report_status (unsigned long ptr);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.4.6 phci_rh_power_off

This checks the power of the root port.

```
static void phci_rh_power_off( phci_hcd *hcd);
```

3.2.4.7 phci_rh_status_data

Reports the status of the root hub port.

```
static int phci_rh_status_data (phci_hcd *hcd, char *buf);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.4.8 phci_rh_control

```
static int phci_rh_control (phci_hcd *hcd,u16 typeReq,u16 wValue,u16 wIndex, char*
buf,u16 wLength);
```

Inputs:

phci_hcd *hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

3.2.5 Data transfer interface

This section provides an overview of data structures, macros and functions related to data transfers on the bus. The setting up, submitting and processing of transfer requests are also explained.

3.2.5.1 Transfer data structures and macros

The USB subsystem uses only one data structure called URB. This structure contains all parameters to set up any USB transfer type. All transfer requests are asynchronously sent to the USB core and the completion of the request is signaled through a callback function.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

The URB structure contains elements common to all transfer types, marked with C. Elements marked > are input parameters, M means mandatory and O means optional. Elements marked < are return values. Elements marked T are transient parameters.

Input and output: all uncommon elements are marked on three columns that represent control, interrupt and isochronous transfers. An X mark indicates that the element will be used with the associated transfer type.

The URB structure may look confusing. There are, however, macros to set up correct parameters.

3.2.5.2 Sample URB structure

```
typedef struct urb
{
        void *hcpriv;           // private data for host controller (don't care)
        struct list_head urb_list;    // list pointer to all active urbs (don't care)
        >CO struct urb* next;         // pointer to next URB
        >CM struct usb_device *dev;   // pointer to associated USB device
        >CM unsigned int pipe;        // pipe information
        <C int status;                // returned status
        TCO unsigned int transfer_flags;
                              //USB_DISABLE_SPD|USB_ISO_ASAP|USB_URB_EARLY_COMPLETE
        >CM void *transfer_buffer;       // associated data buffer
        >CM int transfer_buffer_length;  // data buffer length
        <C int actual_length;            // actual data buffer length
        <X-- unsigned char *setup_packet;// setup packet (control only)
        T-XX int start_frame;            // start frame (iso/irq only)
        >--X int number_of_packets;      // number of packets in this request (iso
        only)
        >-X- int interval;               // polling interval (irq only)
        <--X int error_count;            // number of errors in this transfer (iso
        only)
        >XXX int timeout;                // timeout in jiffies
        >CO void *context;               // context for completion routine
        >CO usb_complete_t complete;     // pointer to completion routine
        >--X iso_packet_descriptor_t iso_frame_desc[0]; // optional iso descriptors
} urb_t, *purb_t;
```

3.2.5.3 Sample ISO packet structure

