PHILIPS

Philips Semiconductors

Connectivity

April 2002

AN10005-01

ISP1161x Embedded Programming Guide

Rev. 1.0

Revision History:

Rev.	Date	Descriptions	Author

We welcome your feedback. Send it to wired.support@philips.com

PHILIPS

This is a legal agreement between you (either an individual or an entity) and Philips Semiconductors. By accepting this product, you indicate your agreement to the disclaimer specified as follows:

DISCLAIMER

PRODUCT IS DEEMED ACCEPTED BY RECIPIENT. THE PRODUCT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, PHILIPS SEMICONDUCTORS FURTHER DISCLAIMS ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANT ABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE PRODUCT AND DOCUMENTATION REMAINS WITH THE RECIPIENT. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL PHILIPS SEMICONDUCTORS OR ITS SUPPLIERS BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE, OR OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THIS AGREEMENT OR THE USE OF OR INABILITY TO USE THE PRODUCT, EVEN IF PHILIPS SEMICONDUCTORS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CONTENTS

1. INTRODUCTION 7

2. ISP1161X SOFTWARE MODELS 8

2.1. HOST-ONLY MODE 8
2.2. DEVICE-ONLY MODE 9
2.3. SIMULTANEOUS HOST-AND-DEVICE MODE 10

3. ISP1161X HARDWARE MODELS 11

3.1 HOST CONTROLLER HARDWARE MODEL 11
3.2. DEVICE CONTROLLER HARDWARE MODEL 12

4. ISP1161X SOFTWARE ARCHITECTURE 13

4.1. USB HOST SOFTWARE ARCHITECTURE 13
4.2. HOST STACK ARCHITECTURE 14
4.3. USB DEVICE SOFTWARE ARCHITECTURE 16

5. PROGRAMMING THE HOST CONTROLLER OF ISP1161X 17

5.1. SOFTWARE ACCESSIBLE HARDWARE COMPONENTS 17
5.2. HC CONTROL AND STATUS REGISTERS 17
5.2.1. Writing and Reading of the 16-Bit and 32-Bit Registers 19
5.3. WRITING AND READING OF THE ATL AND ITL BUFFERS 21
5.4. TYPICAL HARDWARE INITIALIZATION SEQUENCE 22
5.4.1. Detecting the Host Controller 23
5.4.2. Software Resetting the Host Controller 23
5.4.3. Configuring the HcHardwareConfiguration Register 25
5.4.4. Configuring Interrupts 27
5.4.5. Configuring the HcFmInterval Register 30
5.4.6. Configuring Root Hub Registers 30
5.4.7. Setting the ITL and ATL Buffer Lengths 32
5.4.8. Installing INT1 Interrupt Service Routine 33
5.4.9. Setting the Host Controller to the Operational State 34
5.4.10. Setting the Host Controller to Perform USB Enumeration 34
5.5. HOST CONTROLLER DRIVER OPERATION FLOW 36
5.6. ACCESSING THE ATL BUFFER 36
5.6.1. Using SOFITLInt Versus ATLInt 36
5.6.2. Starting Scan of the ATL Buffer by Hardware 39
5.7. ACCESSING THE ITL BUFFER 40
5.8. FLOWCHART OF THE HOST CONTROLLER IN THE OPERATIONAL MODE 41
5.9. SETTING UP PTDS FOR TRANSFERS 42
5.9.1. Control Transfer 45
5.9.2. Bulk, Interrupt and Isochronous Transfers 46
5.10. DATA STRUCTURES FOR LIST PROCESSING 47
5.11. ERROR HANDLING 48

6. PROGRAMMING THE DEVICE CONTROLLER OF ISP1161X 49

6.1. FIRMWARE STRUCTURE OF THE DEVICE CONTROLLER 49
6.1.1. Hardware Abstraction Layer—HAL4SYS.C 50
6.1.2. Hardware Abstraction Layer—HAL4D13.C 50
6.1.3. Interrupt Service Routine—ISR.C 50
6.1.4. Protocol Layer—CHAP_9.C 50

6.1.5. Protocol Layer—D13BUS.C *50*
6.1.6. Main Loop—MAINLOOP.C *50*
6.2. PORTING THE FIRMWARE TO OTHER CPU PLATFORM 50
6.3. DEVELOPING THE FIRMWARE IN THE POLLING MODE 51
6.4. HARDWARE ABSTRACTION LAYER 51
6.4.1. Hardware Abstraction Layer for the System *51*
6.4.2. Hardware Abstraction Layer for the Device Controller of ISP1161x *52*
6.5. INTERRUPT SERVICE ROUTINE 53
6.5.1. Bus Reset *55*
6.5.2. Suspend Change *56*
6.5.3. EOT Handler *56*
6.5.4 Control Endpoint Handler *56*
6.5.5. Control OUT Handler *57*
6.5.6 Control IN Handler *59*
6.5.7. Bulk Endpoint Handler *61*
6.5.8. ISO Endpoint Handler *65*
6.6. MAIN LOOP 67
6.7. STANDARD DEVICE REQUESTS 70
6.7.1. Clear Feature Request *70*
6.7.2. Get Status Request *72*
6.7.3. Set Address Request *73*
6.7.4 Get Configuration Request *74*
6.7.5. Get Descriptor Request *75*
6.7.6. Set Configuration Request *76*
6.7.7. Get and Set Interface Requests *77*
6.7.8. Set Feature Request *77*
6.7.9. Class Request *78*
6.8. VENDOR REQUEST 78
6.8.1. Vendor Request for the Bulk Transfer *78*
6.8.2 CATC Capture of a PIO OUT Transfer *79*
6.8.3. CATC Capture of a PIO IN Transfer *80*
6.8.4. Vendor Request for the ISO Transfer *81*
6.8.5 CATC Capture of an ISO OUT Transfer *81*
6.8.6. CATC Capture of an ISO IN Transfer *82*

7. REFERENCES 82

TABLES

Table 5-1: HC Control and Status Register Summary 18
Table 5-2: HcScratch Register: Bit Allocation 23
Table 5-3: HcCommandStatus Register: Bit Allocation 24
Table 5-4: HcControl Register: Bit Allocation 24
Table 5-5: HcHardwareConfiguration Register: Bit Allocation 26
Table 5-6: HcHardwareConfiguration Register: Bit Description 26
Table 5-7: HcInterruptEnable Register: Bit Allocation 27
Table 5-8: HcµPInterruptEnable Register: Bit Allocation 28
Table 5-9: HcInterruptStatus Register: Bit Allocation 29
Table 5-10: HcµPInterrupt Register: Bit Allocation 29
Table 5-11 HcFmInterval Register: Bit Allocation 30
Table 5-12: HcRhDescriptorA Register: Bit Allocation 31
Table 5-13: HcRhStatus Register: Bit Allocation 32
Table 5-14: HcRhDescriptorB Register: Bit Allocation 32
Table 5-15: USB Transaction Error Codes 48
Table 6-1: Building Blocks Modifications 50

Table 6-2: Interrupt Register: Bit Allocation 54
Table 6-3: Endpoint Status Register: Bit Allocation....58
Table 6-4: Endpoint Status Register: Bit Description....59
Table 6-5: Recommended Endpoint Configuration Register Programming for a Bulk Endpoint....61
Table 6-6: Endpoint Configuration Register: Bit Allocation 61
Table 6-7: Endpoint Configuration Register: Bit Description....61
Table 6-8: Recommended Endpoint Configuration Register Programming for an ISO Endpoint 65
Table 6-9: Mode Register: Bit Allocation 69
Table 6-10: Mode Register: Bit Description 69
Table 6-11: Device Address Register: Bit Allocation 73
Table 6-12: Device Address Register: Bit Description....74
Table 6-13: Device Request 78
Table 6-14: Proprietary Definition of the Sample Firmware and Applet 79
Table 6-15: Device Request 81

FIGURES

Figure 2-1: ISP1161x Host-Only Mode Software Model....8
Figure 2-2: ISP1161x Device-Only Mode Software Model 9
Figure 2-3: ISP1161x Simultaneous Host-and-Device Mode Software Model 10
Figure 3-1: ISP1161x Host Controller Hardware Model....11
Figure 3-2: ISP1161x Device Controller Hardware Model....12
Figure 4-1: USB Host Software Architecture 13
Figure 4-2: Host Stack Architecture....14
Figure 4-3: Host Stack Calling Sequence Example....15
Figure 4-4: USB Device Software Architecture 16
Figure 5-1: 16-Bit Register Access Cycle....19
Figure 5-2: 32-Bit Register Access Cycle....19
Figure 5-3: Code Example for 32-Bit Register Write....20
Figure 5-4: Code Example for 32-Bit Register Read....20
Figure 5-5: Code Example for 16-Bit Register Read....20
Figure 5-6: Code Example for 16-Bit Register Write....21
Figure 5-7: Code Example for Writing to the ATL Buffer....22
Figure 5-8: Code Example for Detecting the Host Controller....23
Figure 5-9: Code Example for Resetting the Host Controller....24
Figure 5-10: Code Example for Setting the Host Controller to the RESET State....24
Figure 5-11: Code Example for Initializing the HcHardwareConfiguration Register....27
Figure 5-12: ISP1161x Host Controller Interrupt Logic 28
Figure 5-13: Code Example for Initializing the Host Controller Interrupt....29
Figure 5-14: Code Example for Initializing the HcDescriptorA Register 31
Figure 5-15: Code Example for Initializing the HcRhStatus Register....31
Figure 5-16: Code Example for Setting the ATL and ITL Buffer Lengths 33
Figure 5-17: Code Example for Setting the Host Controller to the Operational State....34
Figure 5-18: ATLInt Interrupt Flow 37
Figure 5-19: Running the Host Controller with the ATLInt Interrupt....38
Figure 5-20: Running the Host Controller with the SOFITLInt Interrupt....38
Figure 5-21: Code Example for Writing to the ATL Buffer....39
Figure 5-22: Code Example for Reading from the ATL Buffer....40
Figure 5-23: ITL Buffer Access Flow 40
Figure 5-24: Code Example for Writing to the ITL Buffer 41
Figure 5-25: Code Example for Reading from the ITL Buffer 41
Figure 5-26: Host Controller in the Operational State Flow Chart 42
Figure 5-27: PTD Header Fields....45
Figure 5-28: PTD Flow for the Control Transfer....45

Figure 5-29: Data Toggle Bit Setting Example Across Multiple PTDs 46
Figure 5-30: Data Toggle Bit Setting in Multiple PTD Data Packets 47
Figure 5-31: Typical List Structure 47
Figure 5-32: List Processing Data Structure 48
Figure 6-1: Firmware Structure of the ISP1161x Device Controller 49
Figure 6-2: Flowchart of ISR 53
Figure 6-3: Code Example of a Typical ISR 54
Figure 6-4: Code Example to Read the Interrupt Register 55
Figure 6-5: Control Flags 55
Figure 6-6: State Machine of the Control Transfer 56
Figure 6-7: Flowchart of the Control OUT Handler 57
Figure 6-8: Code Example to Check Status of the OUT Endpoint 57
Figure 6-9: Code Example for Reading the Endpoint Status Register 58
Figure 6-10: Code Example for Reading the Contents of an OUT Buffer 58
Figure 6-11: Code Example for Reading the Endpoint Status Register 59
Figure 6-12: Code Example to Check the Status of the IN Endpoint 60
Figure 6-13: Code Example for Writing the Contents to an IN Buffer 60
Figure 6-14: Flowchart of the Control IN Handler 60
Figure 6-15: Code Example for Configuring a Bulk OUT or Bulk IN Endpoint 62
Figure 6-16: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) 62
Figure 6-17: Flowchart of the Bulk OUT Handler 62
Figure 6-18: Code Example for Reading the Endpoint Status Register 63
Figure 6-19: Code Example to Check the Status of the Bulk OUT Endpoint 63
Figure 6-20: Code Example for Reading the Contents of a Bulk OUT Buffer 63
Figure 6-21: Flowchart of the Bulk IN Handler 64
Figure 6-22: Code Example for Reading the Endpoint Status Register 64
Figure 6-23: Code Example to Check the Status of the Bulk IN Endpoint 64
Figure 6-24: Code Example for Writing the Contents into a Bulk IN Buffer 65
Figure 6-25: Code Example for Configuring an ISO OUT or ISO IN Endpoint 65
Figure 6-26: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) 65
Figure 6-27: Flowchart of the ISO OUT Handler 66
Figure 6-28: Flowchart of the ISO IN Handler 66
Figure 6-29: Code Example for Reading the Endpoint Status Register 67
Figure 6-30: Code Example for Reading from an ISO Endpoint Buffer 67
Figure 6-31: Code Example for Writing to an ISO Endpoint Buffer 67
Figure 6-32: Flowchart of the Main Loop 68
Figure 6-33: Code Example for Writing to the Mode Register 69
Figure 6-34: Code Example on Setting SoftConnect 69
Figure 6-35: Flowchart of Clear Feature 70
Figure 6-36: Code Example to Send Zero-Length Packet 71
Figure 6-37: Code Example to Stall or Unstall an Endpoint 71
Figure 6-38: Flowchart of Get Status 72
Figure 6-39: Flowchart of Set Address 73
Figure 6-40: Code Example of the Set Address Routine 73
Figure 6-41: Flowchart of Get Configuration 74
Figure 6-42: Flowchart of Get Descriptor 75
Figure 6-43: Flowchart of Set Configuration 76
Figure 6-44: Flowchart of Get Interface 77
Figure 6-45: Flowchart of Set Interface 77
Figure 6-46: Flowchart of Set Feature 78
Figure 6-47: CATC Capture of a PIO OUT Transfer 79
Figure 6-48: CATC Capture of a PIO IN Transfer 80
Figure 6-49: CATC Capture of an ISO OUT Transfer 81
Figure 6-50: CATC Capture of an ISO IN Transfer 82

1. Introduction

ISP1161x (denotes ISP1161 and ISP1161A) is a single-chip Universal Serial Bus (USB) Host Controller (HC) and Device Controller (DC) that complies with *Universal Serial Bus Specification Rev. 2.0 (Full Speed)*. These two USB controllers—the Host Controller and the Device Controller—share the same microprocessor bus interface. These controllers have the same data bus, but different I/O locations. They also have separate interrupt request output pins, separate direct memory access (DMA) channels that include separate DMA request output pins (DREQ) and DMA acknowledge input pins (DACK). This makes it possible for a microprocessor to control both the USB Host Controller and the USB Device Controller at the same time.

ISP1161x provides two downstream ports for the USB Host Controller and one upstream port for the USB Device Controller. Each downstream port has its own overcurrent (OC) detection input pin and power supply switching control output pin. The upstream port has its own V_{BUS} detection input pin. ISP1161x also provides separate wake-up input pins and suspended status output pins for the USB Host Controller and the USB Device Controller, respectively. This makes power management flexible. The downstream ports for the Host Controller can be connected to any USB compliant USB devices and USB hubs that have USB upstream ports. The upstream port for the Device Controller can be connected to any USB compliant USB host and USB hubs that have USB downstream ports.

The Host Controller is adapted from *Open Host Controller Interface Specification for USB, Release: 1.0a* referred to as OHCI in the rest of this document.

The Device Controller is compliant with most device class specifications, such as Imaging Class, Mass Storage Devices, Communication Devices, Printing Devices and Human Interface Devices. ISP1161x is well suited for embedded systems and portable devices that require a USB host only, a USB device only, or a combined and configurable USB host and USB device capabilities. ISP1161x provides high flexibility to the systems that have it built-in. For example, a system that has ISP1161x built-in allows it not only to be connected to a PC or USB hub that has a USB downstream port, but also to be connected to a device that has a USB upstream port, such as USB printer, USB camera, USB keyboard and USB mouse, among others. ISP1161x enables point-to-point connectivity between embedded systems. An interesting application example is to connect a ISP1161x Host Controller with a ISP1161x Device Controller.

2. ISP1161x Software Models

As ISP1161x can function in one of the three modes—host-only mode, device-only mode and simultaneous host-and-device mode—each mode of operation adopts a different software model.

2.1. Host-Only Mode

The host-only mode software model consists of the host stack, one or more class drivers, zero or more device drivers, and the application. Figure 2-1 shows the data flow and the call hierarchy of the software components in this software model.

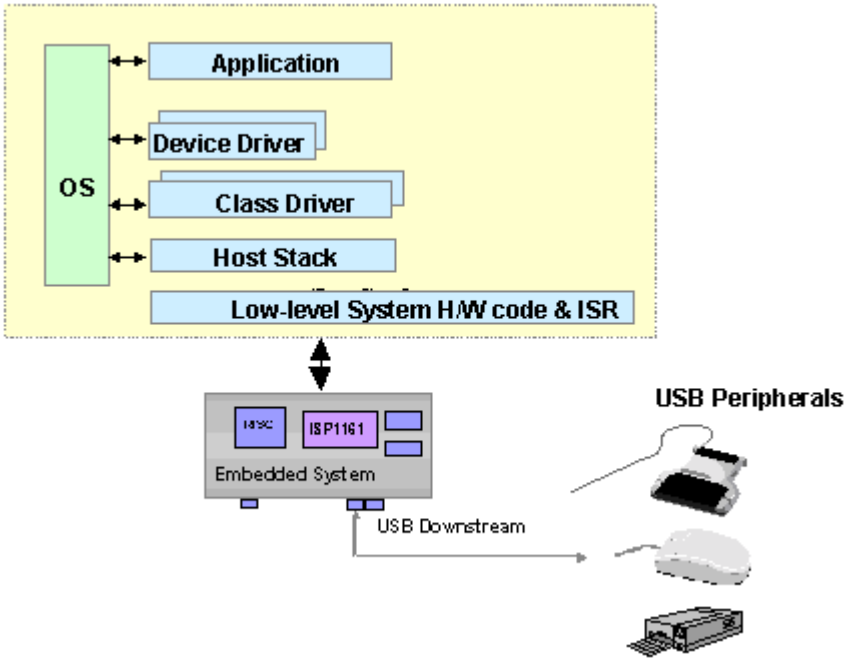


Figure 2-1: ISP1161x Host-Only Mode Software Model

Since a single USB Host Controller can have multiple USB slave devices connected to it, the host-only mode software model can contain multiple class drivers, in which each class driver services each type of USB slave device. Usually, the application accesses class drivers directly to perform USB operations. However, in some cases, it makes sense to have one more layer, dubbed Device Driver, between the class driver and the application. For example, you can have device drivers for different types of printers in which these device drivers access one common USB printer class driver to perform operations on printers.

2.2. Device-Only Mode

The software model for the device-only mode consists of the device stack, a class driver and the application. The data flow and the call hierarchy of the software components in this software model are given in Figure 2-2.

