ISP1763A Linux software

UM0907

User manual

Abstract

This document provides information on the interfaces and data structures that are required to use the ISP1763A host controller, OTG controller and peripheral controller driver layers for the Linux operating system.

Keywords

isp1763a; host controller; peripheral controller; otg controller; usb; universal serial bus

Legal information

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

Contents

1 About this document

1.1 Purpose

This document provides information on the interfaces and data structures that are required to use the ISP1763A host controller, OTG controller, and peripheral controller driver layers for the Linux operating system.

1.2 Revision information

Table 1 Revision history

Date	Rev.	Comments
2010-02-17	1	First version.
2013-04-04	2	Improved the quality of the PDF rendition. No other change in the content.
2013-10-02	3	Applied STMicroelectronics branding. No change in the content

1.3 Reference list

[1] *Universal Serial Bus Specification Rev. 2.0* *www.usb.org*

[2] *ISP1763A Hi-Speed USB OTG controller for portable applications data sheet* *CD00264885*

2 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 STMicroelectronics embedded host controller.

In addition to the host functionality, some embedded systems require 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 involves switching between the host and peripheral functionalities.

The ISP1763A is a USB OTG controller: USB host and USB peripheral. It has one host port, and an OTG port that can be used as either a host or a peripheral.

This document provides information on interfaces and data structures required to use the ISP1763A host controller, OTG controller, and peripheral controller driver layers for the Linux operating system.

