SYNOPSYS®

DesignWare Cores Hi-Speed USB On-The-Go Controller Subsystem Linux Driver Software

User Guide

4334-0 DWC USB 2.0 HSOTG Linux Driver

Version 2.90a
April 2009

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, and Vera are registered trademarks of Synopsys, Inc.

Trademarks (™)

Active Parasitics, AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BOA, BRT, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, DC Expert, DC Professional, DC Ultra, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, Direct RTL, Direct Silicon Access, Discovery, Dynamic-Macromodeling, Dynamic Model Switcher, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Formal Model Checker, FoundryModel, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSIM^{plus}, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Milkyway, ModelSource, Module Compiler, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Raphael, Raphael-NES, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, Softwire, Source-Level Design, Star-RCXT, Star-SimXT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (℠)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

PCI Express is a trademark of PCI-SIG.

All other product or company names may be trademarks of their respective owners.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

IMPORTANT:

Synopsys DWC HS OTG Linux Software Driver and documentation (hereinafter, “Software”) is an UNSUPPORTED proprietary work of Synopsys, Inc. unless otherwise expressly agreed to in writing between Synopsys and you.

The Software IS NOT an item of Licensed Software or Licensed Product under any End User Software License Agreement or Agreement for Licensed Product with Synopsys or any supplement thereto. You are permitted to use and redistribute this Software in source and binary forms, with or without modification, provided that redistributions of source code must retain this notice. You may not view, use, disclose, copy or distribute this file or any information contained herein except pursuant to this license grant from Synopsys. If you do not agree with this notice, including the disclaimer below, then you are not authorized to use the Software.

THIS SOFTWARE IS BEING DISTRIBUTED BY SYNOPSYS SOLELY ON AN “AS IS” BASIS AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE HEREBY DISCLAIMED. IN NO EVENT SHALL SYNOPSYS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

Revision History

Date	Version	Description
April 2009	2.90a	Added support for Scatter/Gather Descriptor DMA in Host mode.
February 2009	2.81a	Added support for: • Two-threshold enhancement • PCI
November 2008	2.80	• Added support for LPM enhancement • Changed driver architecture to be easily ported to other OS
October 2008	2.72a	Added support for: • Isochronous transfers for Slave and Buffer DMA modes • Periodic Transfer Interrupt HW enhancement • Multi Processor Interrupt HW enhancement
June 2008	2.71a	• Added support for NAK/NYET enhancement for Bulk and Control OUT transfers in DMA mode • Changed ARM integrator to IPMate throughout.
May 2008	2.70a	Added support for Scatter/Gather Descriptor DMA mode in Device mode, and Isochronous transfers in Device mode with enabled Scatter/Gather Descriptor DMA mode. Added • "DMA Descriptor Structure" on page 44 Updated • Linux kernel support to 2.6.20.1 throughout. • Descriptor DMA in "Device Initialization" on page 49 • Descriptor DMA in "The dwc_otg_ep0_start_transfer Function" on page 52 • Descriptor DMA in "The dwc_otg_ep0_continue_transfer Function" on page 53 • Descriptor DMA in "The dwc_otg_ep_activate Function" on page 53
February 2007	2.60a	Added thresholding support, including parameters: • thr_ctl[2:0] • tx_thr_len, rx_thr_len. Updated tables B-1 and B-2.

Date	Version	Description
August 2006	2.50a	Added dedicated transmit FIFO support for non-periodic endpoints, including parameters: • en_multiple_tx_fifo • dev_tx_fifo_size_n, (*n* = 1 to 15)
September 2005	2.20a	Linux kernel release 2.6.12.2 replaces release 2.6.9 as the reference operating environment.
July 2005	2.10a	All transfer types are now supported in Host and Device modes, except split isochronous transfers. Support for the following transfer types was added in the 2.10a release from the 2.05a (non-product) release: • Host and device mode isochronous transfers • Device mode interrupt transfers • Split transfers for all transfer types except isochronous • DMA mode support for all transfer types (only Slave mode was available in previous releases)
September 2005	2.10a	Added legal information on page 3.

Preface

Overview

This user guide defines the functionality of the Linux driver software for the Synopsys DesignWare USB 2.0 Hi-Speed On-The-Go (OTG) Controller Subsystem (DWC_otg), release 2.90a. It specifies what the software does, not how it accomplishes its tasks. You can use this driver software as-is, or as a reference for developing drivers for other operating environments.

Hi-Speed USB OTG Controller Subsystem Linux Driver Software corresponds to 4334-0 DWC USB 2.0 HSOTG Linux Driver in the SolvNet database.

The terms "DWC_otg driver" and "driver software" both refer to the Hi-Speed USB OTG Controller Subsystem Linux Driver Software.

User Guide Organization

The chapters and appendixes of this user guide are organized as follows:

Chapter 1, "Product Overview," describes the software architecture, driver software components, and product deliverables.

Chapter 2, "Environment-Specific Features," defines the driver software implementation and functionality for the Linux environment.

Chapter 3, "Core Interface Layer ," describes the Core Interface Layer (CIL) that provides basic services for accessing and managing the DWC_otg hardware.

Chapter 4, "Peripheral Controller Driver," describes the Peripheral Controller Driver (PCD) that translates requests from the Function Driver into appropriate actions on the DWC_otg controller.

Chapter 5, "Host Controller Driver," describes the Host Controller Driver (HCD) that translates requests from the USB Driver into appropriate actions on the DWC_otg controller.

Appendix A, "Performance Analysis," gives an analysis of the DWC_otg core's load on the CPU for a basic transfer.

Appendix B, "ROM Sizing," gives estimates of how much ROM is needed for the DWC_otg driver and other modules for different types of applications.

Reference Documentation

The following standards are related to the Linux Driver Software product:

- *Universal Serial Bus Specification,* Revision 2.0, USB Implementers Forum, April 27, 2000
- *On-The-Go Supplement to the USB 2.0 Specification,* Revision 1.0a, USB Implementers Forum, June 24, 2003

- *On-The-Go Compliance Plan for the USB 2.0 Specification*, Revision 1.0, USB Implementers Forum, August 12, 2003
- *Full and Low Speed Electrical and Interoperability Compliance Test Procedure,* Revision 1.3, USB Implementers Forum, February 2004
- *DesignWare Cores USB 2.0 Hi-Speed On-The-Go (OTG) Databook*, Version 2.20, Synopsys, Inc., June, 2005

Customer Support

To obtain support for your product, choose one of the following:

- Enter a call through SolvNet.
  - Go to https://solvnet.synopsys.com/ManageCase?ccf=1 and provide the requested information, including:
    - Product: DesignWare Cores
    - Sub Product: USB 2.0 OTG
    - Version: 2.90a
    - Subject: HS OTG Linux Driver Software
- Send an e-mail message to support_center@synopsys.com.
  - Include the Product name, Sub Product name, and Version (product release number) in your e-mail so it can be routed correctly.
  - Provide the following additional information, if applicable:
    - For environment setup problems, run the debug_info command in the coreConsultant GUI and include the text file generated.
    - For configuration failures, include the error messages that appear in the coreConsultant GUI console pane.
    - For simulation failures, include a text file with your specific configuration. Generate this text file using the coreConsultant GUI.s write_batch_script command. Also include the log file from the <workspace>/simulation/ directory, the log file for the specific test, and a VPD/VCD waveform dump file.
    - For synthesis failures, include the log file from the <workspace>/syn/ directory.
- Telephone your local support center.
  - United States:

    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - Canada:

    Call 1-650-584-4200 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:

    http://www.synopsys.com/support/support_ctr

1 Product Overview

1.1 Product Overview

The Synopsys DWC_otg core is a USB On-The-Go (OTG) controller subsystem that is compliant with the *On-the-Go Supplement for USB 2.0*, Revision 1.0a. This controller is not EHCI- or OHCI-compliant. To reduce gate count, some features that would be implemented in hardware in an EHCI/OHCI controller are instead implemented in software.

The Linux driver software described in this user guide is for DWC_otg release 2.90a.

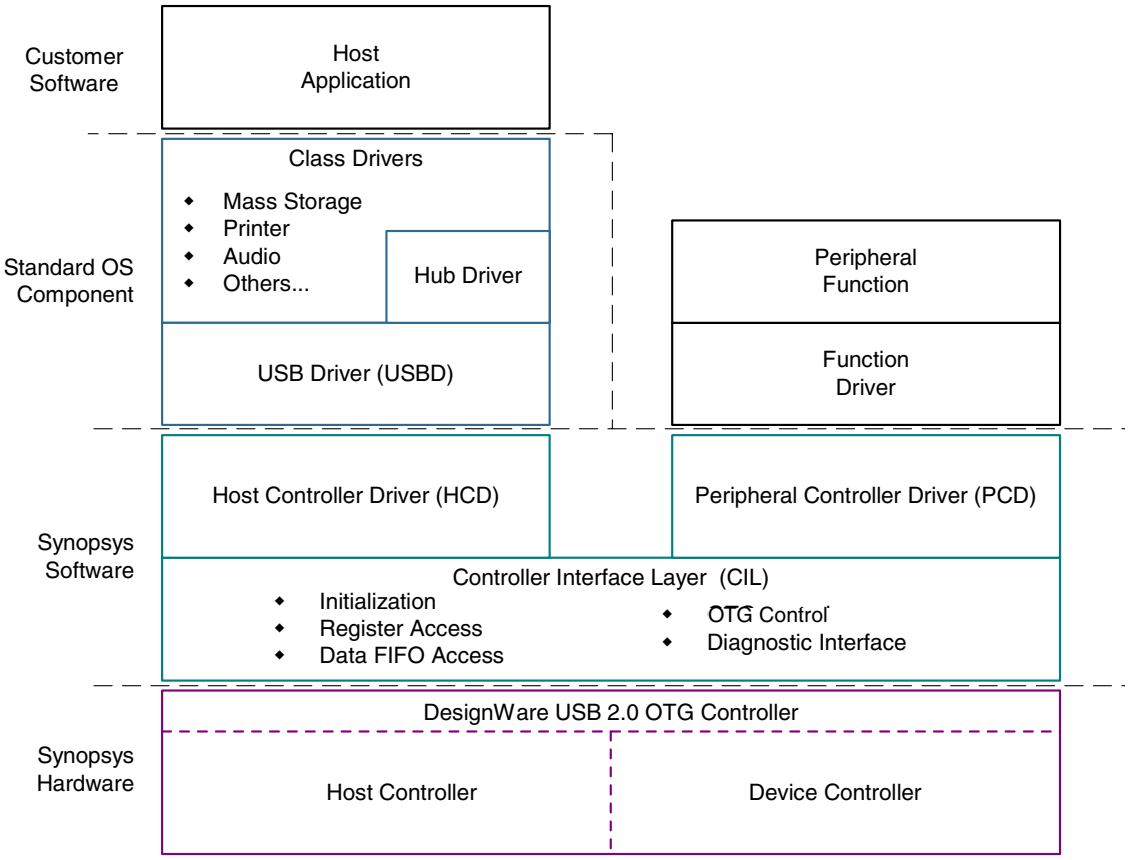
1.2 Software Architecture

Figure 1-1 shows the software architecture for the DWC_otg controller. There are two stacks in the software architecture—the Host Stack and the Peripheral Stack. Brief descriptions of each of these stacks are given below to set the context for the driver software. Since the DWC_otg controller can act as either a host (in Host mode) or a peripheral (in Device mode), portions of both of these stacks are implemented in the DWC_otg controller driver software.

The Host Stack is used to request transfers to or from USB devices when the DWC_otg controller is acting as host. The top-level component in this stack is a Host Application that acts as a producer or consumer of data. The Class Drivers translate application requests into a protocol that is specific to a certain type (or class) of devices. Class Drivers use I/O Request Packets (IRPs) to transfer data to or from USB devices. The USB Driver (USBD) provides services to allow multiple Class Drivers to configure, control, and exchange data with their associated devices. The Hub Driver is also involved in the configuration process. The USBD handles all communication with the Host Controller Driver (HCD), which must interact with the host controller's hardware architecture. The HCD interacts with the host controller to execute USB transfers requested by the USBD.

The Peripheral Stack responds to requests received by a USB device when the OTG component is acting in the role of a peripheral. The Peripheral Function is the sink or source of data requested by the host. The Function Driver handles some USB requests directly. It also provides endpoint read/write data interfaces and notification services to the Peripheral Function. The Peripheral Controller Driver (PCD) interacts with the hardware architecture of the peripheral controller. The PCD interacts with the peripheral controller to transfer data via the USB and notifies the Function Driver of USB requests.

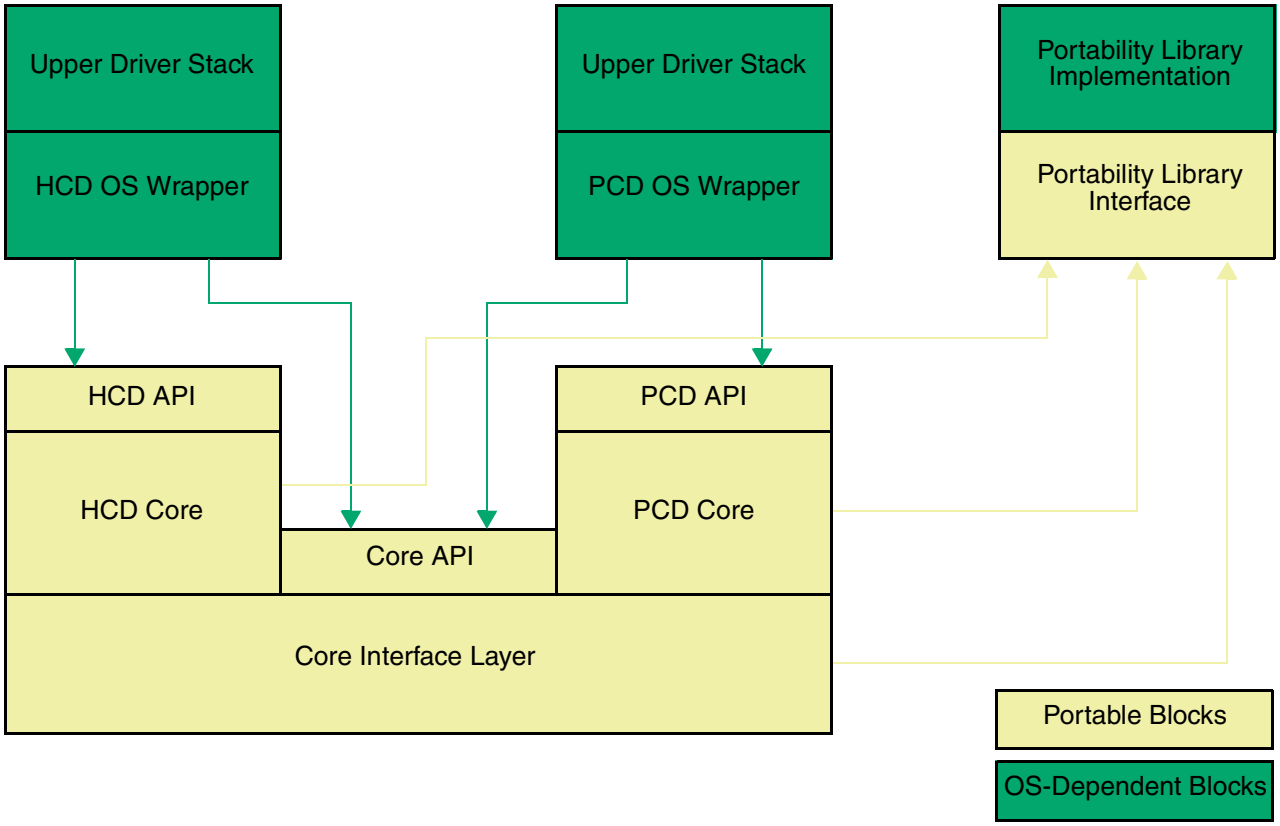
Figure 1-1 DWC_otg Software Architecture



1.2.1 DWC_OTG Driver Architecture

To support portability, the OTG driver has the architecture depicted in Figure 1-2.

Figure 1-2 Driver Architecture



1.3 Driver Software Components

There are three main driver software components: the Host Controller Driver (which includes the HCD core and HCD OS wrapper), the Peripheral Controller Driver (which includes the PCD core and PCD OS wrapper), and the Core Interface Layer (CIL). Basic HCD and PCD functionality is described in Section 1.2, "Software Architecture." Further elaboration of these and a description of the CIL are given below.

The HCD can be viewed as a hardware abstraction layer. In other words, the USBD cannot detect or respond to the underlying hardware of the host controller. It merely transfers data and transmits commands via a software interface with the HCD. Changes made to the host controller do not require changes to this interface (although they do necessitate changes to the internal operation of the HCD).

Similarly, the PCD can be viewed as a hardware abstraction layer. Changes to the peripheral controller hardware require internal changes in the PCD, but do not require changes to the interface between the PCD and the Function Driver.

The Core Interface Layer provides basic services for accessing and managing the DWC_otg hardware. These services are used by both the HCD and the PCD. Services include initialization, register access, data FIFO access, and OTG control. In addition, the CIL provides a basic diagnostic interface that can be used to read and write controller registers.

Separating the Host Stack and Peripheral Stack components allows instantiating only those components required to support the DWC_otg configuration. For example, when you design a product that always operates in Device mode, only the PCD and CIL are required, and the HCD is not instantiated.

1.3.1 Environment Dependencies

All driver components are developed for a Linux operating system using kernel version 2.6.20.1 on an IPMate platform.

The Host Controller Driver (HCD) OS wrapper and Peripheral Controller Driver (PCD) OS wrapper are currently written only for Linux. These components must be modified for other operating environments.

As noted above, changes to the underlying hardware require changes to the internal operation of the driver components. HCD and PCD cores provide their APIs, which are not changed due to hardware changes. All components (HCD core, PCD core, and CIL) are aware of the internal architecture of the controller, register set layout and contents; each of these components may need to be changed to adapt to controller changes. However, for a given operating system (such as Linux), the layers above the HCD and PCD cores do not require any changes. Because these layers use the same API to communicate with the HCD and PCD cores, these cores are considered hardware abstraction layers.

1.4 Deliverables

This section describes the driver components, documentation, and demo software included with the Hi-Speed USB OTG Controller Subsystem Linux Driver Software.

1.4.1 Driver Software

The following components are packaged and released for the DWC_otg controller:

- Host Controller Driver
- Peripheral Controller Driver
- Core Interface Layer

A single module contains all of these components.

These components provide the following support:

- Support for control, bulk, interrupt, and isochronous transfers in both Host and Device modes
- Slave (PIO) mode and integrated DMA mode data FIFO access
- Split transfers (not supported by the DWC_otg controller when Descriptor DMA is enabled)
- USB Suspend and Resume
- On-the-Go Host Negotiation Protocol (HNP) and Session Request Protocol (SRP)

These components are installed in Linux as a single loadable module.

When changes are required to Linux 2.6.20.1 in order to support our configuration of the IPMate platform, a Linux patch will be included in the software deliverables.

1.4.2 Linux Patch

If there are any changes to the Linux 2.6.20.1 kernel that affect support for our IPMate platform configuration, we will include a Linux patch in the software deliverables. Any such patch will be packaged separately from the driver software components, because it is licensed under the GPL (General Public License).

1.4.3 Software Documentation

Source documentation in HTML format will be delivered for the driver components. This includes documentation of the API for each of the driver components.

Instructions for acquiring and installing a Linux kernel of version 2.6.20.1 and applying required patches are included in the release. Standard Linux components are required to run the demo software. These components include:

- Mass storage class driver
- Printer class driver
- USB Core
  - Includes USB driver functionality
  - Includes Hub Driver, which provides hub enumeration and support
- File-Backed Storage Gadget Driver (This is a function driver that acts as a mass storage device and stores/retrieves data from the file system.)

1.4.4 Demo Software

Demo software is delivered with the DWC_otg controller. This software consists of an application that demonstrates the following features:

- Mass storage transfers in Host mode when DWC_otg is connected to a Mass Storage FS/HS digital camera, an FS/HS memory key, or an FS/HS hard disk.
- Printing images to an FS USB printer (HP Photosmart 245 or 375) after the images are transferred from the digital camera.
- Switching from Host to Peripheral mode (and vice-versa) when the A and B ends of the USB cable are swapped.
- Mass storage transfers (FS and HS) in Device mode when DWC_otg is connected as a peripheral to a host system.
- Support for USB Certification testing, including FS-OPT, HS Device, and FS Device testing.

1.4.5 Binaries

Binaries targeted to the IPMate platform are delivered for the following items:

- Linux 2.6.20.1 ARM kernel images for the IPMate
- cramfs image containing the OTG Demo, the OTG Controller driver module, and class drivers required to run the OTG Demo for the IPMate
- Vivi image for IPMate

1.4.6 Portability Library

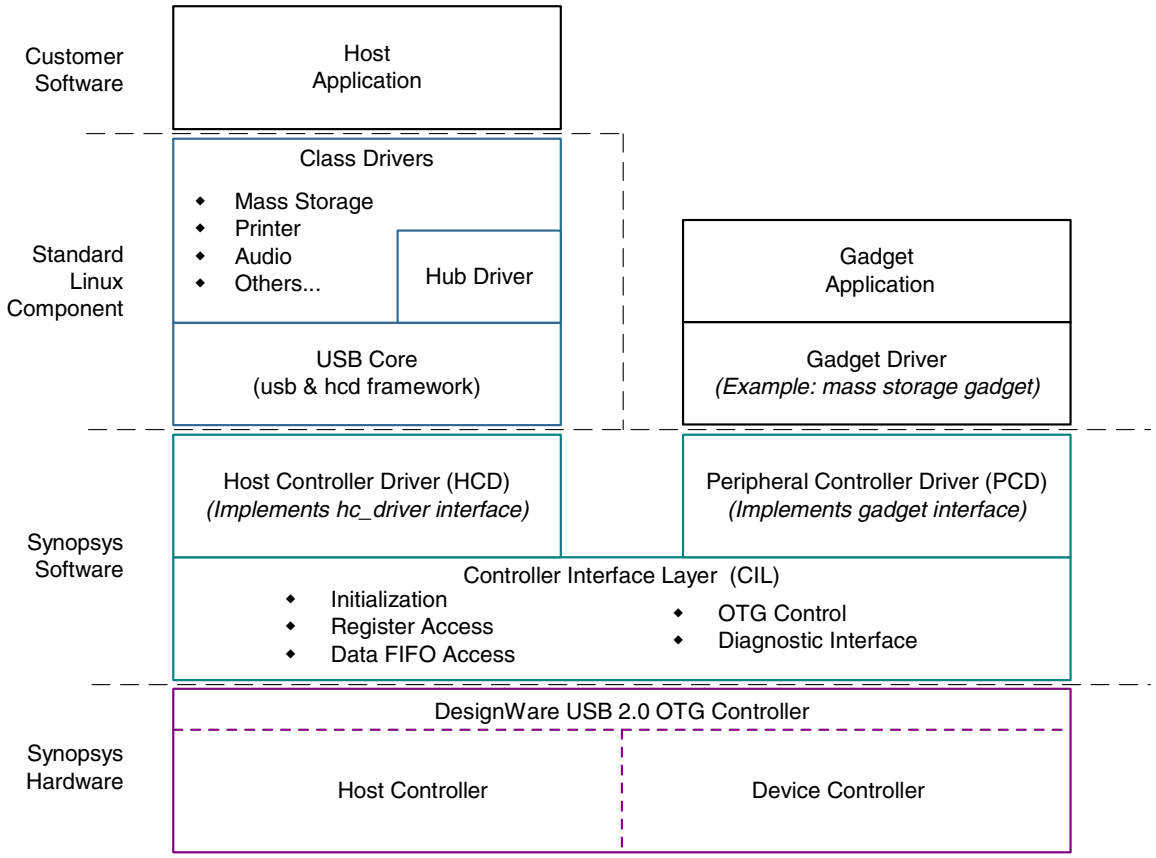
This library provides an OS-independent interface for use by the portable parts of the driver. The implementation of the portability library is OS-dependent.

2 Environment-Specific Features

2.1 Linux Architecture

The Linux architecture, shown in Figure 2-1, is similar to the generic DWC_otg software architecture, shown in Figure 1-1 on page 14.

Figure 2-1 Linux DWC_otg Software Architecture



The USB core component in Linux includes USB driver functionality, plus a framework to support Host Controller Drivers. The HCD framework allows different HCDs to share code. This makes it easier to write new HCDs and reduces the number of bugs. The HCD only implements functionality that requires interaction with the underlying hardware. This is accomplished by defining the hc_driver interface. This interface consists of the functions and data that must be supplied by an HCD in order to plug into the Linux

HCD framework. The Linux version of the DWC_otg HCD will implement the hc_driver interface. See "Linux hc_driver API" on page 79 for more information.