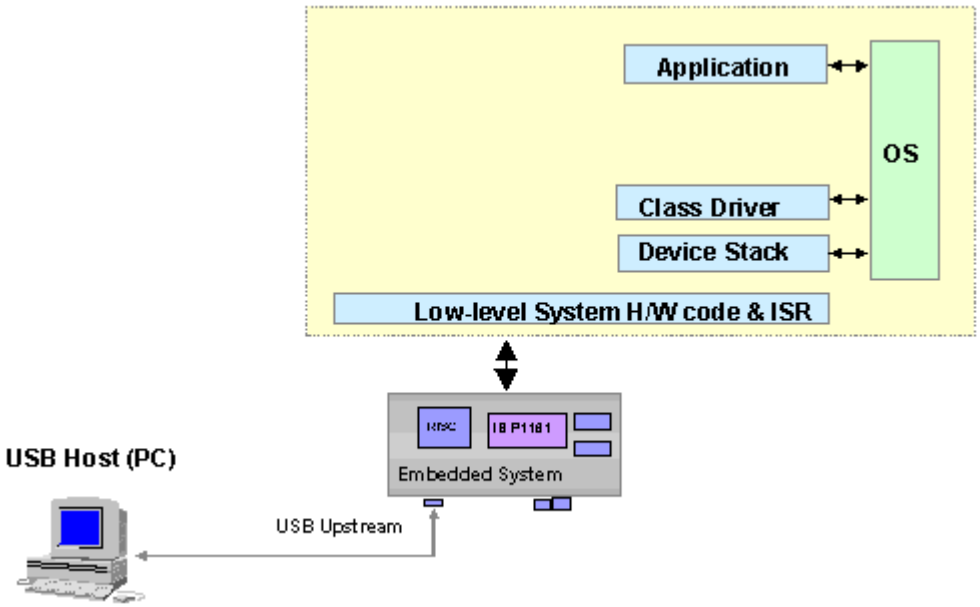


Figure 2-2: ISP1161x Device-Only Mode Software Model

Since a USB slave device performs a single class of functions, there must only be one class driver for a USB slave device. The application accesses the class driver when performing USB operations.

2.3. Simultaneous Host-and-Device Mode

The most versatile mode of ISP1161x is the simultaneous host-and-device mode. The software model for this mode is realized by combining the host-only mode and device-only mode software models into a single model. The resulting software model is depicted in Figure 2-3.

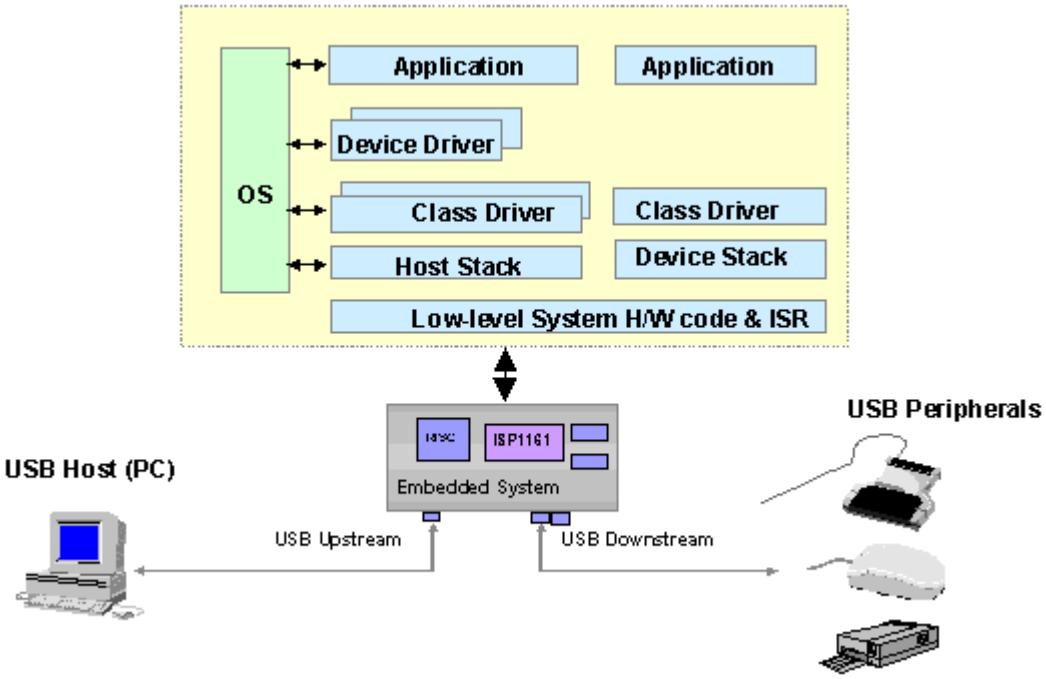


Figure 2-3: ISP1161x Simultaneous Host-and-Device Mode Software Model

In this mode, ISP1161x functions as if there are separate USB Host and Device Controllers. The software model for this mode requires no interdependencies between the host-side portion and the device-side portion of the software. In other words, the device-side software runs totally independent of the host-side software.

3. ISP1161x Hardware Models

3.1. Host Controller Hardware Model

The major difference between the OHCI Host Controller and ISP1161x is that the OHCI Host Controller is a bus-master device whereas ISP1161x is not. In the OHCI Host Controller, the USB data packet is sent from and received in the system memory by the bus master DMA present in the OHCI Host Controller. However, in ISP1161x, the microprocessor is responsible for moving the USB data packet between the system memory, and the ITL and ATL buffers inside ISP1161x. An I/O bus interface and the bus master DMA are eliminated from ISP1161x, and therefore, the term “slave Host Controller”. This is because ISP1161x is intended to be used for embedded applications in which cost and design simplicity are important design considerations for choosing a Host Controller IC.

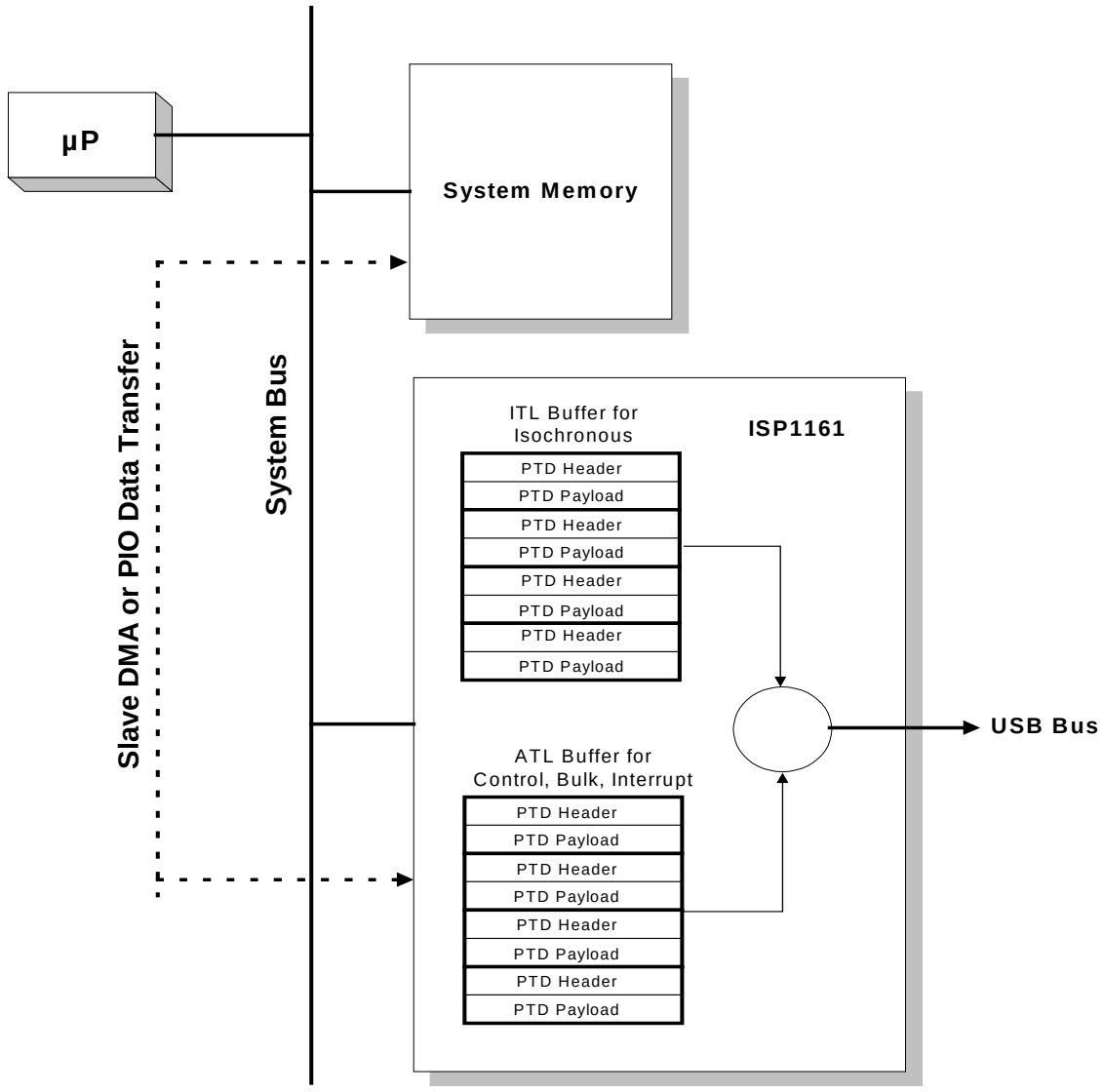


Figure 3-1: ISP1161x Host Controller Hardware Model

3.2. Device Controller Hardware Model

When the Device Controller part of ISP1161x is in operation, the microprocessor moves the USB packet data between the system memory and endpoint FIFOs via Programmed I/O (PIO) or slave DMA built into ISP1161x. USB packets are sent from and received in endpoint FIFOs.

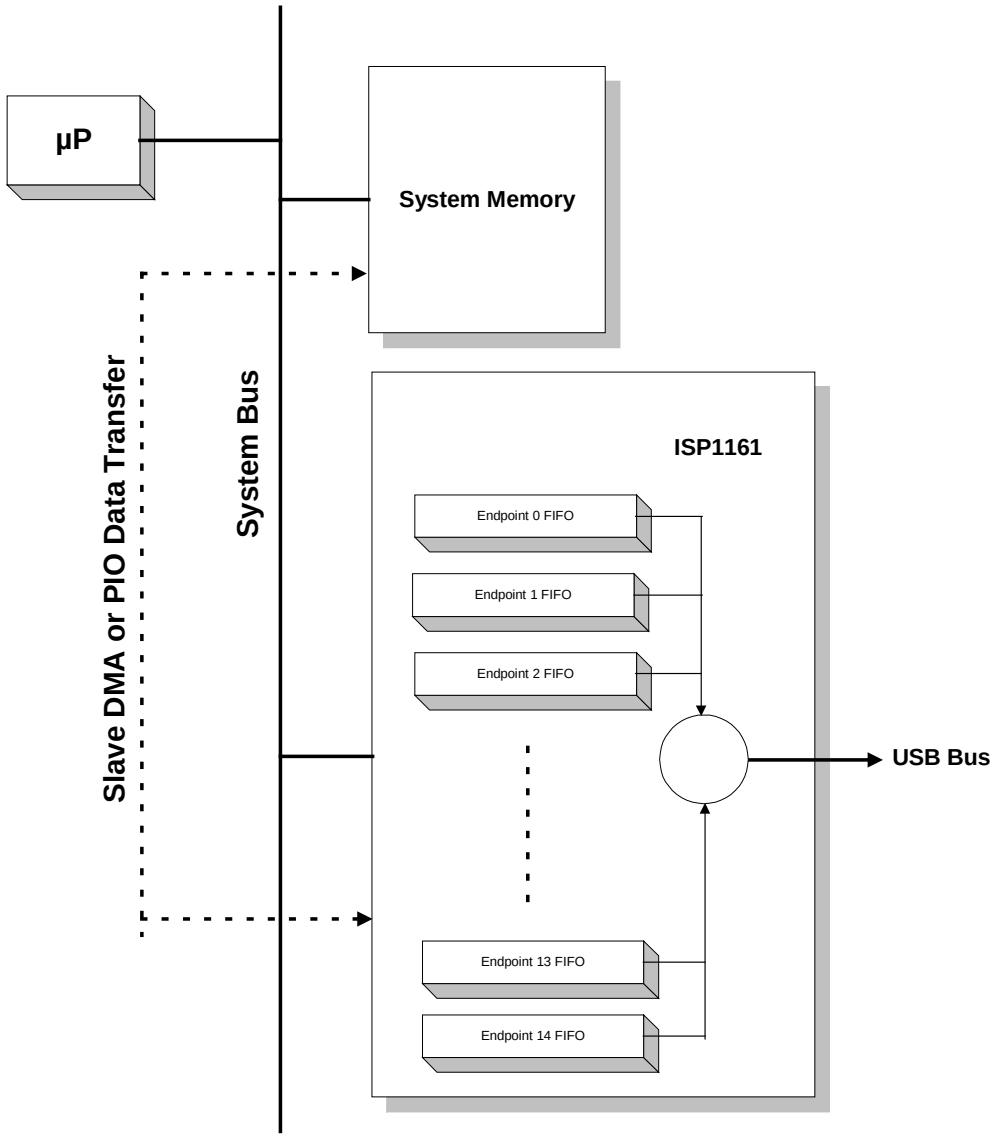


Figure 3-2: ISP1161x Device Controller Hardware Model

4. ISP1161x Software Architecture

4.1. USB Host Software Architecture

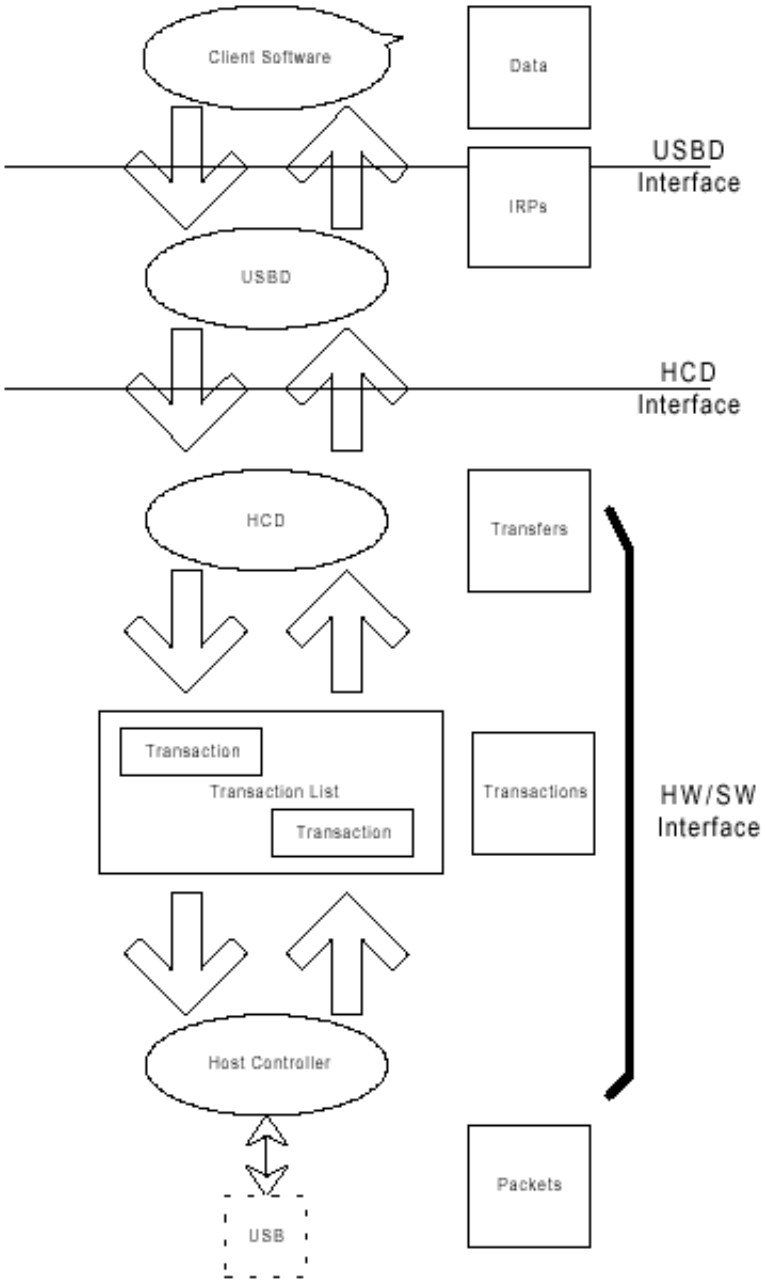


Figure 4-1: USB Host Software Architecture

As can be seen in Figure 4-1, the USB host software architecture includes the Universal Serial Bus Driver (USBD), the Host Controller Driver (HCD) and the client software. The client software can be the application code or USB class drivers. The USBD and the HCD are collectively referred to as the USB host stack. In the USB host stack, the USBD deals with hardware-independent protocol related aspects of USB whereas the HCD deals with hardware-dependent

protocol related aspects of USB. Therefore, it is the HCD that accesses the USB Host Controller hardware. In other words, the HCD drives the Host Controller by manipulating programmable hardware registers inside the Host Controller. This document explains how to program the Host Controller hardware of ISP1161x to enable it to perform HCD functions when it runs as a USB Host Controller.

4.2. Host Stack Architecture

Figure 4-2 shows the major functions built in a USB host stack.

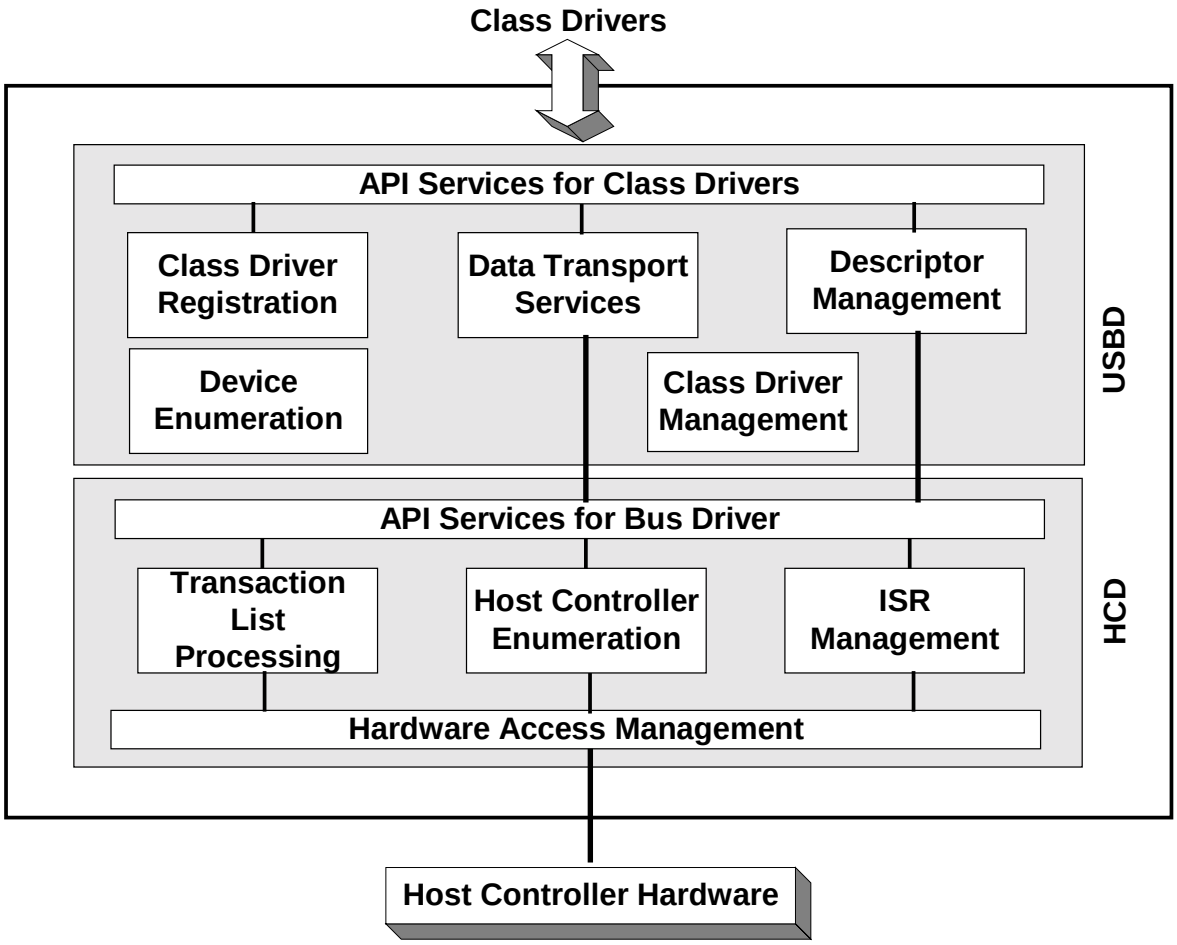


Figure 4-2: Host Stack Architecture

The typical sequence of calls that occurs when performing a USB transfer is as follows:

1. The application initiates a write or read over the USB bus.
2. The class driver calls USBD APIs for the write or read initiated by the application.
3. USBD APIs call HCD APIs on behalf of the calling class driver.
4. HCD APIs cause USB transactions to occur.
5. The class driver is notified that the transfer is complete.
6. The application is notified of the transfer completion.