3 Overview

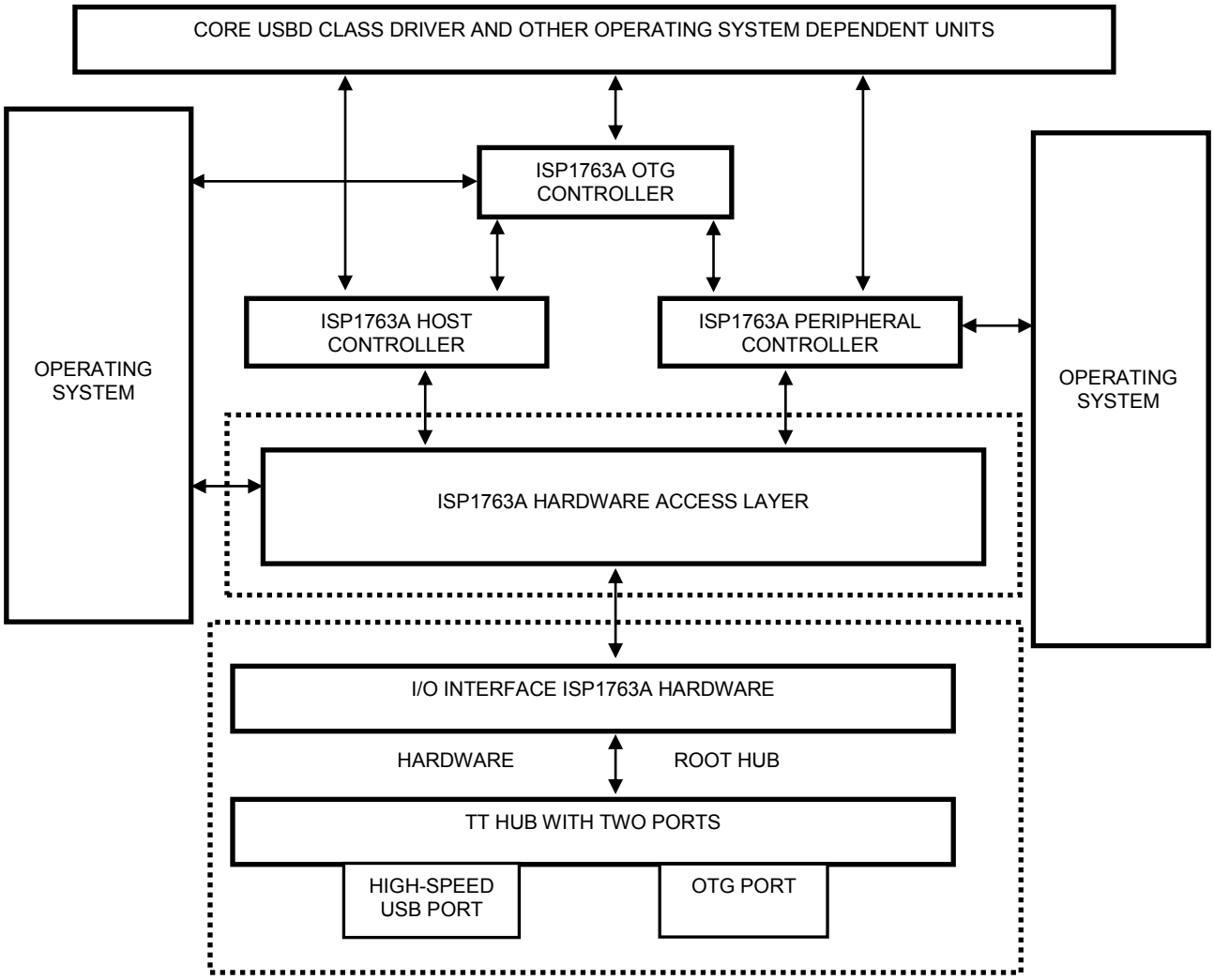

Figure 1 ISP1763A system interface

The following subsections explain each layer in detail.

3.1 ISP1763A hardware access layer

The ISP1763A hardware access layer provides functions to access the ISP1763A hardware and operating system platform related functions. This layer depends on the platform and the ISP1763A hardware. Customers need to port this layer based on the platform they are using. The interface between the ISP1763A hardware access layer and top layers are defined in the API ISP1763A hardware access layer.

3.2 ISP1763A host controller driver

The ISP1763A provides the host-capability function. The ISP1763A 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 under the direction of the software.

3.3 ISP1763A device controller driver

The ISP1763A device controller driver is responsible for data transfer to the connected USB host and manages bus activities. It provides interface to the USB protocol driver and class drivers (device) for data transfer on the USB bus.

3.4 ISP1763A OTG controller driver

The OTG driver maintains the OTG Finite State Machine (FSM) by accessing and controlling OTG controller registers through the HAL. If the device connected to the port is dual-role capable, then the device negotiates the role and acts under the direction of the FSM.

4 Hardware abstraction layer

The hardware abstraction layer provides functions to access the host hardware and OS platform-related functions. Porting of this layer is based on the platform.

4.1 Module management interface

This interfaces to the OS. It is called when loading and unloading the ISP1763A host controller driver to the kernel.

The following functional interface is based on the PCI x86 platform. The functional interface can, however, be modified, depending on the OS.

4.1.1 isp1763_pci_module_init

This function initializes the ISP1763A hardware access driver module. The Linux kernel module manager calls this function.

```
static int __init isp1763_pci_module_init (void)
```

Parameters: None.

Return value:

0: The ISP1763A hardware access driver kernel module is successfully completed.

< 0: The ISP1763A kernel module initialization has failed.

4.1.2 isp1763_pci_module_cleanup

This function de-initializes the ISP1763A hardware access driver module. The Linux kernel module manager calls this function during unloading of this module.

```
static void __exit isp1763_pci_module_cleanup (void)
```

Parameters: None.

Return value: None.

4.2 Controller driver interface

This section includes:

- Driver registration interface
- Resource management interface
- I/O access interface
- Kernel tracing interface

4.2.1 Driver registration interface

4.2.1.1 isp1763_register_driver

This function registers driver access functions to the ISP1763A hardware abstraction layer driver.

```
int isp1763_register_driver(struct isp1763_driver *drv)
```

Parameters:

drv: Pointer to the ISP1763A driver data structure (struct isp1763_driver). The structure has the following elements.

```
typedef struct isp1763_driver {
    char            *name;
    unsigned long   index;  /* HC or DC or OTG */
    isp1763_id      *id;
    int (*probe)(struct isp1763_dev *dev,
            isp1763_id *id
    void (*remove)(struct isp1763_dev *dev);
    void (*suspend)(struct isp1763_dev *dev);
    void (*resume)(struct isp1763_dev *dev);
} isp_1763_driver_t;
```

name: Name of the driver registering to the hardware abstraction layer.

index: Driver type.

probe: The probe function is called by the hardware abstraction layer when it finds the hardware of the type specified by the index. Input parameters to this function are the ISP1763A device data structure (struct isp1763_dev).

remove: This is a removal function. The hardware abstraction layer calls this function when it finds that the hardware is unavailable or inactive. Input parameters to this function are the ISP1763A device data structure (struct isp1763_dev).

suspend: This function is called by the hardware abstraction layer when it finds that the hardware must be suspended. Input parameters to this function are the ISP1763A device data structure (struct isp1763_dev). This function interface is applicable only when the power management is enabled.

resume: This function is called by the hardware abstraction layer when it finds that the hardware must be resumed from the suspended state. Input parameters to this function are the ISP1763A device data structure (struct isp1763_dev). This function interface is applicable only when the power management is enabled.