In Linux, peripheral devices are called gadgets. Thus, the Peripheral Function is called a Gadget Application and the Function Driver is called a Gadget Driver. Linux defines a Gadget API, which is the interface between the Gadget Driver and the Peripheral Controller Driver. Similar to the hc_driver interface, the Gadget API isolates hardware-specific behavior to the Peripheral Controller Driver. The Linux version of the DWC_otg PCD implements the Gadget API. See "Linux Gadget API" on page 63 for more information.

The API of the Core Interface Layer is independent of the operating system. Services provided by this component are the same regardless of the operating system.

2.2 Linux Driver Module

All driver components are contained in a single driver module. The module wrapper code, including module initialization and cleanup functions, is Linux-dependent. This section describes the Linux module functionality.

The dwc_otg module provides the initialization and cleanup entry points for the DWC_otg driver. This module is dynamically installed after Linux is booted using the insmod command. When the module is installed, the The dwc_otg_driver_init Function function is called. When the module is removed (using rmmod), the The dwc_otg_driver_cleanup Function function is called.

This module also defines data structures for the The dwc_otg_driver Data Structure and The dwc_otg_device Data Structure. These structures allow the OTG driver to comply with the standard Linux driver model, in which devices and drivers are registered with a bus driver. This has the benefit that Linux can expose attributes of the driver and device in its special sysfs virtual file system. Users can then read or write files in this file system to perform diagnostics on driver components or the device.

2.2.1 Data Structures

Sections 2.2.1.1–2.2.1.2 describe the data structures used by the DWC_otg driver module.

2.2.1.1 The dwc_otg_driver Data Structure

The dwc_otg_driver structure defines the methods to be called by a bus driver during the life cycle of a device on that bus. Both drivers and devices are registered with a bus driver. The bus driver matches devices to drivers based on information in the device and driver structures. dwc_otg_driver can be registered with either the lm_bus or PCI.

The following structure registers dwc_otg_driver with the lm_bus.

```
struct lm_driver dwc_otg_driver = {
  .drv = {.name = "dwc_otg"},
  .probe = dwc_otg_driver_probe,
  .remove = dwc_otg_driver_remove,
};
```

This bus is specific to the IPMate version of Linux. It is used for devices implemented on ARM logic module boards. In the IPMate development system, the DWC_otg controller is implemented on an FPGA that resides on an ARM Logic Module board.

The IPMate version of Linux also defines an lm_device structure. An lm_device is created at system initialization for each logic module present in the system. One of these logic modules is the board containing the FPGA for the DWC_otg controller. Resources (memory address range and interrupt request number) are allocated for each lm_device at system initialization and stored in the lm_device structure.

The following structure registers dwc_otg_driver with the PCI.

```
static struct pci_driver dwc_otg_driver = {
        .name =         "dwc_otg",
        .id_table =     pci_ids,

        .probe =        dwc_otg_driver_probe,
        .remove =       dwc_otg_driver_remove,

        .driver = {
                .name   = (char*)dwc_driver_name,
        },
};
```

The probe function of The dwc_otg_driver Data Structure is called when The dwc_otg_driver Data Structure is registered. The remove function is called when the The dwc_otg_driver Data Structure is unregistered.

2.2.1.2 The dwc_otg_device Data Structure

The dwc_otg_device structure is a wrapper that encapsulates the driver components used to manage a single DWC_otg controller.

```
typedef struct dwc_otg_device
{
        /** Base address returned from ioremap() */
        void *base;

        #ifdef CONFIG_MACH_IPMATE
        struct lm_device *lmdev;
        #elif  CONFIG_PCI
        int rsrc_start;
        int rsrc_len;
        #endif

        /** Pointer to the core interface structure. */
        dwc_otg_core_if_t *core_if;

        /** Register offset for Diagnostic API.*/
        uint32_t reg_offset;

        /** Pointer to the PCD structure. */
        struct dwc_otg_pcd *pcd;

        /** Pointer to the HCD structure. */
        struct dwc_otg_hcd *hcd;

          /** Flag to indicate whether the common IRQ handler is installed. */
          uint8_t common_irq_installed;

} dwc_otg_device_t;
```

2.2.2 Initialization and Cleanup Functions

Sections 2.2.2.1–2.2.2.4 describe the DWC_otg driver module's initialization and cleanup functions.

2.2.2.1 The dwc_otg_driver_init Function

The dwc_otg_driver_init function is called when the The dwc_otg_driver Data Structure is installed with the insmod command. It registers the The dwc_otg_driver Data Structure structures shown in Section 2.2.1.1. This causes the The dwc_otg_driver_probe Function function to be called.

```
int dwc_otg_driver_init(void)
```

2.2.2.2 The dwc_otg_driver_probe Function

The dwc_otg_driver_probe function is called when an lm_device is bound to a The dwc_otg_driver Data Structure. The dwc_otg_driver_probe function creates the driver components required to control the device (CIL, HCD, and PCD) and initializes the device. Additionally, device and driver attributes are exposed in the sysfs file system.

The driver components are stored in a The dwc_otg_device Data Structure structure. A reference to the The dwc_otg_device Data Structure is saved in the lm_device. This allows the driver to access the The dwc_otg_device Data Structure structure on subsequent calls to driver methods for this device.

```
int dwc_otg_driver_probe(
      #ifdef CONFIG_MACH_IPMATE
      struct lm_device *_dev
      #elif CONFIG_PCI
      struct pci_dev *_dev,  const struct pci_device_id *id
      #endif
)
```

2.2.2.3 The dwc_otg_driver_remove Function

The dwc_otg_driver_remove function is called when an lm_device is unregistered with its bus driver. This happens, for example, when the `rmmod` command is executed. The device may or may not be electrically present. When it is present, the driver stops device processing. Any resources used on behalf of this device are freed.

```
int dwc_otg_driver_remove(
      #ifdef CONFIG_MACH_IPMATE
      struct lm_device *_dev
      #elif CONFIG_PCI
      struct pci_dev *_dev
      #endif
)
```

2.2.2.4 The dwc_otg_driver_cleanup Function

The dwc_otg_driver_cleanup function is called when the driver is removed from the kernel with the rmmod command.

```
void dwc_otg_driver_cleanup(void)
```

2.2.3 Module Parameters

The parameters shown in Table 2-1 may be specified when starting the module. These parameters define how the DWC_otg controller should be configured. Parameter values are passed to the CIL initialization function The dwc_otg_cil_init Function.

Table 2-1 CIL Linux Module Parameters

Parameter Name	Description
otg_cap	Specifies the OTG capabilities. The driver automatically detects this parameter's value when none is specified. • 0: HNP- and SRP-capable (default, when available) • 1: SRP Only-capable • 2: Non-HNP/SRP-capable
dma_enable	Specifies whether to use Slave or DMA mode for accessing the data FIFOs.The driver automatically detects this parameter's value when none is specified. • 0: Slave • 1: DMA (default, when available)
dma_burst_size	The DMA Burst size (applicable only for External DMA mode). Values: 1, 4, 8 16, 32 (default), 64, 128, 256
speed	Specifies the maximum speed of operation in Host and Device modes. The actual speed depends on the speed of the attached device and the value of phy_type. • 0: High-Speed (default, when available) • 1: Full-Speed
host_support_fs_ls_low_power	Specifies whether Low Power mode is supported when attached to a Full-Speed or Low-Speed device in Host mode. • 0: Don't support Low Power mode (default) • 1: Support Low Power mode
host_ls_low_power_phy_clk	Specifies the PHY clock rate in Low Power mode when connected to a Low-Speed device in Host mode. This parameter is applicable only when the host_support_fs_ls_low_power module parameter is set to 1. • 0: 48 MHz (default) • 1: 6 MHz
enable_dynamic_fifo	• 0: Use coreConsultant FIFO size parameters • 1: Allow dynamic FIFO sizing (default)
data_fifo_size	Total number of 4-byte words in the data FIFO memory. This memory includes the RxFIFO, Non-Periodic TxFIFO, and Periodic TxFIFOs. Range: 32 to 32768 (default 8192) **Note:** The total FIFO memory depth in the FPGA configuration is 8192.
dev_rx_fifo_size	Number of 4-byte words in the RxFIFO in Device mode when dynamic FIFO sizing is enabled. Range: 16 to 32768 (default 1064)

Table 2-1 CIL Linux Module Parameters (Continued)

Parameter Name	Description
dev_nperio_tx_fifo_size	Number of 4-byte words in the non-periodic Tx FIFO in device mode when dynamic FIFO sizing is enabled. • When en_multiple_tx_fifo mode is enabled, dev_nperio_tx_fifo_size is used only for endpoint 0 • When en_multiple_tx_fifo mode is not enabled, dev_nperio_tx_fifo_size is used for all non-periodic endpoints. Range: 16 to 32768 (default 1024)
dev_perio_tx_fifo_size_*n* (*n* = 1 to 15)	Number of 4-byte words in each of the Periodic TxFIFOs in Device mode. Range: 4 to 768 (default 256)
host_rx_fifo_size	Number of 4-byte words in the RxFIFO in Host mode when dynamic FIFO sizing is enabled. Range: 16 to 32768 (default 1024)
host_nperio_tx_fifo_size	Number of 4-byte words in the Non-Periodic TxFIFO in Host mode when dynamic FIFO sizing is enabled. Range: 16 to 32768 (default 1024)
host_perio_tx_fifo_size	Number of 4-byte words in the Host Periodic TxFIFO when dynamic FIFO sizing is enabled. Range: 16 to 32768 (default 1024)
max_transfer_size	The maximum transfer size supported in bytes. Range: 2047 to 65535 (default 65535)
max_packet_count	The maximum number of packets in a transfer. Range: 15 to 511 (default 511)
host_channels	The number of host channel registers to use. 1 to 16 (default 12) **Note:** The FPGA configuration supports a maximum of 12 host channels.
dev_endpoints	The number of endpoints in addition to endpoint 0 available for Device mode operations. Range: 1 to 15 (default 6 IN and OUT) **Note:** The FPGA configuration supports a maximum of 6 IN and OUT endpoints in addition to endpoint 0.
phy_type	Specifies the type of PHY interface to use. By default, the driver automatically detects phy_type. • 0: Full-Speed • 1: UTMI+ (default, when available) • 2: ULPI
phy_utmi_width	Specifies the UTMI+ data width. This parameter is only applicable for a PHY_TYPE of UTMI+. Also, this parameter is applicable only when the OTG_HSPHY_WIDTH coreConsultant parameter is set to 8 and 16 bits, meaning that the core has been configured to work at either data path width. Value: 8 or 16 bits (default 16)

Table 2-1 CIL Linux Module Parameters (Continued)

Parameter Name	Description
phy_ulpi_ddr	Specifies whether the ULPI operates at double or single data rate. This parameter is only applicable when PHY_TYPE is ULPI. 0: Single data rate ULPI interface with 8-bit-wide data bus (default) 1: Double data rate ULPI interface with 4-bit-wide data bus
i2c_enable	I^2C interface for full speed PHY. This parameter is applicable only when PHY_TYPE = FS. • 0: Disabled (default) • 1: Enabled
ulpi_fs_ls	ULPI FS/LS mode only. • 0: Disabled (default) • 1: Enabled
ts_dline	Term select D-Line pulsing for all PHYs. • 0: Disabled (default) • 1: Enabled
en_multiple_tx_fifo	Specifies whether dedicated transmit FIFOs are enabled for non periodic IN endpoints in device mode
dev_tx_fifo_size_n (n = 1 to 15)	Number of 4-byte words in each of the Tx FIFOs in device mode when dynamic FIFO sizing is enabled. 4 to 768 (default 256)
thr_ctl[2:0]	Thresholding control flag [2:0] • Bit 0: non-ISO Tx Thresholding - 1: Enabled - 0: Disabled • Bit 1: ISO Tx Thresholding - 1: Enabled - 0: Disabled • Bit 2: Rx Thresholding - 1: Enabled - 0: Disabled
tx_thr_len	Thresholding length for Tx FIFOs
rx_thr_len	Thresholding length for Rx FIFOs
dma_desc_enable	Specifies whether to use Buffer DMA or Descriptor DMA mode for accessing the data FIFOs. The driver automatically detects the value for this parameter if none is specified. • 0: Disabled • 1: Enabled (default)

Table 2-1 CIL Linux Module Parameters (Continued)

Parameter Name	Description
pti_enable	Specifies whether to use the enhanced periodic Transfer Interrupt mode. The driver will automatically detect the value for this parameter if none is specified. • 0: Disabled (default) • 1: Enabled
mpi_enable	Specifies whether to use the Multiprocessor Interrupt mode. The driver will automatically detect the value for this parameter if none is specified. • 0: Disabled (default) • 1: Enabled
lpm_enable	Enable LPM support • 0: LPM Disabled • 1: LPM Enabled (default)
ic_usb_cap	Enable IC_USB capability • 0: IC_USB capability is not enabled (default) • 1: IC_USB capability is enabled
ahb_thr_ratio	Threshold Ratio • 0: AHB Threshold = MAC Threshold • 1: AHB Threshold = 1/2 MAC Threshold • 2: AHB Threshold = 1/4 MAC Threshold • 3: AHB Threshold = 1/8 MAC Threshold

Note Thresholding has not been thoroughly tested with release 2.70a

2.2.4 sysfs Attributes

The Linux sysfs file system provides diagnostic access to the controller hardware and the driver software. This file system is mounted at /sys in the directory structure. Kernel attributes are exported to this file system as regular files. These files may be read to access controller/driver status or written to modify controller/driver state.

Table 2-2 shows the attributes provided by the DWC_otg driver.

Table 2-2 Linux sysfs Attributes

Name	Description	Access
mode	Returns the current mode: • 0: Device mode • 1: Host mode	Read
hnpcapable	Gets or sets the HNP-capable bit in the Core USB Configuration register (GUSBCFG). Read returns the current value.	Read/Write

Table 2-2 Linux sysfs Attributes (Continued)

Name	Description	Access
srpcapable	Gets or sets the SRP-capable bit in the Core USB Configuration register (GUSBCFG). Read returns the current value.	Read/Write
hnp	Initiates the Host Negotiation Protocol. Read returns the status.	Read/Write
srp	Initiates the Session Request Protocol. Read returns the status.	Read/Write
remote_wakeup	On a read, shows the remote wakeup status. On a write, initiates a remote wakeup of the host. When bit 0 is 1 and Remote Wakeup is enabled, the Remote Wakeup signalling bit in the Device Control register (DCTL) is set for 1 millisecond.	Read/Write
buspower	Gets or sets the Power State of the bus • 0: Off • 1: On	Read/Write
bussuspend	Suspends the USB	Read/Write
busconnected	Gets the connection status of the bus	Read
gotgctl	Gets or sets the OTG Control and Status register (GOTGCTL)	Read/Write
gusbcfg	Gets or sets the Core USB Configuration register (GUSBCFG)	Read/Write
grxfsiz	Gets or sets the Receive FIFO Size register (GRXFSIZ)	Read/Write
gnptxfsiz	When en_multiple_tx_fifo is not set: • Gets or sets the Non-Periodic Transmit FIFO Size register (GNPTXFSIZ) When en_multiple_tx_fifo is set: • Gets or sets the Endpoint0 Transmit FIFO Size Register	Read/Write
gpvndctl	Gets or sets the PHY Vendor Control register (GPVNDCTL)	Read/Write
ggpio	Gets the value in the lower 16 bits of the General Purpose I/O register (GGPIO) or sets the upper 16 bits	Read/Write
guid	Gets or sets the value of the User ID register (GUID)	Read/Write
gsnpsid	Gets the value of the Synopsys ID register (GSNPSID)	Read
devspeed	Gets or sets the device speed setting in the Device Configuration register (DCFG)	Read/Write
enumspeed	Gets the device's enumeration speed	Read
hptxfsiz	Gets the value of the Host Periodic Transmit FIFO Size register (HPTXFSIZ)	Read
hprt0	Gets or sets the value in the Host Port Control and Status register ((HPRT)	Read/Write
regvalue	Gets or sets the value of the register at the offset in the regoffset attribute	Read/Write
regoffset	Gets or sets the register offset for the next register access	Read/Write
regdump	Dumps the contents of core registers	Read

Table 2-2 Linux sysfs Attributes (Continued)

Name	Description	Access
spramdump	Dumps the contents of SPRAM	Read
hcddump	Dumps the current HCD state	Read
hcd_frrem	Shows the average value of the Frame Time Remaining field in the Host Frame Number/Frame Time Remaining register (HFNUM) when an SOF interrupt occurs. This can be used to determine the average interrupt latency. Also shows the average Frame Time Remaining value for start_transfer and the “a” and “b” sample points. The “a” and “b” sample points may be used during debugging to determine how long it takes to execute a section of the HCD code.	Read
lpm_response	Gets or sets LPM response mode. Applicable only in device mode. • 0: NYET response to LPM transaction • 1: ACK response to LPM transaction	Read/Write
sleep_status	Shows the core’s sleep state. On write, if in sleep state and in host mode, initiates resume for local device. • 0: Core is not in Sleep state • 1: Core is in Sleep state	Read/Write
inv_sel_hsic	Gets or sets the “HSIC-Invert Select” bit in the GLPMCFG Register. Read returns the current value.	Read/Write
hsic_connect	Gets or sets the “HSIC-Connect” bit in the GLPMCFG Register. Read returns the current value.	Read/Write

Example usage (assuming the DWC_otg controller is on Logic Module 0 in the IPMate):

To get the current mode:

```
cat /sys/devices/lm0/mode
```

To power down the USB:

```
echo 0 > /sys/devices/lm0/buspower
```

3 Core Interface Layer

3.1 Core Interface Layer Overview

The Core Interface Layer (CIL) provides basic services for accessing and managing the DWC_otg hardware. These services are used by both the Host Controller Driver and the Peripheral Controller Driver.

The CIL manages the memory map for the core so that the HCD and PCD do not have to do this separately. The CIL also handles such basic tasks as reading/writing the registers and data FIFOs in the controller. Some of the data access functions provide encapsulation of several operations required to perform a task, such as writing multiple registers to start a transfer. Finally, the CIL performs basic services that are not specific to either the Host or Device modes of operation. These services include managing the OTG Host Negotiation Protocol (HNP) and Session Request Protocol (SRP).

The Core Interface Layer has the following requirements:

- Provide basic controller operations
- Be completely portable
- Use a portability library to abstract the OS services used

Though most of the CIL functionality is used by the appropriate modules' HCD and PCD core layers, the CIL exposes a core API that is also used by the OS wrapper layer, including common initialization, interrupt generic handler, and core parameter access functions.

3.2 Data Structures

This section defines the data structures used by the Core Interface Layer API functions.

3.2.1 Control and Status Register Structures

The structures in Sections 3.2.1.1–3.2.1.3 define the size and relative field offsets for each register in the DWC_otg controller. These structures are not created in memory through normal memory allocation methods. After mapping memory for the controller into the OS memory space, these structures are overlaid on the mapped memory by setting the appropriate base address for each structure. Each register can then be accessed via its address in one of these structures.

The precise method for accessing registers is OS-specific: it may be as simple as directly reading or writing a register field in one of these structures, or it may require passing the mapped register address to a read/write method defined by the OS.

3.2.1.1 Core Global Registers Structure

The dwc_otg_core_global_regs structure defines the size and relative field offsets for the Core Global registers.

```
typedef struct dwc_otg_core_global_regs {
  volatile uint32_t gotgctl;    // OTG Control and Status Register
                                //   000h
  volatile uint32_t gotgint;    // OTG Interrupt Register
                                //   004h
  volatile uint32_t gahbcfg;    // Core AHB Configuration Register
                                //   008h
  volatile uint32_t gusbcfg;    // Core USB Configuration Register
                                //   00Ch
  volatile uint32_t grstctl;    // Core Reset Register
                                //   010h
  volatile uint32_t gintsts;    // Core Interrupt Register
                                //   014h
  volatile uint32_t gintmsk;    // Core Interrupt Mask Register
                                //   018h
  volatile uint32_t grxstsr;    // Receive Status Queue Read
                                // Register (Read Only)
                                //   01Ch
  volatile uint32_t grxstsp;    // Receive Status Queue Read &
                                // POP Register (Read Only)
                                //   020h
  volatile uint32_t grxfsiz;    // Receive FIFO Size Register
                                //   024h
  volatile uint32_t gnptxfsiz;  // Non Periodic Transmit FIFO Size
                                // Register
                                // when en_multiple_tx_fifo is
                                // set ep0 Transmit FIFO size
                                //   028h
  volatile uint32_t gnptxsts;   // Non Periodic Transmit FIFO/Queue
                                // Status Register (Read Only)
                                //   02Ch
  volatile uint32_t gi2cctl;    // I2C Access Register
                                //   030h
  volatile uint32_t gpvndctl;   // PHY Vendor Control Register
                                //   034h
  volatile uint32_t ggpio;      // General Purpose Input/Output Register
                                //   038h
  volatile uint32_t guid;       // User ID Register
                                //   03Ch
  volatile uint32_t gsnpsid;    // Synopsys ID Register (Read Only)
                                //   040h
  volatile uint32_t ghwcfg1;    // User HW Config1 Register
                                // (Read Only)
                                //   044h
  volatile uint32_t ghwcfg2;    // User HW Config2 Register
                                // (Read Only)
                                //   048h
  volatile uint32_t ghwcfg3;    // User HW Config3 Register
                                // (Read Only)
                                //   04Ch
```

```
  volatile uint32_t ghwcfg4;    // User HW Config4 Register
                                // (Read Only)
                                //   050h
  uint32_t reserved[43];        // Reserved
                                //   054h - 0FFh
  volatile uint32_t hptxfsiz;   // Host Periodic Transmit FIFO
                                // Size Register
                                //   100h
  volatile uint32_t
  dptxfsiz_dieptxf[15];
                                  // When en_multiple_tx_fifo is
                                  // not set
                                  // Device Periodic Transmit
                                  // FIFO#n Register (n= 1 to 15)
                                  //   104h - 13Ch
                                  // Otherwise Device Transmit
                                  // FIFO#n Register (n= 1 to 15)
                                  //   104h - 13Ch
} dwc_otg_core_global_regs_t;
```

3.2.1.2 Device Mode Register Structures

The following structures define the size and relative field offsets for Device mode registers.

```
// Device Global Registers 800h-BFFh
typedef struct dwc_otg_dev_global_regs {
  volatile uint32_t dcfg;       // Device Configuration Register
                                //   800h
  volatile uint32_t dctl;       // Device Control Register
                                //   804h
  volatile uint32_t dsts;       // Device Status Register
                                // (Read Only)
                                //   808h
  uint32_t unused;              // Reserved
                                //   80Ch
  volatile uint32_t diepmsk;    // Device IN Endpoint Common
                                // Interrupt Mask Register
                                //   810h
  volatile uint32_t doepmsk;    // Device OUT Endpoint Common
                                // Interrupt Mask Register
                                //   814h
  volatile uint32_t daint;      // Device All Endpoints Interrupt
                                // Register
                                //   818h
  volatile uint32_t daintmsk;   // Device All Endpoints Interrupt
                                // Mask Register
                                //   81Ch
  volatile uint32_t dtknqr1;    // Device IN Token Queue Read
                                //  Register-1 (Read Only)
                                //   820h
  volatile uint32_t dtknqr2;    // Device IN Token Queue Read
                                // Register-2 (Read Only)
                                //   824h
  volatile uint32_t dvbusdis;   // Device VBUS Discharge Register
                                //   828h
```

```
  volatile uint32_t dvbuspulse; // Device VBUS Pulse Register
                                //   82Ch
  volatile uint32_t dtknqr3 dthrctl;
                                // Device IN Token Queue Read
                                // Register-3 (Read Only)
                                // If dedicated FIFOs are enabled
                                // threshold control reg
                                //   830h
  volatile uint32_t dtknqr4 fifoemptymsk;
                                // Device IN Token Queue Read
                                // Register-4 (Read Only)
                                // if dedicated FIFOs are enabled
                                // FIFOs empty intr mask reg.
                                //   834h
      volatile uint32_t deachint;
                                // Device Each Endpoint Interrupt
                                // Register (Read Only)
                                //   838h
      volatile uint32_t deachintmsk;
                                // Device Each Endpoint Interrupt
                                // mask Register (Read/Write)
                                //   83Ch
      volatile uint32_t diepeachintmsk[16];
                                // Device Each In Endpoint
                                // Interrupt mask Register
                                // (Read/Write).
                                //   840h
      volatile uint32_t doepeachintmsk[16];
                                // Device Each Out Endpoint
                                // Interrupt mask Register
                                // (Read/Write).
                                //   834h
} dwc_otg_dev_global_regs_t;

// Device Logical IN Endpoint-Specific Registers 900h-AFCh
typedef struct dwc_otg_dev_in_ep_regs {
  volatile uint32_t diepctl;    // Device IN Endpoint Control
                                // Register
                                //   900h + (ep_num * 20h) + 00h
  uint32_t reserved04;          // Reserved
                                //   900h + (ep_num * 20h) + 04h
  volatile uint32_t diepint;    // Device IN Endpoint Interrupt
                                // Register
                                //   900h + (ep_num * 20h) + 08h
  uint32_t reserved0C;          // Reserved
                                //   900h + (ep_num * 20h) +0Ch
  volatile uint32_t dieptsiz;   // Device IN Endpoint Transfer
                                // Size Register
                                //   900h + (ep_num * 20h) + 10h
  volatile uint32_t diepdma;    // Device IN Endpoint DMA
                                // Address Register
                                //   900h + (ep_num * 20h) + 14h
```

```
      volatile uint32_t dtxfsts // Device In Endpoint Transmit Fifo
                                // Status Register
                                //   900h + (ep_num * 20h) + 18h
      volatile  uint32_diepdmab;// DMA Buffer address register
                                // - 900h + (ep_num * 20h) + 1Ch
} dwc_otg_dev_in_ep_regs_t;

// Device Logical OUT Endpoint-Specific Registers B00h-CFCh
typedef struct dwc_otg_dev_out_ep_regs {
  volatile uint32_t doepctl;    // Device OUT Endpoint Control
                                // Register
                                //   B00h + (ep_num * 20h) + 00h
  volatile uint32_t doepfn;     // Device OUT Endpoint Frame
                                // number Regsiter
                                //   B00h + (ep_num * 20h) + 04h
  volatile uint32_t doepint;    // Device OUT Endpoint Interrupt
                                // Register
                                //   B00h + (ep_num * 20h) + 08h
  uint32_t reserved0C;          // Reserved
                                //   B00 + (ep_num * 20h) +0Ch
  volatile uint32_t doeptsiz;   // Device OUT Endpoint Transfer
                                // Size Register
                                //   B00h + (ep_num * 20h) + 10h
  volatile uint32_t doepdma;    // Device OUT Endpoint DMA
                                // Address Register
                                //   B00h + (ep_num * 20h) + 14h
  uint32_t unused               //   B00h + (ep_num * 20h) + 18h
  volatile uint32_t doepdmab;   // DMA Buffer address register
                                // - B00h + (ep_num * 20h) + 1Ch
} dwc_otg_dev_out_ep_regs_t;
```