```
typedef struct
{
unsigned int offset;        // offset to the transfer_buffer
unsigned int length;        // expected length
unsigned int actual_length;    // actual length after processing
unsigned int status;        // status after processing
} iso_packet_descriptor_t, *piso_packet_descriptor_t;
```

3.2.5.4 Details of the URB structure

pipe [mandatory input parameter]

The pipe element is used to encode the endpoint number and properties. There are macros to create an appropriate pipe value.

The following routines create a pipe for downstream (snd) or upstream (rcv) control transfers to a given endpoint. Argument dev is a pointer to a USB device structure. Argument endpoint is usually 0.

- pipe=usb sndctrlpipe(dev,endpoint)

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

- pipe=usb rcvctrlpipe(dev,endpoint)

The following routines create a pipe for downstream (snd) or upstream (rcv) bulk transfers to a given endpoint. The endpoint is 1 <= endpoint <= 15, depending on active endpoint descriptors.

- pipe=usb sndbulkpipe(dev,endpoint)
- pipe=usb rcvbulkpipe(dev,endpoint)

The following routines create a pipe for downstream (snd) or upstream (rcv) interrupt transfer to a given endpoint. The endpoint is 1 <= endpoint <= 15, depending on active endpoint descriptors.

- pipe=usb sndintpipe(dev,endpoint)
- pipe=usb rcvintpipe(dev,endpoint)

The following routines create a pipe for downstream (snd) or upstream (rcv) isochronous transfers to a given endpoint. The endpoint is 1 <= endpoint <= 15, depending on active endpoint descriptors.

- pipe=usb sndisopipe(dev,endpoint)
- pipe=usb rcvisopipe(dev,endpoint)

transfer buffer [mandatory input parameter]

This element is a pointer to the associated transfer buffer that contains data transferred from or to a device. This buffer must be allocated as a nonpageable contiguous physical memory block. Use void *kmalloc(size t, GFP KERNEL);.

transfer buffer length [mandatory input parameter]

This element specifies the size of the transfer buffer in bytes. For interrupt and control transfers, the value must be less than or equal to the maximum packet size of the associated endpoint.

The maximum packet size can be determined from wMaxPacketSize of an endpoint descriptor. Bulk transfers larger than wMaxPacketSize are automatically split into smaller portions.

complete [optional input parameter]

The USB subsystem asynchronously processes requests. This element allows specifying a pointer to a caller supplied handler function that is called after the request is completed. This handler is used to complete the caller specific part of the request as fast as possible.

context [optional input parameter]

Optionally, a pointer to a request related context structure could be given.

transfer flags [optional input parameter and return value]

A number of transfer flags may be specified to change the behavior when processing the transfer request.

USB disable SPD

This flag disables short packets. A short packet condition occurs if an upstream request transfers less data than the maximum packet size of the associated endpoint.

USB URB early complete

A completion handler is called after the request is processed. Use this flag to call the handler before other linked URBs are resubmitted.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

USB ISO ASAP

When scheduling isochronous requests, this flag informs the Host Controller to start the transfer as soon as possible. If USB ISO ASAP is not specified, a start frame must be given.

It is recommended that you use this flag, if isochronous transfers do not need to be synchronized with the current frame number. The current frame number is an 11-bit counter that increments every millisecond, which is the duration of one frame on the bus.

USB async unlink

When a URB must be cancelled, it can be done synchronously or asynchronously. Use this flag to switch on asynchronous URB unlinking.

USB timeout killed

This flag is set by the Host Controller to mark the URB as killed by timeout. The URB status carries the actual error that caused the timeout.

USB queue bulk

This flag is used to allow queuing for bulk transfers. Normally, only one bulk transfer can be queued for an endpoint of a particular device.

Next [optional input parameter]

It is possible to link several URBs in a chain by using the next pointer. This allows you to send a sequence of USB transfer requests to the USB core. The chain must be terminated using a NULL pointer or the last URB must be linked with the first. This allows to automatically reschedule a number of URBs to transfer a continuous data stream.

Status [return value]

This element carries the status of an ongoing or already finished request. After successfully sending a request to the USB core, status EINPROGRESS = 0, indicating the successful completion of a request. There exist a number of error conditions.

Actual length [return value]

After a request is completed, this element counts the number of bytes transferred. The remaining elements of the URB are specific to the transfer type.

Bulk transfers

No additional parameters need to be specified.

Control transfers

Setup packet [mandatory input parameter]

Control transfers consist of two or three stages.

The first stage is the downstream transfer of the setup packet. This element takes the pointer to a buffer containing the setup data. This buffer must be allocated as a nonpageable contiguous physical memory block. Use void *kmalloc(size t, GFP KERNEL.

The next stage is the data stage, and is followed by the status stage.

Interrupt transfers

Start frame [return value]

This element is returned to indicate the first frame number that the interrupt is scheduled. Setting this value as -1 starts interrupt transfers as soon as possible.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Interval [mandatory input parameter]

This element specifies the interval in milliseconds for the interrupt transfer. Allowed values are 1 <= interval <= 255. Specifying an interval of 0 ms causes a one shot interrupt, no automatic rescheduling is done. For interrupt endpoints, you can find the interrupt interval as element bInterval of an endpoint descriptor.

Isochronous transfers

Start frame [input parameter or return value]

This element specifies the first frame number that the isochronous transfer is scheduled. Setting the start frame allows to synchronize transfers to or from an endpoint. If the USB ISO ASAP flag is specified, this element is returned to indicate the first frame number that the isochronous transfer is scheduled.

Number of packets [mandatory input parameter]

The isochronous transfer requests are sent to the USB core as a set of single requests. A single request transfers a data packet up to the maximum packet size of the specified endpoint (pipe). This element sets the number of packets for the transfer.

Error count [return value]

After the request is completed, URB status is != -EINPROGRESS. This element counts the number of erroneous packets. Detailed information about the single transfer requests can be found in the isochronous frame descriptor structure.

Timeout [input parameter]

A timeout can be specified to automatically remove a URB from the Host Controller schedule. If a timeout occurs, the transfer flag USB TIMEOUT KILLED is set. The actual transfer status carries the USB status that caused the timeout.

ISO frame desc [mandatory input parameter]

This additional array of structures at the end of every isochronous URB sets up the transfer parameters for every single request packet.

Offset [mandatory input parameter]

Specifies the offset address to the transfer buffer for a single request.

Length [mandatory input parameter]

Specifies the length of the data buffer for a single packet. If length is set to 0 for a single request, the USB frame is skipped and no transfer will be initiated. This option can be used to synchronize isochronous data streams.

Actual length [return value]

Returns the actual number of bytes transferred by this request.

Status [return value]

Returns the status of this request.

For the Host Controller to handle the URB, the following functions are used.

3.2.5.5 phci_hcd_submit_urb

Use the following routine to submit the transfer URB for the process that transfers the data over the USB bus.