Return value:

0: Driver registration is successful with the hardware abstraction layer.

< 0: OTG driver registration has failed.

4.2.1.2 isp1763_unregister_driver

This function de-registers controller drivers from the ISP1763A hardware abstraction layer.

```
void isp1763_unregister_driver(struct isp1763_driver *drv)
```

Parameters:

drv: Pointer to the driver registration data structure.

Return value: None.

4.2.2 Resource management interface

4.2.2.1 isp1763_request_irq

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

```
int isp1763_request_irq(void (*handler)(struct isp1763_dev *, void
*),struct isp1763_dev *dev, void *isr_data)
```

Parameters:

handler: This function is called whenever the hardware abstraction layer receives an interrupt on the device interrupt line. Input parameters to this function are the ISP1763A device data structure (dev) and the controller driver ISR data (isr_data).

dev: Pointer to the ISP1763A device data structure (struct isp1763_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.

4.2.2.2 isp1763_free_irq

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

```
void isp1763_free_irq(struct isp1763_dev *dev, void *isr_data)
```

Parameters:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

isr_data: Pointer to the controller data (identifier).

Return value: None.

4.2.3 I/O access interface

4.2.3.1 isp1763_reg_read32

This function reads the 32-bit ISP1763A register.

```
__u32 isp1763_reg_read32(struct isp1763_dev *dev,__u16 reg,__u32
data)
```

Parameters:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

reg: Register index of the ISP1763A device.

data: Temporary variable in which the data read from the register will be placed.

Return value:

32-bit register content.

4.2.3.2 isp1763_reg_write32

This function writes to the 32-bit ISP1763A register.

```
void isp1763_reg_write32(struct isp1763_dev *dev,__u16 reg,__u32
data)
```

Parameters:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

reg: Register index of the ISP1763A device.

data: Data to be written to the ISP1763A.

Return value: None.

4.2.3.3 isp1763_reg_read16

This function reads the 16-bit ISP1763A register.

```
__u16 isp1763_reg_read16(struct isp1763_dev *dev,__u16 reg,__u16
data)
```

Parameter:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

Return value: 16-bit data content.

4.2.3.4 isp1763_reg_write16

This function writes to the 16-bit ISP1763A register.

```
void isp1763_reg_write16(struct isp1763_dev *dev,__u16 reg,__u16
data)
```

Parameters:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

reg: Register index of the ISP1763A device.

data: Data to be written to the ISP1763A.

Return value: None.

4.2.3.5 isp1763_reg_read8

This function reads the 8-bit ISP1763A register.

```
__u8 isp1763_reg_read8(struct isp1763_dev *dev,__u16 reg,__u8
data)
```

Parameter:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

Return value: 8-bit data content.

4.2.3.6 isp1763_reg_write8

This function writes to the 8-bit ISP1763A register.

```
void isp1763_reg_write8(struct isp1763_dev *dev,__u16 reg,__u8
data)
```

Parameters:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev).

reg: Register index of the ISP1763A device.

data: Data to be written to the ISP1763A.

Return value: None.

4.2.3.7 isp1763_mem_read

This function reads from the memory to the ISP1763A.

```
int isp1763_mem_read(struct isp1763_dev *dev, __u32 start_add,
        __u32 end_add, __u32 * buffer, __u32 length, __u16 dir)
```

Parameter:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev) is filled with the memory descriptor buffer supplied.

start_add: Starting address of the memory

end_add: End address

buffer: Buffer pointer

length: Length

dir: Direction (increment or decrement)

Return value:

Operation successful = TRUE

4.2.3.8 isp1763_mem_write

This function writes to the memory from the ISP1763A.

```
int isp1763_mem_write(struct isp1763_dev *dev,  __u32 start_add,
__u32 end_add, __u32 * buffer, __u32 length, __u16 dir)
```

Parameter:

dev: Pointer to the ISP1763A device data structure (struct isp1763_dev) is filled with the memory descriptor buffer supplied.

start_add: Starting address of the memory

end_add: End address

buffer: Buffer pointer

length: Length

dir: Direction (increment or decrement)

Return value:

0: Data read successfully.

< 0: Failed.

5 Host controller interface

The HCD transfers data to connected USB devices and manages root hub ports.