3.2.1.3 Host Mode Register Structures

The following structures define the size and relative field offsets for the Host mode registers.

```
// Host Global Registers 400h-7FFh
typedef struct dwc_otg_host_global_regs {
  volatile uint32_t hcfg;       // Host Configuration Register
                                //   400h
  volatile uint32_t hfir;       // Host Frame Interval Register
                                //   404h
  volatile uint32_t hfnum;      // Host Frame Number /
                                // Frame Remaining Register
                                //   408h
  uint32_t reserved40C;         // Reserved
                                //   40Ch
  volatile uint32_t hptxsts;    // Host Periodic Transmit FIFO/
                                // Queue Status Register
                                //   410h
  volatile uint32_t haint;      // Host All Channels Interrupt
                                // Register
                                //   414h
```

```
  volatile uint32_t haintmsk;   // Host All Channels Interrupt
                                // Mask Register
                                //   418h
  volatile uint32_t hflbaddr;   // Host Frame List Base Address
                                // Register
                                //   41Ch
} dwc_otg_host_global_regs_t;

// Host Channel Specific Registers 500h-5FCh
typedef struct dwc_otg_hc_regs {
  volatile uint32_t hcchar;     // Host Channel 0 Characteristic
                                // Register
                                //   500h + (chan_num * 20h) + 00h
  volatile uint32_t hcsplt;     // Host Channel 0 Split Control
                                // Register
                                //   500h + (chan_num * 20h) + 04h
  volatile uint32_t hcint;      // Host Channel 0 Interrupt
                                // Register
                                //   500h + (chan_num * 20h) + 08h
  volatile uint32_t hcintmsk;   // Host Channel 0 Interrupt
                                // Mask Register
                                //   500h + (chan_num * 20h) + 0Ch
  volatile uint32_t hctsiz;     // Host Channel 0 Transfer Size
                                // Register
                                //   500h + (chan_num * 20h) + 10h
  volatile uint32_t hcdma;      // Host Channel 0 DMA Address
                                // Register
                                //   500h + (chan_num * 20h) + 14h
  uint32_t reserved;            // Reserved
                                //   500h + (chan_num * 20h) + 18h
  volatile uint32_t hcdmab;     // Host Channel 0 DMA Buffer
                                // Address Register
                                // - 500h + (chan_num * 20h) + 1Ch
} dwc_otg_hc_regs_t;
```

3.2.2 OTG Device Interface Structure

The dwc_otg_dev_if structure contains information required to manage the DWC_otg controller acting in device mode. It is the programming view of the controller's device-specific aspects.

```
typedef struct dwc_otg_dev_if
{
    /** Pointer to device Global registers.
     * Device Global Registers starting at offset 800h
     */
    dwc_otg_device_global_regs_t *dev_global_regs;
#define DWC_DEV_GLOBAL_REG_OFFSET 0x800

    /**
     * Device Logical IN Endpoint-Specific Registers 900h-AFCh
     */
    dwc_otg_dev_in_ep_regs_t     *in_ep_regs[MAX_EPS_CHANNELS];
#define DWC_DEV_IN_EP_REG_OFFSET 0x900
#define DWC_EP_REG_OFFSET 0x20
```

```
    /** Device Logical OUT Endpoint-Specific Registers B00h-CFCh */
    dwc_otg_dev_out_ep_regs_t    *out_ep_regs[MAX_EPS_CHANNELS];
#define DWC_DEV_OUT_EP_REG_OFFSET 0xB00

    /* Device configuration information*/
    uint8_t  speed;              /**< Device Speed  0: Unknown, 1: LS, 2:FS, 3: HS */
    uint8_t  num_in_eps;         /**< Number # of Tx EP range: 0-15 exept ep0 */
    uint8_t  num_out_eps;        /**< Number # of Rx EP range: 0-15 exept ep 0*/

    /** Size of periodic FIFOs (Bytes) */
    uint16_t perio_tx_fifo_size[MAX_PERIO_FIFOS];

    /** Size of Tx FIFOs (Bytes) */
    uint16_t tx_fifo_size[MAX_TX_FIFOS];

      /** Thresholding enable flags and length varaiables **/
    uint16_t rx_thr_en;
    uint16_t iso_tx_thr_en;
    uint16_t non_iso_tx_thr_en;

    uint16_t rx_thr_length;
    uint16_t tx_thr_length;

      /**
       * Pointers to the DMA Descriptors for EP0 Control
       * transfers (virtual and physical)
       */

      /** 2 descriptors for SETUP packets */
      uint32_t dma_setup_desc_addr[2];
      dwc_otg_dev_dma_desc_t* setup_desc_addr[2];

      /** Pointer to Descriptor with latest SETUP packet */
      dwc_otg_dev_dma_desc_t* psetup;

      /** Index of current SETUP handler descriptor */
      uint32_t setup_desc_index;

      /** Descriptor for Data In or Status In phases */
      uint32_t dma_in_desc_addr;
      dwc_otg_dev_dma_desc_t* in_desc_addr;

      /** Descriptor for Data Out or Status Out phases */
      uint32_t dma_out_desc_addr;
      dwc_otg_dev_dma_desc_t* out_desc_addr;

} dwc_otg_dev_if_t;
```

3.2.3 OTG Host Interface Structure

The dwc_otg_host_if structure contains information required to manage the DWC_otg controller acting in Host mode. It is the programming view of the controller's host-specific aspects.

```
typedef struct dwc_otg_host_if {
  // Host Global Registers starting at offset 400h
  dwc_otg_host_global_regs_t *host_global_regs;

  // Host Port 0 Control and Status Register at offset 440h
  volatile uint32_t *hprt0;

  // Host Channel Specific Registers 500h-5FCh
  dwc_otg_hc_regs_t          *hc_regs[16];

  // Host configuration information
  uint8_t  num_host_channels;    // range: 1-16
  uint8_t  perio_eps_supported;  // 0: no, 1: yes
  uint16_t perio_tx_fifo_size;   // Only 1 host periodic TxFIFO

} dwc_otg_host_if_t;
```

3.2.4 OTG Core Interface Structure

The dwc_otg_core_if structure contains information required to manage the DWC_otg controller acting in either Host or Device mode. It is the programming view of the controller as a whole.

```
typedef struct dwc_otg_core_if
{
    /** Parameters that define how the core should be configured.*/
    dwc_otg_core_params_t      *core_params;

    /** Core Global registers starting at offset 000h. */
      dwc_otg_core_global_regs_t *core_global_regs;

    /** Device-specific information */
    dwc_otg_dev_if_t           *dev_if;
    /** Host-specific information */
    dwc_otg_host_if_t          *host_if;

      /** Value from SNPSID register */
      uint32_t snpsid;

      /*
       * Set to 1 if the core PHY interface bits in USBCFG have been
       * initialized.
       */
      uint8_t phy_init_done;

      /*
       * SRP Success flag, set by srp success interrupt in FS I2C mode
       */
      uint8_t srp_success;
      uint8_t srp_timer_started;

    /* Common configuration information */
      /** Power and Clock Gating Control Register */
      volatile uint32_t *pcgcctl;
#define DWC_OTG_PCGCCTL_OFFSET 0xE00
```

```
    /** Push/pop addresses for endpoints or host channels.*/
    uint32_t *data_fifo[MAX_EPS_CHANNELS];
#define DWC_OTG_DATA_FIFO_OFFSET 0x1000
#define DWC_OTG_DATA_FIFO_SIZE 0x1000

    /** Total RAM for FIFOs (Bytes) */
    uint16_t total_fifo_size;
    /** Size of Rx FIFO (Bytes) */
    uint16_t rx_fifo_size;
    /** Size of Non-periodic Tx FIFO (Bytes) */
    uint16_t nperio_tx_fifo_size;


      /** 1 if DMA is enabled, 0 otherwise. */
      uint8_t dma_enable;

      /** 1 if Descriptor DMA mode is enabled, 0 otherwise. */
      uint8_t           dma_desc_enable;

      /** 1 if PTI Enhancement mode is enabled, 0 otherwise. */
      uint8_t           pti_enh_enable;

      /** 1 if MPI Enhancement mode is enabled, 0 otherwise. */
      uint8_t           multiproc_int_enable;

      /** 1 if dedicated Tx FIFOs are enabled, 0 otherwise. */
      uint8_t           en_multiple_tx_fifo;

      /** Set to 1 if multiple packets of a high-bandwidth transfer is in
       * process of being queued */
      uint8_t queuing_high_bandwidth;

      /** Hardware Configuration -- stored here for convenience.*/
      hwcfg1_data_t hwcfg1;
      hwcfg2_data_t hwcfg2;
      hwcfg3_data_t hwcfg3;
      hwcfg4_data_t hwcfg4;

      /** Host and Device Configuration -- stored here for convenience.*/
      hcfg_data_t hcfg;
      dcfg_data_t dcfg;

        /** The operational State, during transations
     * (a_host>>a_peripherial and b_device=>b_host) this may not
     * match the core but allows the software to determine
     * transitions.
     */
    uint8_t op_state;

      /**
       * Set to 1 if the HCD needs to be restarted on a session request
       * interrupt. This is required if no connector ID status change has
       * occurred since the HCD was last disconnected.
```

```
       */
      uint8_t restart_hcd_on_session_req;

    /** HCD callbacks */
    /** A-Device is a_host */
#define A_HOST                          (1)
    /** A-Device is a_suspend */
#define A_SUSPEND                       (2)
    /** A-Device is a_peripherial */
#define A_PERIPHERAL (3)
    /** B-Device is operating as a Peripheral. */
#define B_PERIPHERAL                    (4)
    /** B-Device is operating as a Host. */
#define B_HOST (5)

    /** HCD callbacks */
    struct dwc_otg_cil_callbacks *hcd_cb;
    /** PCD callbacks */
    struct dwc_otg_cil_callbacks *pcd_cb;

      /** Device mode Periodic Tx FIFO Mask */
      uint32_t p_tx_msk;
      /** Device mode Periodic Tx FIFO Mask */
      uint32_t tx_msk;

      /** Workqueue object used for handling several interrupts */
      struct workqueue_struct *wq_otg;

      /** Work object used for handling "Connector ID Status Change" Interrupt */
      struct work_struct w_conn_id;

      /** Work object used for handling "Wakeup Detected" Interrupt */
      struct delayed_workw_wkp;
  /** Lx state of device */
   dwc_otg_lx_state_e lx_state;

} dwc_otg_core_if_t;
```

3.2.5 Endpoint Structure

The dwc_ep structure represents the state of a single endpoint when the DWC_otg controller is in Device mode. It contains the data items required for an endpoint to be activated and transfer packets.

```
typedef struct dwc_ep {
      /** EP number used for register address lookup */
      uint8_t num;
      /** EP direction 0 = OUT */
      unsigned is_in:1;
      /** EP active. */
      unsigned active:1;

      /**
       * Periodic Tx FIFO # for IN Eps.
 * For INTR EP set to 0 to use non-periodic Tx FIFO
```

```
     * If dedicated Tx FIFOs are enabled for all
     * IN Eps - Tx FIFO # FOR IN Eps
     */
          unsigned tx_fifo_num:4;
          /** EP type: 0 - Control, 1 - ISOC, 2 - BULK,3 - INTR */
          unsigned type:2;
    #define DWC_OTG_EP_TYPE_CONTROL   0
    #define DWC_OTG_EP_TYPE_ISOC      1
    #define DWC_OTG_EP_TYPE_BULK      2
    #define DWC_OTG_EP_TYPE_INTR      3
    /** DATA start PID for INTR and BULK EP */
          unsigned data_pid_start:1;
          /** Frame (even/odd) for ISOC EP */
          unsigned even_odd_frame:1;
          /** Max Packet bytes */
          unsigned maxpacket:11;

          /** Max Transfer size */
          uint32_t maxxfer;

          /** @name Transfer state */
          /** @{ */

          /**
           * Pointer to the beginning of the transfer
     * buffer - do not modify during transfer.
           */

          dwc_dma_t dma_addr;

          dwc_dma_t dma_desc_addr;
          dwc_otg_dev_dma_desc_t *desc_addr;

          uint8_t *start_xfer_buff;
          /** pointer to the transfer buffer */
          uint8_t *xfer_buff;
          /** Number of bytes to transfer */
          unsigned xfer_len:19;
          /** Number of bytes transferred. */
          unsigned xfer_count:19;
          /** Sent ZLP */
          unsigned sent_zlp:1;
          /** Total len for control transfer */
          unsigned total_len:19;

          /** stall clear flag */
          unsigned stall_clear_flag:1;

    #ifdef DWC_UTE_CFI
          /* The buffer mode */
          data_buffer_mode_e buff_mode;

          /* The chain of DMA descriptors.
           * MAX_DMA_DESCS_PER_EP will be allocated for each active EP.
```

```
	 */
	dwc_otg_dev_dma_desc_t *descs;

	/* The DMA address of the descriptors chain start */
	dma_addr_t descs_dma_addr;
	/** This variable stores the length of the last enqueued request */
	uint32_t cfi_req_len;
#endif			//DWC_UTE_CFI
	/** Allocated DMA Desc count */
	uint32_t desc_cnt;

#ifdef DWC_EN_ISOC
	/**
	 * Variables specific for ISOC EPs
	 *
	 */
	/** DMA addresses of ISOC buffers */
	dwc_dma_t dma_addr0;
	dwc_dma_t dma_addr1;

	dwc_dma_t iso_dma_desc_addr;
	dwc_otg_dev_dma_desc_t *iso_desc_addr;

	/** pointer to the transfer buffers */
	uint8_t *xfer_buff0;
	uint8_t *xfer_buff1;

	/** number of ISOC Buffer is processing */
	uint32_t proc_buf_num;
	/** Interval of ISOC Buffer processing */
	uint32_t buf_proc_intrvl;

	/** Data size for regular frame */
	uint32_t data_per_frame;

	/** Data size for pattern frame */
	uint32_t data_pattern_frame;
	/** Frame number of pattern data */
	uint32_t sync_frame;

	/** bInterval */
	uint32_t bInterval;

	/** ISO Packet number per frame */
	uint32_t pkt_per_frm;

	/** Next frame num for which will be setup DMA Desc */
	uint32_t next_frame;

	/** Number of packets per buffer processing */
	uint32_t pkt_cnt;

	/** Info for all isoc packets */
	iso_pkt_info_t *pkt_info;
```

```
        /** current pkt number */
        uint32_t cur_pkt;

        /** current pkt number */
        uint8_t *cur_pkt_addr;

        /** current pkt number */
        uint32_t cur_pkt_dma_addr;
#endif /* DWC_EN_ISOC */
/** @} */
} dwc_ep_t;
```

3.2.6 Host Channel Structure

The dwc_hc structure represents the state of a single host channel when the DWC_otg controller is in Host mode. It contains the data items required to transfer packets to an endpoint via a host channel.

```
typedef struct dwc_hc
{
        /** Host channel number used for register address lookup */
        uint8_t  hc_num;

        /** Device to access */
        unsigned dev_addr : 7;

        /** EP to access */
        unsigned ep_num : 4;

        /** EP direction. 0: OUT, 1: IN */
        unsigned ep_is_in : 1;

        /**
         * EP speed.
         * One of the following values:
         *     - DWC_OTG_EP_SPEED_LOW
         *     - DWC_OTG_EP_SPEED_FULL
         *     - DWC_OTG_EP_SPEED_HIGH
         */
        unsigned speed : 2;
#define DWC_OTG_EP_SPEED_LOW0
#define DWC_OTG_EP_SPEED_FULL1
#define DWC_OTG_EP_SPEED_HIGH2

        /**
         * Endpoint type.
         * One of the following values:
         *     - DWC_OTG_EP_TYPE_CONTROL: 0
         *     - DWC_OTG_EP_TYPE_ISOC: 1
         *     - DWC_OTG_EP_TYPE_BULK: 2
         *     - DWC_OTG_EP_TYPE_INTR: 3
         */
        unsigned ep_type : 2;
```

```
	/** Max packet size in bytes */
	unsigned max_packet : 11;

	/**
	 * PID for initial transaction.
	 * 0: DATA0,<br>
	 * 1: DATA2,<br>
	 * 2: DATA1,<br>
	 * 3: MDATA (non-Control EP),
	 *    SETUP (Control EP)
	 */
	unsigned data_pid_start : 2;
#define DWC_OTG_HC_PID_DATA0 0
#define DWC_OTG_HC_PID_DATA2 1
#define DWC_OTG_HC_PID_DATA1 2
#define DWC_OTG_HC_PID_MDATA 3
#define DWC_OTG_HC_PID_SETUP 3

	/** Number of periodic transactions per (micro)frame */
	unsigned multi_count: 2;

	/** @name Transfer State */
	/** @{ */

	/**
	 * In Buffer DMA mode this buffer will be used
	 * if xfer_buff is not DWORD-aligned.
	 */
	dwc_dma_t align_buff;
	/** Pointer to the current transfer buffer position. */
	uint8_t *xfer_buff;
	/** Total number of bytes to transfer. */
	uint32_t xfer_len;
	/** Number of bytes transferred so far. */
	uint32_t xfer_count;
	/** Packet count at start of transfer.*/
	uint16_t start_pkt_count;

	/**
	 * Flag to indicate whether the transfer has been started.
	 * Set to 1 if it has been started, 0 otherwise.
	 */
	uint8_t xfer_started;

	/**
	 * Set to 1 to indicate that a PING request should be issued on
	 * this channel. If 0, process normally.
	 */
	uint8_t do_ping;

	/**
	 * Set to 1 to indicate that the error count for this
	 * transaction is non-zero. Set to 0 if the error count is 0.
	 */
```

```
uint8_t error_state;

/**
 * Set to 1 to indicate that this channel should be halted
 * the next time a request is queued for the channel.
 * This is necessary in slave mode if no request queue space
 * is available when an attempt is made to halt the channel.
 */
uint8_t halt_on_queue;

/**
 * Set to 1 if the host channel has been halted, but the
 * core is not finished flushing queued requests. Otherwise 0.
 */
uint8_t halt_pending;

/**
 * Reason for halting the host channel.
 */
dwc_otg_halt_status_e halt_status;

/*
 * Split settings for the host channel.
 */
uint8_t do_split;          /**< Enable split for the channel */
uint8_t complete_split;    /**< Enable complete split */
uint8_t hub_addr;          /**< Address of high speed hub */

uint8_t port_addr;         /**< Port of the low/full speed device */
/** Split transaction position
 * One of the following values:
 *    - DWC_HCSPLIT_XACTPOS_MID
 *    - DWC_HCSPLIT_XACTPOS_BEGIN
 *    - DWC_HCSPLIT_XACTPOS_END
 *    - DWC_HCSPLIT_XACTPOS_ALL */
uint8_t xact_pos;

/** Set when the host channel does a short read. */
uint8_t short_read;

/**
 * Number of requests issued for this channel since it was assigned to
 * the current transfer (not counting PINGs).
 */
uint8_t requests;

/**
 * Queue Head for the transfer being processed by this channel.
 */
struct dwc_otg_qh *qh;

/** @} */

/** Entry in list of host channels. */
```

```
        DWC_CIRCLEQ_ENTRY(dwc_hc)hc_list_entry;

        /** @name Descriptor DMA support */
        /** @{ */

        /** Number of Transfer Descriptors. */
        uint8_t ntd;

        /** Descriptor List physical address. */
        dwc_dma_t desc_list_addr;

        /** Scheduling micro-frame bitmap. */
        uint8_t schinfo;

        /** @} */
} dwc_hc_t;
```

3.2.7 DMA Descriptor Structure

The following structure represents bit fields for the following Scatter/Gather DMA Descriptors in device mode:

- Non ISO OUT/IN
- ISO OUT
- ISO IN

```
/**
 * This union represents the bit fields in the DMA Descriptor
 * status quadlet for device. Read the quadlet into the <i>d32</i>
 * member then set/clear the bits using the <i>b</i>it, <i>b_iso_out</i>
 * and <i>b_iso_in</i> elements.
 */
typedef union dev_dma_desc_sts

{
            /** raw register data */
            uint32_t d32;
            /** quadlet bits */
            struct {
            /** Received number of bytes */
            unsigned bytes : 16;

            unsigned reserved16_22 : 7;
            /** Multiple Transfer - only for OUT EPs */
            unsigned mtrf : 1;
            /** Setup Packet received - only for OUT EPs */
            unsigned sr : 1;
            /** Interrupt On Complete */
            unsigned ioc : 1;
            /** Short Packet */
            unsigned sp : 1;
            /** Last */
            unsigned l : 1;
            /** Receive Status */
```

```
            unsigned sts : 2;
            /** Buffer Status */
            unsigned bs : 2;
            } b;

#ifdef DWC_EN_ISOC
            /** iso out quadlet bits */
            struct {
            /** Received number of bytes */
            unsigned rxbytes : 11;

            unsigned reserved11 : 1;
            /** Frame Number */
            unsigned framenum : 11;
            /** Received ISO Data PID */
            unsigned pid : 2;
            /** Interrupt On Complete */
            unsigned ioc : 1;
            /** Short Packet */
            unsigned sp : 1;
            /** Last */
            unsigned l : 1;
            /** Receive Status */
            unsigned rxsts : 2;
            /** Buffer Status */
            unsigned bs : 2;
            } b_iso_out;

            /** iso in quadlet bits */
            struct {
            /** Transmited number of bytes */
            unsigned txbytes : 12;
            /** Frame Number */
            unsigned framenum : 11;
            /** Transmited ISO Data PID */
            unsigned pid : 2;
            /** Interrupt On Complete */
            unsigned ioc : 1;
            /** Short Packet */
            unsigned sp : 1;
            /** Last */
            unsigned l : 1;
            /** Transmit Status */
            unsigned txsts : 2;
            /** Buffer Status */
            unsigned bs : 2;
            } b_iso_in;
#endif //DWC_EN_ISOC
} dev_dma_desc_sts_t;

/**
 * Device DMA Descriptor structure
 *
 * DMA Descriptor structure contains two quadlets:
```

```
 * Status quadlet and Data buffer pointer.
 */
typedef struct dwc_otg_dev_dma_desc
{
      /** DMA Descriptor status quadlet */
      dev_dma_desc_sts_t      status;
      /** DMA Descriptor data buffer pointer */
      uint32_t                buf;
}     dwc_otg_dev_dma_desc_t;
```

The following structure represents bit fields for Scatter/Gather DMA Descriptors in host mode.

```
/**
 * This union represents the bit fields in the DMA Descriptor
 * status quadlet for host mode. Read the quadlet into the <i>d32</i>
 * member then set/clear the bits using the <i>b</i>it elements.
 */
typedef union host_dma_desc_sts
{
      /** raw register data */
      uint32_t d32;
      /** quadlet bits */

      /* for non-isochronous */
      struct {
            /** Number of bytes */
            unsigned n_bytes : 17;
            /**
 * QTD offset to jump when Short Packet
 * received - only for IN EPs
 */
            unsigned qtd_offset : 6;
            /**
 * Set to request the core to jump to alternate QTD
 * if Short Packet received - only for IN EPs
 */
            unsigned a_qtd : 1;
            /**
 * Setup Packet bit. When set indicates that buffer
 * contains setup packet - only for OUT EPs
 */
            unsigned sup : 1;
            /** Interrupt On Complete */
            unsigned ioc : 1;
            /** End of List */
            unsigned eol : 1;
            unsigned reserved27 : 1;
            /** Rx/Tx Status */
            unsigned sts : 2;
      #define DMA_DESC_STS_PKTERR1
            unsigned reserved30 : 1;
            /** Active Bit */
            unsigned a : 1;
      } b;
```

```
        /* for isochronous */
        struct {
              /** Number of bytes */
              unsigned n_bytes : 12;
              unsigned reserved12_24 : 13;
              /** Interrupt On Complete */
              unsigned ioc : 1;
              unsigned reserved26_27 : 2;
              /** Rx/Tx Status */
              unsigned sts : 2;
              unsigned reserved30 : 1;
              /** Active Bit */
              unsigned a : 1;
        } b_isoc;
} host_dma_desc_sts_t;

/**
 * Host-mode DMA Descriptor structure
 *
 * DMA Descriptor structure contains two quadlets:
 * Status quadlet and Data buffer pointer.
 */
typedef struct dwc_otg_host_dma_desc
{
        /** DMA Descriptor status quadlet */
        host_dma_desc_sts_tstatus;
        /** DMA Descriptor data buffer pointer */
        uint32_tbuf;
} dwc_otg_host_dma_desc_t;
```

3.3 Core Interface Layer Initialization

Sections 3.3.1–3.3.2 describe the CIL functions that support initialization of the CIL Driver component and the DWC_otg controller.