The following example shows the call sequence from the class driver to the USBD and the HCD in a host stack implementation.

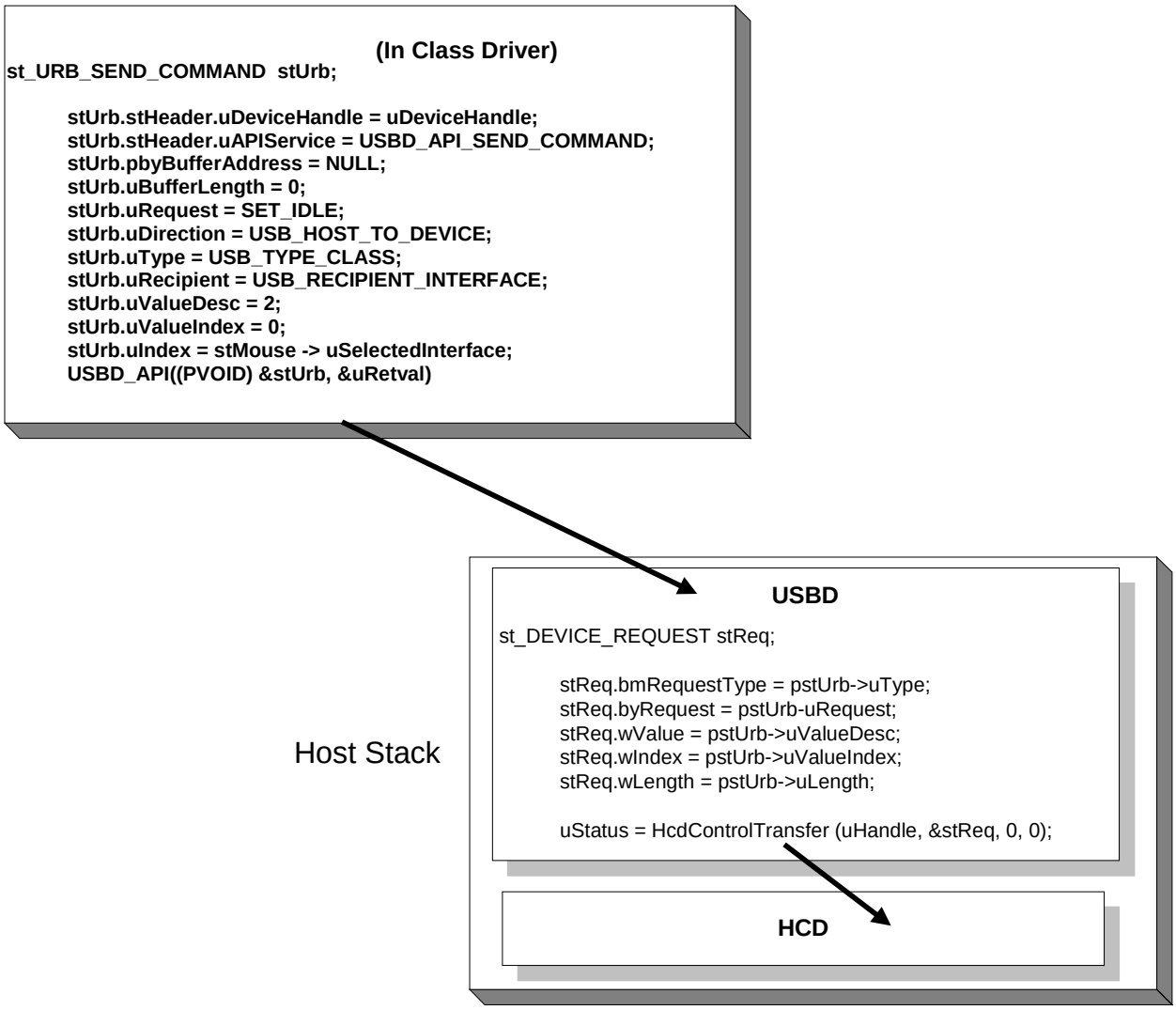


Figure 4-3: Host Stack Calling Sequence Example

In this example, the *USBD_API()* call (in the class driver box) is the calling mechanism for calling USBD APIs. The *HcdControlTransfer()* function is one of the available HCD APIs and it does a control transfer.

4.3. USB Device Software Architecture

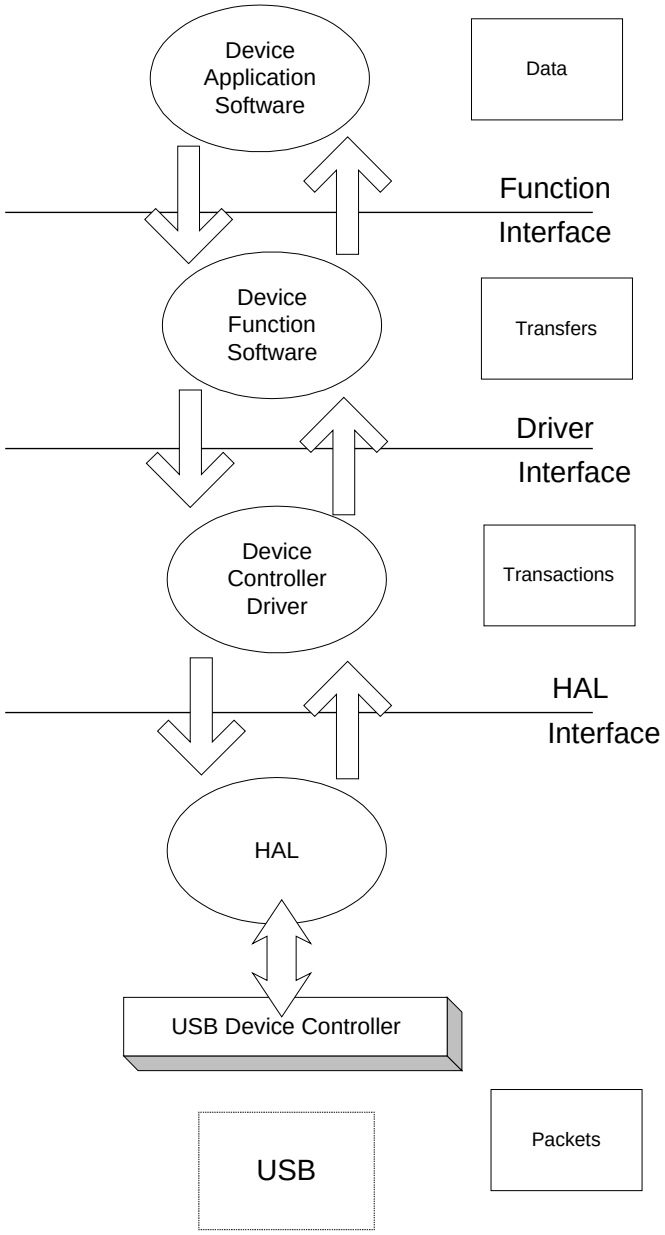


Figure 4-4: USB Device Software Architecture

A USB device is a slave device, and its job is to respond to requests sent by the host. The Device Controller Driver responds to requests on its own, if these requests are standard requests, that is, USB standard requests. If these requests are function specific, that is, class requests, the Device Controller Driver passes them to the device function software. The device function software in conjunction with the device application software handle these requests and send responses to the host. Logically, the Device Controller Driver interacts with the USBD on the host side and the device function software interacts with the host class driver.

5. Programming the Host Controller of ISP1161x

5.1. Software Accessible Hardware Components

The major hardware components of the Host Controller in ISP1161x that are accessible by software are:

- HC control and status registers
- ATL buffer
- ITL buffer.

5.2. HC Control and Status Registers

The HC control and status registers in ISP1161x include a set of operational OHCI compliant registers (32-bit wide) and a set of ISP1161x specific registers (16-bit wide). Each read/write register has a set of two offset indices: one for the read access and the other for the write access. Read-only or write-only registers have only one offset index. For convenience, the command-write operation, can be ORed with 0x80, so that only one value is required to be defined for each register. The offset indices for the HC control and status registers are given in Table 5-1.

Table 5-1: HC Control and Status Register Summary

Command (Hex)		Register	Width	Functionality
Read	Write			
00	N/A	HcRevision	32	HC Control and Status Registers
01	81	HcControl	32	
02	82	HcCommandStatus	32	
03	83	HcInterruptStatus	32	
04	84	HcInterruptEnable	32	
05	85	HcInterruptDisable	32	
0D	8D	HcFmInterval	32	HC Frame Counter Registers
0E	N/A	HcFmRemaining	32	
0F	N/A	HcFmNumber	32	
11	91	HcLSThreshold	32	
12	92	HcRhDescriptorA	32	HC Root Hub Registers
13	93	HcRhDescriptorB	32	
14	94	HcRhStatus	32	
15	95	HcRhPortStatus[1]	32	
16	96	HcRhPortStatus[2]	32	
20	A0	HcHardwareConfiguration	16	HC DMA and Interrupt Control Registers
21	A1	HcDMAConfiguration	16	
22	A2	HcTransferCounter	16	
24	A4	HcµPInterrupt	16	
25	A5	HcµPInterruptEnable	16	
27	N/A	HcChipID	16	HC Miscellaneous Registers
28	A8	HcScratch	16	
N/A	A9	HcSoftwareReset	16	
2A	AA	HcITLBufferLength	16	HC Buffer RAM Control Registers
2B	AB	HcATLBufferLength	16	
2C	N/A	HcBufferStatus	16	
2D	N/A	HcReadBackITL0Length	16	
2E	N/A	HcReadBackITL1Length	16	
40	C0	HcITLBufferPort	16	
41	C1	HcATLBufferPort	16	

5.2.1. Writing and Reading of the 16-Bit and 32-Bit Registers

Since the data bus in ISP1161x is 16-bit wide, 32-bit registers are read or written in two data cycles. Figure 5-1 illustrates a 16-bit register access cycle.

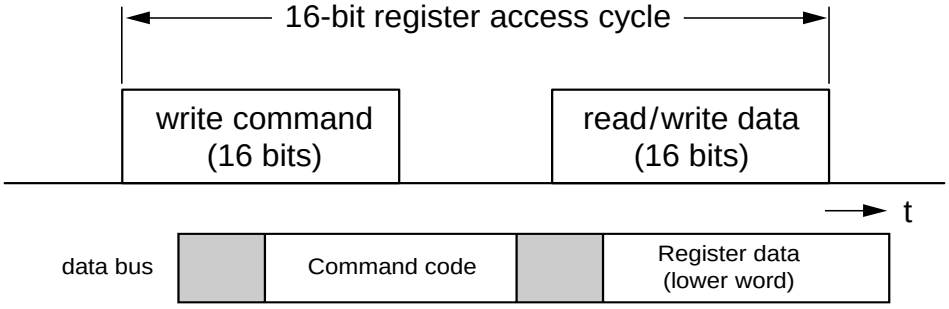


Figure 5-1: 16-Bit Register Access Cycle

In Figure 5-1, a *command code* is the offset index of the register being accessed. Therefore, for example, to write a value into the *HcScratch* register, the HCD will put the offset index of A8H on the data bus followed by a single 16-bit value. To read the *HcScratch* register, the HCD will put the offset index of the register on the data bus and read a single 16-bit data from the data bus.

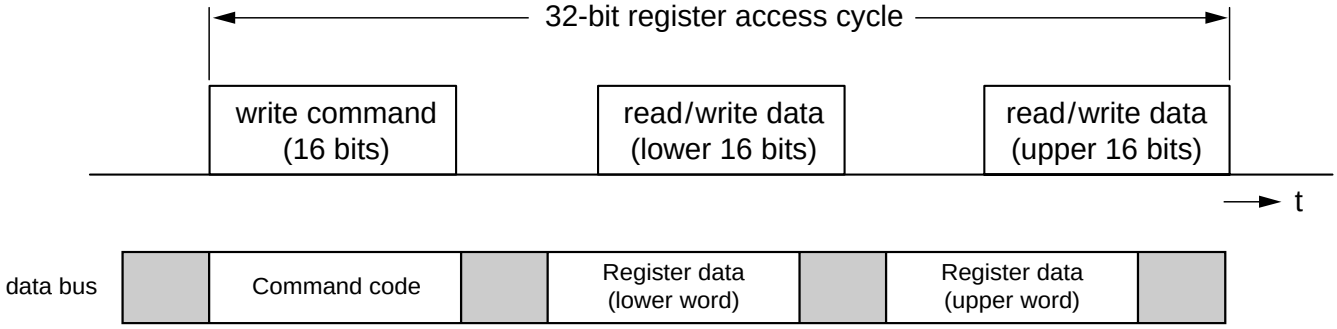


Figure 5-2: 32-Bit Register Access Cycle

To write to a 32-bit register, the HCD will put the offset index of the register on the data bus followed by two consecutive 16-bit data. To read, the HCD will put the offset index of the register on the data bus and read two consecutive 16-bit data from the data bus.

The sample code in Figure 5-3 shows a 32-bit register access with ISP1161x connected to an ISA bus in the x86 platform with two ISA ports assigned to the Host Controller of ISP1161x: the command and data ports.

```
#define DATA_PORT     0x290   // Use the PC's I/O address 0x290 for the Host Controller
                              // data port
#define COMMAND_PORT  0x292;  // Use the PC's I/O address 0x292 for the Host Controller
                              // command port

unsigned long  uReg, uRegData, uData;
```

The *HcControl* register writes the offset index.

```
uReg = 0x81;                  // HcControl write is 0x81
uRegData = 0x00010020;
```

Write the offset index to the command port.

```
outw(COMMAND_PORT, uReg);
```

Write data to the data port.
Write the lower 16-bit data first.

```
uData = uRegData & 0x0000FFFF;
```

```
outw(DATA_PORT, uData);
```

Write the higher 16-bit data. For 16-bit register write, this step is not necessary.

```
uData = (uRegData & 0xFFFF0000) >> 16;     // AND followed by right bit shift of 16 bits
outw(DATA_PORT, uData);
```

Figure 5-3: Code Example for 32-Bit Register Write

In the preceding example, the command and data ports are 16-bit wide. The *outw()* function is an x86 assembly routine that writes a 16-bit data to the specified port.

The following example code reads data from a 32-bit register.

```
unsigned long   uRegData;
```

The *HcControl* register writes the offset index.

```
uHcControlReg = 0x01;                // HcControl register read is 0x01
```

Write the offset index to the command port.

```
outw(COMMAND_PORT, uHcControlReg);
```

Read the lower 16-bit data from the data port.

```
uData = inw(DATA_PORT);
```

Save the lower 16-bit data.

```
uRegData = uData & 0x0000FFFF;
```

Read the higher 16-bit data and concatenate the lower and higher 16-bit data.
For 16-bit read, this step is not required.

```
uData = inw(DATA_PORT);
uRegData |= (uData & 0x0000FFFF) << 16;
```

Figure 5-4: Code Example for 32-Bit Register Read

The function in*w()* is an x86 assembly routine that reads a 16-bit data from the specified port.

The code example in Figure 5-5 reads a 16-bit value from a 16-bit register.

```
unsigned long   uRegData;
```

The *HcScratch* register reads the offset index.

```
uHcScratchReg = 0x28;
```

Write the offset index to the command port.

```
outw(COMMAND_PORT, uHcScratchReg);
```

Read the 16-bit register value from the data port.

```
uData = inw(DATA_PORT);
```

Figure 5-5: Code Example for 16-Bit Register Read

The code example in Figure 5-6 writes a 16-bit value to a 16-bit register.

```
unsigned long  uData;
```

The *HcScratch* register writes the offset index.

```
uScratchReg = 0xA8;
uData = 0xAA55;
```

Write the offset index to the command port.

```
    outw(COMMAND_PORT, uScatchReg);
```

Write the 16-bit data to the data port.

```
    outw(DATA_PORT, uData);
```

Figure 5-6: Code Example for 16-Bit Register Write

Throughout this document, pseudo function calls—WRITE_32BIT_REG(), READ_32BIT_REG(), WRITE_16BIT_REG() and READ_16BIT_REG()—will be used in code examples to depict read/write access to ISP1161x internal registers.

5.3. Writing and Reading of the ATL and ITL Buffers

The ATL and ITL buffers are physically located in the FIFO buffer RAM inside ISP1161x. Each buffer contains a list of PTDs that the Host Controller hardware uses to send or receive USB packets to or from USB slave devices. As part of scheduling USB transfers, the HCD constructs PTDs in the system memory and then moves the constructed PTDs to the ATL or ITL buffer. The Host Controller hardware allows software to access each buffer as if they are separate hardware buffers. The HCD accesses the ATL buffer by using the hardware registers—*HcTransferCounter* (22H/A2H) and *HcATLBufferPort* (41H/C1H)—and the ITL buffer by using *HcTransferCounter* and *HcITLBufferPort* (40H/C0H).

The example code in Figure 5-7 shows how to write and read to and from the ATL buffer in the PIO mode.

```
void fnv1161AtlWrite(char * pbyChar, unsigned long uTotalByte)
{
        unsigned long           uTotalDoubleWord;
        unsigned long          * puLong;
        unsigned long           uIndex;
        unsigned long           uData1;
        unsigned long           uData2;
```

Program the 16-bit transfer counter register: *HcTransferCounter*. Make sure that bit 7 of the register offset index is 1.

```
        outw(COMMAND_PORT, HcTransferCounter | 0x80);
        outw(DATA_PORT, uTotalByte);
```

Express the total number of bytes to be transferred in terms of double word.
Typecast the byte aligned data buffer to double word aligned buffer.

```
        uTotalDoubleWord = uTotalByte >> 2;
        puLong = (unsigned long *) pbyChar;
```

Write the *HcATLBufferPort* register offset index to the command port.
Make sure that bit 7 of the register offset index is 1.

```
        outw(COMMAND_PORT ,HcATLBufferPort | 0x80));
```

Wait a while before writing data bytes. Each *iodelay()* causes 1 system tick delay.
There must be a minimum of 300 ns delay between the command and data phases.

```
        iodelay();
        iodelay();
        iodelay();
```

Disable all hardware interrupts during the data write.

```
        cli();
```

Write data to the ATL buffer by writing to the data port 16 bits at a time.

```
        for (uIndex = 0; uIndex < uTotalDoubleWord; uIndex ++)
        {
                uData1 = puLong[uIndex] & 0x0000FFFF;
                uData2 = (puLong[uIndex] & 0xFFFF0000) >> 16;

                outw(DATA_PORT,uData1);         // Write lower 16-bit first
                outw(DATA_PORT,uData2);         // Write higher 16-bit
```

```
            // There must be a minimum of a 112 ns delay between data phases.
            iodelay();
        }
        Enable all hardware interrupts when the write is done.
        sti();
}
```

Figure 5-7: Code Example for Writing to the ATL Buffer

5.4. Typical Hardware Initialization Sequence

When the ISP1161x hardware is powered on, the Host Controller Driver (HCD) must go through the following hardware initialization steps to set the Host Controller into the operational state.