The transfers from the USB core driver in the form of the USB Request Block (URB) are scheduled over the USB bus in a round-robin method. The standard Linux ver. 2.6.20 Enhanced Host Controller Interface (EHCI) driver is modified to suit the ISP1763A architecture.

The main reason to follow the Linux ver. 2.6.20 host controller architecture is to utilize the horizontal and vertical traversal rules set by the EHCI driver. To transfer data to or from the EHCI, the hardware schedule list that is traversed by the hardware and scheduled over the USB bus is used. The ISP1763A, however, uses the software-driven interrupt-based scheduler that is responsible to schedule PTDs into the ISP1763A hardware and complete transfers required for the USBD.

Scheduling a transfer over the ISP1763A means scheduling the PTD over the shared memory of the ISP1763A. Transfer scheduling is done by the software and filled in the shared memory of the ISP1763A for hardware execution.

The ISP1763A host controller memory is divided into two parts: data payload area and header area. The data payload contains data to be transferred, and the header contains the PTD, which dictates how the transfer is to be performed by the hardware.

The ISP1763A host controller traverses the memory in a linear method. A PTD is dynamically added and removed from the endpoint list by using the Skip bit. Using this bit, the host controller determines whether to access the respective PTD. The driver uses the Done bit to check the completion status of the PTD.

PTD structures are translations of Linux ver. 2.6.20 EHCI data structures that are optimized for the ISP1763A, which is a slave host controller and has no bus master capability.

PTD data structures are designed to provide maximum flexibility required by USB, minimize memory traffic, and reduce hardware and software complexity.

The ISP1763A controller executes transactions for devices by using a simple and shared memory schedule. The schedule consists of data structures organized into three lists:

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

The system software maintains two schedules for the host controller: periodic and asynchronous.

The ISP1763A has a maximum of 16 ISO, 16 INTL, and 16 ATL PTDs. These PTDs are used as channels to transfer data from the shared memory to the USB bus. These channels are allocated dynamically to various pipes, which mean PTDs are not fixed to one pipe. It is allotted to another pipe on completion of all URBs in the current pipe.

Multiple transfers are scheduled to the shared memory for various endpoints by traversing the next link pointer provided by the endpoint data structure, until it reaches the end of the endpoint list. There are three endpoint lists: ISO, INTL, and ATL. If the schedule is enabled, then the host controller executes the ISO schedule, followed by the INTL schedule, and then the ATL schedule.

These lists are traversed and scheduled by the software, according to the EHCI traversal rule. The host controller executes the scheduled ISO, INTL, and ATL PTDs.

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 group or individual PTDs. When the logical condition of the Done bit is true in the register, it means that the PTD is completed.

There are four types of interrupts in the ISP1763A: ISO, INTL, ATL, and SOF.

The following sections explain HCD interfaces in detail:

- Module management and controller routines
- Memory management interface
- Root hub and internal hub management
- Data transfer interface

5.1 Module management and controller routines

5.1.1 pehci_hcd_reset

This function is used to reset the host controller, which contains the following three operations:

1. Write logic 1 to bit 0 of the SW Reset register (B8h) and wait for it to be self cleared.
2. Write logic 1 to bit 1 of the SW Reset register (B8h), and wait for it to be self cleared.
3. Write logic 1 to bit 1 of the USBCMD register (8Ch) and wait for it to be self cleared.

5.1.2 pehci_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 pehci_hcd_start(struct usb_hcd  *usb_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. pehci_hcd_reset(hcd)

2. Enable the interrupt. pehci_hcd_enable_interrupts(hcd)
3. Initialize map buffers. pehci_hcd_init_map_buffers(hcd)

The HCD sets last PTD bits of all schedule types by writing to HC_ISO_PTD_LASTPTD_REG (A8h), HC_INT_PTD_LASTPTD_REG (AEh), and HC_ATL_PTD_LASTPTD_REG (B4h).

```
/*set last maps, for iso its only 1, else 32 tds bitmap*/
isp1763_reg_write16(pehci_hcd->dev, pehci_hcd-
>regs.atltdlastmap, 0x8000);
isp1763_reg_write16(pehci_hcd->dev, pehci_hcd-
>regs.inttdlastmap, 0x80);
isp1763_reg_write16(pehci_hcd->dev, pehci_hcd-
>regs.isotdlastmap, 0x01);
```

Initially ISO transfers are not 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 not active.

```
/*iso transfers are not active*/
pehci_hcd->next_uframe = -1;
pehci_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 “running” but do not process anything yet. Initialize the timer for the root hub polling. Poll until the device is 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 */
usb_hcd->state =  HC_STATE_RUNNING;
```

This completes the host controller initialization. The SOF will now be on the internal root hub port. This allows detection of the port status change with a connection status change to allow the internal hub to enumerate.