3.3.1 The dwc_otg_cil_init Function

The dwc_otg_cil_init function is called to initialize the DWC_otg CSR data structures. The register addresses in the device and host structures are initialized from the base address supplied by the caller. The calling function must make OS calls to get the base address of the DWC_otg controller registers. The core_params argument holds the parameters that specify how the core shall be configured.

```
dwc_otg_core_if_t *dwc_otg_cil_init(const uint32 *reg_base_addr)
```

3.3.2 The dwc_otg_core_init Function

The dwc_otg_core_init function initializes the DWC_otg controller registers and prepares the core for Device mode or Host mode operation.

```
void dwc_otg_core_init(dwc_otg_core_if_t *otg_core_if)
```

Sections 3.3.2.1–3.3.2.3 describe the following initialization sequences:

- “Host and Device Initialization” on page 48
- “Device Initialization” on page 49

❖ "Host Initialization" on page 50

3.3.2.1 Host and Device Initialization

The The dwc_otg_core_init Function function performs the following steps to initialize DWC_otg Core Global registers that are used in both Host and Device modes:

1. Reads the GHWCFG1/2/3 registers to find the configuration parameters selected for the DWC_otg core.
2. Programs the following fields in the GAHBCFG register:
   - ✦ DMA Mode bit (only when the OTG_ARCHITECTURE parameter is set to Internal/External DMA)
   - ✦ AHB Burst Length field (only when the OTG_ARCHITECTURE parameter is set to Internal/External DMA)
   - ✦ Global Interrupt Mask bit = 1
   - ✦ Non-Periodic TxFIFO Empty Level, applicable only when the core is operating in Slave mode and en_multiple_tx_fifo is not set.
   - ✦ Periodic TxFIFO Empty Level, applicable only when the core is operating in Slave mode and en_multiple_tx_fifo is not set.
   - ✦ TxFIFO Empty Level, applicable only when the core is operating in Slave mode and en_multiple_tx_fifo is set.
3. Programs the following fields in GUSBCFG register:
   - ✦ HNP-Capable bit (only when the OTG_MODE parameter is set to OTG)
   - ✦ SRP-Capable bit (unless the OTG_MODE parameter is set to Device Only)
   - ✦ ULPI DDR Selection bit (only when the OTG_HSPHY_INTERFACE parameter is selected for ULPI)
   - ✦ External HS PHY or Internal FS Serial PHY Selection bit (unless "None" is selected for the OTG_FSPHY_INTERFACE parameter)
   - ✦ ULPI or UTMI+ Selection bit (unless "None" is selected for the OTG_HSPHY_INTERFACE parameter)
   - ✦ PHY Interface bit (only when the OTG_HSPHY_INTERFACE parameter is selected for UTMI+ or ULPI)
   - ✦ HS/FS Timeout Calibration field
   - ✦ USB Turnaround Time field
   - ✦ If LPM support is enabled, sets LPM Capable bit.
4. Enables the following interrupts in the GINTMSK register:
   - ✦ Mode Mismatch interrupt
   - ✦ OTG interrupt
   - ✦ RxFIFO Non-Empty interrupt
   - ✦ Resume/Remote Wakeup Detected interrupt
   - ✦ Connector ID Status Change interrupt
   - ✦ Disconnect interrupt

- Suspend interrupt

5. Depending on the core's OTG capabilities, enables the following interrupts in the GINTMSK register:
   - Session Request/New Session Detected interrupt (when the core is SRP-capable)
   - LPM Transaction received interrupt (if LPM support is enabled)
6. Reads the GINTSTS.Current Mode bit to determine whether the core is in Host or Device mode and performs host or device initialization, accordingly.

3.3.2.2 Device Initialization

1. Programs the following fields in the DCFG register:
   - Device Speed
   - Nonzero-Length Status OUT Handshake
   - IN Token Out of Sequence NAK Mode (when non-periodic IN endpoints are supported)
   - Periodic Frame Interval (when periodic endpoints are supported)
   - Descriptor DMA Enable field (If DMA and Dedicated TX FIFO modes are used). Set up the Data FIFO RAM for each of the FIFOs (only if dynamic FIFO sizing is enabled):
   - Programs the GRXFSIZ register to be able to receive control OUT data and SETUP data. At a minimum, this must equal 1 max packet size of control endpoint 0 + 1 DWORD (for the status of the control OUT data packet) + 10 DWORDs (for SETUP packets).
   - Programs the GNPTXFSIZ register to be able to transmit control IN data. At a minimum, this must equal 1 max packet size of control endpoint 0.
   - Program the DPTXFSIZ*n* registers to select the start address of each of the device periodic transmit FIFOs (only when en_multiple_fifo_en is not set).
   - Program the DIEPTXF*n* registers to select the start address of each of the device transmit FIFOs (only when en_multiple_fifo_en is set).
2. Enable the following interrupts in the GINTMSK register:
   - Early Suspend interrupt
   - USB Reset interrupt
   - Enumeration Done interrupt
   - End of Periodic Frame interrupt
   - IN Token Received interrupt
   - Endpoint Mismatch interrupt when en_multiple_fifo_en is not set.
   - IN Endpoints interrupt
   - OUT Endpoints interrupt
   - Incomplete Isochronous IN Transfer interrupt
   - Incomplete Isochronous OUT Transfer interrupt
3. Program the following fields in the DCFG register.
   - Ignore Frame Number for PTI mode
   - Global Continue on BNA

3.3.2.3 Host Initialization

1. Enable the following interrupts in the GINTMSK register:
   - SOF interrupt (not in Descriptor DMA mode)
   - Port interrupt
   - Host Channels interrupt
2. Program the HCFG register to select Full-Speed or High-Speed host operation.
3. Set up the Data FIFO RAM for each of the FIFOs (only when dynamic FIFO sizing is enabled):
   - Program the GRXFSIZE register to select the size of the Receive FIFO.
   - Program the GNPTXFSIZE register to select the size and the start address of the Non-Periodic Transmit FIFO for non-periodic transactions.
   - Program the HPTXFSIZ register to select the size and start address of the Host Periodic Transmit FIFO for periodic transactions.
4. Set HPRT0.PrtPwr to 1'b1. This drives VBUS on the USB.

The remaining initialization occurs in the interrupt handler for the HPRT0.PrtConnDet and HPRT0.PrtEnChng interrupts.

- HPRT0.PrtConnDet is asserted when a device is connected to the host port. The interrupt handler issues a reset on the port.
- HPRT0.PrtEnChng is asserted when the host port is enabled after the reset sequence. The interrupt handler reads HPRT0.PrtSpd to get the speed of the device. When the device is operating at full speed or low speed and FS/LS Low Power mode is enabled, the HCFG.FSLSPclkSel bit is set to select the PHY frequency. When the PHY clock frequency has been changed for power savings, another reset is issued on the port.

See "Port Interrupt" for more detail on the Port Interrupt handler.

3.4 Device Operations

Sections 3.4.1–3.4.2 describe the CIL functions that support managing the DWC_otg controller in Device mode.

3.4.1 Global Device Operations

The following functions apply to the device as a whole. They are not specific to a particular endpoint.

3.4.1.1 The dwc_otg_read_setup_packet Function

The dwc_otg_read_setup_packet function reads a SETUP packet (2 DWORDs) from the RxFIFO into the destination buffer. This function is called from the RxFIFO Non-Empty (RxFLvl) Interrupt routine (see page 73) when a SETUP packet has been received in Slave mode.

```
void dwc_otg_read_setup_packet(dwc_otg_core_if_t *core_if,
                                                    const uint32_t *dest)
```

3.4.1.2 The dwc_otg_get_frame_number Function

The dwc_otg_get_frame_number function gets the current USB frame number, which is the frame number from the last SOF packet.

```
uint32_t dwc_otg_get_frame_number(dwc_otg_core_if_t *core_if)
```

3.4.1.3 The dwc_otg_wakeup Function

The dwc_otg_wakeup function starts a USB protocol session, when no session is in progress. When a session is already in progress, but the device is suspended, remote wakeup signaling is started.

```
void dwc_otg_wakeup(dwc_otg_core_if_t *core_if)
```

3.4.1.4 The dwc_otg_dump_dev_registers Function

The dwc_otg_dump_dev_registers function reads and displays the device registers.

```
void dwc_otg_dump_dev_registers(dwc_otg_core_if_t *core_if)
```

3.4.2 Endpoint Operations

The following functions are used to manage endpoint operations.

3.4.2.1 The dwc_otg_ep0_activate Function

The dwc_otg_ep0_activate function enables endpoint 0 OUT to receive SETUP packets and configures endpoint 0 IN for transmitting packets. It is normally called when the Enumeration Done interrupt occurs.

```
void dwc_otg_ep0_activate(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

- Read the DSTS register to determine the device's enumeration speed.
- Program the Max Packet Size field in the DIEPCTL0 register. This step configures control endpoint 0. The max packet size for a control endpoint depends on the device's enumeration speed.
- Program the DOEPCTL0 register to enable control OUT endpoint 0; to receive a SETUP packet: DOEPCTL0.Endpoint Enable = 1.
- Clear the global non-periodic IN NAK.

3.4.2.2 The dwc_otg_ep0_start_transfer Function

The dwc_otg_ep0_start_transfer function sets up an endpoint 0 control transfer request and starts the transfer.

```
void dwc_otg_ep0_start_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

- For IN transfers, packets are loaded into the appropriate TxFIFO in the TxFIFO Empty ISR.
- For OUT transfers, packets are unloaded from the RxFIFO in the Rx Status Queue ISR.

IN Transfers:

1. Check for room in the Tx Status Queue. When full, the request is started later in the IN endpoint ISR.
2. Write the transfer size and packet count to the DIEPTSIZ0 register.

Note The DIEPCTL0/DOEPCTL0 registers only have one bit for the packet count.

3. If Buffer DMA is enabled, write the address of the data in the DIEPDMA0 register.
4. If Descriptor DMA is enabled, setup the Descriptor DMA, write the Descriptor address in the DIEPDMA0 register.
5. Enable the EP by setting the enable and clear NAK bits in the DIEPCTL0 register.
6. If DMA is NOT enabled, enable the nptxempty interrupt if en_multiple_tx_fifo is not set, and EP0 Tx FIFO empty interrupt if it is set, so the data will be loaded into the TxFIFO.

OUT Transfers:

1. Write the transfer size and packet count to the DOEPTSIZ0 register.
2. If Buffer DMA is enabled, write the address of the data in the DOEPDMA0 register.

3. If Descriptor DMA is enabled, setup the Descriptor DMA, write the Descriptor address into DOEPDMA0 register.
4. Enable the EP by setting the enable and clear NAK bits in the DOEPCTL0 register.

3.4.2.3 The dwc_otg_ep0_continue_transfer Function

The dwc_otg_ep0_continue_transfer function continues control IN transfers started by The dwc_otg_ep0_start_transfer Function (above), when the transfer does not fit in a single packet.

Note The DIEPCTL0/DOEPCTL0 registers only have one bit for the packet count.

```
void dwc_otg_ep0_continue_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

It does the same IN Transfer actions as shown in The dwc_otg_ep0_start_transfer Function, above.

3.4.2.4 The dwc_otg_ep_activate Function

The dwc_otg_ep_activate function activates an endpoint. The Device Endpoint Control register for the endpoint is configured as defined in "Endpoint Structure" on page 38

Note dwc_otg_ep_activate is not used for endpoint 0.

```
void dwc_otg_ep_activate(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

Program the characteristics of the required endpoint into the following fields of the DIEPCTL*n* (IN or IN/OUT endpoints) or DOEPCTL*n* (OUT or IN/OUT endpoints) register:

- Maximum Packet Size
- USB Active Endpoint = 1
- Endpoint Start Data Toggle (For interrupt and bulk endpoints)
- Endpoint Type
- Tx FIFO Number (1 if en_multiple_fifo_en is not set and assigns fifo if en_multiple_fifo_en is set)
- If performing in MPI mode, unmask these interrupts:
  - For OUT EPs
    - DOEPEACHMSK[n].SETUP = 1
    - DOEPEACHMSK[n].XferCompl = 1
    - DOEPEACHMSK[n].ahberr = 1
  - For IN EPs
    - DIEPEACHMSK[n].epdisable = 1
    - DIEPEACHMSK[n].XferCompl = 1
    - DIEPEACHMSK[n].TimeOut = 1
    - DIEPEACHMSK[n].epdisable = 1
    - DIEPEACHMSK[n]. Ahberr = 1

Note

- **When en_multiple_fifo_en is not set**, Interrupt IN endpoints can be configured as periodic or non-periodic endpoints depending on the application. The core treats an IN endpoint as a non-periodic endpoint when the DIEPCTL*n*.TxFNum field is set to 0. Otherwise, a separate Periodic FIFO must be allocated using coreConsultant. The number of this FIFO must be programmed into the DIEPCTL*n*.TxFNum field. Configuring an interrupt IN endpoint as a non-periodic endpoint saves the extra Periodic FIFO area.
- **When en_multiple_fifo_en is set for all IN endpoints**, a separate transmit FIFO must be allocated

3.4.2.5 The dwc_otg_ep_deactivate Function

The dwc_otg_ep_deactivate function deactivates an endpoint by clearing the USB Active Endpoint bit in the Device Endpoint Control register. Also, in MPI mode, this function masks the Endpoint Interrupts. Note that this function is not used for Endpoint 0. EP0 cannot be deactivated.

```
void dwc_otg_ep_deactivate(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

3.4.2.6 The dwc_otg_ep_start_transfer Function

The dwc_otg_ep_start_transfer function sets up a data transfer for an endpoint and starts the transfer.

For IN transfers, it loads a packet into the appropriate FIFO. When this is a multiple packet transfer, the additional packets are loaded in the ISR.

For OUT transfers, the packets are unloaded from the RxFIFO in the ISR.

```
void dwc_otg_ep_start_transfer(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

IN Transfers:

1. Check for room in the Tx Status Queue. When full, the request will be started later in the IN endpoint ISR.
2. Write the transfer size and packet count to the DIEPTSIZ register.
3. If DMA is enabled, write the address of the data in the DIEPDMA register.
4. If Descriptor DMA is enabled, set up Descriptor DMA, write the Descriptor address into the DIEPDMAn register.
5. Enable the endpoint by setting the Endpoint Enable and Clear NAK bits in the DIEPCTL register.
6. When DMA is not enabled, enable the nptxempty/txfempty interrupt, so the data is loaded into the TxFIFO.

OUT Transfers:

1. Writes the transfer size and packet count to the DIEPTSIZ register.
2. If DMA is enabled, write the address of the data in the DIEPDMA register.
3. If Descriptor DMA is enabled, set up Descriptor DMA, write the Descriptor address into the DOEPDMAn register.
4. Enable the endpoint by setting the Endpoint Enable and Clear NAK bits in the DIEPCTL register.

3.4.2.7 The dwc_otg_ep_start_zl_transfer Function

The dwc_otg_ep_start_zl_transfer function initiates zero-length transfer on non-control EPs for transfers with size divisible to max packet size and with usb_request's zero field is set to 1

```
void dwc_otg_ep_start_zl_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

3.4.2.8 dThe wc_otg_ep_write_packet Function

The dwc_otg_ep_write_packet function writes a packet into the TxFIFO associated with an endpoint.

- When en_multiple_tx_fifo is not set for non-periodic endpoints, the Non-Periodic TxFIFO is written.
- For periodic endpoints, the Periodic TxFIFO associated with the endpoint is written with all packets for the next (micro)frame.
- Otherwise, for all EPs the Tx FIFO associated with the endpoint is written

```
void dwc_otg_ep_write_packet(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

3.4.2.9 The dwc_otg_ep_set_stall Function

Sets the endpoint to STALL.

```
void dwc_otg_ep_set_stall(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

3.4.2.10 The dwc_otg_ep_clear_stall Function

Clears the STALL on the endpoint.

```
void dwc_otg_ep_clear_stall(dwc_otg_core_if *core_if, dwc_ep_t *ep)
```

3.4.2.11 The dwc_otg_iso_ep_start_transfer Function

Starts Isochronous transfer.

```
void dwc_otg_iso_ep_start_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

3.4.2.12 The dwc_otg_iso_ep_start_buf_transfer Function

Initializes Isochronous transfer per each data processing interval between gadget and PCD (PTI enhancement mode).

```
void dwc_otg_iso_ep_start_buf_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

3.4.2.13 The dwc_otg_iso_ep_start_ddma_transfer Function

Initializes Isochronous transfer per each data processing interval between gadget and PCD (Descriptor DMA mode).

```
void dwc_otg_iso_ep_start_ddma_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *dwc_ep)
```

3.4.2.14 The dwc_otg_iso_ep_start_frm_transfer Function

Initializes Isochronous transfer per each (micro)frame.

```
void dwc_otg_iso_ep_start_buf_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

3.4.2.15 The dwc_otg_iso_ep_stop_transfer Function

Stops Isochronous transfer.

```
void dwc_otg_iso_ep_stop_transfer(dwc_otg_core_if_t *core_if, dwc_ep_t *ep)
```

3.5 Host Operations

Sections 3.5.1–3.5.2 describe the externally visible CIL functions that support managing the DWC_otg controller in Host mode.

3.5.1 Global Host Operations

The following function applies to the host as a whole. It is not specific to a particular host channel.

3.5.1.1 The dwc_otg_dump_host_registers Function

The dwc_otg_dump_host_registers function reads and displays the host registers.

```
void dwc_otg_dump_host_registers(dwc_otg_core_if_t *core_if)
```

3.5.2 Host Channel Operations

The following functions manage host channel operations.

3.5.2.1 The dwc_otg_hc_init Function

The dwc_otg_hc_init function prepares a host channel for transferring packets to or from a specific endpoint.

```
void dwc_otg_hc_init(dwc_otg_core_if *core_if, dwc_hc_t *hc)
```

1. Program the HAINTMSK register to enable the selected channel's interrupts.
2. Program the HCINTMSK register to enable the transaction-related interrupts of interest in the Host Channel Interrupt register.
3. Program the selected channel's HCCHAR*n* register with the device's endpoint characteristics, such as type, speed, direction, and so forth.
4. For split transactions, program the HCSPLT*n* register of the selected channel with the split characteristics (Do Complete Split, Transaction Position, Hub Address, Port Address).

3.5.2.2 The dwc_otg_hc_halt Function

The dwc_otg_hc_halt function attempts to halt a host channel. It should only be called in Slave mode or to abort a transfer in either Slave mode or DMA mode. Under normal circumstances in DMA mode, the controller halts the channel when the transfer is complete or a condition occurs that requires application intervention.

```
void dwc_otg_hc_halt(dwc_otg_core_if *core_if, dwc_hc_t *hc
                     int halt_status)
```

In Slave mode, this function checks for a free request queue entry, then sets the Channel Enable and Channel Disable bits of the Host Channel Characteristics register (HCHAR*n*) of the specified channel to initiate the halt. When there is no free request queue entry, this function sets only the Channel Disable bit of the HCCHAR*n* register to flush requests for this channel. In the latter case, this function sets a flag to indicate that the host channel must be halted when a request queue slot is open.

In DMA mode, this function always sets the Channel Enable and Channel Disable bits of the HCCHAR*n* register. The controller ensures there is space in the request queue before submitting the halt request.

Some time may elapse before the core flushes any posted requests for this host channel and halts. The Channel Halted interrupt handler completes the deactivation of the host channel. See "Channel Halted" for a description of what happens when the Channel Halted interrupt occurs.

If the host channel is not active when this function is called, no action is taken.

3.5.2.3 The dwc_otg_hc_start_transfer Function

The dwc_otg_hc_start_transfer function sets up a data transfer for a host channel and starts the transfer. It may be called in either Slave mode or Buffer DMA mode. In Slave mode, the caller must ensure that there is sufficient space in the Request Queue and TxFIFO.

```
void dwc_otg_hc_start_transfer(dwc_otg_core_if *core_if, dwc_hc_t *hc)
```

For an OUT transfer in Slave mode, it loads a data packet into the appropriate FIFO. When necessary, additional data packets are loaded in the Host ISR.

For an IN transfer in Slave mode, a data packet is requested. The data packets are unloaded from the RxFIFO in the Host ISR. When necessary, additional packets are requested in the Host ISR.

For a PING transfer in Slave mode, the Do Ping bit is set in the HCTSIZ register, along with a packet count of 1, and the channel is enabled. This causes a single PING transaction to occur. Other fields in HCTSIZ are simply set to 0, because no data transfer occurs in this case.

For a PING transfer in Buffer DMA mode, the HCTSIZ register is initialized with all the information required to perform the subsequent data transfer. In addition, the Do Ping bit is set in the HCTSIZ register. In this case, the controller performs the entire PING protocol, then starts the data transfer.

For starting transfers other than a PING transfer in Slave mode, this function performs the following steps:

1. For a periodic transfer, programs the Odd Frame bit of the HCCHAR*n* register to select the (micro)frame for the transfer.
2. Programs the HCTSIZ*n* register of the selected channel with the total transfer size in bytes and the expected number of packets, including short packets. It also programs the PID field with the initial data PID (to be used on the first OUT transaction or expected from the first IN transaction).
3. For split transactions, sets the Split Enable bit of the selected channel's HCSPLT*n* register.
4. Programs the HCDMA*n* register of the selected channel with the buffer start address (applicable only for the Internal DMA mode).
5. Sets the Channel Enable bit of HCCHAR*n* to 1. For an IN transfer, this allows the controller to start the USB transaction.
6. For an OUT transfer in Slave mode, loads one data packet into the appropriate TxFIFO. After this data packet is loaded, the controller can start the USB transaction.

3.5.2.4 The dwc_otg_hc_continue_transfer Function

The dwc_otg_hc_continue_transfer function continues a data transfer that was started by a previous call to The dwc_otg_hc_start_transfer Function. The caller must ensure there is sufficient space in the request queue and TxFIFO. This function must only be called in Slave mode. In DMA mode, the controller acts autonomously to complete transfers programmed to a host channel.

```
int dwc_otg_hc_continue_transfer(dwc_otg_core_if_t *_core_if, dwc_hc_t *_hc)
```

For an OUT transfer, a new data packet is loaded into the appropriate FIFO when any data remains to be queued. For an IN transfer, another data packet is always requested. For the SETUP phase of a control transfer, this function does nothing.

This function returns 1 when a new request is queued, 0 when no more requests are required for this transfer.