Note: In addition to the hardware initialization steps described later, the HCD must also initialize necessary data structures in between the hardware initialization steps. The requirements for the initialization of data structures differ depending on the underlying operating system and description of data structures is outside the scope of this document.

1. Detecting the Host Controller
2. Software resetting the Host Controller
    a. Setting the Host Controller to the RESET state
3. Configuring the *HcHardwareConfiguration* register
    a. Setting the interrupt output polarity
    b. Setting the interrupt trigger mode between level triggered and edge triggered
    c. Enabling the global interrupt INT1
    d. Setting DMA related modes, if DMA is used
        i. DACK input polarity
        ii. DREQ output polarity
4. Configuring interrupts
    a. USB specific interrupts
        i. Master interrupt enable
        ii. Root hub status change interrupt
        iii. Frame number overflow interrupt
        iv. Unrecoverable error interrupt
        v. Resume detect interrupt
        vi. Start-of-Frame (SOF) interrupt
        vii. Scheduling overrun interrupt
    b. Host Controller related interrupts
        i. Clock ready interrupt
        ii. Host Controller suspend interrupt
        iii. OPR register interrupt
        iv. All EOT interrupt
        v. ATL done interrupt

vi. SOF ITL done interrupt

5. Configuring the *HcControl* register
    a. Setting remote wake-up enable
    b. Setting remote wake-up connected
6. Configuring the *HcFmInterval* register
7. Configuring the root hub registers
    a. *HcRhDescriptorA* register
    b. *HcRhDescriptorB* register
    c. *HcRhStatus* register
8. Setting the ITL and ATL buffer lengths
9. Installing the INT1 interrupt service routine
10. Setting the Host Controller to the operational state.

5.4.1. Detecting the Host Controller

The detection of the Host Controller is done by the HCD by writing a value to the *HcScratch* register (see Table 5-2), reading from the *HcScratch* register and comparing the expected and actual values of the register. If the two values match, the HCD concludes that the Host Controller is present. The correct *HcChipID* read can also be used as an extra condition for detection of the Host Controller.

Table 5-2: HcScratch Register: Bit Allocation

READ INDEX—28H; WRITE INDEX—A8H

Bit	15	14	13	12	11	10	9	8
Symbol	Scratch[15:8]							
Reset	0	0	0	0	0	0	0	0
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	Scratch[7:0]							
Reset	0	0	0	0	0	0	0	0
Access	R/W							

The pseudocode for detecting an ISP1161x Host Controller is given in Figure 5-8.

```
WRITE_16BIT_REG(HcScratch, 0x55AA);
uData = READ_16BIT_REG(HcScratch);

if (uData == 0x55AA)
{
        uData = READ_16BIT_REG(HcChipID)

        // The high byte of the chip ID for ISP1161x.
        if (uData & 0xFF00) == 0x6100
                foundISP1161x;
)
else
        NotFoundISP1161x;
```

Figure 5-8: Code Example for Detecting the Host Controller

5.4.2. Software Resetting the Host Controller

The software reset of the Host Controller involves two steps:

1. Resetting the Host Controller

2. Setting the Host Controller to the RESET state.

The HCD resets the Host Controller by setting the HCR bit in the *HcCommandStatus* register (see Table 5-3). Since it takes a while (about 10 µs) to reset the Host Controller, the HCD must wait for at least 10 µs before it proceeds. A pseudocode for resetting the Host Controller is given in Figure 5-9.

```
// Read the contents of the HcCommandStatus register.
uValue = READ_32BIT_REG(HcCommandStatus);

// Set the HCR bit
uValue |= 0x00000001;

WRITE_32BIT_REG(hcCommandStatus, uValue);

// Wait until reset is done. When reset is done, the HCR bit is set to logic 0.
While (READ_32BIT_REG(HcCommandStatus & 0x00000001));
```

Figure 5-9: Code Example for Resetting the Host Controller

Table 5-3: HcCommandStatus Register: Bit Allocation

READ INDEX—02H; WRITE INDEX—82H

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	00H							
Access	R							
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved						SOC[1:0]	
Reset	0	0	0	0	0	0	0	0
Access	R						R	
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved							HCR
Reset	0	0	0	0	0	0	0	0
Access	R/W							

Once the Host Controller is reset, the HCD must set the Host Controller to the RESET state by writing 00B to the HCFS field in the *HcControl* register (see Table 5-4). This step completes resetting of the Host Controller.

```
uValue = READ_32BIT_REG(HcControl);

// When writing a new value to the HcControl register, the state of other bits in the register
// must be preserved by writing 1 to the bits already set to 1 in the register.
uValue &= ~0x000000C0;

// 00B in bit[7:6] => RESET state
uValue |= 0x00000000

WRITE_32BIT_REG (HcControl, uValue);
```

Figure 5-10: Code Example for Setting the Host Controller to the RESET State

Table 5-4: HcControl Register: Bit Allocation

READ INDEX—01H; WRITE INDEX—81H

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	00H							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	23	22	21	20	19	18	17	16
Symbol	reserved							
Reset	00H							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	15	14	13	12	11	10	9	8
Symbol	reserved					RWE	RWC	reserved
Reset	0	0	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	7	6	5	4	3	2	1	0
Symbol	HCFS[1:0]		reserved					
Reset	0	0	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

HCFS (Host Controller Functional State)—Bits[7 to 6]

- 00B—RESET
- 01B—RESUME
- 10B—OPERATIONAL
- 11B—SUSPEND.

5.4.3. Configuring the HcHardwareConfiguration Register

This register controls the characteristics of the Host Controller hardware behavior. The bit settings in this register vary depending on how the system board is designed. All bits except bits[12:10] have a power-up value. Bits[12:10] must be set properly depending on how the system board is designed. The bit[0] controls the state of the INT1 pin of the Host Controller, which is the interrupt output pin for the Host Controller side of ISP1161x. For interrupts to be enabled in the Host Controller, the bit[0] must be set. With the bit[0] of the *HcHardwareConfiguration* register set to logic 1, the bits in the *HcμPInterruptEnable* register control the activation of each interrupt available in the Host Controller.

Table 5-5: HcHardwareConfiguration Register: Bit Allocation

READ INDEX—20H; WRITE INDEX—A0H

Bit	15	14	13	12	11	10	9	8
Symbol		reserved		2_DownstreamPort15Kresistorsel	SuspendClkNotStop	AnalogOCEnable	reserved	DACKMode
Reset	0	0	0	0	0	0	0	0
Access		R/W		R/W	R/W	R/W	R/W	R/W
Bit	7	6	5	4	3	2	1	0
Symbol	EOTInputPolarity	DACKInputPolarity	DREQOutputPolarity	DataBusWidth		InterruptOutputPolarity	InterruptPinTrigger	InterruptPinEnable
Reset	0	0	0	0	1	0	0	0
Access	R/W	R/W	R/W	R/W		R/W	R/W	R/W

The bit description of the *HcHardwareConfiguration* register is given in Table 5-6.

Table 5-6: HcHardwareConfiguration Register: Bit Description

Bit	Symbol	Description
15 to 13	-	reserved
12	2_DownstreamPort15KresistorSel	**0** — use external 15 kΩ resistors for downstream ports **1** — use built-in resistors for downstream ports
11	SuspendClkNotStop	**0** — Clk can be stopped **1** — clock can not be stopped
10	AnalogOCEnable	**0** — use external OC detection. Digital input **1** — use on-chip OC detection. Analog input
9	-	reserved
8	DACKMode	**0** — normal operation. DACK1 is used with read and write signals. Power-up value. **1** — reserved
7	EOTInputPolarity	**0** — active LOW. Power-up value **1** — active HIGH
6	DACKInputPolarity	**0** — active LOW. Power-up value **1** — active HIGH
5	DREQOutputPolarity	**0** — active LOW **1** — active HIGH. Power-up value
4 to 3	DataBusWidth[1:0]	**01** — 16 bits **Others** — reserved
2	InterruptOutputPolarity	**0** — active LOW. Power-up value **1** — active HIGH
1	InterruptPinTrigger	**0** — interrupt is level-triggered. Power-up value **1** — Interrupt is edge-triggered.
0	InterruptPinEnable	**0** — power-up value **1** — pin Global Interrupt INT1 is enabled

In the ISA-based ISP1161x evaluation board, all bit fields can be set to power-up values except the InterruptOutputPolarity bit. The interrupt output polarity is active HIGH. The following code example programs the *HcHardwareConfiguration* register for the ISA-based ISP1161x evaluation board connected to a personal computer (PC) motherboard.

```
#define         INTERRUPT_PIN_ENBLE           0x0001  // INT1 pin in ISP1161x
#define         INTERRUPT_OUTPUT_POLARITY     0x0004  // Active HIGH
ULONG           uData;

// Read the register.
uData = READ_16BIT_REG(HcHardwareConfiguration);

// Active HIGH enables global interrupt pin INT1.
uData |= (INTERRUPT_PIN_ENABLE | INTERRUPT_OUTPUT_POLARITY);

WRITE_16BIT_REG(HcHardwareConfiguration, uData);
```

Figure 5-11: Code Example for Initializing the HcHardwareConfiguration Register

5.4.4. Configuring Interrupts

The Host Controller in ISP1161x has two groups of interrupt sources. The first group includes interrupts generated by USB events, such as Start-of-Frame, scheduling overrun and root hub status change. The occurrence of these interrupts is controlled by the combination of the *HcInterruptEnable* and *HcInterruptDisable* registers, and the status of each of these interrupts is indicated in the *HcInterruptStatus* register.

Table 5-7: HcInterruptEnable Register: Bit Allocation

READ INDEX—04H; WRITE INDEX—84H

Bit	31	30	29	28	27	26	25	24
Symbol	MIE				reserved			
Reset	0	0	0	0	0	0	0	0
Access	R/W							
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved	RHSC	FNO	UE	RD	SF	reserved	SO
Reset	0	0	0	0	0	0	0	0
Access	R/W							

The second group includes interrupts that occur as a result of changes in the state of the Host Controller. For example, the suspension of the Host Controller generates an interrupt. Also, any combination of interrupts in the first group is a source for an interrupt included in the second group. Figure 5-12 shows the relationship between these two groups of interrupts.

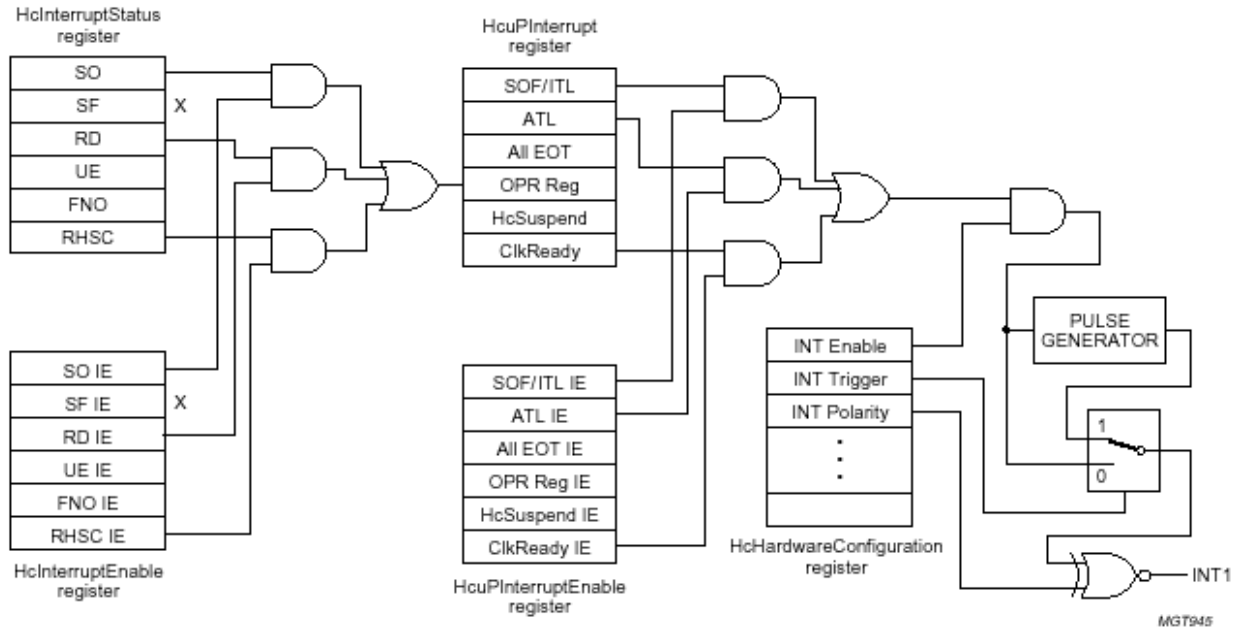


Figure 5-12: ISP1161x Host Controller Interrupt Logic

As can be seen in the block diagram, the propagation of the first group of interrupts that can be enabled via the *HcInterruptEnable* register is controlled by the OPRInterruptEnable bit in the *HcμPInterruptEnable* register (see Table 5-8).

Table 5-8: HcμPInterruptEnable Register: Bit Allocation

READ INDEX—25H; WRITE INDEX—A5H

Bit	15	14	13	12	11	10	9	8
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved	ClkReady	HC Suspended Enable	OPR Interrupt Enable	reserved	EOT Interrupt Enable	ATL Interrupt Enable	SOF Interrupt Enable
Reset	0	0	0	0	0	0	0	0
Access	R/W							

When initializing the interrupts available in the Host Controller of ISP1161x, it is recommended that you initialize the interrupts in the *HcμPinterruptEnable* register before initializing the interrupts in the *HcInterruptEnable* register. The following code segment initializes all the interrupts in the Host Controller.

```
#define OPR_Reg         0x0010
#define SOFITLInt       0x0001

// Clear all pending interrupts.
WRITE_16BIT_REG(Hc•PInterrupt, 0xFFFF);
```

```
// Enable the OPR and SOF interrupts.
WRITE_16BIT_REG(HcdPInterruptEnable, OPR_Reg | SOFITLInt);

// Disable all USB specific interrupts.
WRITE_32BIT_REG(HcInterruptDisable, 0x0000007F);

#define SF            0x00000004
#define RHSC    0x00000040
#define MIE     0x80000000

// Enable the SOF and Master Interrupts.
WRITE_32BIT_REG(HcInterruptEnable, SF | RHSC | MIE);
```

Figure 5-13: Code Example for Initializing the Host Controller Interrupt

To clear pending USB specific interrupts (that is, the first group of interrupts), a value of 1 must be written to the interrupt bit position to be cleared in the *HcInterruptStatus* register (see Table 5-9). For example, the following code clears the root hub status change (RHSC) interrupt bit:

```
WRITE_32BIT_REG(HcInterruptStatus, RHSC);
```

Table 5-9: HcInterruptStatus Register: Bit Allocation

READ INDEX—03H; WRITE INDEX—83H

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved	RHSC	FNO	UE	RD	SF	reserved	SO
Reset	0	0	0	0	0	0	0	0
Access	R/W							

To clear pending Host Controller related interrupts (that is, the second group of interrupts), a value of 1 must be written to the interrupt bit position to be cleared in the *HcμPInterrupt* register (see Table 5-10). For example, the following code clears the OPR_Reg interrupt:

```
WRITE_16BIT_REG(Hc•PInterrupt, OPR_Reg);
```

Table 5-10: HcμPInterrupt Register: Bit Allocation

READ INDEX—24H; WRITE INDEX—A4H

Bit	15	14	13	12	11	10	9	8
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved	ClkReady	HC Suspended	OPR_Reg	reserved	AllEOT Interrupt	ATLInt	SOFITLInt
Reset	0	0	0	0	0	0	0	0
Access	R/W							

Note: Since ISP1161x is a frame-based slave Host Controller, the microprocessor must update Philips Transfer Descriptors (PTD) in the ATL and/or ITL buffers for every frame. It is strongly recommended that the SOFITLInt interrupt (enabled in the *HcμPInterruptEnable* register) in conjunction with the SF interrupt (enabled in the *HcInterruptEnable* register) be used as a means to update PTDs in the ATL and ITL buffers. When the SOFITLInt interrupt is used, the ATLInt interrupt must be disabled because enabling the ATLInt interrupt results in two interrupts occurring in every frame.

5.4.5. Configuring the *HcFmInterval* Register

The recommended values for FrameInterval (FI) and FSLargestDataPacket (FSMPS) are 0x2EDF and 0x2778, respectively. Therefore, the following code will write these two values to the register.

```
WRITE_32BIT_REG(HcFmInterval, 0x2EDF | (0x2778 << 16));
```

Table 5-11 HcFmInterval Register: Bit Allocation

READ INDEX—0DH; WRITE INDEX—8DH

Bit	31	30	29	28	27	26	25	24
Symbol	FIT				FSMPS[14:8]			
Reset	0	0	0	0	0	0	0	0
Access	R/W				R/W			
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol				FSMPS[7:0]				
Reset	0	0	0	0	0	0	0	0
Access				R/W				
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved				FI[13:8]			
Reset	0	0	1	0	1	1	1	0
Access	R/W				R/W			
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol				FI[7:0]				
Reset	1	1	0	1	1	1	1	1
Access				R/W				

5.4.6. Configuring Root Hub Registers

At the time of initialization, the following three root hub specific registers must be initialized: *HcRhDescriptorA*, *HcRhDescriptorB* and *HcRhStatus*.

In the *HcRhDescriptorA* register (see Table 5-12), all bit fields except the DT bit are implementation specific (IS). For the ISA-based ISP1161x evaluation board, the following bit fields must be initialized as given:

- The recommended value for the POTPGT (PowerOnToPowerGoodTime) field is 25, which gives 50 ms power-on-to-power-good time.
- The OCPM (OverCurrentProtectionMode) bit must be set to logic 0 because the overcurrent status is reported collectively for all downstream ports.

Table 5-12: HcRhDescriptorA Register: Bit Allocation

READ INDEX—12H; WRITE INDEX—92H

Bit	31	30	29	28	27	26	25	24
Symbol	POTPGT[7:0]							
Reset	IS							
Access	R/W							
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved							
Reset	00H							
Access	R/W							
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol		reserved		NOCP	OCPM	DT	NPS	PSM
Reset	0	0	0	IS	IS	0	IS	IS
Access		R		R/W	R/W	R	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol			reserved				NDP[1:0]	
Reset	0	0	0	0	0	0	IS	
Access			R				R	

The code example to initialize the *HcRhDescriptorA* register for the ISA-based ISP1161x evaluation board is given in Figure 5-14.

```
#define POWER_ON_TO_POWER_GOOD_TIME   50
ULONG          uData = 0;

uData = 0x00000200 ;

// Must use an even value.
uData |= ((POWER_ON_TO_POWER_GOOD_TIME / 2) << 24);

WRITE_32BIT_REG (HcRhDescriptorA, uData);
```

Figure 5-14: Code Example for Initializing the HcDescriptorA Register

With the *HcRhDescriptorA* register initialized, the LPSC (LocalPowerStatusChange) bit in the *HcRhStatus* register (see Table 5-13) must be set to logic 1 to turn on power to all ports because the power switching mode is set to global power-on in the *HcRhDescriptorA* register. All other bits in the *HcRhStatus* register are set to logic 0.

```
// LPSC <= 1
uData = 0x00010000

WRITE_32BIT_REG(HcRhStatus, uData);
```

Figure 5-15: Code Example for Initializing the HcRhStatus Register

Table 5-13: HcRhStatus Register: Bit Allocation

READ INDEX—14H; WRITE INDEX—94H

Bit	31	30	29	28	27	26	25	24
Symbol	CRWE				reserved			
Reset	0				0			
Access	W				R			
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol			reserved				CCIC	LPSC
Reset			0				0	0
Access			R				R/W	R/W
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	DRWE				reserved			
Reset	0	0	0	0	0	0	0	0
Access	R/W				R			
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol			reserved				OCI	LPS
Reset	0	0	0	0	0	0	0	0
Access			R				R	R/W

For the ISA-based ISP1161x evaluation board, the PPCM field (see Table 5-14) must be set to logic 0 because the power switching mode is global power-on and the DR field (see Table 5-14) must also be set to logic 0 because devices can be detached from the root hub ports.