5.1.3 pehci_hcd_init_map_buffers

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 ISP1763A 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*/
    }
```

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;/*header address of this td */
    u32 ptd_data_addr; /*data addr of this td to write in & 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 the host controller is started.

5.1.4 pehci_hcd_start_controller

This routine will start the host controller by setting the RUN/STOP bit in the USBCMD register to RUN and waiting for the handshake to set this bit, 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.

5.1.5 pehci_hcd_suspend

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

```
void pehci_hcd_suspend(struct isp1763_dev *dev)
```

Parameters:

dev: Pointer to the ISP1763A device structure (struct isp1763_dev).

Return value: void.

5.1.6 pehci_hcd_resume

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

```
void pehci_hcd_resume(struct isp1763_dev *dev)
```

Parameters:

dev: Pointer to the ISP1763A device structure (struct isp1763_dev).

Return value: void.

5.1.7 pehci_hcd_stop

This routine sets the controller into the stop state.

```
pehci_hcd_stop(struct usb_hcd *usb_hcd)
```

Parameters:

dev: Pointer to the ISP1763A hcd structure (struct usb_hcd).

Return value: void.

5.1.8 pehci_hcd_irq

Interrupt handler for interrupts: SOF, ITL, and ATL, and responsible for the corresponding activities.

```
irqreturn_t pehci_hcd_irq(struct isp1763_dev *dev, void
*__irq_data, struct pt_regs *regs)
```

Parameters:

dev: Pointer to the ISP1763A device structure (struct isp1763_dev).

regs: Pointer to the register.

__irq_data -> usb_hcd ->pehci_hcd: Pointer to the ISP1763A host controller data structure (struct pehci_hcd *hcd).

5.2 Memory management interface

The memory allocation for various sections in the ISP1763A is given in Table 2.

Table 2 Memory allocation for various sections

Memory map	CPU address	Memory address
ISO	0400h to 05FFh	0000h to 007Fh
INTL	0800h to 09FFh	0080h to 00FFh
ATL	0C00h to 0DFFh	0100h to 017Fh
Payload	1000h to 5FFFh	0180h to 0B7Fh

The ISP1763A has 24 kB of on-chip memory that must be mapped on the CPU address: 4 kB PTD area and 20 kB payload area.

5.2.1 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.

5.2.2 phci_hcd_mem_alloc

```
static void phci_hcd_mem_alloc(u32 size,struct isp1763_mem_addr
memptr,u32 flag)
```

Input:

u32 size: Size of the memory required.

struct isp1763_mem_addr *memptr: The structure to be filled.

u32 flag: Used for the dynamic memory allocation.

Return value: void.

5.2.3 phci_hcd_mem_free

This function frees the memory based on allocation.

```
static void phci_hcd_mem_free(struct isp1763_mem_addr *memptr)
```

Input:

struct isp1763_mem_addr *memptr: The structure to be freed.

Return value: void.

Using memory banks for the operation, when the HCD is mapping the EHCI TD to the enhanced PTD, the buffer allocation is based on the length. Once a transfer is completed, the buffer is used for next transfers.

5.3 Root hub and internal hub management

The ISP1763A host controller of the USB bus is required to implement the root hub. The operational register space contains port registers that contain the minimum hardware status and control needed to manage the internal root hub of a port.

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

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.

The 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 requires pseudo descriptors to be created and the hub enumerated with the hub driver of the system because usually a hub is enumerating and powering the ports.

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

On enumeration of the root hub, the ISP1763A will detect a connection of the internal hub on port 1, the only internal port. The detection is passed to the hub and the USB core driver. The hub driver returns the appropriate URB, enumerating the internal hub that has three ports.

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.

5.4 Data transfer interface

5.4.1 pehci_hcd_urb_enqueue

Called by the submit_urb routine. This is used to submit a USB Request Block (URB) for a data transfer request across the USB bus.

```
pehci_hcd_urb_enqueue(struct usb_hcd *usb_hcd,struct
usb_host_endpoint *ep,struct urb *urb,gfp_t mem_flags)
```

Inputs:

usb_hcd *usb_hcd: HCD structure.

usb_host_endpoint *ep: EP data structure

struct urb *urb: USB request structure.

Return:

int: Status of completion.

5.4.2 pehci_hcd_urb_dequeue

Used to unlink the URB that was previously queued.