3.5.2.5 The dwc_otg_hc_start_transfer_ddma Function

The dwc_otg_hc_start_transfer_ddma function does the setup for a data transfer for a host channel in Descriptor DMA mode and starts the transfer. The descriptor list for this channel must be initialized before

calling this function. For Interrupt and Isochronous transfers, the FrameList must also be initialized before starting the transfer. This function must be called only in Descriptor DMA mode.

```
void dwc_otg_hc_start_transfer_ddma(dwc_otg_core_if *core_if, dwc_hc_t *hc)
```

For starting transfers in Descriptor DMA mode, this function performs the following steps:

1. Programs the PID field of the HCTSIZn register for the selected channel with the initial data PID (to be used on the first OUT transaction or expected from the first IN transaction).
2. For a PING transfer, sets the Do Ping bit in the HCTSIZ register. In this case, the controller performs the entire PING protocol, then starts the data transfer.
3. Programs the NTD field of the HCTSIZn register with the number of transfer descriptors. For Bulk, Control and Interrupt transfers the actual number of descriptors is programmed. For Isochronous transfers, NTD is set to maximum allowed value, which depends on the speed of the connected device.
4. Programs the SCHED_INFO field of the HCTSIZn register with the scheduling bitmap for Interrupt and Isochronous transfers. For full-speed, the scheduling bitmap is always 0xff
5. Programs the HCDMAn register of the selected channel with the descriptor list starting address.
6. Sets the "Channel Enable" bit of HCCHARn to 1.

3.5.2.6 The dwc_otg_hc_write_packet Function

The dwc_otg_hc_write_packet function writes a packet into the TxFIFO associated with the Host Channel. For a channel associated with a non-periodic endpoint, this function writes to the Non-Periodic TxFIFO. For a channel associated with a periodic endpoint, this function writes to the Periodic TxFIFO. This function should only be called in Slave mode.

```
void dwc_otg_hc_write_packet(dwc_otg_core_if *core_if, dwc_hc_t *hc)
```

3.5.2.7 The dwc_otg_hc_cleanup Function

The dwc_otg_hc_cleanup function clears the transfer state for a host channel. It is called after a transfer is done and the host channel is being released.

```
void dwc_otg_hc_cleanup(dwc_otg_core_if_t *_core_if, dwc_hc_t *_hc)
```

Clear channel interrupt enables and any unhandled channel interrupt conditions.

3.6 Common Operations

The following operations describe the CIL functions that support managing the DWC_otg controller in either Device or Host mode.

3.6.1 The dwc_otg_mode Function

The dwc_otg_mode function returns the mode of the operation (Host or Device).

```
//returns 0 - Device Mode, 1 - Host Mode
uint32_t dwc_otg_mode(dwc_otg_core_if_t *core_if)

uint8_t dwc_otg_is_device_mode(dwc_otg_core_if_t *core_if)
uint8_t dwc_otg_is_host_mode(dwc_otg_core_if_t *core_if)
```

3.6.2 The dwc_otg_read_packet Function

The dwc_otg_read_packet function reads a packet from the RxFIFO into the destination buffer.

To read SETUP data, use The dwc_otg_read_setup_packet Function.

```
dwc_otg_read_packet(dwc_otg_core_if_t *core_if,
                    const uint8_t *dest, const uint16_t bytes)
```

3.6.3 The dwc_otg_dump_global_registers Function

The dwc_otg_dump_global_registers function reads and displays the global registers.

```
void dwc_otg_dump_global_registers(dwc_otg_core_if_t *core_if)
```

3.6.4 The dwc_otg_enable_common_interrupts Function

The dwc_otg_enable_common_interrupts function initializes the common interrupts, used in both Device and Host modes.

```
void dwc_otg_enable_common_interrupts(dwc_otg_core_if_t *_core_if)
```

3.6.5 The dwc_otg_enable_device_interrupts Function

The dwc_otg_enable_device_interrupts function enables the Device mode interrupts.

```
void dwc_otg_enable_device_interrupts(dwc_otg_core_if_t *_core_if)
```

3.6.6 The dwc_otg_enable_global_interrupts Function

The dwc_otg_enable_global_interrupts function enables the controller's Global interrupt in the Core AHB Configuration register (GAHBCFG).

```
void dwc_otg_enable_global_interrupts(
                            dwc_otg_core_if_t *_core_if )
```

3.6.7 The dwc_otg_disable_global_interrupts Function

The dwc_otg_disable_global_interrupts function disables the controller's Global interrupt in the Core AHB Configuration register (GAHBCFG).

```
void dwc_otg_disable_global_interrupts(
                              dwc_otg_core_if_t *_core_if )
```

3.6.8 The dwc_otg_disable_host_interrupts Function

The dwc_otg_disable_host_interrupts function disables the Host mode interrupts.

```
void dwc_otg_enable_host_interrupts(dwc_otg_core_if_t *_core_if)
```

3.7 Register Access

Sections 3.7.1–3.7.8 describe the CIL functions that support access to the DWC_otg registers. These functions may be used in either Device or Host mode.

3.7.1 The dwc_otg_read_core_intr Function

The dwc_otg_read_core_intr function returns the contents of the Core Interrupt register (GINTSTS), masked by the contents of the Core Interrupt Mask register (GINTMSK).

```
uint32_t dwc_otg_read_core_intr(dwc_otg_core_if_t *core_if)
```

3.7.2 The dwc_otg_read_otg_intr Function

The dwc_otg_read_otg_intr function reads the OTG Interrupt register (GOTGINT).

```
uint32_t dwc_otg_read_otg_intr(dwc_otg_core_if_t *core_if)
```

3.7.3 The dwc_otg_read_dev_all_in_ep_intr Function

The dwc_otg_read_dev_all_in_ep_intr function reads the Device All Endpoints Interrupt register (DAINT) and returns the IN endpoint interrupt bits masked by the contents of the Device All Endpoints Interrupt Mask register (DAINTMSK) for non-MPI mode, or Device Each Endpoints Interrupt Mask register (DEACHINTMSK) for MPI mode.

```
dwc_otg_read_dev_all_in_ep_intr(dwc_otg_core_if_t *core_if)
```

3.7.4 The dwc_otg_read_dev_all_out_ep_intr Function

The dwc_otg_read_dev_all_out_ep_intr function reads the Device All Endpoints Interrupt register (DAINT) and returns the OUT endpoint interrupt bits masked by the contents of the Device All Endpoints Interrupt Mask register (DAINTMSK) for non-MPI mode, or Device Each Endpoints Interrupt Mask register (DEACHINTMSK) for MPI mode.

```
dwc_otg_read_dev_all_out_ep_intr(dwc_otg_core_if_t *core_if)
```

3.7.5 The dwc_otg_read_dev_in_ep_intr Function

The dwc_otg_read_dev_in_ep_intr function returns the contents of the Device IN Endpoint Common Interrupt register (DIEPINT*n*) masked by the contents of the Device In Endpoint-*n* Interrupt Mask register (DIEPMSK) for non-MPI mode, or Device Each In Endpoints Interrupt Mask register (DIEPEACHINTMSKn) for MPI mode.

```
uint32_t dwc_otg_read_dev_in_ep_intr(dwc_otg_core_if_t *core_if,
                                     dwc_ep_t *ep)
```

3.7.6 The dwc_otg_read_dev_out_ep_intr Function

The dwc_otg_read_dev_out_ep_intr function returns the contents of the Device OUT Endpoint Common Interrupt register (DOEPINT*n*) masked by the contents of the Device Out Endpoint-*n* Interrupt Mask register (DOEPMSK) for non-MPI mode, or Device Each Out Endpoints Interrupt Mask register (DOEPEACHINTMSKn) for MPI mode. For PTI mode, this function also returns PktDrpSts bit value.

```
uint32_t dwc_otg_read_dev_out_ep_intr(dwc_otg_core_if_t *core_if,
                                      dwc_ep_t *ep)
```

3.7.7 The dwc_otg_read_host_all_channels_intr Function

The dwc_otg_read_host_all_channels_intr function reads the Host All Channels Interrupt register (HAINT).

```
uint32_t dwc_otg_read_host_all_channels_intr(dwc_otg_core_if_t *core_if,)
```

3.7.8 The dwc_otg_read_host_channel_intr Function

The dwc_otg_read_host_channel_intr function reads the Host Channel Interrupt register for a given host channel (HCINT*n*).

```
uint32_t dwc_otg_read_host_channel_intr(dwc_otg_core_if_t *core_if,
                                                          dwc_hc_t *hc)
```

3.8 Common Interrupt Service Routine

Sections 3.8.1–3.8.7 describe the operations performed for each interrupt the CIL handles.

3.8.1 Mode Mismatch Interrupt

The dwc_otg_mode_mismatch_intr function logs a debug message:

"WARNING: Mode Mismatch Interrupt: currently in <device/host> mode."

```
int32_t dwc_otg_mode_mismatch_intr(dwc_otg_core_if_t *core_if)
```

3.8.2 OTG Interrupt

The dwc_otg_handle_otg_intr function services OTG interrupts. It reads the OTG Interrupt register (GOTGINT) to determine which interrupt has occurred.

```
int32_t dwc_otg_handle_otg_intr(dwc_otg_core_if_t *core_if)
```

3.8.3 USB Suspend Interrupt

The dwc_otg_handle_usb_suspend_intr function indicate that the Suspend state has been detected on the USB. For HNP the USB Suspend interrupt signals the change from "a_peripherial" to "a_host". When power management is enabled, the core is put in Low Power mode.

```
int32_t dwc_otg_handle_usb_suspend_intr(dwc_otg_core_if_t *_core_if)
```

3.8.4 Connector ID Status Change Interrupt

The dwc_otg_handle_conn_id_status_change_intr function indicates that the cable has been inserted or removed from the connector. For OTG this may be an A-cable or a B-connector. The OTG Interrupt Register (GOTCTL) is read to determine whether this is a Device to Host mode transition or a Host mode to Device mode transition.

```
int32_t dwc_otg_handle_conn_id_status_change_intr(
                                                  dwc_otg_core_if_t *core_if)
```

3.8.5 New Session Detected Interrupt

The dwc_otg_handle_session_req_intr function is asserted when the core has detected the initiation of Session Request Protocol on the USB. This indicates that a device is initiating the Session Request Protocol to request the host to turn on bus power so a new session can begin. The interrupt handler responds by turning on bus power. When the DWC_otg controller is in Low Power mode, the interrupt handler brings the controller out of Low Power mode before turning on bus power.

```
int32_t dwc_otg_handle_session_req_intr(dwc_otg_core_if_t *core_if)
```

3.8.6 Disconnect Detected Interrupt

The dwc_otg_hcd_handle_disconnect_intr function indicates that a device has been disconnected from the root port. The driver sets its internal Connect Status Change flag and clears the Disconnect Detected bit.

```
int32_t dwc_otg_hcd_handle_disconnect_intr(dwc_otg_core_if_t *core_if)
```

If a device has been disconnected, the host software layers above the Host Controller Driver perform the appropriate cleanup actions.

The hub driver is notified when it reads the root hub's status via the Status Change endpoint. The hub driver issues a GetPortStatus command to the root hub to determine that a connect status change has

occurred and to find the current port connect status. The driver's Connect Status Change flag is cleared when the hub driver issues a ClearPortFeature(C_PORT_CONNECTION) command to the host port.

This interrupt also indicates role switching during the Host Negotiation Protocol.

3.8.7 Remote Wakeup Detected Interrupt

The dwc_otg_handle_wakeup_detected_intr function indicates that the DWC_otg controller has detected a remote wakeup sequence. When the DWC_otg controller is in Low Power mode, the interrupt handler must bring the controller out of Low Power mode. The controller automatically begins resume signaling. The interrupt handler schedules a time to stop resume signaling.

```
int32_t dwc_otg_handle_wakeup_detected_intr(dwc_otg_core_if_t *core_if)
```

3.8.8 LPM Transaction Received Interrupt

This interrupt indicates that host has sent LPM Extended transaction and device has responded with an ACK handshake. For the host mode DWC_otg core issues this interrupt only for local device.

```
int32_t dwc_otg_handle_lpm_intr(dwc_otg_core_if_t *core_if)
```

4 Peripheral Controller Driver

4.1 Peripheral Controller Driver Overview

The Peripheral Controller Driver (PCD) is responsible for translating requests from the Function Driver into the appropriate actions on the DWC_otg controller. The PCD isolates the Function Driver from the specifics of the controller by providing an API to the Function Driver. This API may vary between operating systems, but it remains constant within a given OS. "Function Driver Interface" describes this API for supported operating systems.

A USB device responds to commands issued from the host. "Standard USB Command Processing" on page 72 describes the handling of standard USB commands within the DWC_otg software environment.

An important PCD function is managing interrupts generated by the DWC_otg controller. The behavior of each DWC_otg Device mode interrupt is described in "Function Driver Interface" on page 63

4.2 Function Driver Interface

This section describes the API that the PCD's OS Wrapper layer presents to the Function Driver, which is operating system-dependent. The PCD OS Wrapper layer acts as an intermediate layer between the Function Driver and the PCD core, translating Function Driver requests into PCD core function calls. The OS Wrapper layer is currently implemented only for Linux.

4.2.1 Linux Gadget API

The Peripheral Controller Driver for Linux implements the Gadget API, so that the existing gadget drivers can be used (Gadget Driver is the Linux term for a function driver.)

The Linux Gadget API is defined in the header file, linux/usb_gadget.h. The following data structures define the functions implemented in the PCD to provide the interface. The USB Endpoint Operations API is defined in the usb_ep_ops structure, and the USB Controller API is defined in the usb_gadget_ops structure.

```
/* endpoint-specific parts of the api to the usb controller
 * hardware.  Unlike the urb model, (de)multiplexing layers are
 * not required.  (so this api could slash overhead if used on the
 * host side...)
 * note that device side usb controllers commonly differ in how
 * many endpoints they support, as well as their capabilities.
 */
struct usb_ep_ops {
      int (*enable) (struct usb_ep *ep,
```

```
            const struct usb_endpoint_descriptor *desc);
      int (*disable) (struct usb_ep *ep);

      struct usb_request *(*alloc_request) (struct usb_ep *ep,
            gpf gfp_flags);
      void (*free_request) (struct usb_ep *ep,
            struct usb_request *req);

      void *(*alloc_buffer) (struct usb_ep *ep, unsigned bytes,
            dma_addr_t *dma, gpf gfp_flags);
      void (*free_buffer) (struct usb_ep *ep, void *buf,
            dma_addr_t dma,
            unsigned bytes);
      // NOTE:  on 2.6, drivers may also use dma_map() and
      // dma_sync_single_*() to directly manage dma overhead.

      int (*queue) (struct usb_ep *ep, struct usb_request *req,
            gpf gfp_flags);
      int (*dequeue) (struct usb_ep *ep, struct usb_request *req);

      int (*set_halt) (struct usb_ep *ep, int value);
      int (*fifo_status) (struct usb_ep *ep);
      void (*fifo_flush) (struct usb_ep *ep);
};

/* The rest of the api to the controller hardware: device
 * operations, which don't involve endpoints (or i/o).
 */
struct usb_gadget_ops {
      int   (*get_frame)(struct usb_gadget *);
      int   (*wakeup)(struct usb_gadget *);
      int   (*set_selfpowered) (struct usb_gadget *,
                        int is_selfpowered);
      int   (*vbus_session) (struct usb_gadget *, int is_active);
      int   (*vbus_draw) (struct usb_gadget *, unsigned mA);
      int   (*pullup) (struct usb_gadget *, int is_on);
      int   (*ioctl)(struct usb_gadget *,
                        unsigned code, unsigned long param);
      int   (*lpm_support) (struct usb_gadget *);
};
```

The gadget API has been extended to implement the gadget/PCD interface for isochronous transfers differently than other types of transfer. This includes defining several new types: usb_iso_request and usb_isoc_ep_ops. The gadget API has also been extended with the lpm_support function, adding LPM support.

The usb_gadget_iso_packet_descriptor structure contains information about each ISO packet after transmission—if data was transmitted, actual size of transfer, and so forth.

```
struct usb_gadget_iso_packet_descriptor
{
      unsigned int                  offset;
      unsigned int                  length; /* expected length  */
```

```
        unsigned int                    actual_length;
        unsigned int                    status;
};
```

usb_iso_request struct represents an isochronous transfer request information. Once an isochronous transfer is started, it does not stop until the gadget driver stops it or the device is disconnected. For isochronous transfers, double buffering is used: while one buffer is being transferred on the USB, the other's data is being exchanged between the gadget driver and PCD. Several parameters for isochronous transfer are set in this structure:

- **buf_proc_intrvl**: Time interval (number of frames) for data exchange, i.e., interval for calling process_buffer. This variable should be divisible to bInterval (in terms of number of frames, not power of 2).
- **data_per_frame**: size of data to be transmitted in regular (micro)frame
- **data_pattern_frame**: size of data to be transmitted in sync (micro)frame, frame number to start transfer.
- **sync_frame**: sync frame number
- **start_frame**: frame number from which transfer should be started, effective if flags set to 0
- **flags**:
  - When set to 0, PCD should start transfer from frame number mentioned in start_frame,
  - When not 0, and when flag REQ_ISO_ASAP is set, PCD should start transfer as soon as possible not depending on frame number
- **process_buffer**: performs data exchange between gadget and PCD, called periodically when buf_proc_intrvl is expired
- **buf0, buf1**: 2 data buffers
- **iso_packet_desc0, iso_packet_desc1**: two isochronous packet descriptor chains, set by the PCD after transfer is performed, status field is set to:
  - 0 when an isochronous packet was transferred normally
  - ENODATA when data was not transferred for any reason

```
struct usb_iso_request {
        void                    *buf0;
        void                    *buf1;
        dma_addr_t              dma0;
        dma_addr_t              dma1;

        uint32_t                buf_proc_intrvl;

        unsigned                no_interrupt:1;
        unsigned                zero:1;
        unsigned                short_not_ok:1;

        uint32_t                sync_frame;
        uint32_t                data_per_frame;
        uint32_t                data_pattern_frame;
        uint32_t                start_frame;
        uint32_t                flags;
```

```
      void              (*process_buffer)(struct usb_ep*,
                                struct usb_iso_request*);

      void              *context;

      int               status;

      struct usb_gadget_iso_packet_descriptor *iso_packet_desc0;
      struct usb_gadget_iso_packet_descriptor *iso_packet_desc1;
};
```

Isoc Specific Endpoints operations are defined in usb_isoc_ep_ops. These functions start and stop isochronous transfers. Once stared by calling iso_ep_start(), isochronous transfers will not stop until iso_ep_stop() function is called or device is disconnected.

```
struct usb_isoc_ep_ops {
      struct            usb_ep_ops ep_ops;

      int               (*iso_ep_start)(struct usb_ep*, struct usb_iso_request*, int);
      int               (*iso_ep_stop)(struct usb_ep*, struct usb_iso_request*);

      struct
      usb_iso_request*  (*alloc_iso_request)(struct usb_ep* ep, int packets,int
gfp_flags);
      void              (*free_iso_request)(struct usb_ep* ep, struct usb_iso_request
*req);
};
```

4.2.1.1 USB Endpoint Operations

Sections 4.2.1.1.1–4.2.1.1.13 describe the behavior of the Gadget API endpoint operations implemented in the DWC_otg driver software. Detailed descriptions of the generic behavior of each of these functions can be found in the Linux header file: include/linux/usb_gadget.h.

The Gadget API provides wrapper functions for each function pointer defined in usb_ep_ops. The Gadget Driver calls the wrapper function, which then calls the underlying PCD function. Sections 4.2.1.1.1–4.2.1.1.13 are named according to the wrapper functions. Within each section, the corresponding DWC_otg PCD function name is specified.

API functions not described below are not implemented.

4.2.1.1.1 The usb_ep_enable Function

The Gadget Driver calls the usb_ep_enable function for each endpoint to be configured for the current configuration (SET_CONFIGURATION).

This function

1. initializes the dwc_otg_ep_t data structure (in dwc_otg_pcd.h)
2. allocates DMA Descriptors
3. calls The dwc_otg_ep_activate Function.

```
int dwc_otg_pcd_ep_enable(struct usb_ep *ep,
                          const struct usb_endpoint_descriptor *desc)
```

4.2.1.1.2 The usb_ep_disable Function

The usb_ep_disable function is called when an endpoint is disabled due to a disconnect or change in configuration. Any pending requests terminate with a status of –ESHUTDOWN.

This function

1. modifies the dwc_otg_ep_t data structure for this endpoint
2. frees DMA Descriptors
3. calls The dwc_otg_ep_deactivate Function.

```
int dwc_otg_pcd_ep_disable(struct usb_ep *ep)
```

4.2.1.1.3 The usb_ep_alloc_request Function

The usb_ep_alloc_request function allocates a request object to use with the specified endpoint.

```
struct usb_request *dwc_otg_pcd_alloc_request(struct usb_ep *ep,
                                              gfp gfp_flags)
```

4.2.1.1.4 The usb_ep_free_request Function

The usb_ep_free_request function frees a request object.

```
void dwc_otg_pcd_free_request(struct usb_ep *ep,
                              struct usb_request *req)
```

4.2.1.1.5 The usb_ep_alloc_buffer Function

The usb_ep_alloc_buffer function allocates an I/O buffer for a transfer to or from the specified endpoint.

```
void *dwc_otg_pcd_alloc_buffer(struct usb_ep *ep, unsigned bytes,
                               dma_addr_t *dma, gfp gfp_flags)
```

4.2.1.1.6 The usb_ep_free_buffer Function

The usb_ep_free_buffer function frees an I/O buffer that was allocated by The usb_ep_alloc_buffer Function.

```
void dwc_otg_pcd_free_buffer(struct usb_ep *ep, void *buf,
                             dma_addr_t dma, unsigned bytes)
```

4.2.1.1.7 The usb_ep_queue Function

The usb_ep_queue function submits an I/O request to an endpoint.

```
int dwc_otg_pcd_ep_queue(struct usb_ep *ep,
                         struct usb_request *req, int gfp_flags)
```

Note

- When the request completes, the request's completion callback is called to return the request to the driver.
- Any endpoint, except control endpoints, can have multiple requests pending.
- Once submitted, a request cannot be examined or modified.
- Each request is turned into one or more packets.
- A bulk endpoint can queue any amount of data; the transfer is packetized.
- Zero-length packets are specified with the Request Zero flag.

4.2.1.1.8 The usb_ep_dequeue Function

The usb_ep_dequeue function cancels an I/O request from an endpoint.

```
int dwc_otg_pcd_ep_dequeue(struct usb_ep *ep,
                           struct usb_request *req)
```

4.2.1.1.9 The usb_ep_set_halt, usb_ep_clear_halt Function

The usb_ep_set_halt function stalls an endpoint.

The usb_ep_clear_halt function clears an endpoint halt and resets its data toggle.

Both of these functions are implemented with the same underlying function. The behavior depends on the value argument. 1 means set_halt, 0 means clear_halt.

```
int dwc_otg_pcd_ep_set_halt(struct usb_ep *ep, int value)
```

4.2.1.1.10 The usb_iso_ep_start Function

The usb_iso_ep_start function starts isochronous transfers on an endpoint. PCD periodically calls ISO request process_iso_buffer function for data exchange between PCD and Gadget

```
int dwc_otg_pcd_iso_ep_start(struct usb_ep *_ep, struct usb_iso_request *_req, int
_gfp_flags)
```

4.2.1.1.11 The usb_iso_ep_stop Function

The usb_iso_ep_stop function stops isochronous transfers on an endpoint.

```
int dwc_otg_pcd_iso_ep_stop(struct usb_ep *_ep, struct usb_iso_request *_iso_req)
```

4.2.1.1.12 The usb_alloc_iso_request Function

This function allocates an isochronous request object to use with the specified endpoint.

```
struct usb_iso_request *dwc_otg_pcd_alloc_iso_request(struct usb_ep *_ep,int
packets,int gfp_flags)
```

4.2.1.1.13 The usb_free_iso_request Function

Stops iso transfers on Endpoint.

```
void dwc_otg_pcd_free_iso_request(struct usb_ep *_ep, struct usb_iso_request *req)
```

4.2.1.2 Gadget Operations

The following Gadget operations are implemented in the DWC_otg PCD. Functions in the API that are not described below are not implemented.

The Gadget API provides wrapper functions for each of the function pointers defined in usb_gadget_ops. The Gadget Driver calls the wrapper function, which then calls the underlying PCD function.The following gadget operations are named according to the wrapper functions. Within each section, the corresponding DWC_otg PCD function name is specified.