The code example to initialize the *HcRhDescriptorB* register is as follows:

```
WRITE_32BIT_WRITE(HcRhDescriptorB, 0x00000000);
```

Table 5-14: HcRhDescriptorB Register: Bit Allocation

READ INDEX—13H; WRITE INDEX—93H

Bit	31	30	29	28	27	26	25	24
Symbol					reserved			
Reset	-	-	-	-	-	-	-	-
Access	-	-	-	-	-	-	-	-
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol			reserved				PPCM[2:0]	
Reset	-	-	-	-	-		IS	
Access	-	-	-	-	-	R/W	R/W	R/W
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol					reserved			
Reset	-	-	-	-	-	-	-	-
Access	-	-	-	-	-	-	-	-
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol			reserved				DR[2:0]	
Reset	-	-	-	-	-		IS	
Access	-	-	-	-	-	R/W	R/W	R/W

5.4.7. Setting the ITL and ATL Buffer Lengths

The Host Controller in ISP1161x has 4 kbytes of internal FIFO buffer RAM that can be divided into the ATL and ITL buffers by the *HcATLBufferLength* and *HcITLBufferLength* registers. The ITL buffer is further divided into the ITL0 and ITL1 buffers of equal size, programmed in the *HcITLBufferLength* register, to form a ping pong structure. At minimum, the ATL buffer must exist because the ATL buffer is used for the control, interrupt and Bulk transfers. The presence of

the ITL buffer is optional. The following code example sets the ATL buffer length to 2 KB and the ITL buffer length to 2 KB, which results in the ITL0 and ITL1 buffers being 1 KB each.

```
WRITE_16BIT_REG(HcITLBufferLength, 1024);
WRITE_16BIT_REG(HcATLBufferLength, 2048);
```

Figure 5-16: Code Example for Setting the ATL and ITL Buffer Lengths

5.4.8. Installing INT1 Interrupt Service Routine

If one or more interrupts occur in the Host Controller, the microprocessor is alerted of interrupts through the INT1 pin in ISP1161x. The INT1 pin is usually connected to an interrupt controller through which the microprocessor receives an interrupt from ISP1161x. In the ISA-based ISP1161x evaluation board, the INT1 pin is an input to the two-chip cascaded 8259A programmable interrupt controller (PIC) in the PC motherboard. On the ISA-based ISP1161x evaluation board, the INT1 pin is usually set to IRQ10. When the ISP1161x evaluation board is used in the PC motherboard, the HCD must program the 8259A PIC so that the INT1 pin is connected to the IRQ10 channel in the PIC. The following code example programs the cascaded PICs on the PC motherboard.

```
#define        PIC1_OCW1      0x21    // ISA port address for operation control word 1 in the
                                      // first PIC
#define        PIC2_OCW1      0xA1    // ISA port address for operation control word 1 in the
                                      // second PIC

void configurePIC (ULON       uIrqLevel)
{
ULONG    PICMaskBit[]={1, 2, 4, 8, 16, 32, 64, 128};
ULONG    uData;
ULONG    uIntPort;

       // Set the mask bit for the corresponding IRQ level.
       // Read the current mask bits from the operation control word 1 in PIC
       // and set the mask bit for the IRQ level for INT1.
       if (uIrqLevel < 8)
        {
              // If the IRQ level for INT1 is between IRQ0 and IRQ7
              uData = (ULONG) inb(PIC1_OCW1);
              uData |= PicMaskBit[uIntLevel];
              outb(PIC1_OCW1, uData);
        }
       else
       {      // If the IRQ level for INT1 is between IRQ8 and IRQ15
              uData = (ULONG) inb(PIC2_OCW1);
              uData |= PicMaskBit[uIrqLevel - 8];
              outb(PIC2_OCW1, uData);
       }

       // Set the interrupt triggering mode to level triggering by setting the appropriate bit
       // in the ELCR register in the PIC.
       if (uIrqLevel < 8)
              uIntPort = 0x4d0;
       else
       {
              uIntPort = 0x4d1;
              uIrqLevel -= 8;
       }
       uData = (ULONG) inb(uIntPort);

       uData |= PicMaskBit[uIntLevel];
       outb(uIntPort, uData);

}
```

Once the interrupt controller is properly configured, an interrupt service routine must be installed for the target interrupt request level. The facility to connect an interrupt service routine to a particular interrupt request level is usually provided by the host operating system. For example; in Linux®, the system call *request_irq()* is used to install an interrupt service routine.

5.4.9. Setting the Host Controller to the Operational State

The next step in initializing the Host Controller is to set the Host Controller to the "operational" state from the "reset" state. The transition from the "operational" state to the "reset" state causes the Host Controller to start generating Start-of-Frame (SOF) at 1 ms intervals. The following code sets the Host Controller to the "operational" state.

```
uValue = READ_32BIT_REG(HcControl);

// When writing a new value to the HcControl register, the state of the other bits in the
// register must be preserved by writing 0 to the bits already set to logic 1 in the register.

uValue &= 0x000000C0;

// 10B in bits[7:6] => Operational state
uValue |= 0x00000080

WRITE_32BIT_REG (HcControl, uValue);
```

Figure 5-17: Code Example for Setting the Host Controller to the Operational State

5.4.10. Setting the Host Controller to Perform USB Enumeration

Upon setting the relevant registers as mentioned earlier, the Host Controller is ready to perform USB enumeration. For more detailed information on USB enumeration, refer to the *Universal Serial Bus Specification Revision 2.0 (full-speed).*

The pseudocode is as follows.

```
// Performs enumeration of the USB device connected to ISP1161x //
if (HcRhPortStatus[i] & 0x00000001)   // Detection of the connected device
{
wait_ms(100);                 // Wait at least 100 ms to allow completion of insertion
write_32bit_reg(HcRhPortStatus[i], 0x00000010);     // Set port reset

wait_ms(10);                  // Wait for reset recovery time. Min is 10 ms.
port_enable();                // Set HcRh registers to enable USB ports
               {
               write_32bit_reg(HcRhPortStatus1,0x00000102); // Set Port1 PortEnableStatus and
                                                            // PortPowerStatus to '1'
               write_32bit_reg(HcRhPortStatus2,0x00000102); // Set Port2 PortEnableStatus and
                                                            // PortPowerStatus to '1'
               write_32bit_reg(HcRhDescriptorA,0x00000B01); // Set NumberDownstreamPort,
                                                            // OCProtection etc. to '1'
               write_32bit_reg(HcRhDescriptorB,0x00000000); // Device removable and control
                                                            // by Global power switch
               }
       void set_address(old_addr, new_addr);        // A unique device address has been assigned
               {
               // Send out first control Setup packet
               make_control_ptd(cbuf_ptr, SETUP, 0, 0, 8, 0, old_addr);
               send_control(cbuf_ptr,rb_ptr,0x0500,new_addr,0x0000,0x0000);

               // Send out control Status packet
               make_control_ptd(cbuf_ptr,IN,0,0,0,1,old_addr);
send_control(cbuf_ptr,rb_ptr,0x0000,0x0000,0x0000,0x0000);  // Send zero-length packet to
                                                           // complete transfer
               }
       void set_config(int addr,int config)                 // Configure the device
{
 // Send out first control Setup packet
 make_control_ptd(cbuf_ptr,SETUP,0,0,8,0,addr);
 send_control(cbuf_ptr,rb_ptr,0x0900,config,0x0000,0x0000);

 // Send out control Status packet
 make_control_ptd(cbuf_ptr,IN,0,0,0,1,addr);
 send_control(cbuf_ptr,rb_ptr,0x0000,0x0000,0x0000,0x0000); // Send zero-length packet to
                                                            // complete transfer
}
}
//-------------------------------------------------------------------------------------------
void make_control_ptd(unsigned int *rptr, char type_ptd,char last,char ep,unsigned int max,char
tog,char addr)
{
 ptd2send.CompletetionCode=0x0;       // Set Completion Code = 0000. No Errors.
 ptd2send.active_bit=1;       // Enable execution of transactions by the Host Controller.
 ptd2send.toggle=tog;
```

```
 ptd2send.ActualBytes=0;      // Set to zero. This field is filled by the Host Controller to
                              // reflect how many bytes are sent or received.
 ptd2send.endpoint=ep;
 ptd2send.last_ptd=1;
 ptd2send.speed=port1speed;   // Indicates speed of the endpoint
 ptd2send.MaxPacketSize=max;
 ptd2send.TotalBytes=max;
 ptd2send.pid= type_ptd;
 ptd2send.format=0;
 ptd2send.fm=0;
 ptd2send.FunctionAddress=addr;

 c_ptd[0]=     (ptd2send.CompletetionCode           &0x0000)<<12
                  |(ptd2send.active_bit      &0x0001)<<11
                  |(ptd2send.toggle   &0x0001)<<10   // Shift bit 10 bits to the left
                  |(ptd2send.ActualBytes   &0x03FF); // 10 bits of ActualBytes in bytes 0 and 1
                                                     // of PTD
 c_ptd[1]=     (ptd2send.endpoint     &0x000F)<<12
                  |(ptd2send.last_ptd &0x0001)<<11
                  |(ptd2send.speed    &0x0001)<<10
                  |(ptd2send.MaxPacketSize&0x03FF);  // 10 bits of MaxPacketSize in bytes 1 and 2
                                                    // of PTD

 c_ptd[2]=     (0x0000                      &0x000F)<<12
                  |(ptd2send.pid      &0x0003)<<10
                  |(ptd2send.TotalSize       &0x03FF);      // 10 bits of TotalSize in bytes 3
                                                            // and 4 of PTD

 c_ptd[3]=     (ptd2send.fm                 &0x00FF)<<8
                  |(ptd2send.format   &0x0001)<<7
                  |(ptd2send.FunctionAddress     &0x007F);
}
//-----------------------------------------------------------------------------------------
void send_control(unsigned int *a_ptr,unsigned int *r_ptr,unsigned int d0,unsigned int
d1,unsigned int d2,unsigned int d3)
{
 abuf[0]=*(a_ptr+0);
 abuf[1]=*(a_ptr+1);
 abuf[2]=*(a_ptr+2);
 abuf[3]=*(a_ptr+3);
 abuf[4]=d0;
 abuf[5]=d1;
 abuf[6]=d2;
 abuf[7]=d3;
 nptr=abuf;
 write_atl(nptr,8);                          // Write 16 bytes
 do
  {
   if(port1speed==1){read_atl(r_ptr, 8);}    // Read 16 bytes
   if(port1speed==0){read_atl(r_ptr,36);}    // Read 72 bytes
   active_bit=(*r_ptr)&(0x0800);             // Check active bit. The Host Controller sets the
                                             // bit to 0 after PTD is finished
   active_bit=active_bit>>11;
   cnt--;
   pwait(wait_time);
  }
 while((cnt>2)   &&   (active_bit!=0));
}
//-----------------------------------------------------------------------------------------
void write_atl(unsigned int *a_ptr, unsigned int data_size)
{
 write_register16(Com16_HcTransferCounter,data_size*2);
 outport(g_1161_command_address,Com16_HcATLBufferPort|0x80);
 cnt=0;
 do
  {
   outport(g_1161_data_address,*(a_ptr+cnt));
   cnt++;
  }
 while(cnt<(data_size));
}
//-----------------------------------------------------------------------------------------
void read_atl(unsigned int *a_ptr, unsigned int data_size)
{
  write_register16(Com16_HcTransferCounter,data_size*2);
 outport(g_1161_command_address,Com16_HcATLBufferPort);
 cnt=0;
 do
```

```
  {
   *(a_ptr+cnt)=inport(g_1161_data_address);
   cnt++;
  }
 while(cnt<(data_size));
}
```

5.5. Host Controller Driver Operation Flow

The Host Controller Driver (HCD) has two functions. First, the HCD builds PTDs in a certain data structure in the system memory on being called by a higher-level component, such as the USB bus driver through its API functions. Second, the SOFITLInt interrupt service routine moves any “done” PTDs from the ATL and/or ITL buffers into the system memory and furthermore, moves pending PTDs from the system memory to the ATL and/or ITL buffers. The SOFITLInt interrupt service routine is invoked once every frame because of the SOFITLInt interrupt (see Section 5.4.4). Once the ATL and/or ITL buffers are updated, the Host Controller hardware resumes processing of PTDs in the two buffers when a new frame begins.

5.6. Accessing the ATL Buffer

The HCD can access the ATL buffer to update PTDs only when the Host Controller hardware stops scanning the buffer. The hardware stops scanning the ATL buffer under the following two conditions:

- When all the PTDs in the ATL buffer are done (The active bit in the PTD header is set to logic 0.).

  Or,

- When an ATLInt interrupt occurs; meaning that the ATL buffer scanning duration expires (FSMPS[14:0] has the value of the duration). The FSMPS[14:0] duration is typically about 85% of the duration of a 1 ms frame.

5.6.1. Using SOFITLInt Versus ATLInt

If the ISP1161x Host Controller is used for only Bulk or interrupt or both devices, the programmer has the choice of using either the SOFITLInt or ATLInt interrupt as an indication to access the ATL buffer. However, for isochronous devices, the SOFITLInt interrupt must be used because the ATLInt interrupt cannot detect 1 ms frame boundaries.

It is, therefore, strongly recommended that you enable only the SOFITLInt interrupt when building a host stack that supports all USB device types. Otherwise, there will be two interrupts—SOFITLInt and ATLInt—for every 1 ms frame.

The following timing diagram illustrates a flow of events in the context of the HCD and hardware when the ATLInt interrupt is used.

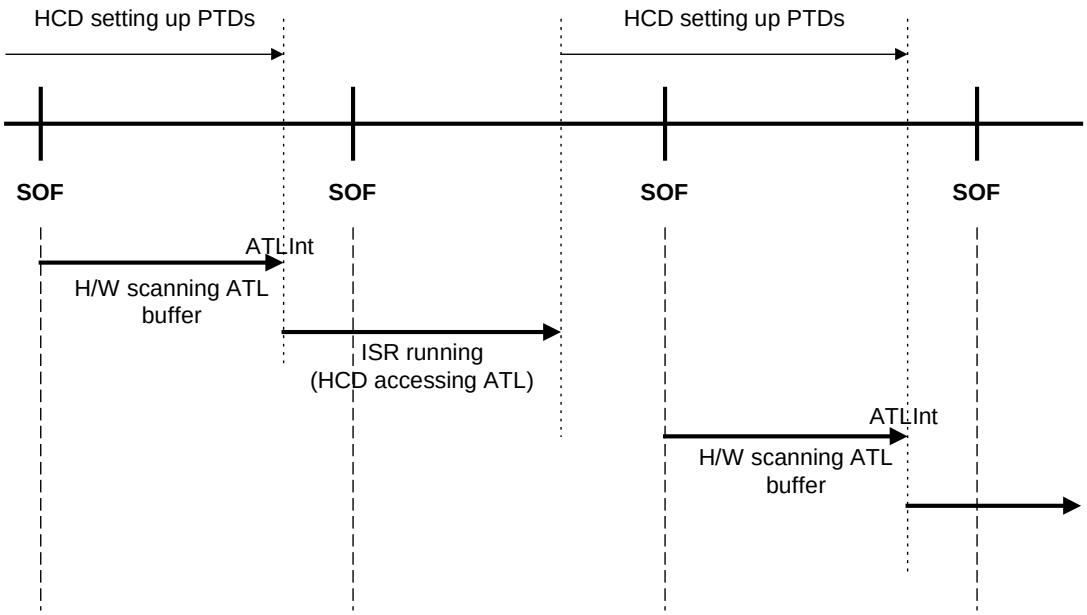


Figure 5-18: ATLInt Interrupt Flow

In the timing diagram in Figure 5-18, it is assumed that the hardware scans the ATL buffer until the FSMPS duration expires. This means that the ATL buffer still has uncompleted PTDs when the FSMPS duration expires. The ATLInt interrupt may occur sooner that the FSMPS duration time if PTDs in the ATL buffer get completed before the duration time expires. When there are no more PTDs in the ATL buffer, the ATLInt interrupt does not occur.

As can be seen from the timing diagram in Figure 5-18, there will be some time for ISR to run before the next frame starts. If ISR is done and the ATL buffer is updated with new PTDs before the next frame begins, USB transactions can occur in every frame. However, if the execution of ISR and setting up of new PTDs in the ATL buffer cross into the next frame, hardware waits until a new full frame begins. The timing diagram in Figure 5-19 illustrates the case in which USB transactions occur in every frame.

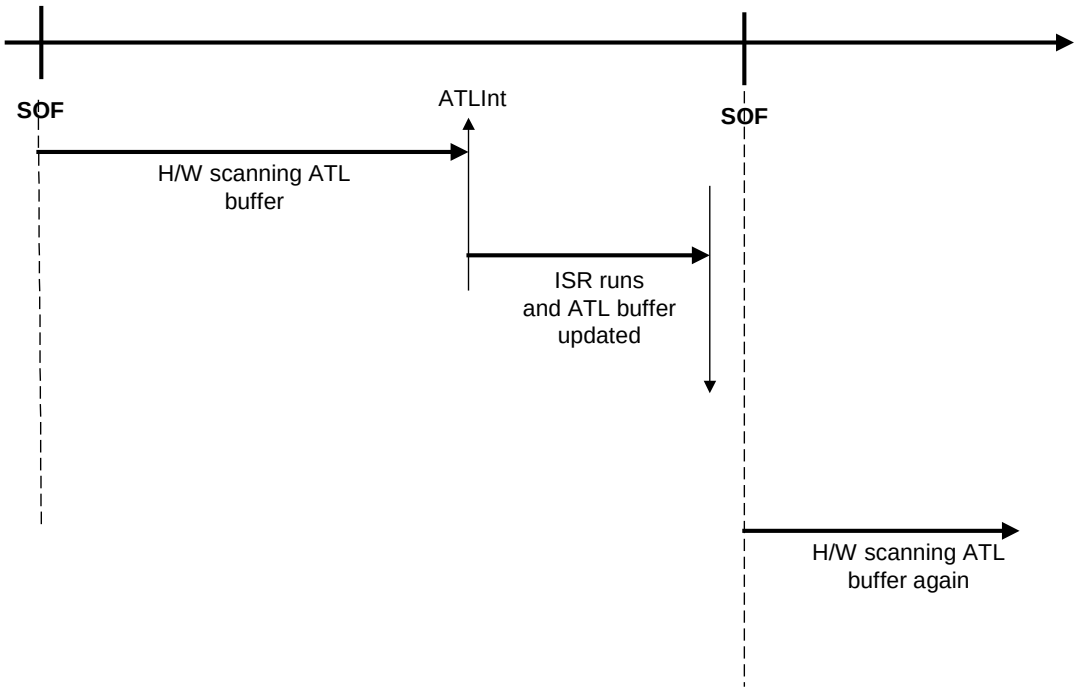


Figure 5-19: Running the Host Controller with the ATLInt Interrupt

An undesirable side effect of using the ATLInt interrupt to access the ATL buffer is that the ATLInt interrupt may interrupt the microprocessor too many times in short intervals, if the ATL buffer consistently contains PTDs that cause short USB transactions only.