```
static int pehci_hcd_urb_dequeue(struct usb_hcd  *usb_hcd, struct
urb *urb)
```

Inputs:

usb_hcd *usb_hcd: HCD structure.

struct urb *urb: USB request structure.

Return:

int: Completion status.

6 Device controller driver interface

The following section explains in detail the APIs of the Device Controller Driver (DCD).

6.1 Module management interface

This interface is with the operating system. It is called at the time of loading or unloading of the ISP1763A DCD.

6.1.1 pdc_module_init

```
static int __init pdc_module_init(void)
```

This function registers the PDC driver to the ISP1763A HAL driver, which in turn calls the probe function when the device is found. The Linux kernel module manager calls this function.

Parameters:

None

Return value:

0: Successful completion of the mass storage class driver kernel module

< 0: Initialization failed

6.1.2 pdc_module_cleanup

```
static void __exit pdc_module_cleanup(void)
```

This function is used to de-initialize the DCD module. The Linux kernel module manager calls this function during unloading of this module. It unregisters the DCD from the ISP1763A HAL layer.

Parameters: None

Return value: None

6.2 Interface between DCD and USB class driver

6.2.1 set_config

```
INT32 (*set_config)(VOID *priv, UCHAR ubConfig)
```

Parameters:

priv: Placeholder

ubConfig: Current configuration

Return value:

Negative: Failure

Non-negative: Success

Handles configuration of the mass storage device. It enables or disables endpoints.

6.2.2 class_vendor

```
INT32 (*class_vendor) (VOID *priv, UCHAR *pubSetup, UCHAR
**pubData, UINT16 *pusLength)
```

This is a call back function to the class driver's class-specific request handler.

Parameters:

priv: Placeholder

pubSetup: USB control request

pubData: Request buffer

pusLength: Length of the data buffer

Return value:

Negative: Failure

Non-negative: Success

Handles class requests for respective class devices.

6.2.3 set_intf

```
INT32 (*set_intf) (VOID *priv, UCHAR ubIntf, UCHAR ubAltSet)
```

This is a callback handler for the set interface request.

Parameters:

priv: Placeholder

ubIntf: Interface Index

ubAltSet: Interface value

Return value:

0 on success else failure

It enables the interface of the class device.

6.2.4 pdc_register_class_drv

```
int pdc_register_class_drv(struct pdc_class_drv *drv)
```

Parameters:

drv: Clients interface data structure

Returns value:

Zero for success.

Non-zero for failure.

Registers client drivers to DCD.

6.2.5 pdc_deregister_class_drv

```
void pdc_deregister_class_drv(struct pdc_class_drv *drv)
```

Parameters:

drv: Clients interface data structure

Returns value: Nothing

De-registers client drivers from DCD.

6.2.6 pdc_submit_urb

```
Int pdc_submit_urb(struct pdc_urb *urb_req)
```

Parameters:

urb_req: URB request data structure

Return value:

0 on success else failure

This function handles the submission of transfer URBs to endpoint buffers.

6.2.7 pdc_cancel_urb

```
Int pdc_cancel_urb(struct pdc_urb *urb_req)
```

Parameters:

urb_req: URB request data structure

Return value:

0 on success else failure

This function cancels all the pending URBs if any.

6.2.8 pdc_open_pipe

```
pdc_pipe_handle_t pdc_open_pipe(struct pdc_pipe_desc * pipe_desc)
```

Parameters:

pipe_desc: Pipe descriptor or endpoint data structure

Return value:

Returns the pipe handle for the specified endpoint.

This function enables the endpoint and returns the pipe handle.

6.2.9 pdc_close_pipe

```
Void pdc_close_pipe(pdc_pipe_handle_t pipe_handle)
```

Parameters:

pipe_handle: Pipe descriptor or endpoint data structure

Return value: Nothing

This function disables the endpoint and invalidates the pipe handle.

6.2.10 pdc_pipe_operation

```
int pdc_pipe_operation(struct pdc_pipe_opr *pipe_opr)
```

Parameters:

pipe_opr: Pipe operation data structure

Return value:

Returns 0 on success else failure.

This function is used to stall and un-stall endpoints. It is also used to get the current endpoint pipe status.

6.3 Interface between DCD and hardware abstraction layer

Following are interfaces that interact with the hardware access layer. You may be required to customize interfaces to interact with your platforms.

6.3.1 pdc_read8