4.2.1.2.1 The usb_gadget_get_frame Function

The usb_gadget_get_frame function gets the USB frame number of the last SOF.

```
int dwc_otg_pcd_get_frame(struct usb_gadget *gadget)
```

4.2.1.2.2 The usb_gadget_wakeup Function

The usb_gadget_wakeup function initiates Session Request Protocol (SRP) to wake the host when no session is in progress. When a session is already in progress, but the device is suspended, remote wakeup signaling is started.

```
int dwc_otg_pcd_wakeup(struct usb_gadget *gadget)
```

4.2.1.2.3 The usb_gadget_test_lpm_support Function

Tests LPM support by underlying PCD.

```
int dwc_otg_pcd_test_lpm_enabled(struct usb_gadget *gadget)
```

4.3 PCD Core API

PCD OS wrapper communicates with PCD Core via PCD Core API. The PCD Core API is defined in the dwc_otg_pcd_if.h header file.

Almost all the functions require dwc_otg_pcd_t pointer to be passed. This pointer obtained via dwc_otg_pcd_init function call.

The following data structures are defined in dwc_otg_pcd_if.h header file:

```
typedef int (*dwc_completion_cb_t)(dwc_otg_pcd_t *pcd,
      void *ep_handle, void *req_handle, int32_t status,
      uint32_t actual);
typedef int (*dwc_isoc_completion_cb_t)(dwc_otg_pcd_t *pcd, void
      *ep_handle, void *req_handle, int proc_buf_num);
typedef int (*dwc_setup_cb_t)(dwc_otg_pcd_t *pcd, uint8_t *bytes);
typedef int (*dwc_disconnect_cb_t)(dwc_otg_pcd_t *pcd);
typedef int (*dwc_connect_cb_t)(dwc_otg_pcd_t *pcd, int speed);
typedef int (*dwc_suspend_cb_t)(dwc_otg_pcd_t *pcd);
typedef int (*dwc_sleep_cb_t)(dwc_otg_pcd_t *pcd);
typedef int (*dwc_resume_cb_t)(dwc_otg_pcd_t *pcd);
typedef int (*dwc_hnp_params_changed_cb_t)(dwc_otg_pcd_t *pcd);
typedef int (*dwc_reset_cb_t)(dwc_otg_pcd_t *pcd);

struct dwc_otg_pcd_function_ops
{
        dwc_connect_cb_t connect;
        dwc_disconnect_cb_t disconnect;
        dwc_setup_cb_t setup;
        dwc_completion_cb_t complete;
        dwc_isoc_completion_cb_t isoc_complete;
        dwc_suspend_cb_t suspend;
        dwc_sleep_cb_t sleep;
        dwc_resume_cb_t resume;
        dwc_reset_cb_t reset;
        dwc_hnp_params_changed_cb_t hnp_changed;
};
```

The PCD OS wrapper must pass the struct dwc_otg_pcd_function_ops pointer to dwc_otg_pcd_start function. The PCD core calls a certain callback function whenever the corresponding event occurs.

The PCD core also defines the PCD interrupt handler function, which must be called on every hardware interrupt.

4.3.1 The dwc_otg_pcd_init Function

This function is called to allocate and initialize the PCD core.

```
extern dwc_otg_pcd_t *dwc_otg_pcd_init(dwc_otg_core_if_t *core_if)
```

4.3.2 The dwc_otg_pcd_remove Function

This function frees PCD core allocated by the dwc_otg_pcd_init function.

```
extern void dwc_otg_pcd_remove(dwc_otg_pcd_t *pcd)
```

4.3.3 The dwc_otg_pcd_start Function

This function binds the Function Driver to the PCD core.

```
extern void dwc_otg_pcd_start(dwc_otg_pcd_t *pcd,
const struct dwc_otg_pcd_function_ops *fops)
```

4.3.4 The dwc_otg_pcd_ep_enable Function

This function enables an endpoint in the PCD. The endpoint is described by the ep_desc which has the same structure as a USB endpoint descriptor. The ep_handle parameter refers to the endpoint from other API functions and callbacks.

```
extern int dwc_otg_pcd_ep_enable(dwc_otg_pcd_t *pcd,
const uint8_t *ep_desc, void *ep_handle)
```

4.3.5 The dwc_otg_pcd_ep_disable Function

This function disables the endpoint referenced by ep_handle.

```
extern int dwc_otg_pcd_ep_disable(dwc_otg_pcd_t *pcd,
void *ep_handle)
```

4.3.6 The dwc_otg_pcd_ep_queue Function

This function is called to queue a data transfer request onto the endpoint referenced by ep_handle.

```
extern int dwc_otg_pcd_ep_queue(dwc_otg_pcd_t *pcd,
void *ep_handle, uint8_t *buf, dwc_dma_t dma_buf,
uint32_t buflen, int zero, void *req_handle,
int atomic_alloc)
```

4.3.7 The dwc_otg_pcd_ep_dequeue Function

This function de-queues the specified data transfer request, which has not been completed yet.

```
extern int dwc_otg_pcd_ep_dequeue(dwc_otg_pcd_t *pcd,
void *ep_handle, void *req_handle)
```

4.3.8 The dwc_otg_pcd_ep_halt Function

This function halts (STALLs) or clears an endpoint.

```
extern int dwc_otg_pcd_ep_halt(dwc_otg_pcd_t *pcd, void *ep_handle,
int value)
```

4.3.9 The dwc_otg_pcd_handle_intr Function

This function should be called on every hardware interrupt.

```
extern int32_t dwc_otg_pcd_handle_intr(dwc_otg_pcd_t *pcd)
```

4.3.10 The dwc_otg_pcd_get_frame_number Function

This function returns current frame number.

```
extern int dwc_otg_pcd_get_frame_number(dwc_otg_pcd_t *pcd)
```

4.3.11 The dwc_otg_pcd_iso_ep_start Function

This function starts isochronous transfers on the endpoint referenced by ep_handle.

```
extern int dwc_otg_pcd_iso_ep_start(dwc_otg_pcd_t *pcd,
void *ep_handle, uint8_t *buf0, uint8_t *buf1,
dwc_dma_t dma0, dwc_dma_t dma1, int sync_frame,
int dp_frame, int data_per_frame,   int start_frame,
int buf_proc_intrvl, void *req_handle,
int atomic_alloc)
```

4.3.12 dwc_otg_pcd_iso_ep_stop

This function stops isochronous transfers on the endpoint referenced by ep_handle.

```
extern int dwc_otg_pcd_iso_ep_stop(dwc_otg_pcd_t *pcd,
void *ep_handle, void *req_handle)
```

4.3.13 The dwc_otg_pcd_get_iso_packet_params Function

This function gets the isochronous packet status.

```
extern void dwc_otg_pcd_get_iso_packet_params(dwc_otg_pcd_t *pcd,
void *ep_handle, void *iso_req_handle, int packet,
int *status, int *actual, int *offset)
```

4.3.14 The dwc_otg_pcd_get_iso_packet_count Function

This function gets the isochronous packet count.

```
extern int dwc_otg_pcd_get_iso_packet_count(dwc_otg_pcd_t *pcd,
void *ep_handle, void *iso_req_handle)
```

4.3.15 The dwc_otg_pcd_wakeup Function

This function starts SRP if no session is in progress. If a session is already in progress, but the device is suspended, this function initiates remote wakeup signaling.

```
extern int dwc_otg_pcd_wakeup(dwc_otg_pcd_t *pcd)
```

4.3.16 The dwc_otg_pcd_is_lpm_enabled Function

This function returns a 1 if LPM support is enabled; 0 otherwise.

```
extern int dwc_otg_pcd_is_lpm_enabled(dwc_otg_pcd_t *pcd)
```

4.3.17 The dwc_otg_pcd_get_rmwkup_enable Function

This function returns 1 if remote wakeup is allowed; 0 otherwise.

```
extern int dwc_otg_pcd_get_rmwkup_enable(dwc_otg_pcd_t *pcd)
```

4.3.18 The dwc_otg_pcd_initiate_srp Function

This function initiates SRP.

```
extern void dwc_otg_pcd_initiate_srp(dwc_otg_pcd_t *pcd)
```

4.3.19 The dwc_otg_pcd_remote_wakeup Function

This function starts Remote Wakeup signaling.

```
extern void dwc_otg_pcd_remote_wakeup(dwc_otg_pcd_t *pcd, int set)
```

4.3.20 The dwc_otg_pcd_is_dualspeed Function

This function returns a 1 if the device is dual-speed; 0 otherwise.

```
extern uint32_t dwc_otg_pcd_is_dualspeed(dwc_otg_pcd_t *pcd)
```

4.3.21 The dwc_otg_pcd_is_otg Function

This function returns a 1 if the device is OTG; 0 otherwise.

```
extern uint32_t dwc_otg_pcd_is_otg(dwc_otg_pcd_t *pcd)
```

4.3.22 hnp_param functions

These functions get HNP parameters.

```
extern uint32_t get_b_hnp_enable(dwc_otg_pcd_t *pcd)
extern uint32_t get_a_hnp_support(dwc_otg_pcd_t *pcd)
extern uint32_t get_a_alt_hnp_support(dwc_otg_pcd_t *pcd)
```

4.4 Standard USB Command Processing

In Linux, USB commands are processed in two places: the PCD and the Gadget Driver (for example, the File-Backed Storage Gadget Driver).

Table 4-1 USB Commands

Command	Driver	Description
GET_STATUS	PCD	Command is processed as defined in Chapter 9 of the *Universal Serial Bus Specification*, Revision 2.0.
CLEAR_FEATURE	PCD	The Device and Endpoint requests are processed, and interface requests are ignored.
SET_FEATURE	PCD and Gadget Driver	The Device and Endpoint requests are processed by the PCD. Interface requests are passed to the Gadget Driver.
SET_ADDRESS	PCD	Program the Device Configuration register (DCFG) register with the received device address.
GET_DESCRIPTOR	Gadget Driver	Return the requested descriptor.

Table 4-1 USB Commands (Continued)

Command	Driver	Description
SET_DESCRIPTOR	Gadget Driver	Optional—not implemented by any of the existing Gadget drivers.
SET_CONFIGURATION	Gadget Driver	Disable all endpoints and enable endpoints for new configuration.
GET_CONFIGURATION	Gadget Driver	Return the current configuration.
SET_INTERFACE	Gadget Driver	Disable all endpoints and enable endpoints for new configuration.
GET_INTERFACE	Gadget Driver	Return the current interface.
SYNC_FRAME	Gadget Driver	Receives Sync Frame number for Isoc endpoints.

When the SETUP Phase Done interrupt is asserted for a control endpoint, the SETUP transaction bytes are processed by the PCD. Calling the Gadget Driver's setup function from the PCD processes the gadget SETUP commands.

4.5 Device Interrupt Service Routine

The PCD handles device interrupts. Many conditions can cause a device interrupt. When an interrupt occurs, the device interrupt service routine determines the cause of the interrupt and invokes the appropriate function to handle it. These interrupt-handling functions are described as follows.

All interrupt registers are processed from LSB to MSB.

4.5.1 Start of Frame Interrupt (SOF)

This function handles Start of Frame (SOF) interrupts. The handler for this interrupt is not implemented.

4.5.2 RxFIFO Non-Empty (RxFLvl) Interrupt

The dwc_otg_pcd_handle_rx_status_q_level_intr function services the RxFIFO Non-Empty interrupt, which indicates that there is a least one packet in the RxFIFO. The packets are moved from the FIFO to memory, where they are processed when the Endpoint Interrupt register (DIEPINT*n*/DOEPINT*n*) indicates Transfer Completed or SETUP Phase Done. This interrupt is only enabled in Slave mode.

```
int32_t dwc_otg_pcd_handle_rx_status_q_level_intr(dwc_otg_pcd_t *_pcd)
```

This function repeats the following steps until the Rx Status Queue is empty:

1. Reads the Receive Status Pop register (GRXSTSP) to get packet info.
2. When the RxFIFO is empty, clears the interrupt and exit.
3. When the RxFIFO contains a SETUP packet, calls The dwc_otg_read_setup_packet Function to copy the SETUP data to the buffer.
4. When the RxFIFO contains an OUT data packet, calls The dwc_otg_read_packet Function to copy the data to the destination buffer.

4.5.3 Non-Periodic TxFIFO Empty Interrupt

The dwc_otg_pcd_handle_np_tx_fifo_empty_intr function services Non-Periodic TxFIFO Empty interrupts, which occur when the Non-Periodic TxFIFO is empty or half-empty. The active request is checked for the next packet to be loaded into the Non-Periodic TxFIFO.

The learning queue is processed to determine the endpoint of the last IN token. The Non-Periodic TxFIFO is then filled with the data to be transmitted from the endpoint. As many complete packets as will fit are loaded into the Non-Periodic TxFIFO.

This interrupt is enabled only in Slave mode, when en_multiple_tx_fifo is not set.

```
int32_t dwc_otg_pcd_handle_np_tx_fifo_empty_intr(dwc_otg_pcd_t *_pcd)
```

4.5.4 Early Suspend Interrupt

The dwc_otg_pcd_handle_early_suspend_intr function services Early Suspend interrupts, which indicate that an Early Suspend condition has been detected on the USB. At this time, the interrupt handler simply clears the interrupt condition and takes no other action.

```
int32_t dwc_otg_pcd_handle_early_suspend_intr(dwc_otg_pcd_t *_pcd)
```

4.5.5 USB Reset Interrupt

The dwc_otg_pcd_handle_usb_reset_intr function services USB Reset interrupts, which occur when a USB reset is detected. When the USB Reset interrupt occurs, the device state is set to Default and the endpoint 0 state is set to Idle.

```
int32_t dwc_otg_pcd_handle_usb_reset_intr(dwc_otg_pcd_t *_pcd)
```

The interrupt handler performs the following steps:

1. Clears Remote Wakeup signaling.
2. Sets the NAK bit for all OUT endpoints (DOEPCTL*n*.SNAK = 1).
3. Flushes the TxFIFO and Learning Queue.
4. Unmasks the following interrupt bits:

   For non-MPI mode

   - DAINTMSK.INEP0 = 1 (control IN endpoint 0)
   - DAINTMSK.OUTEP0 = 1 (control OUT endpoint 0)
   - DOEPMSK.SETUP = 1
   - DOEPMSK.XferCompl = 1
   - DOEPMSK.ahberr = 1
   - DOEPMSK.epdisable = 1
   - DIEPMSK.XferCompl = 1
   - DIEPMSK.TimeOut = 1
   - DIEPMSK.epdisable = 1
   - DIEPMSK.Ahberr = 1

   For MPI mode

   - DEACHINTMSK.INEP0 = 1 (Control 0 IN endpoint)
   - DEACHINTMSK.OUTEP0 = 1 (Control 0 OUT endpoint)
   - DOEPEACHMSK[0].SETUP = 1
   - DOEPEACHMSK[0].XferCompl = 1
   - DOEPEACHMSK[0].ahberr = 1
   - DIEPEACHMSK[0].epdisable = 1

- DIEPEACHMSK[0].XferCompl = 1
- DIEPEACHMSK[0].TimeOut = 1
- DIEPEACHMSK[0].epdisable = 1
- DIEPEACHMSK[0]. Ahberr = 1

5. Resets the Device Address field in Device Configuration Register (DCFG).
6. To receive a SETUP packet, programs the following fields in the endpoint-specific registers for control OUT endpoint 0:
   - DOEPTSIZ0.SetUpCount = 3 (to receive up to 3 back-to-back SETUP packets)
   - DOEPTSIZ0.Packet Count = 1
   - DOEPTSIZE0.Transfer Size = 24 bytes (to receive up to 3 back-to-back SETUP data packets)
   - In DMA mode, programs the Device OUT Endpoint 0 DMA Address register (DOEPDMA0) with a memory address to store any SETUP data packets received and enable the endpoint
7. Clears the USB Reset interrupt.

At this point, all initialization required to receive SETUP packets is complete, except for enabling the control 0 OUT endpoint.

4.5.6 Enumeration Done Interrupt

The int32_t dwc_otg_pcd_handle_enum_done_intr interrupt indicates that USB speed enumeration has completed.

```
int32_t dwc_otg_pcd_handle_enum_done_intr(dwc_otg_pcd_t *_pcd)
```

The interrupt handler reads the device status register (DSTS) and sets the device speed in the data structure by performing the following steps:

1. Sets up endpoint 0 to receive SETUP packets by calling dwc_ep0_activate().
2. Sets the EP0 state to EP0_IDLE.
3. Sets the speed in the Gadget structure.

4.5.7 Isochronous OUT Packet Dropped Interrupt

This interrupt indicates that the ISO OUT Packet was dropped due to Rx FIFO full or Rx Status Queue Full. The handler for this interrupt is not implemented.

```
int32_t dwc_otg_pcd_handle_isoc_out_packet_dropped_intr
                                        (dwc_otg_pcd_t *_pcd)
```

4.5.8 End of Periodic Frame Interrupt

This interrupt indicates the end of the portion of the micro-frame for periodic transactions. The handler for this interrupt is not implemented.

```
int32_t dwc_otg_pcd_handle_end_periodic_frame_intr(
                                        dwc_otg_pcd_t *_pcd)
```

4.5.9 IN Token Received Interrupt

This interrupt indicates that the IN Token Sequence Learning Queue is not empty. This interrupt is disabled.

4.5.10 Endpoint Mismatch Interrupt

The dwc_otg_pcd_handle_ep_mismatch_intr function services Endpoint Mismatch interrupts, which indicate that the endpoint of the packet on the top of the Non-Periodic TxFIFO does not match endpoint of the IN token received.

```
int32_t dwc_otg_pcd_handle_ep_mismatch_intr(
                                        dwc_otg_pcd_t *_pcd)
```

The Device IN Token Queue registers are read to determine the order the IN tokens have been received. The Non-Periodic TxFIFO is flushed, so it can be reloaded in the order presented in the IN Token Queue.

This interrupt is enabled only when en_multiple_tx_fifo is not set.

4.5.11 IN Endpoint Interrupt

The dwc_otg_pcd_handle_in_ep_intr function indicates that an IN endpoint has a pending interrupt.

```
int32_t dwc_otg_pcd_handle_in_ep_intr(dwc_otg_pcd_t *_pcd)
```

The interrupt handler performs the following steps:

1. Reads the Device All Endpoints Interrupt register (DAINT).
2. Repeats the following for each IN endpoint interrupt bit set (from LSB to MSB):
   a. Read the Device Endpoint Interrupt (DIEPINTn) register
   b. If “Transfer Complete” call the request complete function.
   c. If “Endpoint Disabled” complete the EP disable procedure.
   d. If “AHB Error Interrupt” log error.
   e. If “Time-out Handshake” log error.
   f. If “IN Token Received when TxFIFO Empty” write packet to Tx FIFO.
   g. If “IN Token EP Mismatch” (disable, this is handled by EP Mismatch Interrupt).
   h. If “Tx FIFO Empty” call write_empty_fifo() function.
   i. If “BNA” and Global Continue on BNA is not set, disable EP.
   j. If “NAK” perform actions needed.

4.5.12 OUT Endpoint Interrupt

The dwc_otg_pcd_handle_out_ep_intr function services Out Endpoint interrupts, which interrupt indicate that an OUT endpoint has a pending interrupt.

```
int32_t dwc_otg_pcd_handle_out_ep_intr(dwc_otg_pcd_t *_pcd)
```

The interrupt handler performs the following steps:

1. Reads the Device All Endpoint Interrupt register (DAINT).
2. Repeats the following for each OUT endpoint interrupt bit set (from LSB to MSB):
   a. Read the Device Endpoint Interrupt (DOEPINTn) register.
   b. If “Transfer Complete,” call the Request Complete function (for PTI mode enabled and PktDrpSts bit is set, call handler for Out Packet Dropped).
   c. If “Endpoint Disabled,” complete the EP disable procedure.
   d. If “AHB Error Interrupt,” log error.

e. If “Setup Phase Done,” process Setup Packet (See “Standard USB Command Processing” on page 72).

f. If “BNA,” and Global Continue on BNA is not set, disable EP.

g. If “Status Phase Detected,” IN token for Control Write status phase is received. Enable IN EP for sending zero-length status packet.

h. If “Babble,” perform actions needed.

i. If “NYET,” perform actions needed.

j. If “NAK,” perform actions needed.

4.5.13 Incomplete Isochronous IN Transfer Interrupt

The dwc_otg_pcd_handle_incomplete_isoc_in_intr function indicates that one of the following conditions occurred while the device was transmitting an isochronous transaction:

- Corrupted IN token for isochronous endpoint
- Packet not complete in the FIFO

```
int32_t dwc_otg_pcd_handle_incomplete_isoc_in_intr(dwc_otg_pcd_t *_pcd)
```

The handler function finds the endpoint asserted by this interrupt, marks the packet as lost, and re-enables the endpoint for the next packet.

4.5.14 Incomplete Isochronous OUT Transfer Interrupt

This interrupt indicates that the core has dropped an Isochronous OUT packet. The following conditions can be the cause:

- FIFO Full, the entire packet would not fit in the FIFO.
- CRC Error
- Corrupted Token

```
int32_t dwc_otg_pcd_handle_incomplete_isoc_out_intr(dwc_otg_pcd_t *_pcd)
```

The handler function finds the endpoint asserted by this interrupt, marks the packet as lost, and re-enables the endpoint for the next packet.

5 Host Controller Driver

5.1 Host Controller Driver Overview

The Host Controller Driver (HCD) translates requests from the USB Driver into appropriate actions on the DWC_otg controller. The HCD isolates the USBD from the specifics of the controller by providing an API to the USBD. This API may vary between operating systems, but it remains constant within a given OS. "USB Driver Interface" on page 79 describes this API for supported operating systems.

An important function of the HCD is managing interrupts generated by the DWC_otg controller. The behavior of each DWC_otg Host mode interrupt is described in "Host Interrupt Service Routine" on page 92

5.2 USB Driver Interface

This section describes the API that the HCD's OS Wrapper layer presents to the USBD for each supported operating system. Currently, Linux is the only OS supported. The HCD OS wrapper acts as an intermediate layer between the USBD and HCD core.

5.2.1 Linux hc_driver API

The Host Controller Driver for Linux implements the hc_driver API. This API is defined in the header file drivers/usb/core/hcd.h in the 2.6.20.1 Linux source tree. The hc_driver data structure below defines the interface.

This interface is relatively new to Linux. It performs only those actions that require detection of controller hardware differences. All other host controller code is contained within the common kernel code in drivers/usb/core/hcd.c.

```
struct hc_driver {
      const char*description;                                  /* "ehci-hcd" etc */

      /* irq handler */
      irqreturn_t (*irq) (struct usb_hcd *hcd, struct pt_regs *regs);

      int    flags;
#define HCD_MEMORY 0x0001                                        /* HC regs use memory

                                                              (else I/O) */
#define HCD_USB11                    0x0010                     /* USB 1.1 */
#define HCD_USB2                     0x0020                     /* USB 2.0 */

      /* called to init HCD and root hub */
```

```
        int     (*reset) (struct usb_hcd *hcd);
        int    (*start) (struct usb_hcd *hcd);

        /* NOTE:  these suspend/resume calls relate to the HC as
         * a whole, not just the root hub; they're for bus glue.
         */
        /* called after all devices were suspended */
        int     (*suspend) (struct usb_hcd *hcd, u32 state);

        /* called before any devices get resumed */
        int     (*resume) (struct usb_hcd *hcd);

        /* cleanly make HCD stop writing memory and doing I/O */
        void    (*stop) (struct usb_hcd *hcd);

        /* return current frame number */
        int     (*get_frame_number) (struct usb_hcd *hcd);

        /* memory lifecycle */
        struct usb_hcd *(*hcd_alloc) (void);
        void    (*hcd_free) (struct usb_hcd *hcd);

        /* manage i/o requests, device state */
        int     (*urb_enqueue) (struct usb_hcd *hcd, struct urb *urb,
           int mem_flags);
        int     (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);

        /* hw synch, freeing endpoint resources that urb_dequeue can't */
        void  (*endpoint_disable) (struct usb_hcd *hcd,
                     struct hcd_dev *dev, int bEndpointAddress);

        /* root hub support */
        int          (*hub_status_data) (struct usb_hcd *hcd, char *buf);
        int          (*hub_control) (struct usb_hcd *hcd,
                           u16 typeReq, u16 wValue, u16 wIndex,
                           char *buf, u16 wLength);
        int          (*hub_suspend) (struct usb_hcd *);
        int          (*hub_resume) (struct usb_hcd *);
        int          (*start_port_reset) (struct usb_hcd *,
                       unsigned port_num);
};
```

The following sections describe the behavior of the hc_driver API functions implemented in the DWC_otg driver software. API functions not described below are not implemented.