Whereas using the SOFITLInt interrupt allows USB transactions to occur in every frame, using the SOFITLInt interrupt implies that USB transaction occur in every other frame, in which one frame is consumed by ISR. This is illustrated in the timing diagram in Figure 5-20.

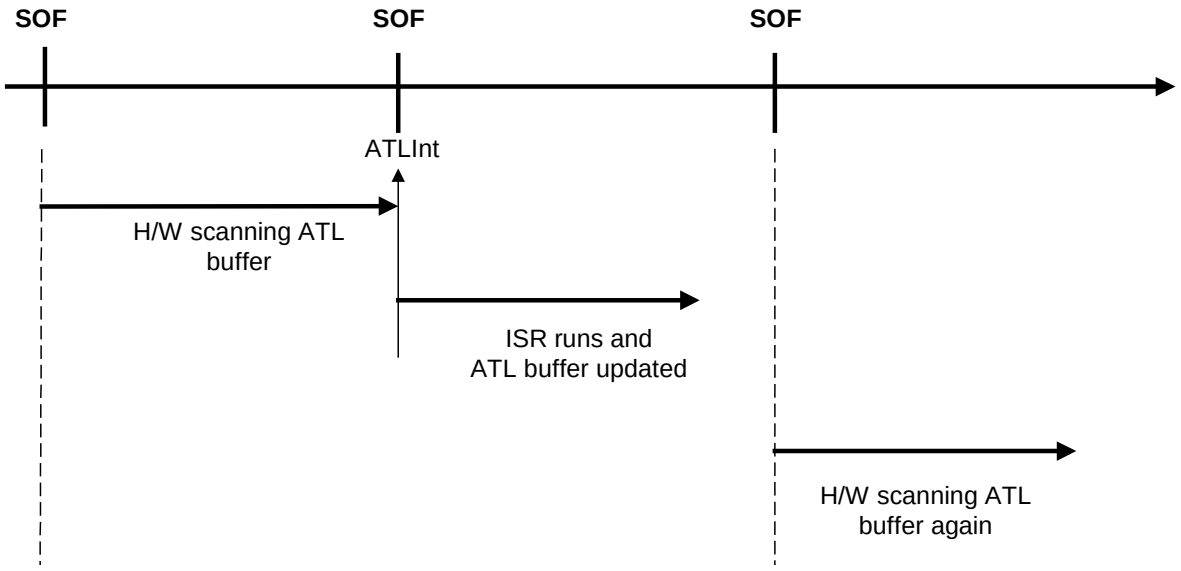


Figure 5-20: Running the Host Controller with the SOFITLInt Interrupt

5.6.2. Starting Scan of the ATL Buffer by Hardware

The Host Controller hardware starts scanning the ATL buffer when the HCD writes data to the ATL buffer via the *HcATLBufferPort* register for the number of bytes specified in the *HcTransferCounter* register. When the write is completed, the ATLBufferFull bit in the *HcBufferStatus* register is set to logic 1. The transition of the ATLBufferFull bit from logic 0 to logic 1 causes the hardware to start scanning the ATL buffer to process PTDs in the ATL buffer. When the ATLInt interrupt occurs, meaning that the hardware stops scanning the ATL buffer, the ATLBufferDone bit in the *HcBufferStatus* register is set to logic 1, which is an indication to the HCD that it can now access the ATL buffer.

The following pseudocode illustrates write to the ATL buffer.

```
void writeToATLBuffer (char * pbuffer, ULONG uTotalBytes)
{
ULONG          uTotalDoubleWord;
ULONG          * puBuffer;
ULONG          uData1, uData2, uIndex;

        // Write the length of write to the HcTransferCounter register in number of bytes.
        WRITE_32BIT_REG(HcTransferCounter, uTotalBytes)

        // Access data four bytes at a time and typecast the buffer pointer accordingly.
        uTotalDoubleWord = uTotalBytes >> 2;
        puBuffer = (ULONG *) pbuffer;

        // Send the write index of the HcATLBufferPort register to the Host Controller.
        outw(COMMAND_PORT, 0xC1)

        // Delay for 3 system ticks.
        iodelay()
        iodelay()
        iodelay()

        // Critical section. Disable all interrupts */
        DISABLE_INTERRUPTS();

        for (uIndex=0; uIndex < uTotalDoubleWord; ++uIndex)
        {
                // Get lower and higher half words.
                uData1 = puBuffer[uIndex] & 0x0000FFFF;
                uData2 = puBuffer[uIndex] & 0xFFFF0000;

                // Write lower-half word followed by higher-half word to the ATL buffer
                outw(DATA_PORT, uData1);
                outw(DATA_PORT, uData2);

                iodelay();
        }

        // Out of the critical section. Allow interrupts to happen again.
        ENABLE_INTERRUPTS();

}
```

Figure 5-21: Code Example for Writing to the ATL Buffer

The following pseudocode illustrates read from the ATL buffer.

```
void readFromATLBuffer (char * pbuffer, ULONG uTotalBytes)
{
ULONG          uTotalDoubleWord;
ULONG          * puBuffer;
ULONG          uData1, uData2, uIndex;

        // Write the length of read to the HcTransferCounter register in number of bytes.
        WRITE_32BIT_REG(HcTransferCounter, uTotalBytes)

        // Access data four bytes at a time and typecast the buffer pointer accordingly.
        uTotalDoubleWord = uTotalBytes >> 2;
        puBuffer = (ULONG *) pbuffer;

        // Send the read index of the HcATLBufferPort register to the Host Controller.
        outw(COMMAND_PORT, 0x41)

        // Delay for 3 system ticks.
```

```
        iodelay()
        iodelay()
        iodelay()

        // Critical section. Disable all interrupts */
        DISABLE_INTERRUPTS();

        for (uIndex=0; uIndex < uTotalDoubleWord; ++uIndex)
        {
                // Read lower- and higher-half words from the ATL buffer.
        uData1 =inw(DATA_PORT);
        uData2 =inw(DATA_PORT);

        // Store data into the doubleword buffer.
        puBuffer[uIndex] = (uData1 & 0x0000FFFF) | ((uData2 & 0xFFFF0000) << 16);

        iodelay();
}

// Out of critical section. Allow interrupts to happen again.
ENABLE_INTERRUPTS();

}
```

Figure 5-22: Code Example for Reading from the ATL Buffer

5.7. Accessing the ITL Buffer

The ITL buffer can be accessed by the HCD at any time because of the ping pong buffer structure of the ITL buffer. While the ping buffer is being accessed by the HCD, the Host Controller hardware can access the pong buffer and vice-versa. The timing diagram in Figure 5-23 illustrates how the ping pong buffer of the ITL buffer is accessed.

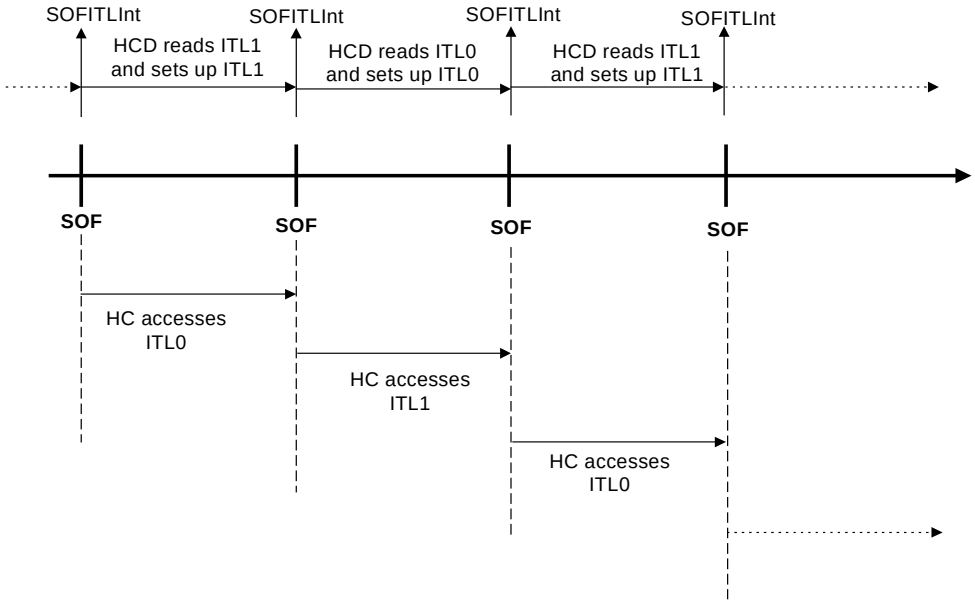


Figure 5-23: ITL Buffer Access Flow

The following code example shows how to write data from the system memory to the ITL buffer.

```
void writeToITLBuffer (char * pbuffer, ULONG uTotalBytes)
{
ULONG           uTotalDoubleWord;
ULONG           * puBuffer;
ULONG           uData1, uData2, uIndex;

        // Write the length of write to the HcTransferCounter register in number of bytes.
        WRITE_32BIT_REG(HcTransferCounter, uTotalBytes)
```

```
        // Access data four bytes at a time and typecast the buffer pointer accordingly.
        uTotalDoubleWord = uTotalBytes >> 2;
        puBuffer = (ULONG *) pbuffer;

        // Send the write index of the HcITLBufferPort register to the Host Controller.
        outw(COMMAND_PORT, 0xC0)

        // Critical section. Disable all interrupts */
        DISABLE_INTERRUPTS();

        for (uIndex=0; uIndex < uTotalDoubleWord; ++uIndex)
        {
                // Get lower- and higher-half words.
                uData1 = puBuffer[uIndex] & 0x0000FFFF;
                uData2 = puBuffer[uIndex] & 0xFFFF0000;

                // Write lower-half word followed by higher-half word to the ITL buffer.
                outw(DATA_PORT, uData1);
                outw(DATA_PORT, uData2);

        }

        // Out of critical section. Allow interrupts to happen again.
        ENABLE_INTERRUPTS();

}
```

Figure 5-24: Code Example for Writing to the ITL Buffer

The code example in Figure 5-25 shows how to read data from the ITL buffer to the system memory.

```
void readFromITLBuffer (char * pbuffer, ULONG uTotalBytes)
{
ULONG           uTotalDoubleWord;
ULONG           * puBuffer;
ULONG           uData1, uData2, uIndex;

        // Write the length of read to the HcTransferCounter register in number of bytes.
        WRITE_32BIT_REG(HcTransferCounter, uTotalBytes)

        // Access data four bytes at a time and typecast the buffer pointer accordingly.
        uTotalDoubleWord = uTotalBytes >> 2;
        puBuffer = (ULONG *) pbuffer;

        // Send the read index of the HcITLBufferPort register to the Host Controller.
        outw(COMMAND_PORT, 0x40)

        // Critical section. Disable all interrupts */
        DISABLE_INTERRUPTS();

        for (uIndex=0; uIndex < uTotalDoubleWord; ++uIndex)
        {
                // Read lower- and higher-half words from the ITL buffer.
                uData1 =inw(DATA_PORT);
                uData2 =inw(DATA_PORT);

                // Store data into the doubleword buffer.
                puBuffer[uIndex] = (uData1 & 0x0000FFFF) | ((uData2 & 0xFFFF0000) << 16);
         }

        // Out of critical section. Allow interrupts to happen again.
        ENABLE_INTERRUPTS();

}
```

Figure 5-25: Code Example for Reading from the ITL Buffer

5.8. Flowchart of the Host Controller in the Operational Mode

Once set in the operational mode, the Host Controller goes into a series of steps as shown in the flowchart in . The ITL buffer is processed first, followed by the interrupt and the ATL buffer.

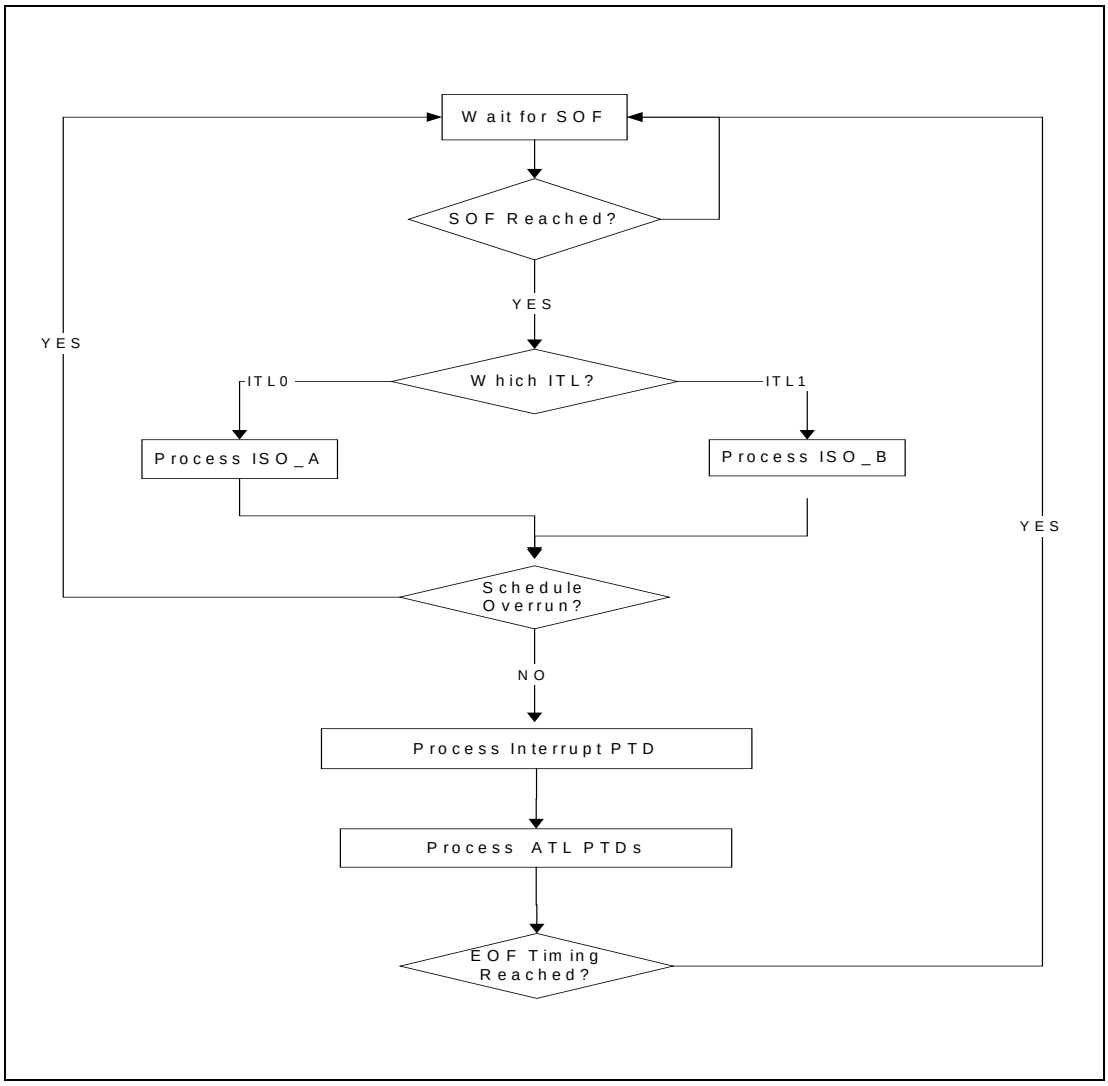


Figure 5-26: Host Controller in the Operational State Flow Chart

5.9. Setting Up PTDs for Transfers

PTDs for the control, Bulk and interrupt transfers are placed in the ATL buffer, and PTDs for the isochronous transfer are placed in the ITL buffer. In the ATL buffer, a combination of the control, Bulk and interrupt transfer PTDs can be placed in the ATL buffer destined for multiple endpoints in the same or different devices. In the ITL buffer, there can be multiple PTDs placed in the buffer for different isochronous endpoints in the same or different devices, but there must be only one PTD placed in the buffer for the same isochronous endpoint. If there happens to be more than one PTD for the same endpoint, the Host Controller hardware will send the same number of isochronous packets as that of PTDs to the same endpoint. This is a violation of the USB specification that requires one isochronous packet per frame. Since there is no hardware checking, the HCD must ensure that there is only one PTD for the same endpoint in the ITL buffer. The 8-byte PTD header fields are shown in Figure 5-27.

Bit	7	6	5	4	3	2	1	0
Byte 0	ActualBytes[7:0]							
Byte 1	CompletionCode[3:0]				Active	Toggle	ActualBytes[9:8]	
Byte 2	MaxPacketSize[7:0]							
Byte 3	EndpointNumber[3:0]				Last	Speed	MaxPacketSize[9:8]	
Byte 4	TotalBytes[7:0]							
Byte 5	reserved				DirectionPID[1:0]		TotalBytes[9:8]	
Byte 6	Format	FunctionAddress[6:0]						
Byte 7	reserved							

Symbol	Access	Description
ActualBytes[9:0]	R/W	Contains the number of bytes that were transferred for this PTD

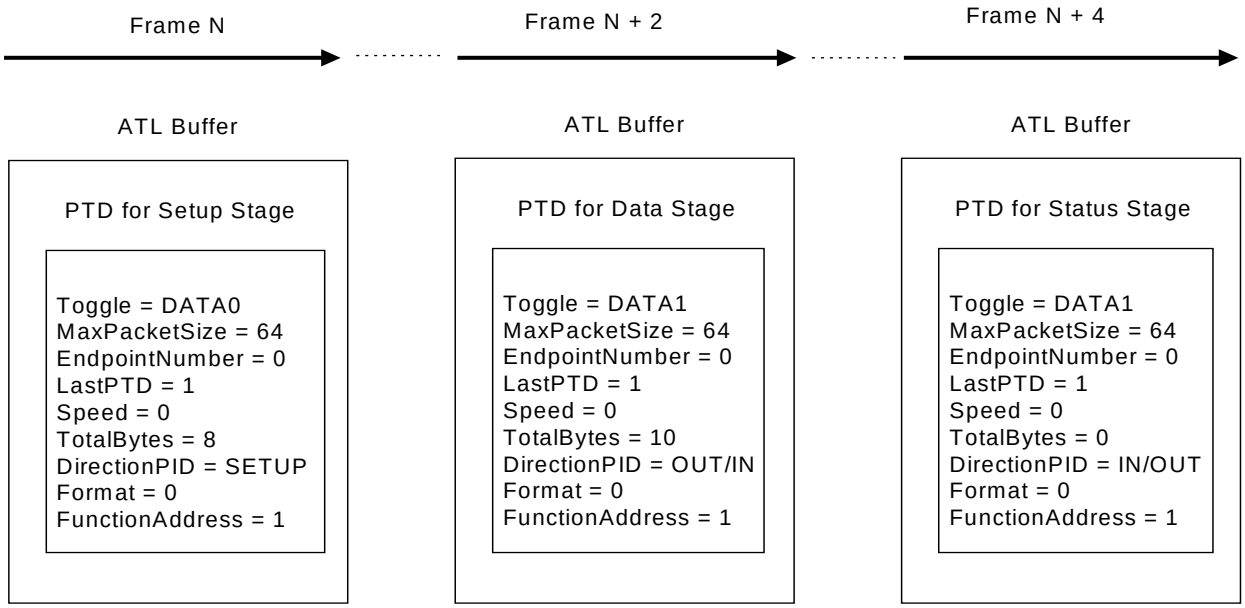
Symbol	Access			Description
CompletionCode[3:0]	R/W	0000	NoError	General TD or isochronous data packet processing completed with no detected errors.
		0001	CRC	Last data packet from endpoint contained a CRC error.
		0010	BitStuffing	Last data packet from endpoint contained a bit stuffing violation.
		0011	DataToggleMismatch	Last packet from endpoint had data toggle PID that did not match the expected value.
		0100	Stall	TD was moved to the Done queue because the endpoint returned a STALL PID.
		0101	DeviceNotResponding	Device did not respond to token (IN) or did not provide a handshake (OUT).
		0110	PIDCheckFailure	Check bits on PID from endpoint failed on data PID (IN) or handshake (OUT)
		0111	UnexpectedPID	Received PID was not valid when encountered or PID value is not defined.
		1000	DataOverrun	The amount of data returned by the endpoint exceeded either the size of the maximum data packet allowed from the endpoint (found in MaximumPacketSize field of ED) or the remaining buffer size.
		1001	DataUnderrun	The endpoint returned is less than MaximumPacketSize and that amount was not sufficient to fill the specified buffer.
		1010	reserved	-
		1011	reserved	-
		1100	BufferOverrun	During an IN, the HC received data from an endpoint faster than it could be written to system memory.
		1101	BufferUnderrun	During an OUT, the HC could not retrieve data from the system memory fast enough to keep up with the USB data rate.
Active	R/W	Set to logic 1 by firmware to enable the execution of transactions by the HC. When the transaction associated with this descriptor is completed, the HC sets this bit to logic 0, indicating that a transaction for this element should not be executed when it is next encountered in the schedule.		
Toggle	R/W	Used to generate or compare the data PID value (DATA0 or DATA1). It is updated after each successful transmission or reception of a data packet.		
MaxPacketSize[9:0]	R	The maximum number of bytes that can be sent to or received from the endpoint in a single data packet.		
EndpointNumber[3:0]	R	USB address of the endpoint within the function.		
Last(PTD)	R	Last PTD of a list (ITL or ATL). A logic 1 indicates that the PTD is the last PTD.		
(Low)Speed	R	Speed of the endpoint: **S = 0** — full speed **S = 1** — low speed		
TotalBytes[9:0]	R	Specifies the total number of bytes to be transferred with this data structure. For Bulk and Control only, this can be greater than MaximumPacketSize.		