```
__u8 pdc_read8(__u16 reg)
```

Parameters:

reg: Register address

Return value: Returns 8-bit value on success

This function reads the specified register and returns 8 bit value.

6.3.2 pdc_write8

```
void pdc_write8(__u16 reg, __u8 data)
```

Parameters:

reg: Register address

data: 18-bit data to be written into the specified register

Return value: Nothing

This function writes specified 8-bit value into the specified register.

6.3.3 pdc_read16

```
__u16 pdc_read16(__u16 reg)
```

Parameters:

reg: Register address

Return value: Returns 16-bit value on success

This function reads the specified register and returns 16-bit value.

6.3.4 pdc_write16

```
void pdc_write16(__u16 reg, __u16 data)
```

Parameters:

reg: Register address

data: 16-bit data to be written into the specified register

Return value: Nothing

This function writes specified 16-bit value into the specified register.

6.3.5 pdc_read32

```
static __inline__ __u32 pdc_read32(__u16 reg)
```

Parameters:

reg: Register address

Return value: Returns 32-bit value on success.

This function reads the specified register and returns 32-bit value.

6.3.6 pdc_write32

```
static __inline__ void pdc_write32(__u16 reg, __u32 data)
```

Parameters:

reg: Register address

data: 32-bit data to be written into the specified register

Return value: Nothing

This function writes specified 32-bit value into the specified register.

6.3.7 readendpoint

```
static int readendpoint(int endpoint, u8 * buffer, int length)
```

Parameters:

endpoint: Endpoint number

buffer: Buffer in which the endpoint data is to be read

length: Length of the data to read

Return: Returns the buffer length on success

This function reads the endpoint data from endpoint buffers.

6.3.8 writeendpoint

```
static void writeendpoint(int endpoint, u8 * buffer, int length)
```

Parameters:

endpoint: Endpoint number

buffer: Endpoint data buffer to be written

length: Length of the data to write

Return: Nothing

This function writes the endpoint data into endpoint buffers.

7 OTG controller interface

OTG is a supplement to *Universal Serial Bus Specification Rev. 2.0* that augments existing USB peripherals by adding to these peripherals limited host capability to support other targeted USB peripherals. It is primarily targeted at portable devices because it addresses concerns related to such devices, such as small connector and low power. Non-portable devices, even standard hosts, can also benefit from OTG features.

The ISP1763A OTG controller is designed to perform all tasks specified in the OTG supplement. It supports Host Negotiation Protocol (HNP) and Session Request Protocol (SRP) for dual-role devices. The ISP1763A uses software implementation of HNP and SRP for maximum flexibility. A set of OTG registers provides control and status monitoring capabilities to support software HNP and SRP.

The OTG driver controls the activities on the OTG port by using the HCD port control interface.

7.1 Module management

This interfaces to the OS and is called at the time of loading and unloading the ISP1763A OTG controller driver to the kernel. The following functional interface is based on the PCI x86 platform Linux platform, and can be modified, depending on the operating system.

7.1.1 usb_otg_module_init

This function initializes the ISP1763A hardware access driver module. The Linux kernel module manager calls this function.

```
static int __init usb_otg_module_init (void)
```

Parameters: None.

Return value:

0: The ISP1763A hardware access driver kernel module is successfully completed.

< 0: The ISP1763 kernel module initialization has failed.

7.1.2 usb_otg_module_cleanup

This function de-initializes the ISP1763A hardware access driver module. The Linux kernel module manager calls this function during the unloading of this module.

```
void __exit usb_otg_module_cleanup (void)
```

Parameters: None.

Return value: None.

7.2 OTG controller routines

7.2.1 otgfsm_pdc_notif

This function processes the notification from the above layer, such as peripheral and hub driver on behalf of the OTG FSM.

```
void otgfsm_pdc_notif(void *priv, unsigned long notif, unsigned
long data)
```

Parameters:

priv: Pointer to the ISP1763A OTG structure.

notif: Integer value indicating device state suspend, reset, or resume.

data: Integer value indicating device HNP enable or support.

Return value: None.

7.2.2 usb_otg_isr_handler

This function is the interrupt handler for OTG interrupts.