5.2.1.1 The start Function

The start function initializes the DWC_otg controller and its root hub to prepare for Host mode operation. It then activates the root port. This function returns 0 on success and a negative error code on failure.

```
int dwc_otg_hcd_start(struct usb_hcd *hcd)
```

5.2.1.2 The stop Function

The stop function halts DWC_otg Host mode operations cleanly. USB transfers and memory reads and writes are stopped.

```
void dwc_otg_hcd_stop(struct usb_hcd *hcd)
```

5.2.1.3 The get_frame_number Function

The get_frame_number function returns the current frame number.

```
int dwc_otg_hcd_get_frame_number(struct usb_hcd *hcd)
```

5.2.1.4 The hcd_alloc Function

The hcd_alloc function allocates a dwc_otg_hcd structure, which contains a struct usb_hcd field, and returns the struct usb_hcd element for use by the common kernel code.

```
struct usb_hcd *dwc_otg_hcd_alloc()
```

The dwc_otg_hcd structure contains fields needed by the DWC_otg HCD functions described in this section. One of these fields is the usb_hcd structure common to all host controller drivers. DWC_otg HCD functions receive a usb_hcd argument and convert it to a dwc_otg_hcd structure to access DWC_otg-specific fields.

5.2.1.5 The hcd_free Function

The hcd_free function frees the dwc_otg_hcd structure that contains the struct usb_hcd field.

```
void dwc_otg_hcd_free(struct usb_hcd *hcd)
```

5.2.1.6 The urb_enqueue Function

The urb_enqueue function starts processing a USB transfer request specified by a USB Request Block (URB). mem_flags indicates the type of memory allocation to use while processing this URB.

```
int dwc_otg_hcd_urb_enqueue(struct usb_hcd *hcd, struct urb *urb,
                            int mem_flags)
```

See /usr/include/linux/usb.h in your Linux distribution for detailed information on the URB fields. See the description of usb_submit_urb in .../drivers/usb/core/urb.c in your Linux distribution for a full description of URB processing. Some key points of request processing are listed below:

- When the request completes, the completion callback specified in the URB is called asynchronously. The status of the request is stored in the URB.
- URBs may be submitted to an endpoint before previous URBs complete.
- In general, a device driver should not access an URB after it has been submitted until the complete function is called. Status fields for isochronous and interrupt requests can be accessed immediately after an URB is submitted.

This function's main role is to schedule the transfer by adding it to a list of transfers for the specified endpoint. In addition to separating transfers by endpoint, the scheduler separates transfers by type—control, bulk, isochronous, and interrupt.

Processing the transfer lists and executing USB transfers is the responsibility of the host interrupt handler routines. These functions are responsible for sending packets, processing received packets, and handling transfer completions or transfer errors. See "Host Interrupt Service Routine" on page 92 for more information on the host interrupt handler routines.

5.2.1.7 The urb_dequeue Function

The urb_dequeue function aborts or cancels a USB transfer request. urb_dequeue always returns 0 to indicate success.

```
int dwc_otg_hcd_urb_dequeue(struct usb_hcd *hcd, struct urb *urb)
```

This function also starts the process of removing the URB from the endpoint transfer list. The transfer may already be in progress, so the complete function for the URB is called when cancellation is finished.

5.2.1.8 The endpoint_disable Function

The endpoint_disable function frees the resources in the DWC_otg controller related to a given endpoint, and clears state in the HCD related to the endpoint. Any URBs for the endpoint must already be dequeued.

```
void dwc_otg_hcd_endpoint_disable(struct usb_hcd *hcd
                          struct hcd_dev *dev, int bEndpointAddress)
```

This function may be called, for example, when a SET_CONFIGURATION or SET_INTERFACE command is processed. Any endpoint state in the HCD (such as endpoint type, max packet size, data toggle state, and so forth) is cleared. When a host channel is delegated to this endpoint, it is released.

5.2.1.9 The irq Function

The irq function handles Host mode interrupts for the DWC_otg controller, and returns IRQ_NONE when there is no interrupt to handle. The IRQ function returns IRQ_HANDLED when there is a valid interrupt.

```
irqreturn_t dwc_otg_hcd_irq(struct usb_hcd *hcd, struct pt_regs *regs)
```

This method is called when a DWC_otg Host mode interrupt occurs. See "Host Interrupt Service Routine" for more information on Host mode interrupt handling.

5.2.1.10 The hub_status_data Function

The hub_status_data function creates a Status Change bitmap for the root hub and root port. The bitmap is returned in buf. Bit 0 is the status change indicator for the root hub. Bit 1 is the status change indicator for the single root port. The function returns 1 when either change indicator is 1; otherwise, it returns 0.

```
int dwc_otg_hcd_hub_status_data(struct usb_hcd *hcd, char *buf)
```

The change bit for the root hub is the logical OR of the following bits:

- Local Power Status Change
- Overcurrent Change

Currently, both of these bits are always 0, so the change bit for the root hub is always 0.

The change bit for the root port is the logical OR of the following bits:

- Connect Status Change
- Port Enable/Disable Change
- Suspend Change
- Overcurrent Indicator Change
- Reset Change

5.2.1.11 The hub_control Function

The hub_control function handles hub class-specific requests.

```
int dwc_otg_hcd_hub_control(struct usb_hcd *hcd, u16 typeReq,
                            u16 wValue, u16 wIndex, char *buf, u16 wLength)
```

Table 5-1 shows the requests handled by this function. See Section 11.24.2 of the *Universal Serial Bus Specification*, Revision 2.0 for more detail on these requests.

Table 5-1 Root Hub Class-Specific Requests

Request	Description
ClearHubFeature	**Supported Features**
	C_HUB_LOCAL_POWER No action required. Root Hub Local Power Source is always 0 (good). The Local Power Status Change bit is also always 0 (no change).
	C_HUB_OVER_CURRENT No action required. The Hub Overcurrent bit is always 0 because the hub reports overcurrent on a per-port basis. The Hub Overcurrent Change bit is also always 0 (no change).
ClearPortFeature	**Supported Features**
	PORT_ENABLE: Clears the Port Enable bit in the Host Port Control and Status Register.
	PORT_SUSPEND: Sets the Port Resume bit in the Host Port Control and Status Register. Schedules a time to clear the Port Resume bit so that the core will stop driving the resume signal on the USB.
	PORT_L1: Sets the Port Resume bit in the Host Port Control and Status Register. The Port Resume will be clear by Core automatically.
	PORT_POWER: Clears the Port Power bit in the Host Port Control and Status Register.
	PORT_INDICATOR: No action required. The DWC_otg controller does not support a Port Indicator.
	C_PORT_CONNECTION: Clears the driver's internal Connect Status Change flag.
	C_PORT_RESET: Clears the driver's internal Port Reset Change flag, which is set when reset signaling on the host port is complete.
	C_PORT_ENABLE: Clears the driver's internal Port Enable/Disable Change flag.
	C_PORT_SUSPEND: Clears the driver's internal Port Suspend Change flag, which is set when resume signaling on the host port is complete.
	C_PORT_L1: Clears the driver's internal PORT_L1 change flag.
	C_PORT_OVER_CURRENT: Clears the driver's internal port overcurrent change

Table 5-1 Root Hub Class-Specific Requests (Continued)

Request	Description
GetHubDescriptor	**Descriptor Fields**
	bDescLength: 9 bytes
	bDescriptorType: 29H
	bNbrPorts: 1
	wHubCharacteristics: • D1–D0: 00 (ganged power switching) • D2: 0 (not part of a compound device) • D4–D3: 01 (individual port overcurrent protection) • D6–D5: 00 (TT not applicable for root hub) • D7: 0 (port indicators not supported)
	bPwrOn2PwrGood: 1 (2 ms) This value depends on system electrical characteristics. It should be set to an appropriate value for a root hub port on your system. Possible values are 0–255, with each unit representing 2 ms. The value is the time from the beginning of the power-on sequence on the port until power is good on that port. The USB system software uses this value to determine how long to wait before accessing a powered-on port.
	bHubContrCurrent: 0 (for root hub)
	DeviceRemovable: 0x00 (1 port, device is removable)
	PortPwrCtrlMask: 0xff (1 port)
GetHubStatus	**Status Fields**
	Local Power Source: Always 0 (local power supply good)
	Overcurrent: Always 0 (overcurrent reported per-port)
	Local Power Status Change: Always 0 (no change)
	Overcurrent Change: Always 0 (no change)

Table 5-1 Root Hub Class-Specific Requests (Continued)

Request	Description
GetPortStatus	**Status Fields**
	Current Connect Status Returns the value of the Port Connect Status bit in the Host Port Control and Status register (HPRT).
	Port Enabled/Disabled Returns the value of the Port Enable bit in the Host Port Control and Status register (HPRT).
	Suspend Returns the value of the Port Suspend bit in the Host Port Control and Status register (HPRT).
	L1 Returns the value of the PORT_L1 bit.
	Overcurrent Returns the value of the Port Overcurrent Active bit in the Host Port Control and Status register (HPRT).
	Reset Returns the value of the Port Reset bit in the Host Port Control and Status register (HPRT).
	Port Power Returns the value of the Port Power bit in the Host Port Control and Status register (HPRT).
	Low-Speed Device Attached Returns 1 when the Port Speed field in the Host Port Control and Status register (HPRT) is 2‘b10 (Low-Speed). Returns 0 otherwise.
	High-Speed Device Attached Returns 1 when the Port Speed field in the Host Port Control and Status register (HPRT) is 2‘b00 (High-Speed). Returns 0 otherwise.
	Port Test Mode Returns 1 when the Port Test Control field in the Host Port Control and Status register (HPRT) is non-zero. Returns 0 otherwise.
	Port Indicator Control Always returns 0.
	Connect Status Change Returns the value of the driver’s internal Connect Status Change flag. This flag is set when the Port Connect Detected bit in the Host Port Control and Status register (HPRT) is set or the Disconnect Detected Interrupt bit in the Global Interrupt Status register (GINTSTS) is set.

Table 5-1 Root Hub Class-Specific Requests (Continued)

Request	Description
GetPortStatus *(continued)*	**Status Fields**
	Port Enable/Disable Change Returns the value of the driver's internal Port Enable/Disable Change flag. This flag is set when the Port Enable/Disable Change bit in the Host Port Control and Status register (HPRT) is set and the current Port Enable status is 0.
	Suspend Change Returns the value of the driver's internal Port Suspend Change flag. This flag is set when Resume signaling on the host port is completed.
	L1 Change Returns the value of the driver's internal Port L1 Change flag. This flag is set when the host completes the L1 exit process.
	Overcurrent Indicator Change Returns the value of the driver's internal Port Overcurrent Change flag.This flag is set when the Port Overcurrent Changed bit in the Host Port Control and Status register (HPRT) is set.
	Reset Change Returns the value of the driver's internal Port Reset Change flag. This flag is set when Reset signaling on the host port is completed.
SetHubFeature	**Supported Features**
	No features are supported for this request. This request is treated as a no-op for all features.
SetPortFeature	**Supported Features**
	PORT_RESET Sets the Port Reset bit in the Host Port Control and Status register (HPRT). Schedules a time to clear the Port Reset bit so that the core stops driving the Reset signal on the USB.
	PORT_SUSPEND Sets the Port Suspend bit in the Host Port Control and Status register (HPRT).
	PORT_POWER Sets the Port Power bit in the Host Port Control and Status register (HPRT).
	PORT_TEST Sets the specified value in the Port Test Control field in the Host Port Control and Status register (HPRT).
	PORT_INDICATOR No action required. The DWC_otg controller does not support a port indicator.
SetandTestPortFeature	**Supported Features**
	PORT_L1: Sends LPM transaction to the local device.

5.3 HCD Core API

The HCD OS wrapper communicates with the HCD core using the HCD core API. The HCD core API is defined in the dwc_otg_pcd_if.h header file.

The following data structures are defined in the dwc_otg_hcd_if.h header file.

```
typedef int (*dwc_otg_hcd_start_cb_t)(dwc_otg_hcd_t *hcd);
typedef int (*dwc_otg_hcd_disconnect_cb_t)(dwc_otg_hcd_t *hcd);
typedef int (*dwc_otg_hcd_hub_info_from_urb_cb_t)
      (dwc_otg_hcd_t *hcd, void *urb_handle,
      uint32_t *hub_addr, uint32_t *port_addr);
typedef int (*dwc_otg_hcd_speed_from_urb_cb_t)
      (dwc_otg_hcd_t *hcd, void *urb_handle);
typedef int (*dwc_otg_hcd_complete_urb_cb_t)
      (dwc_otg_hcd_t *hcd, void *urb_handle,
      dwc_otg_hcd_urb_t *dwc_otg_urb,
      uint32_t status);
typedef int (*dwc_otg_hcd_get_b_hnp_enable) (dwc_otg_hcd_t *hcd);

struct dwc_otg_hcd_function_ops
{
        dwc_otg_hcd_start_cb_t                  start;
        dwc_otg_hcd_disconnect_cb_t             disconnect;
        dwc_otg_hcd_hub_info_from_urb_cb_t      hub_info;
        dwc_otg_hcd_speed_from_urb_cb_t         speed;
        dwc_otg_hcd_complete_urb_cb_t           complete;
        dwc_otg_hcd_get_b_hnp_enable            get_b_hnp_enable;
};
```

5.3.1 HCD Core API functions

The HCD core provides the following functions, which can be used by the OS wrapper.

5.3.1.1 The dwc_otg_hcd_alloc_hcd Function

This function allocates dwc_otg_hcd_t structure and returns pointer on it.

```
extern dwc_otg_hcd_t *dwc_otg_hcd_alloc_hcd(void)
```

5.3.1.2 The dwc_otg_hcd_init Function

This function should be called to initiate HCD core.

```
extern int dwc_otg_hcd_init(dwc_otg_hcd_t *hcd,
dwc_otg_core_if_t *core_if)
```

5.3.1.3 The dwc_otg_hcd_remove Function

This function frees the HCD.

```
extern void dwc_otg_hcd_remove(dwc_otg_hcd_t *hcd)
```

5.3.1.4 The dwc_otg_hcd_handle_intr Function

This function must be called on every hardware interrupt.

```
extern int32_t dwc_otg_hcd_handle_intr(dwc_otg_hcd_t *dwc_otg_hcd)
```

5.3.1.5 The dwc_otg_hcd_get_priv_data Function

This function returns private data that had been set by dwc_otg_hcd_set_priv_data

```
extern void *dwc_otg_hcd_get_priv_data(dwc_otg_hcd_t *hcd)
```

5.3.1.6 The dwc_otg_hcd_set_priv_data Function

This function sets private data.

```
extern void dwc_otg_hcd_set_priv_data(dwc_otg_hcd_t *hcd,
void *priv_data)
```

5.3.1.7 The dwc_otg_hcd_start Function

This function initializes the HCD core.

```
extern int dwc_otg_hcd_start(dwc_otg_hcd_t *hcd,
struct dwc_otg_hcd_function_ops *fops)
```

5.3.1.8 The dwc_otg_hcd_stop Function

This function halts the DWC_otg host mode operations cleanly. USB transfers are stopped.

```
extern void dwc_otg_hcd_stop(dwc_otg_hcd_t *hcd)
```

5.3.1.9 The dwc_otg_hcd_hub_control Function

This function handles hub class-specific requests.

```
extern int dwc_otg_hcd_hub_control(dwc_otg_hcd_t *dwc_otg_hcd,
u16 typeReq, u16 wValue, u16 wIndex,
char *buf, u16 wLength)
```

5.3.1.10 The dwc_otg_hcd_otg_port Function

This function returns the OTG port number.

```
extern uint32_t dwc_otg_hcd_otg_port(dwc_otg_hcd_t *hcd)
```

5.3.1.11 The dwc_otg_hcd_is_b_host Function

If the core is currently acting as a B host, this function returns 1; 0 otherwise.

```
extern uint32_t dwc_otg_hcd_is_b_host(dwc_otg_hcd_t *hcd)
```

5.3.1.12 The dwc_otg_hcd_get_frame_number Function

This function returns the current frame number.

```
extern int dwc_otg_hcd_get_frame_number(dwc_otg_hcd_t *hcd)
```

5.3.1.13 The dwc_otg_hcd_dump_state Function

This function dumps the hcd state.

```
extern void dwc_otg_hcd_dump_state(dwc_otg_hcd_t *hcd)
```

5.3.1.14 The dwc_otg_hcd_dump_frrem Function

This function dumps the average frame remaining at SOF. This can be used to determine the average interrupt latency. Frame remaining is also shown for start transfer and two additional sample points.

This function is not currently implemented.

```
extern void dwc_otg_hcd_dump_frrem(dwc_otg_hcd_t *hcd)
```

5.3.1.15 The dwc_otg_hcd_send_lpm Function

This function sends LPM transaction to the local device.

```
extern int dwc_otg_hcd_send_lpm(dwc_otg_hcd_t *hcd,
uint8_t devaddr, uint8_t hird, uint8_t bRemoteWake)
```

5.3.1.16 The dwc_otg_hcd_urb_alloc Function

This function allocates memory for dwc_otg_hcd_urb structure. Allocated memory must be freed by calling the dwc_free function.

```
extern dwc_otg_hcd_urb_t *dwc_otg_hcd_urb_alloc(dwc_otg_hcd_t *hcd,
int iso_desc_count, int atomic_alloc)
```

5.3.1.17 The dwc_otg_hcd_urb_set_pipeinfo Function

This function sets pipe information in the URB.

```
extern void dwc_otg_hcd_urb_set_pipeinfo(dwc_otg_hcd_urb_t hcd_urb,
uint8_t devaddr, uint8_t ep_num, uint8_t ep_type,
uint8_t ep_dir, uint16_t mps)
```

5.3.1.18 The dwc_otg_hcd_urb_set_params Function

This function sets dwc_otg_hcd_urb parameters.

```
extern void dwc_otg_hcd_urb_set_params(dwc_otg_hcd_urb_t *urb,
void *urb_handle, void *buf, dwc_dma_t dma,
uint32_t buflen, void *sp, dwc_dma_t sp_dma,
uint8_t zero, uint8_t iso_asap, uint16_t interval)
```

5.3.1.19 The dwc_otg_hcd_urb_get_status Function

This function gets status from dwc_otg_hcd_urb

```
extern uint32_t dwc_otg_hcd_urb_get_status(
dwc_otg_hcd_urb_t *dwc_otg_urb)
```

5.3.1.20 The dwc_otg_hcd_urb_get_actual_length Function

This function gets the actual length from dwc_otg_hcd_urb

```
extern uint32_t dwc_otg_hcd_urb_get_actual_length(
dwc_otg_hcd_urb_t *dwc_otg_urb)
```

5.3.1.21 The dwc_otg_hcd_urb_get_error_count Function

This function gets the error count from dwc_otg_hcd_urb (only for isochronous URBs).

```
extern uint32_t dwc_otg_hcd_urb_get_error_count(
dwc_otg_hcd_urb_t *dwc_otg_urb)
```

5.3.1.22 The dwc_otg_hcd_urb_set_iso_desc_params Function

This function sets the isochronous descriptor offset and length.

```
extern void dwc_otg_hcd_urb_set_iso_desc_params(
dwc_otg_hcd_urb_t *dwc_otg_urb,
int desc_num, uint32_t offset, uint32_t length)
```

5.3.1.23 The dwc_otg_hcd_urb_get_iso_desc_status Function

This function gets the isochronous descriptor status specified by desc_num

```
extern uint32_t dwc_otg_hcd_urb_get_iso_desc_status(
dwc_otg_hcd_urb_t *dwc_otg_urb, int desc_num)
```

5.3.1.24 The dwc_otg_hcd_urb_enqueue Function

Queue URB. After transfer is completed, the complete callback will be called with the URB status.

```
int dwc_otg_hcd_urb_enqueue(dwc_otg_hcd_t *dwc_otg_hcd,
      dwc_otg_hcd_urb_t *dwc_otg_urb, void **ep_handle)
```

5.3.1.25 The dwc_otg_hcd_urb_dequeue Function

Dequeue the specified URB.

```
int dwc_otg_hcd_urb_dequeue(dwc_otg_hcd_t *dwc_otg_hcd,
      dwc_otg_hcd_urb_t *dwc_otg_urb)
```

5.4 Select and Queue Transactions

The HCD selects transactions to execute, and queues transactions to the DWC_otg controller.

5.4.1 Select Transactions

This function selects transactions from the HCD transfer schedule and assigns them to available host channels. It is called from HCD interrupt handler functions. Because the transfer schedule is shared between the HCD The urb_enqueue Function function and the interrupt handlers, care must be taken in handling transfer lists in the HCD. The return value indicates the types of transactions selected (periodic, non-periodic, both, or none).

```
dwc_otg_transaction_type_e
dwc_otg_hcd_select_transactions(dwc_otg_core_if_t *core_if)
```

Selecting transactions to assign to available host channels depends on the following factors:

- The number and type of transfers currently scheduled in the HCD
- The number of host channels available

If there are any periodic transactions to be executed in the next (micro)frame, they are assigned to host channels. (Periodic transactions are always assigned to host channels one (micro)frame before they are scheduled to execute.) The HCD reserves a host channel for each periodic transfer to ensure that a channel is always available when the transfer is scheduled to execute. In addition, the HCD ensures that the total committed bandwidth for periodic transactions is less than the maximum bandwidth allowed in the USB specification.

After assigning periodic transactions for the next (micro)frame, non-periodic transactions for the current (micro)frame are assigned to host channels. At least one host channel is always available for non-periodic transactions.

Multiple transactions may be assigned to a host channel for a single transfer request. For periodic transfers, up to three transactions may be assigned for a high-speed, high-bandwidth endpoint. For bulk endpoints, it is more efficient to assign multiple transactions at once rather than to wait for one transaction to finish before starting the next. However, there is a trade-off between the number of endpoints that can be serviced in a (micro)frame and the number of transactions that are assigned to a host channel simultaneously for each endpoint.

As transactions are assigned to a host channel, the channel is initialized as described in "The dwc_otg_hc_init Function" on page 56.

Channel assignment logic in Descriptor DMA mode differs from Buffer DMA logic.

Bulk transfers are not assigned to the channel until all the transfer requests constituting a single Scatter/Gather request are queued. Channel for Isochronous transfers remains assigned to the appropriate endpoint until session completion. For all endpoint types other than Isochronous, the channel is released when the DWC_otg controller completes transactions corresponding to the last descriptor within the list, or if some error occurs (such as a STALL or Babble error). In Descriptor DMA mode, this function is called from urb_enqueue function as well, unlike in Buffer DMA mode, when transaction selection is performed on each (micro) frame interrupt.

5.4.2 Queue Transactions

The dwc_otg_hcd_queue_transactions function processes the currently active host channels and queues transactions for these channels to the DWC_otg controller. It is called from HCD interrupt handler functions. The tr_type argument specifies whether periodic, non-periodic, or both types of transactions are queued.

```
void dwc_otg_hcd_queue_transactions(dwc_otg_core_if_t *core_if,
                                    dwc_otg_transaction_type_e tr_type)
```

The number and type of transactions queued depends on the following factors:

- The value of the tr_type argument
- Periodic and Non-Periodic TxFIFO space available
- Periodic and Non-Periodic Request Queue space available
- Slave or DMA mode

Note that host channels for periodic and non-periodic transactions are processed separately because they have separate TxFIFOs and request queues.

For each of these transaction types, this function cycles through the associated host channels and determines the action to take.

The action required for each host channel depends on the state of that channel. For example, if in Slave or Buffer DMA the host channel has been initialized, but no transactions have started, this function starts the transfer on the host channel as described in "The dwc_otg_hc_start_transfer Function" on page 57. When the transfer associated with the host channel is in a PING state, this function issues a PING request.

In Slave mode, this function performs the following actions:

- If all the data packets for an OUT transfer have been queued and no retries are required, that transfer is skipped during processing. For IN transfers, another request is always queued when there is space available in the request queue. This ensures that another request is issued when the previous request received a NAK response. Since NAK interrupts for IN requests may be missed under certain conditions[1], it's important to issue IN requests until the transfer completes or a transfer error occurs.
- Processes active host channels of each transaction type in a round-robin order to provide fair access to all active transfers. Processing resumes where it left off in the previous call and continues until there is not enough space in the TxFIFO or the request queue for the current transaction type.

1. A NAK interrupt may be missed when multiple IN requests are issued back-to-back in the same (micro)frame. When both requests receive a NAK response, the second NAK may occur before the first NAK interrupt has been handled. In this case, the second NAK does not cause an interrupt.

- If more requests must be queued to complete the periodic or non-periodic transactions, it enables the Periodic and/or Non-Periodic TxFIFO Empty interrupts.

In Buffer DMA mode, all active channels are started. The DWC_otg controller automatically issues requests and transfers data between memory and the data FIFOs as Request Queue and FIFO space becomes available.