Symbol	Access		Description
DirectionPID[1:0]	R	00	SETUP
		01	OUT
		10	IN
		11	reserved
Format	R	The format of this data structure. If this is a Control, Bulk or Interrupt endpoint, then Format = 0. If this is an Isochronous endpoint, then Format = 1.	
FunctionAddress[6:0]	R	The is the USB address of the function containing the endpoint that this PTD refers to.	

Figure 5-27: PTD Header Fields

5.9.1. Control Transfer

Control transfers require extra care by the HCD because a control transfer has two or three transaction stages—Setup, Data and Status—in which each stage must be completed in order. PTDs for the Setup, Data and Status stages cannot be placed in the ATL buffer in the same USB frame. This is because the ISP1161x Host Controller is a frame-based Host Controller, which means the Host Controller hardware tries to process as many PTDs as possible in the ATL buffer during the allotted time in a single frame. The HCD must check for the completion of the PTD for the current transaction stage before placing a PTD for the next transaction in an ensuing frame. Figure 5-29 illustrates the PTD flow for a control transfer.



- The HCD is assumed to place only one PTD in the ATL buffer for each transaction stage (LastPTD = 1).
- Device assumption:
  - 64-byte control endpoint
  - Full speed
  - Device address is 1
  - Data stage has 10-byte data

Figure 5-28: PTD Flow for the Control Transfer

In the frame N+1, the HCD will process the completed PTD for the Setup stage transaction. The HCD will process the completed PTD for the Data stage transaction in the frame N+3 (see Figure 5-28). This scenario is valid when the SOFITLInt interrupt is used as an indication to process the ATL buffer at the 1 ms interval. A control transfer may omit the Data stage transaction.

5.9.2. Bulk, Interrupt and Isochronous Transfers

DirectionPID is either IN or OUT for these transfers. TotalBytes may be larger than the length of an intended endpoint. In this case, the Host Controller hardware automatically sends an IN or OUT token preceding each max-packet-sized data packet with the correct data toggle bit for each data packet. The HCD must take care of the setting of the data toggle bit in the ensuing PTDs. The Host Controller hardware updates the data toggle bit field in the PTD only when the data packet is delivered successfully. Therefore, when the HCD retires an erroneous data packet, the HCD must take into account the fact that the data toggle bit field for the erroneous packet was left unchanged.
Figure 5-29 illustrates the setting of the data toggle bit field across multiple PTDs.

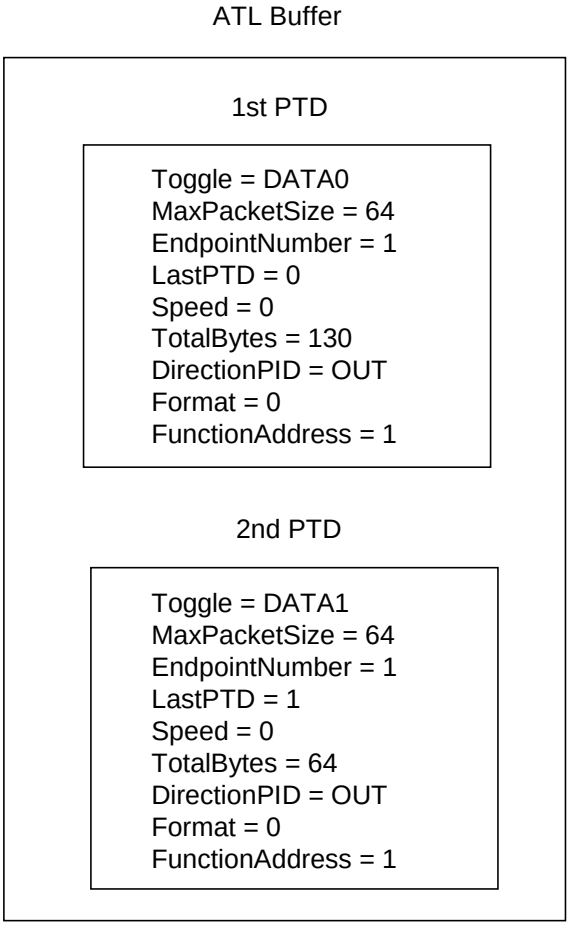


Figure 5-29: Data Toggle Bit Setting Example Across Multiple PTDs

The example above assumes the Bulk OUT endpoint size to 64 bytes. The 1st PTD has 130 bytes, and the 2nd PTD has 64 bytes to transfer to the device addressed 1. The 1st PTD will cause the Host Controller hardware to generate a total of three packets and the hardware will generate one packet from the 2nd PTD as shown in Figure 5-30.

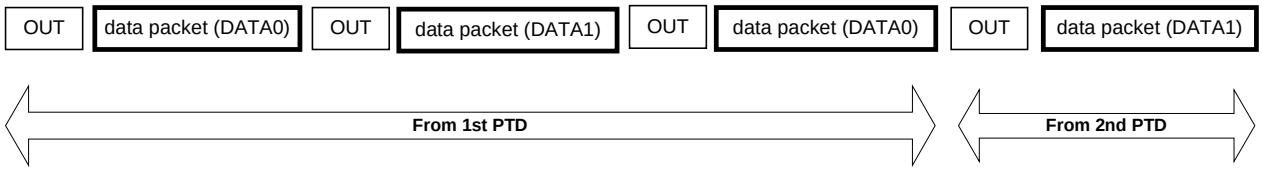


Figure 5-30: Data Toggle Bit Setting in Multiple PTD Data Packets

As shown in Figure 5-30 , the data toggle bit field must be set to DATA1 in the 2nd PTD.

5.10. Data Structures for List Processing

Before the HCD copies PTDs from the system memory to the ATL or ITL buffer, the HCD must build and keep track of PTDs through a collection of data structures. Normally, the responsibility for keeping track of the devices connected to a Host Controller lies with the bus driver. At any given point in time, the bus driver must have an understanding of what devices remain connected, what device is being disconnected and what device is being connected. Retaining this information requires elaborate data structures. Descriptions of these data structures will not be covered in this document because these are beyond the scope of the goal of this document.

The responsibility of the HCD in comparison to the bus driver is to keep track of all endpoints in all the connected devices with the attributes of each endpoint, such as the endpoint maximum packet size, the endpoint address and the device address to which an endpoint belongs. In addition, the HCD must manage the creation of new PTDs for each endpoint and the processing of the PTDs that have been completed. Employing an efficient architecture of data structures is the key to the speedy operation of a Host Controller.

One example of such a data structure would be something similar to the data structure used in the implementation of the OHCI Host Controller [*Open Host Controller Interface Specification for USB, Release: 1.0a* available at www.usb.org]. The data structure is composed of three endpoint lists (control endpoint, Bulk endpoint and interrupt endpoint), a PTD list for each endpoint and a “Done Queue” list. The interrupt endpoint list takes on a different structure as compared to the control and Bulk endpoint lists, which takes the form of a tree structure.

Each list is pointed by a global pointer variable in the absence of any hardware register that can hold the address of the first Endpoint (EP) queue head in the list (see Figure 5-31). Each EP queue header points to a PTD list. A PTD list holds PTDs waiting to be processed by the Host Controller. PTDs are moved in the ATL buffer—in the control, Bulk and interrupt transfers—by the HCD. Once PTDs are placed in the ATL buffer, the Host Controller hardware processes the PTDs in the next frame.

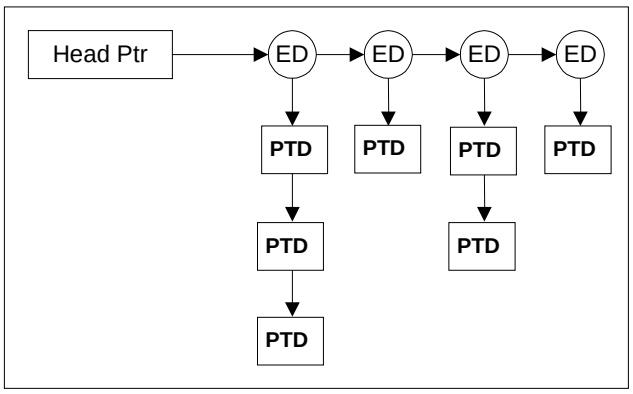


Figure 5-31: Typical List Structure

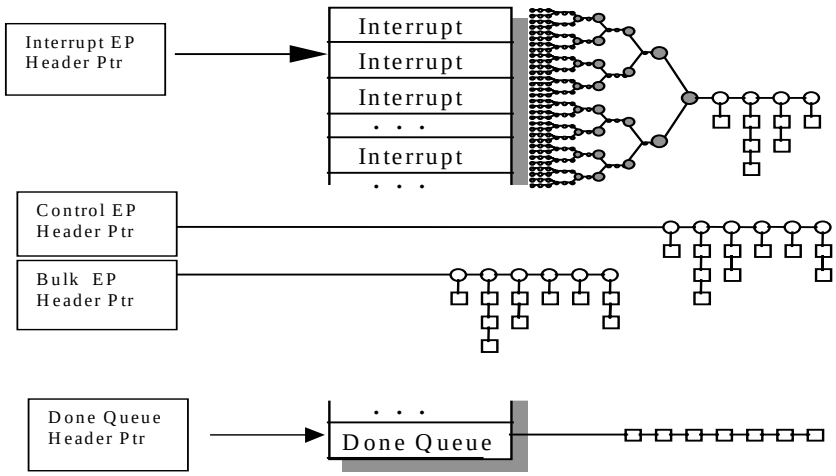


Figure 5-32: List Processing Data Structure

For more details on the algorithm for processing interrupt transfers; refer to *Open Host Controller Interface Specification for USB, Release: 1.0a.*

5.11. Error Handling

The Host Controller hardware reports any error that occurs during the execution of a PTD via the CompletionCode[3:0] field in the PTD. There are a total of 11 possible errors that can occur. Of the 11 possible errors, all except one error—data underrun error—are fatal errors that cause the USB transaction to fail. The following table lists these errors, the causes for these errors in an OUT transaction and the treatment of these errors by the Host Controller in an IN transaction.

Table 5-15: USB Transaction Error Codes

Fatal Errors	Error Code	IN Token	OUT Token
ERROR_CRC	01	No ACK sent	Not applicable
ERROR_Bitstuffing	02	No ACK sent	Not applicable
ERROR_DatatTogglingMismatch	03	ACK sent	Not applicable
ERROR_Stall	04	No ACK sent	The host received Stall from the device.
ERROR_DeviceNotResponding	05	No ACK sent	The host did not receive a handshake reply within 18-bit time, or a bad SYNC pulse.
ERROR_PIDCheckFailure	06	No ACK sent	Not applicable
ERROR_UnExpectedPID	07	No ACK sent	Corrupted ACK, STALL or NAK
ERROR_DataOverRun	08	NAK sent	Not applicable
Non-Fatal Error (Warning)			
ERROR_DataUnderRun	09	ACK sent	Not applicable
ERROR_BufferOverrun	0C	—	—
ERROR_BufferUnderrun	0D	—	—

For all errors, the data toggle bit is still toggled and updated by the Host Controller hardware. The HCD must take the state of the data toggle bit if and when it retries the failed PTD. This is because the data toggle bit is changed in spite of an error.

For more details on error handling, refer to the Software section of the ISP1161x Frequently Asked Questions document.

6. Programming the Device Controller of ISP1161x

The Device Controller (DC) of ISP1161x is a core based on Philips ISP1181 Device Controller, which is a full-speed USB interface device with up to 14 configurable endpoints. You can access the Device Controller of ISP1161x via the PIO mode or DMA transfer with up to 16-bytes per cycle. It has 2462 bytes of dedicated internal FIFO memory. The type and FIFO size of each endpoint can be individually configured, depending on the required packet size. The isochronous and Bulk endpoints are double-buffered for increased data throughput.

The Device Controller of ISP1161x can implement peripheral functions, such as printers, scanners, external mass storage (Zip® drive) devices and digital still cameras, to transfer data to and from the PC host. The system CPUs in these peripherals are extremely busy handling many tasks, such as device control, data and image processing. The firmware of the Device Controller is designed to be fully interrupt-driven. While the system CPU is doing its foreground task, the USB transfer is handled in the background. This assures best transfer rate and better software structure, and also simplifies programming and debugging.

The description on programming the Device Controller of ISP1161x is based on the firmware code of the ISP1161x ISA evaluation kit. The operating system used is DOS. Therefore, the Hardware Abstraction layer focuses on the ISA bus access.

6.1. Firmware Structure of the Device Controller

The firmware for the evaluation board consists of two major portions: the processing of information and the interrupt service routine. The Hardware Abstraction layer just moves data from hardware to memory space to be processed by the Main Loop as shown in Figure 6-1.

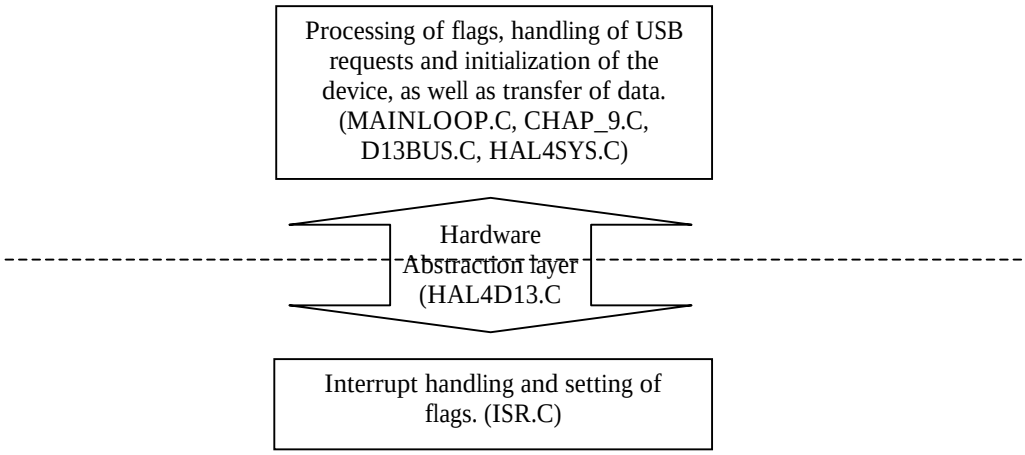


Figure 6-1: Firmware Structure of the ISP1161x Device Controller

As can be seen in Figure 6-1, the firmware structure can be divided into the following six building blocks:

- Hardware Abstraction Layer—HAL4SYS.C
- Hardware Abstraction Layer—HAL4D13.C
- Interrupt Service Routine—ISR.C
- Protocol Layer—CHAP_9.C
- Protocol Layer—D13BUS.C
- Main Loop—MAINLOOP.C.

6.1.1. Hardware Abstraction Layer—HAL4SYS.C

This is the lowest-layer code in the firmware that performs hardware-dependent I/O access of the Device Controller of ISP1161x, as well as the evaluation board hardware. When porting the firmware to other CPU platforms, this part of the code always needs modifications or additions.

6.1.2. Hardware Abstraction Layer—HAL4D13.C

To further simplify programming with the Device Controller of ISP1161x, the firmware defines a set of command interfaces that encapsulate all the functions used to access the Device Controller of ISP1161x. When porting the firmware to other operation systems, this portion of the code must be modified.

6.1.3. Interrupt Service Routine—ISR.C

This part of the code handles interrupt generated by the Device Controller of ISP1161x. It retrieves data from the ISP1161x Device Controller's internal FIFO to CPU memory and sets up proper event flags to inform the Main Loop program to process.

6.1.4. Protocol Layer—CHAP_9.C

This Protocol layer handles standard USB device request, which is defined in the Chapter 9 of USB Specification Rev. 2.0. The firmware implementation of the USB device request is described in more detail in Section 6.7.

6.1.5. Protocol Layer—D13BUS.C

This Protocol layer handles specific vendor requests. Examples are the Bulk transfer and the isochronous (ISO) transfer.

6.1.6. Main Loop—MAINLOOP.C

The Main Loop checks event flags and passes to appropriate the subroutine for further processing. It also contains the code for human interface, such as the keyboard scan.

6.2. Porting the Firmware to Other CPU Platform

Table 6-1 shows the modifications that must be done to building blocks. There are two levels of porting. The first level is the Standard Device Request, that is, USB Chapter 9 only, which is to allow the firmware to pass enumeration by supporting standard USB requests. The second level is the full product development. This involves product-specific firmware code, that is, Vendor Request.

Table 6-1: Building Blocks Modifications

File Name	Chapter 9 Only	Product Level
HAL4SYS.C	Port to hardware specific	Port to hardware specific
HAL4D13.C	Port to hardware specific	No change
ISR.C	No change	Add product specific processing to the Generic and Main endpoints
CHAP_9.C	No change	Product specific USB descriptors
D13BUS.C	No change	Add vendor request supports, if necessary
MAINLOOP.C	Depending on the CPU and the system, ports, timer and interrupt initialization must be rewritten	Add product specific Main Loop processing

6.3. Developing the Firmware in the Polling Mode

To develop the firmware in the polling mode, add the following lines of code to the Main Loop:

```
if(interrupt_pin_low)
       fn_usb_isr();
```

Normally, Interrupt Service Routine (ISR) is initiated by the hardware. In the polling mode, the Main Loop detects the status of the interrupt pin, and invokes ISR, if necessary.