```
void static usb_otg_isr_handler(struct isp1763_dev *dev, void *
isr_data)
```

Parameters:

dev: Pointer to the ISP1763A device structure.

isr_data: Pointer to the ISR data handler of the driver.

Return value: None.

7.2.3 otgfsm_current_state

This function prints the current state and OTG controller register values. The function will be called by the application through the usb_otgdev_ioctl function.

```
void otgfsm_current_state(otg_fsm_t * fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.4 otgfsm_deininit

This function de-initializes the OTG FSM state.

```
void otgfsm_deininit(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.5 otgfsm_init

This function initializes the FSM. It also sets the state of the OTG based on the ID pin.

```
void otgfsm_init(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None

7.2.6 otgfsm_dp_pullup

This function controls the local pull-up.

```
void otgfsm_dp_pullup(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.7 otgfsm_dp_pulldown

This function controls the local pull-down.

```
void otgfsm_dp_pulldown(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.8 otgfsm_local_vbus

This function controls the local V_{BUS}.

```
void otgfsm_local_vbus(otg_fsm_t *fsm_data, __u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.9 otgfsm_otg_se0_en

This function controls the 2 ms SE0 state. It triggers bus reset during the BWAIT_ACON state.

```
void otgfsm_otg_se0_en(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.10 otgfsm_run

This function runs the OTG FSM. It loops the state machine until the current state and previous state of OTG are the same.

```
void otgfsm_run(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.11 otgfsm_run_Adevice

This function runs the A-device FSM.

```
static void otgfsm_run_Adevice(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.12 otgfsm_run_Bdevice

This function runs the B-device FSM.

```
static void otgfsm_run_Bdevice(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.13 otgfsm_set_state

This function sets the variable of the FSM, based on the input from the application through usb_otgdev_ioctl. Example, bus request.

```
void otgfsm_set_state(otg_fsm_t *fsm_data, __u8 cmd)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

cmd: Integer value indicating the state to set the host, suspend, bus drop, idle, and peripheral.

Return value: None

7.2.14 otgfsm_status_probe

This function reads and updates OTG FSM variables from the OTG interrupt source register.

```
void otgfsm_status_probe(otg_fsm_t *fsm_data)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

Return value: None.

7.2.15 otgfsm_sw_sel_hc_dc

This function selects which ATX is connected to the OTG port: host controller or peripheral controller. By default, the peripheral controller is selected.

```
void otgfsm_sw_sel_hc_dc(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.16 otgfsm_vbus_chrg

This function charges V_{BUS}. Used to send V_{BUS} pulsing in session request protocol.

```
void otgfsm_vbus_chrg(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None

7.2.17 otgfsm_vbus_drv

This function is to drive V_{BUS}.

```
void otgfsm_vbus_drv(otg_fsm_t *fsm_data,__u8 ctrl_flag)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ctrl_flag: Integer value indicating control true or false.

Return value: None.

7.2.18 OtgHal_AccessCtrlReg

This function is used to set and clear bits in the OTG Control register.

```
unsigned short OtgHal_AccessCtrlReg(otg_fsm_t *fsm_data,unsigned
char
ubAccessType,u32 ulAccessFields)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ubAccessType: Char type value indicating access type read, set, and clear register.

ulAccessFields: Integer value indicating the value to be set in the register.

Return value:

1: Success.

0: Failure.

7.2.19 phOtgHal_ConfigHwForFsmState

This function configures the hardware when the OTG state machine exits the current state and enters the new state.

```
static void phOtgHal_ConfigHwForFsmState(otg_fsm_t *fsm_data,
UCHAR
ubFsmState,UCHAR ubConfigCode)
```

Parameters:

fsm_data: Pointer to the OTG FSM state machine.

ubFsmState: Char type value indicating the FSM state.

ubConfigCode: Char type value indicating configuration code init or exit.

Return value: None.

Glossary

API	Application Programming Interface
ATL	Asynchronous Transfer List
ATX	Analog USB Transceiver
DCD	Device Controller Driver
EHCI	Enhanced Host Controller Interface
EP	Endpoint
FSM	Finite State Machine
HAL	Hardware Abstraction Layer
HCD	Host Controller Driver
HNP	Host Negotiation Protocol
H/W	Hardware
INTL	Interrupt
ISO	Isochronous
ISR	Interrupt Service Routine
MSCD	Mass Storage Controller Driver
OS	Operating System
OTG	On-The-Go
PTD	Proprietary Transfer Descriptor
SOF	Start-Of-Frame
SRP	Session Request Protocol
TD	Transfer Descriptor
URB	USB Request Block
USB	Universal Serial Bus