```
static int  phci_hcd_submit_urb( struct urb *urb)
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

3.2.5.6 phci_hcd_submit_urb

Use the following routine to submit the transfer URB for the process that transfers the data over the USB bus. This routine maps the transfer over multiple TDs and links them for the transfer.

```
static int  phci_hcd_submit_urb( struct urb *urb)
```

3.2.5.7 phci_hcd_completeurb

Once the transfer is completed, this routine is called to complete the URB transfer.

```
static void phci_hcd_completeurb(phci_hcd *hcd, struct urb *urb, struct pt_regs
*regs);
```

3.2.5.8 phci_hcd_urb_dequeue

The host calls this routine if it wishes to remove the URB mostly to abort the transfer. This requires careful removal of the transfer from the schedule.

```
static int phci_hcd_urb_dequeue(phci_hcd *hcd, struct urb *urb, urb_priv_t *urb_priv)
```

4. OTG stack interface

This interfaces to the USB OTG driver and is used to control OTG port activities.

4.1.1 phci_register_otg

This function is used to register OTG driver notification functions to the ISP176x HCD. The notification function is called on completing the OTG enumeration.

```
int   phci_register_otg(phci_otg_data_t *otg_data)
```

Parameters:

otg_data: Pointer to the OTG driver registration information data structure (phci_otg_data_t). The structure has the following elements:

```
typedef struct {
   void  *priv_data;
   void  (*otg_notif)(void *otg_priv);
   void  *hc_priv_data;
} phci_otg_data_t;
```

priv_data: The OTG driver private data pointer. This pointer is used as an input parameter for the OTG notification function.

otg_notif: The OTG notification function to send the OTG enumeration result.

hc_priv_data: The ISP176x HCD private data. The ISP176x HCD fills this field when the registration is successful. This will be used as an input parameter for the other HCD function calls.

Return value:

0 Successful registration of the OTG driver with the HCD.

<0 OTG driver registration has failed.

4.1.2 phci_unregister_otg

This function is used to deregister the OTG driver notification functions from the ISP176x HCD.

```
void  phci_unregister_otg(phci_otg_data_t  *otg_data)
```

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Parameters:

otg_data: Pointer to the OTG driver registration information data structure (phci_otg_data_t).

Return value: None.

4.1.3 phci_otg_port_control

This function is used to control the OTG port activities through the virtual root hub portion of the ISP176x HCD.

```
void  phci_otg_port_control(void *priv, __u8  cmd, __u32 *data)
```

Parameters:

priv: Pointer to the ISP176x HCD private data. The OTG driver will get this pointer during its registration with the HCD.

data: Pointer to the port control data. Table 3: shows the possible values of this field.

Return value: None.

Table 3: Possible values of port control data

Value	Description
OTG_PORT_OPEN_PORT	Open OTG port for USB bus driver operations
OTG_PORT_GET_ENUM	Get OTG port enumeration results
OTG_PORT_SUSPEND	Suspend OTG port
OTG_PORT_DISCONNECT_PORT	Disconnect OTG port for USB bus driver operations
OTG_PORT_OPEN_PORT_IMM	Open OTG port immediately for USB bus driver operations

5. Abbreviations

Table 4: Abbreviations

Abbreviation	Description
API	Application Programming Interface
ATL	Acknowledged Transfer List
DMA	Direct Memory Access
FSM	Finite-State Machine
HCD	Host Controller Driver
INTL	INTerrupt List
ITL	Isochronous (ISO) Transfer List
ISO	ISOchronous
ISR	Interrupt Service Routine
OS	Operating System
OTG	On-The-Go
PIO	Parallel Input Output
PTD	Philips Transfer Descriptor

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

Abbreviation	Description
SOF	Start-Of-Frame
TT	Transaction Translator
URB	USB Request Block
USB	Universal Serial Bus

6. References

- *Universal Serial Bus Specification Revision 2.0*
- *On-The-Go supplement to the USB2.0 specification Rev 1.0*
- *ISP1760 Hi-Speed Universal Serial Bus host controller for embedded applications* data sheet
- *ISP1761 Hi-Speed Universal Serial Bus On-The-Go controller* data sheet.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

7. Disclaimers

Life support — These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. When the product is in full production (status 'Production'), relevant changes will be communicated via a Customer Product/Process Change Notification (CPCN). Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

© Koninklijke Philips Electronics N.V. 2005. All rights reserved.

8. Contents

1. Introduction 3
1.1 ISP176x peripheral hardware 4
1.2 ISP176x host hardware 4
1.3 ISP176x Hardware Access Layer 5
1.4 ISP176x Host Controller Driver 5
1.5 ISP176x Peripheral Controller Driver 5
1.6 ISP176x OTG Controller driver 5
2. Hardware Access Layer 5
2.1 Starting the Host Controller 5
2.2 Module management interface 6
2.3 isp176x_hal_module_init 6
2.4 ISP176x Controller Driver interface 7
2.5 Driver registration interface 7
2.6 Resource management interface 8
2.7 I/O access Interface 9
2.8 Kernel tracing interface 12
2.9 Common structures 12
3. Host Controller interface 13
3.1 Module management 13
3.2 ISP176x host management service 14
4. OTG stack interface 34
5. Abbreviations 35
6. References 36
7. Disclaimers 37
8. Contents 38

© Koninklijke Philips Electronics N.V. 2005

All rights are reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent- or other industrial or intellectual property rights.

Date of release: 7 April 2005

Published in The Netherlands