In Descriptor DMA mode the dwc_otg_hcd_start_xfer_ddma function is called. This function performs the following actions:

- Initializes descriptor list
- For periodic transfers, updates FrameList entries, corresponding to the assigned channel number, based on endpoint b-interval value.
- Calls dwc_otg_hc_start_transfer_ddma function (described in "The dwc_otg_hc_start_transfer_ddma Function" on page 57) to start the transfer.

5.5 Host Interrupt Service Routine

The HCD handles host interrupts. Many conditions can cause a host interrupt. When an interrupt occurs, the host interrupt service routine determines the cause of the interrupt and invokes the appropriate function to handle it. These interrupt handling functions are described in Sections 5.5.1–5.5.5.

All interrupt registers are processed from LSB to MSB. As each interrupt condition is handled, the corresponding interrupt status bit is cleared.

5.5.1 SOF Interrupt

The dwc_otg_hcd_handle_sof_intr function services Start-of-Frame interrupts in Host mode. Non-periodic transactions may be queued to the DWC_otg controller for the current (micro)frame. Periodic transactions may be queued to the controller for the next (micro)frame.

```
int32_t dwc_otg_hcd_handle_sof_intr(dwc_otg_hcd_t *dwc_otg_hcd)
```

This function performs the following steps:

- Activates any currently inactive periodic transfers that are scheduled to be executed in the next (micro)frame.
- Calls dwc_otg_hcd_select_transactions to select transactions from the HCD transfer schedule and assigns them to available host channels, as described in "Select Transactions" on page 90
- Calls dwc_otg_hcd_queue_transactions to queue transactions to the DWC_otg controller as described in "Queue Transactions" on page 91

In Descriptor DMA mode, the SOF Interrupt is masked because transaction selection and execution logic does not rely on this interrupt.

5.5.2 RxFIFO Non-Empty (RxFLvl) Interrupt

The dwc_otg_hcd_handle_rx_status_q_level_intr function services RxFIFO Non-Empty interrupts, which indicate that there is at least one packet in the RxFIFO. The packet(s) are moved from the FIFO to memory when the DWC_otg controller is operating in Slave mode.

```
int32_t dwc_otg_hcd_handle_rx_status_q_level_intr(
                                        dwc_otg_hcd_t *dwc_otg_hcd)
```

This function performs the following steps:

1. Reads the Receive Status Pop Register (GRXSTSP) to get packet status information.
2. When the packet was successfully received and the controller is operating in Slave mode, calls The dwc_otg_read_packet Function to copy the data to its destination buffer.

When the status read from GRXSTSP does not indicate data is available in the RxFIFO, the status must be discarded. Popping the IN Transfer Completed, Channel Halted, or Data Toggle Error statuses causes an interrupt, so the function returns without doing anything further.

5.5.3 Non-Periodic TxFIFO Empty Interrupt

The dwc_otg_hcd_handle_np_tx_fifo_empty_intr function services TxFIFO Empty interrupts, which occur when the Non-Periodic TxFIFO is half-empty. More data packets may be written to the FIFO for OUT transfers. More requests may be written to the non-periodic request queue for IN transfers. This interrupt is enabled only in Slave mode.

```
int32_t dwc_otg_hcd_handle_np_tx_fifo_empty_intr(
                                        dwc_otg_hcd_t *dwc_otg_hcd)
```

This function queues more non-periodic transactions by calling dwc_otg_hcd_queue_transactions. See "Queue Transactions" on page 91

5.5.4 Periodic TxFIFO Empty Interrupt

The dwc_otg_hcd_handle_perio_tx_fifo_empty_intr function services Periodic TxFIFO Empty interrupts, which occur when the Periodic TxFIFO is half-empty. More data packets may be written to the FIFO for OUT transfers. More requests may be written to the Periodic Request Queue for IN transfers. This interrupt is enabled only in Slave mode.

```
int32_t dwc_otg_hcd_handle_perio_tx_fifo_empty_intr(
                                        dwc_otg_hcd_t *dwc_otg_hcd)
```

This function queues more periodic transactions by calling dwc_otg_hcd_queue_transactions. See "Queue Transactions" on page 91

5.5.5 Port Interrupt

The dwc_otg_hcd_handle_port_intr function services Port interrupts, which are caused by many conditions. It determines which interrupt conditions have occurred and handles them appropriately.

```
int32_t dwc_otg_hcd_handle_port_intr(dwc_otg_hcd_t *dwc_otg_hcd)
```

This function reads the Host Port Control and Status register (HPRT) and performs the actions described in Sections 5.5.5.1–5.5.5.3 for each interrupt status bit set in the register.

5.5.5.1 Port Connect Detected

This status indicates that a device has been connected to the root port. The driver sets its internal Connect Status Change flag and clears the Port Connect Detected bit. The Hub Driver is notified when it reads the root hub's status via the Status Change endpoint. The Hub Driver issues a GetPortStatus command to the root hub to determine that a Connect Status Change has occurred and to find the current Port Connect status. The driver's Connect Status Change flag is cleared when the Hub Driver issues a ClearPortFeature(C_PORT_CONNECTION) command to the host port.

If a device has been connected, the Hub Driver initiates a port reset by issuing a SetPortFeature(PORT_RESET) command to the host port. When this occurs, the driver sets the Port Reset bit

in the Host Port Control and Status register (HPRT). In addition, the driver schedules a time to clear the Port Reset bit so that the core stops driving the reset signal on the USB. When the Port Reset bit is cleared, the core enables the port and sets the Port Enable bit in the Host Port Control and Status register.

5.5.5.2 Port Enable/Disable Change

This status indicates that the host port has been enabled or disabled. The Hub Driver requires notification only when the root port is disabled (see Section 11.24.2.7.2.2 in the *Universal Serial Bus Specification*, Revision 2.0). The core conditions that can automatically disable the host port are overcurrent and device disconnect. Both conditions result in status change notifications to the Hub Driver.

If the root port is enabled when this interrupt is triggered, the interrupt handler performs the following steps:

1. If the driver is configured to support Low Power mode and the attached device is a full-speed or low-speed device, the interrupt handler sets USBCFG.PhyLPwrClkSel = 1 to select a 48-MHz PHY clock for power savings. For a low-speed device, the handler sets HCFG.FSLSPclkSel to either 48 MHz or 6 MHz, depending on the driver configuration parameters. For a full-speed device, HCFG.FSLSPclkSel is always set to 48 MHz in Low Power mode.
2. If the driver is not configured to support Low Power mode, or the attached device is a high-speed device, the interrupt handler sets USBCFG.PhyLPwrClkSel = 0 to select a 480-MHz PHY clock.
3. If any clock rates have been changed, the interrupt handler issues another USB bus reset. The interrupt handler does not set the driver's internal Port Reset Change flag.
4. If no clock rates have been changed, the interrupt handler sets the driver's internal Port Reset Change flag. This flag indicates that the reset is complete. The hub driver detects this when it issues a GetPortStatus command to the host port.
5. The interrupt handler clears the Port Enable/Disable Change bit and returns.

If the root port is disabled when this interrupt is triggered, the driver sets its internal Port Enable/Disable Change flag and clears the Port Enable/Disable Change bit. The Hub Driver is notified of this change when it reads the root hub's status via the Status Change endpoint. The Hub Driver issues a GetPortStatus command to the root hub to determine that a Port Enable/Disable Change has occurred and to find the current port enabled status. The driver's internal Port Enable/Disable Change flag is cleared when the Hub Driver issues a ClearPortFeature(C_PORT_ENABLE) command to the host port.

5.5.5.3 Port Overcurrent Change

The Port Overcurrent Change status indicates that the host port overcurrent status has changed. The driver sets its internal Port Overcurrent Change flag and clears the Port Overcurrent Change bit in the HPRT register (HPRT). The Hub Driver is notified of this change when it reads the root hub's status via the Status Change endpoint. The Hub Driver issues a GetPortStatus command to the root hub to determine that a port overcurrent change has occurred and to find the overcurrent status. The driver's internal Port Overcurrent Change flag is cleared when the Hub Driver issues a ClearPortFeature(C_PORT_OVER_CURRENT) command to the host port.

5.5.6 Host Channels Interrupt

The dwc_otg_hcd_handle_hc_intr function indicates that one or more host channels has a pending interrupt. There are many conditions that can cause each host channel interrupt. This function determines which conditions have occurred for each host channel interrupt and handles them appropriately.

```
int32_t dwc_otg_hcd_handle_hc_intr(dwc_otg_hcd_t *dwc_otg_hcd)
```

This function reads the Host All Channels Interrupt register (HAINT) to determine which host channels caused the interrupt. For each host channel interrupt bit set, this function reads the corresponding Host Channel Interrupt register (HCINT*n*). This function performs the actions described in Sections 5.5.6.1–5.5.6.11 for each bit set in the HCINT*n* register.

5.5.6.1 Transfer Complete

The Transfer Complete status indicates that all packets programmed to be transferred on the host channel have successfully completed. The interrupt handler updates transfer information for the USB request associated with the host channel. When the entire request is complete, the interrupt handler sets the status of the request and informs the USB Driver that the request is complete. In Linux, this is done by calling the complete callback function passed in with the request.

In Slave mode, the interrupt handler calls The dwc_otg_hc_halt Function to halt the host channel. This makes the channel available for other transfers. For IN transfers, it flushes any remaining IN requests that may have been scheduled before the transfer completed.

5.5.6.2 Channel Halted

In Slave mode, the Channel Halted status normally occurs when the driver explicitly requests a channel halt by setting the Channel Disable bit in the Host Channel Characteristic register (HCCHAR*n*). This may happen for many reasons, including a Transfer Complete interrupt, a STALL response, a NAK response, a NYET response, or an error condition. In these cases, the Channel Halted interrupt indicates that the DWC_otg controller has finished its cleanup.

To release the channel, the interrupt handler performs the following steps:

1. Ensures that any state associated with the host channel is cleaned up.
2. Calls dwc_otg_hcd_select_transactions to select more transactions to be executed as described in "Select Transactions" on page 90
3. Makes the host channel available for other transfers by calling dwc_otg_hcd_queue_transactions to queue more transactions to the DWC_otg controller as described in "Queue Transactions" on page 91

In Buffer DMA mode, this status indicates that the core has finished processing transactions on a channel. In this case, the other bits in the HCINT*n* register determine what actions need to be taken. When this interrupt occurs, the HCINT*n* register is read and control is transferred to the appropriate interrupt handler. After other handling is complete, the actions above are performed to release the channel for use by other transfers.

In Descriptor DMA mode, this status indicates that the core has finished processing transactions on a descriptor with End-Of-List (EOL) bit set, or the core has detected an error condition when processing any of the programmed descriptors.

5.5.6.3 AHB Error

The AHB Error status indicates that an AHB error occurred while transferring data to or from memory in DMA mode. The interrupt handler aborts the USB request with an error status (-EIO). Debug information is displayed to indicate the memory address at which the error occurred.

In Buffer DMA mode, the interrupt handler makes the host channel available for other transfers by calling dwc_otg_hc_halt as described in "The dwc_otg_hc_halt Function" on page 56.

In Descriptor DMA mode, the core disables the channel on receiving an AHB error. The interrupt handler does not call dwc_otg_hc_halt. The driver just releases the host channel for use by other transfers.

5.5.6.4 STALL Response Received

The STALL Response Received status indicates that a STALL response was received for the last host channel transaction. The interrupt handler aborts the USB request with an error status (-EPIPE). For bulk and interrupt transfers, the data toggle for the associated endpoint is reset to 0 when this occurs.

In Slave mode, the interrupt handler halts the host channel by calling The dwc_otg_hc_halt Function.

In Buffer DMA mode, the interrupt handler releases the host channel for use by other transfers as described in "Channel Halted" on page 95.

In Descriptor DMA mode, all successfully executed USB requests (associated descriptors) are completed with success status. Any failed request is aborted with an error status (-EPIPE).

5.5.6.5 NAK Response Received

The NAK Response Received status indicates that a NAK response was received for the last host channel transaction. The interrupt handler handles this according to the type of transaction that received the NAK response. The type of transaction is determined by reading the HCCHAR*n* register. In Descriptor DMA mode, the driver does not service the NAK response because the core takes care of handling it.

Sections 5.5.6.5.1–5.5.6.5.4 describe NAK processing under the various possible conditions.

5.5.6.5.1 NAK for Control or Bulk OUT Transactions

In Slave mode, a NAK may occur for a PING transaction in addition to a bulk or control transaction. The interrupt handler performs the following steps:

1. Resets the error count.
2. For a high-speed device, sets the state in the host channel structure to indicate that a PING is pending for this channel. This causes the PING protocol to be started when dwc_otg_hcd_queue_transactions is called for this transfer.
3. Halts the channel by calling The dwc_otg_hc_halt Function. When the transfer is restarted, the packet that prompted the NAK is retransmitted.

In Buffer DMA mode, the driver does not service a NAK response for Control and Bulk OUT transactions because the core takes care of rewinding of buffer pointers and re-initializing the channel. The driver does check for a NAK condition in the case of a transaction Error, to reset the error count.

5.5.6.5.2 NAK for Control or Bulk IN Transaction

This interrupt is enabled only when the error count is non-zero. In this case, the error count is reset. Normally, there is no need to handle NAKs on control or bulk IN transactions. Requests are continually queued until the transfer completes.

5.5.6.5.3 NAK for Interrupt IN or OUT Transaction

When an interrupt transaction receives a NAK reply, it is not retried until its next scheduled period. The interrupt handler resets the error count and makes the host channel available for other transfers. In Slave mode, halt the channel by calling The dwc_otg_hc_halt Function. In Buffer DMA mode, release the host channel for use by other transfers as described in "Channel Halted" on page 95

5.5.6.5.4 NAK for SSPLIT/CSPLIT Transaction

The interrupt handler performs the following steps:

1. Sets the state in the host channel structure to indicate that the split transaction must be restarted.
2. When the NAK occurred on a complete split, resets the error count.

3. In Slave mode, halts the channel by calling The dwc_otg_hc_halt Function. In DMA mode, the handler releases the host channel for use by other transfers as described in "Channel Halted" on page 95 The transfer will be restarted where it left off.

5.5.6.6 ACK Response Received

This status indicates that an ACK response was received for the last host channel transaction. In some cases, this interrupt is masked or ignored. In other cases, the interrupt handler must perform some actions when an ACK is received. These situations are described in Sections 5.5.6.6.1–5.5.6.6.3. Unless otherwise noted, handling is the same in both Slave and Buffer DMA modes. In Descriptor DMA mode, the driver does not service the ACK response because the core takes care of handling it.

5.5.6.6.1 ACK for PING Transaction in Slave Mode

The ACK interrupt is enabled when performing the PING protocol in Slave mode.

The interrupt handler performs the following steps:

1. Resets the error count.
2. Clears the PING state in the host channel structure. This causes a normal control or bulk transfer to be started when dwc_otg_hcd_queue_transactions is called for this transfer.
3. Halts the channel by calling The dwc_otg_hc_halt Function. When the transfer is restarted, the packet that received a NAK reply is retransmitted.

5.5.6.6.2 ACK After Transfer Error

The ACK interrupt is enabled after a transfer error occurs. In this case, reset the error count and disable the ACK interrupt.

5.5.6.6.3 ACK for SSPLIT Transaction

The ACK interrupt is enabled when a SSPLIT transaction is issued. When the ACK occurs, the interrupt handler schedules a CSPLIT transaction to be executed at a later time.

5.5.6.7 NYET Response Received

This status indicates that a NYET response was received for the last host channel transaction. This response is only valid for high-speed control and bulk OUT transactions and for split transactions. In Descriptor DMA mode, the driver does not service the NYET response because the core takes care of handling it.

Sections 5.5.6.7.1–5.5.6.7.2 describe NYET processing under the various possible conditions.

5.5.6.7.1 NYET for Control or Bulk OUT Transaction

In Slave mode, the interrupt handler performs the following steps:

1. Resets the error count.
2. Sets the state in the host channel structure to indicate that a PING is pending for this channel. This causes the PING protocol to be started when dwc_otg_hcd_queue_transactions is called for this transfer.
3. Halts the channel by calling The dwc_otg_hc_halt Function. The transfer is restarted where it left off.

In Buffer DMA mode, the driver does not service a NYET response, since the core takes care of rewinding of buffer pointers and re-initializing the channel. The driver does check for a NYET condition in the case of a transaction Error, to reset the error count.

5.5.6.7.2 NYET for CSPLIT Transaction

The interrupt handler reschedules the CSPLIT transaction to be executed at a later time. For periodic transactions, retry the entire split transaction in a subsequent full-speed frame when there is not enough time left in the current full-speed frame.

5.5.6.8 Transaction Error

This status indicates that a transaction error occurred on the last host channel transaction. This can be caused by a CRC error, a PID check error, a bit stuffing error, or a timeout. In Descriptor DMA mode, the driver does not handle the Transaction Error status.

5.5.6.8.1 Transaction Error for Control or Bulk Transaction

The interrupt handler performs the following steps:

1. Increments the error count.
2. For a high-speed device, sets the state in the host channel structure to indicate that a PING is pending for this channel. This causes the PING protocol to be started when dwc_otg_hcd_queue_transactions is called for this transfer.
3. In Slave mode, the interrupt handler halts the channel by calling The dwc_otg_hc_halt Function.

   In Buffer DMA mode, the handler releases the host channel for use by other transfers as described in "Channel Halted" on page 95. When three transaction errors occur for the same packet, the interrupt handler aborts the USB request with an error status (-EPROTO) and the host channel is halted. Otherwise, the transaction is retried.

5.5.6.8.2 Transaction Error for Interrupt Transaction

The interrupt handler performs the following steps:

1. Increments the error count.
2. For complete split transactions, sets the transfer state to retry the entire split transaction at a later time.
3. In Slave mode, the interrupt handler halts the channel by calling The dwc_otg_hc_halt Function.

   In Buffer DMA mode, the handler releases the host channel for use by other transfers as described in "Channel Halted" on page 95. When three transaction errors have occurred for the same packet, the interrupt handler aborts the USB request with an error status (-EPROTO) and the host channel is halted. Otherwise, the transaction is retried.

5.5.6.9 Excessive Transaction Errors

This status indicates that three consecutive transaction errors occurred on the USB bus. This (XCS_XAC) interrupt is valid only in Descriptor DMA mode. In this case, all successfully executed USB requests (associated descriptors) are completed with success status, a failed request is aborted with an error status (-PROTO). The XCS_XACT interrupt is not generated for Isochronous channels.

5.5.6.10 Babble Error

This status indicates that a packet babble or frame babble error was received for the last host channel transaction. Packet babble occurs for IN transactions when the device transmits a data packet that is larger than the max packet size. Frame babble occurs when an IN transaction is in progress at the EOF2 timing point of a frame. Both are considered fatal errors.

The interrupt handler aborts the USB request with an error status (-EOVERFLOW).

In Slave mode, it halts the channel by calling The dwc_otg_hc_halt Function.

In Buffer DMA mode, it releases the host channel for use by other transfers as described in "Channel Halted" on page 95.

In Descriptor DMA mode, all successfully executed USB requests (associated descriptors) are completed with success status, failed requests are aborted with an error status (-EOVERFLOW).

5.5.6.11 Frame Overrun

This status indicates that the controller did not start a periodic transaction because the transaction would not have finished before the EOF1 time in the current (micro)frame. This could indicate a temporary problem (such as the AHB being busy) or that the bandwidth allocated for periodic transactions is too high.

The interrupt handler retries the transfer at a later time. In Slave mode, it halts the channel by calling The dwc_otg_hc_halt Function. In Buffer DMA mode, it releases the host channel for use by other transfers as described in "Channel Halted" on page 95. In Descriptor DMA mode, the driver does not handle the Frame Overrun status.

A
Performance Analysis

This appendix contains results of performance and CPU usage testing.

A.1 Testing Environment

Tests were performed with 2.90a driver and 2.90a hardware on Linux kernel 2.6.20.1.

Device mode performance environment:

- Host: Linux based PC (2.81-GHz Pentium 4 CPU) with EHCI host controller.
- Device: IPMate board with DWC_otg operating in device mode (max_transfer_size set to 128KB).
- Gadget drivers: File Storage and RAM Storage gadgets (buflen set to 128KB).

Host mode performance environment:

- Host: IPMate board with DWC_otg operating in host mode.
- Device: IPMate board with DWC_otg operating in device mode. Gadget driver: RAM Storage gadget.

IPMate board has the following parameters.

- ARM9 core clock: 202.8 MHz
- System bus clock:
  - AHB: 101.4 MHz
  - APB: 50.7 MHz
- DRAM clock: 101.4 MHz
- OTG IP clock (and secondary AHB bus): 33 MHz
- System memory size: 64 MB

A.2 Test Results for HS OTG Linux Driver Software

Tables A-1 through A-3 summarize results when copying a 20 MB file to and from removable storage, using the dbench utility. To measure the CPU load, we used the Linux OProfile utility.

Table A-1 Device Mode With RAM Storage Gadget

	Slave Mode		Buffer DMA		Descriptor DMA	
	Speed (MB/s)	CPU Load (%)[a]	Speed (MB/s)	CPU Load (%)[a]	Speed (MB/s)	CPU Load (%)[a]
Read	4.87	95	33.2	5	33.20	5
Write	4.82	93	26.2	6	26.20	7

a. CPU loading calculated using DWC_otg, dwc_common_port_lib and handle_IRQ_event threads.

Table A-2 Device Mode With File Storage Gadget

	Slave Mode		Buffer DMA		Descriptor DMA	
	Speed (MB/s)	CPU Load (%)[a]	Speed (MB/s)	CPU Load (%)[a]	Speed (MB/s)	CPU Load (%)[a]
Read	4.62	73	12.76	4	12.52	4.5
Write	3.57	72	11.34	3.5	11.48	3

a. CPU loading calculated using DWC_otg, dwc_common_port_lib and handle_IRQ_event threads.

Table A-3 Host Mode as a Device OTG With RAM Storage Gadget

	Slave Mode		Buffer DMA		Descriptor DMA	
	Speed (MB/s)	CPU Load[a] (%)	Speed (MB/s)	CPU Load[b] (%)	Speed (MB/s)	CPU Load (%)
Read	3.69	70	13.25	38	16.81	4.8
Write [b]	6.59	79	15.30	42	18.72	6

a. CPU loading calculated using DWC_otg, dwc_common_port_lib and handle_IRQ_event threads.
b. Performance measured using dbench utility with '--direct' option specified.

B ROM Sizing

B.1 Overview

This appendix provides estimates of how much ROM the DWC_otg driver and other modules require. Depending on your application, only certain modules are required. For example, a complete USB host stack requires the DWC_otg driver, Linux USB core module, and a class driver (a printer driver class, for example).

B.2 Estimated ROM Sizes

Table B-1 Estimated ROM Size for Host Applications

Modules (Host Stack)	Total (KB)	ROM (KB)	Description
dwc_otg	126 [a]	123 [a]	DWC_otg driver (HCD and PCD)
dwc_common_port_lib	16	15	Portability library
usbcore	98	93	Linux USB core module
usb-storage	51	43	Needed for mass storage devices
sd_mod	11	10	Needed for mass storage devices
sg	21	20	Needed for mass storage devices
scsi_mod	90	79	Needed for mass storage devices
usblp	9	8	Needed for printers

a. The sizes for dwc_otg are listed with full functionality. See Tables B-3 and B-4 for details.

Table B-2 Estimated ROM Size for Device Applications

Modules (Device Stack)	Total (KB)	ROM (KB)	Description
dwc_otg	126 [a]	123 [a]	DWC_otg driver (HCD and PCD)
dwc_common_port_lib	16	15	Portability library
g_file_storage	21	19	Linux example mass storage function driver

a. The sizes for dwc_otg are listed with full functionality. See Tables B-3 and B-4 for details.

Table B-3 Estimated ROM Size for DWC_otg With Debug Option Enabled (All Debug Messages Enabled)

Modules (Host Stack)	Total (KB)	ROM (KB)	Description
Full	165	162	-
Device Mode Isochronous Transfers Disabled	156	152	Device mode isochronous transfers disabled
Device Only	116	113	Host mode disabled
Device Only with Isochronous Transfers Disabled	107	104	Host mode disabled, Device mode isochronous transfers disabled
Host Only	120	118	Device mode disabled

Table B-4 Estimated ROM Size for DWC_otg With Debug Option Disabled (Some Debug Messages Disabled)

Modules (Host Stack)	Total (KB)	ROM (KB)	Description
Full	126	123	-
Device Mode Isochronous Transfers Disabled	116	114	Device mode Isochronous transfers disabled
Device Only	93	91	Host mode disabled
Device Only with Isochronous Transfers Disabled	84	81	Host mode disabled, Device mode Isochronous transfers disabled
Host Only	91	88	Device mode disabled