6.4. Hardware Abstraction Layer

6.4.1. Hardware Abstraction Layer for the System

This layer contains the lowest-layer functions that must be changed on different CPU platforms. The function prototypes in the Hardware Abstraction layer for the system are as follows:

```
Hal4Sys_AcquireTimer0(void);
Hal4Sys_ReleaseTimer0(void);
interrupt Hal4Sys_Isr4Timer(void);

void Hal4Sys_AcquireKeypad(void);
void Hal4Sys_ReleaseKeypad(void);

void Hal4Sys_WaitinUS(IN OUT ULONG time);
void Hal4Sys_WaitinMS( IN OUT ULONG time);

void Hal4Sys_ControlLEDPattern( UCHAR LEDpattern);
void Hal4Sys_ControlD13Interrupt( BOOLEAN InterruptEN);
```

For example, the subroutine to acquire the system timer is as follows:

```
void Hal4Sys_AcquireTimer0(void)
{
       if(bD13flags.bits.verbose)
       printf("enter Hal4Sys_AcquireTimer0\n");

       Hal4Sys_OldIsr4Timer = getvect(0x8);
       setvect(0x8, Hal4Sys_Isr4Timer);

       if(bD13flags.bits.verbose)
       printf("exit Hal4Sys_AcquireTimer0\n");
}
```

6.4.2. Hardware Abstraction Layer for the Device Controller of ISP1161x

The following functions are defined as the Device Controller command interface of ISP1161x to simplify the device programming. These are implementations of the ISP1161x Device Controller command set, which is defined in the ISP1161x datasheet.

```
Hal4D13_SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex);
Hal4D13_GetEndpointConfig(UCHAR bEPIndex);

Hal4D13_SetAddressEnable(UCHAR bAddress, UCHAR bEnable);
Hal4D13_GetAddress(void);

Hal4D13_SetMode(UCHAR bMode);
Hal4D13_GetMode(void);

Hal4D13_SetDevConfig(USHORT wDevCnfg);
Hal4D13_GetDevConfig(void);

Hal4D13_SetIntEnable(ULONG dIntEn);
Hal4D13_GetIntEnable(void);

Hal4D13_SetDMAConfig(USHORT wDMAConfig);
Hal4D13_GetDMAConfig(void);
Hal4D13_SetDMACounter(USHORT wDMACounter);
Hal4D13_GetDMACounter(void);

Hal4D13_ResetDevice(void);

Hal4D13_WriteEndpoint(UCHAR bEPIndex, UCHAR * buf, USHORT len);
Hal4D13_ReadEndpoint(UCHAR bEPIndex, UCHAR * buf, USHORT len);

Hal4D13_SetEndpointStatus(UCHAR bEPIndex, UCHAR bStalled);
Hal4D13_GetEndpointStatusWInteruptClear(UCHAR bEPIndex);
Hal4D13_ValidBuffer(UCHAR bEPIndex);
Hal4D13_ClearBuffer(UCHAR bEPIndex);

Hal4D13_AcknowledgeSETUP(void );

Hal4D13_GetErrorCode(UCHAR bEPIndex);
Hal4D13_LockDevice(UCHAR bTrue);

Hal4D13_ReadChipID(void);
Hal4D13_ReadCurrentFrameNumber(void);

Hal4D13_ReadInterruptRegister(void);
```

6.5. Interrupt Service Routine

The Device Controller of the ISP1161x firmware is fully interrupt-driven. The flowchart of Interrupt Service Routine (ISR) is given in Figure 6-2.

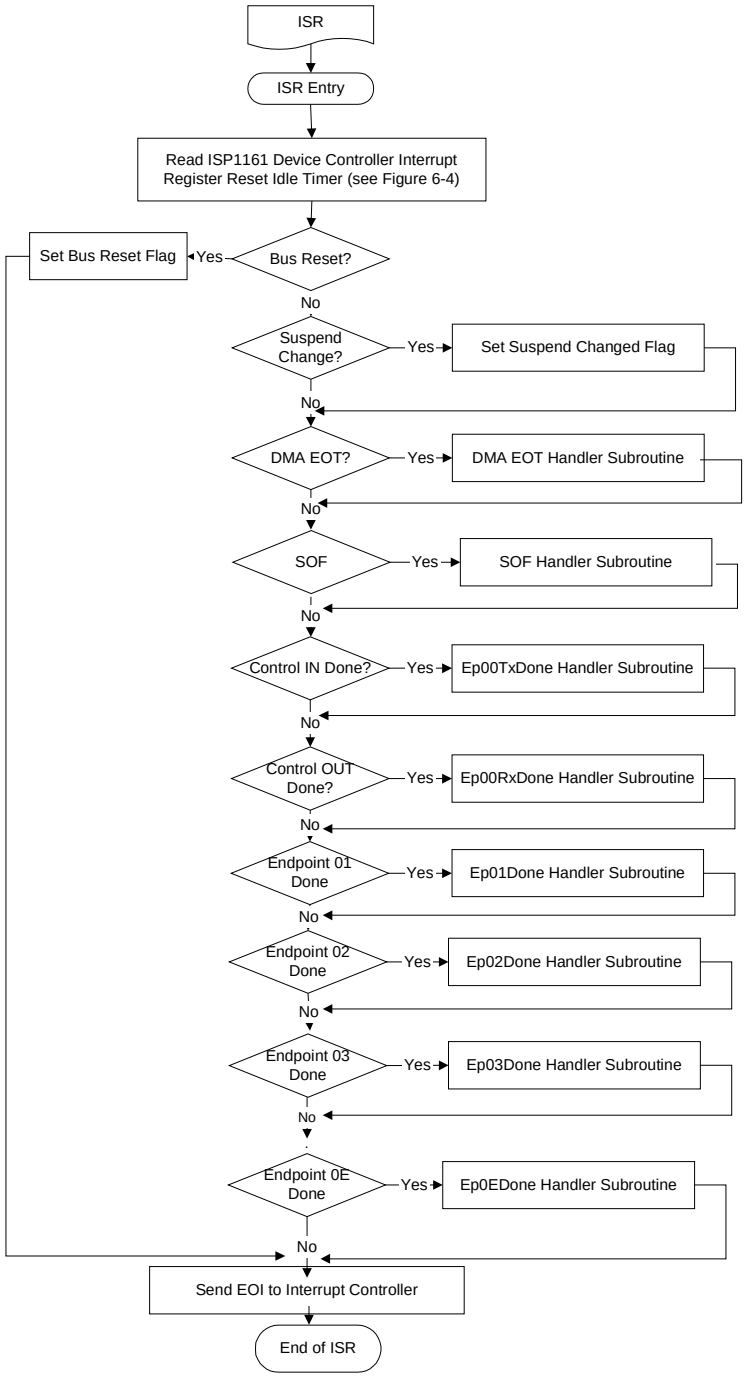


Figure 6-2: Flowchart of ISR

Table 6-2: Interrupt Register: Bit Allocation

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	R	R	R
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	EP14	EP13	EP12	EP11	EP10	EP9	EP8	EP7
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	R	R	R
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EP6	EP5	EP4	EP3	EP2	EP1	EP0IN	EP0OUT
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	R	R	R
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	BUSTATUS	SP_EOT	PSOF	SOF	EOT	SUSPND	RESUME	RESET
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	R	R	R

Note: A logic 1 indicates that an interrupt occurred on the respective bit.

Figure 6-3 contains the pseudocode of a typical Interrupt Service Routine.

```
void fn_usb_isr(void)
{
        ULONG   i_st;

        i_st = ReadInterruptRegister();  /* See Figure 6-4 on reading the Interrupt register */
        if(i_st != 0) {

                        if(i_st & D13REG_INTSRC_BUSRESET)
                                Isr_BusReset();

                        else if(i_st & D13REG_INTSRC_SUSPEND)
                                Isr_SuspendChange();  /* This function sets suspend changed flag */

                        else if(i_st & D13REG_INTSRC_EOT)
                                Isr_DmaEot(); /* DMA EOT handler subroutine */

                        else if(i_st & (D13REG_INTSRC_SOF|D13REG_INTSRC_PSEUDO_SOF))
                                Isr_SOF();  /* SOF handler subroutine */

                        else
                        {
                                if(i_st & D13REG_INTSRC_EP0IN)
                                        Isr_Ep00TxDone();       /* Ep00TxDone handler subroutine */
                                                                /* (control IN EP) */
                                if(i_st & D13REG_INTSRC_EP0OUT)
                                        Isr_Ep00RxDone();       /* Ep00RxDone handler subroutine */
                                                                /* (control OUT EP) */
                                if(i_st & D13REG_INTSRC_EP01)
                                        Isr_Ep01Done();         /* Ep01Done handler subroutine */
                                if(i_st & D13REG_INTSRC_EP02)
                                        Isr_Ep02Done();         /* Ep02Done handler subroutine */
                                if(i_st & D13REG_INTSRC_EP03)
                                        Isr_Ep03Done();         /* Ep03Done handler subroutine */
                                /* Add interrupts as and when needed */

                                if(i_st & D13REG_INTSRC_EP0E)
                                        Isr_Ep0EDone();         /* Ep0EDone handler subroutine */
                        }
                        }
}
```

Figure 6-3: Code Example of a Typical ISR

A pseudocode to read the Interrupt register is given in Figure 6-4.

```
ULONG ReadInterruptRegister(void)
{
        ULONG i = 0;
        outport(D13_COMMAND_PORT, Read_Int_Register);   /* Read the Read_Int_Register = 0xC0 */
        i = inport(D13_DATA_PORT);                      /* Read the lower word */
        i += (((ULONG)inport(D13_DATA_PORT)) << 16);    /* OR the lower word with the upper */
                                                        /* word to form a ULONG variable */
        return i;                                        /* Return the Interrupt register */
}
```

Figure 6-4: Code Example to Read the Interrupt Register

At the entrance of ISR, the firmware uses the Read Interrupt register to decide the source of the interrupt and then to dispatch it to the appropriate subroutines for processing. ISR communicates with the foreground Main Loop through event flags "D13FLAGS" and data buffers "CONTROL_XFER".

```
        typedef union _D13FLAGS
        {
                struct _D13FSM_FLAGS
                {

                        IRQL_1 UCHAR            bus_reset               : 1;
                        IRQL_1 UCHAR            suspend                 : 1;
                        IRQL_1 UCHAR            DCP_state               : 4;
                        IRQL_1 UCHAR            setup_dma               : 1;
                        IRQL_1 UCHAR            timer                   : 1;
                } bits;
                ULONG value;
        } D13FLAGS;

        typedef struct _CONTROL_XFER
        {
                IRQL_1 DEVICE_REQUEST           DeviceRequest;
                IRQL_1 USHORT                   wLength;
                IRQL_1 USHORT                   wCount;
                IRQL_1 ADDRESS                  Addr;
                IRQL_1 UCHAR                    dataBuffer[MAX_CONTROLDATA_SIZE];

        } CONTROL_XFER, * PCONTROL_XFER;

Where,
        typedef struct _device_request
        {
                UCHAR bmRequestType;
                UCHAR bRequest;
                USHORT wValue;
                USHORT wIndex;
                USHORT wLength;
        } DEVICE_REQUEST;
```

Figure 6-5: Control Flags

The task splitting between ISR and the Main Loop is that ISR collects data from the internal buffer of the ISP1161x Device Controller and moves the data packet to a data buffer. When ISR has collected enough data, it informs the Main Loop that data is ready for processing. The Main Loop processes the data from the data buffer.

The following sections explain the various event handlers.

6.5.1. Bus Reset

The bus reset does not require any special processing within ISR. ISR sets the “bus_reset” flag in D13FLAGS and then exits.

6.5.2. Suspend Change

Suspend does not require special processing within ISR. ISR sets the suspend flag in D13FLAGS and then exits.

6.5.3. EOT Handler

For information on EOT handler, contact the Philips Semiconductors' support team at wired.support@philips.com

6.5.4. Control Endpoint Handler

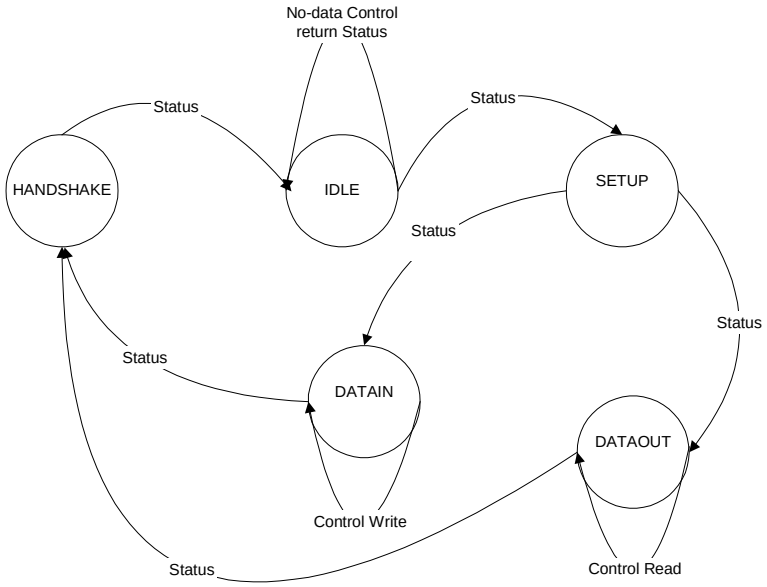


Figure 6-6: State Machine of the Control Transfer

The control transfer always begins with the Setup stage and is followed by an optional Data stage. The Data stage can be one or more IN or OUT transactions. Finally, it ends with the Status stage, that is, HANDSHAKE. Figure 6-6 shows the various states of transitions on control endpoints. The firmware uses these five states to handle the control transfer correctly.

6.5.5. Control OUT Handler

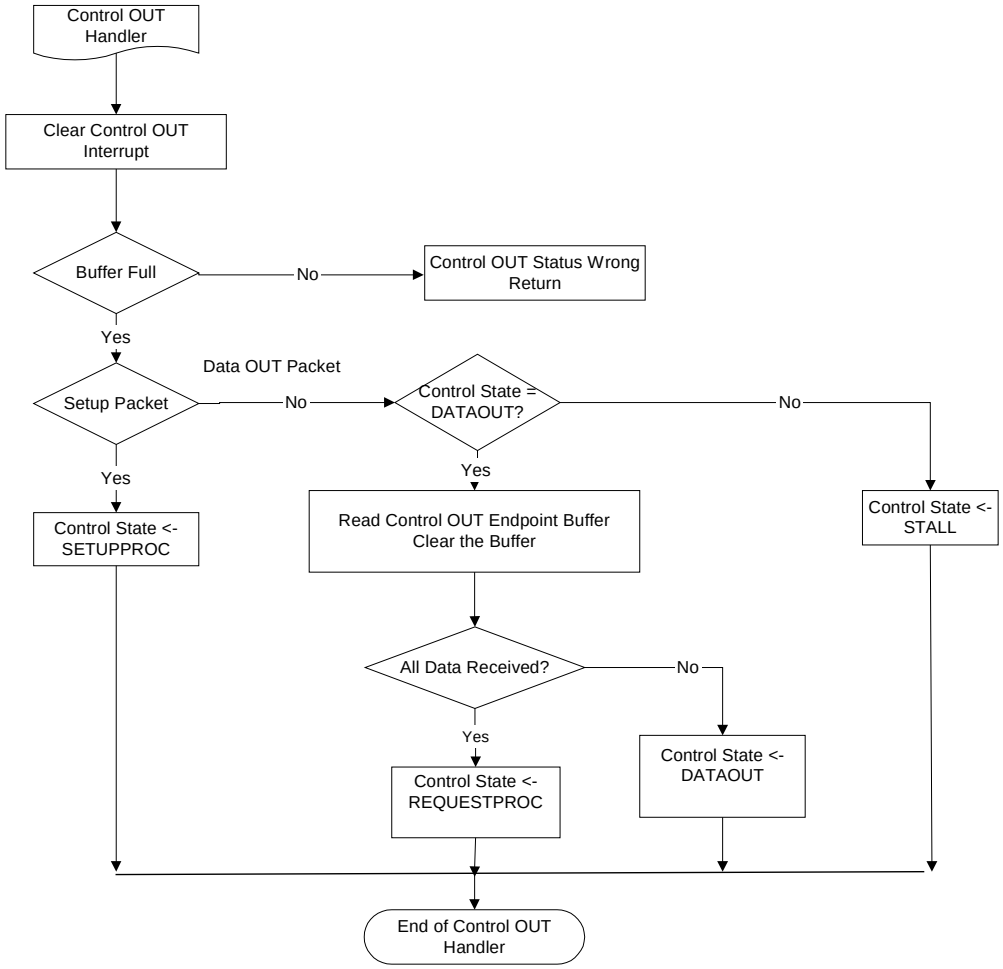


Figure 6-7: Flowchart of the Control OUT Handler

The microprocessor must clear the control OUT interrupt bit on the Device Controller of ISP1161x and verify whether this endpoint is full. Figure 6-8 contains a pseudocode to check whether the OUT endpoint is full. This is done by issuing a Read Endpoint Status command (code 0x50) that clears the control OUT interrupt bit of the Interrupt register, and at the same time returns status information. Figure 6-9 shows a pseudocode to read the Endpoint Status register (see Table 6-3 and Table 6-4). This clears the corresponding endpoint interrupt. If the status information reports a Setup packet (SETUPT bit (bit 2) of the Endpoint Status register), the "SETUPPROC" state will be set for the Main Loop to process. Otherwise, the microprocessor extracts the content of the data OUT packet buffer by reading the control endpoint. Figure 6-10 contains a pseudocode to read the contents of an OUT buffer. After making sure all the data is received, the handler sets the Device Controller of ISP1161x to the "REQUESTPROC" state.

```
EP_Status = Read_Endpoint_Status(0x00) /* Endpoint status of EP0 */
if(EP_Status & 0x20)                   /* Check whether the primary buffer is full or not */
{
       /* Proceed with the program flow */
}
```

Figure 6-8: Code Example to Check Status of the OUT Endpoint

```
UCHAR Read_Endpoint_Status( UCHAR EPIndex)
{
        UCHAR c;
        outport(D13_COMMAND_PORT, READ_EP_ST + EPIndex); /* READ_EP_ST = 0x50 */
        c = (UCHAR)(inport(D13_DATA_PORT) & 0x0ff);
        return c;
}
```

Figure 6-9: Code Example for Reading the Endpoint Status Register

A typical pseudocode to read the contents of an OUT buffer is given in Figure 6-10.

```
USHORT Read_Endpoint (UCHAR EPIndex , USHORT* PTR , USHORT LENGTH)
{
        USHORT  j,i;
        /* Select endpoint */
        outport(D13_COMMAND_PORT , READ_EP+EPIndex);    /* READ_EP = 0x10 */
        j = inport(D13_DATA_PORT);     /* Read the length in bytes inside the OUT buffer */
        if( j > LENGTH)
                j = LENGTH;
        for(i=0 ; i<j ; i++)
        {       /* Read buffer */
                *(PTR+i) = inport(D13_DATA_PORT);
        }
        /* Clear buffer */
        outport(D13_COMMAND_PORT , CLEAR_BUFF+ EPIndex);   /* CLEAR_BUFF = 0x70 */
        return j;
}
```

Figure 6-10: Code Example for Reading the Contents of an OUT Buffer

Table 6-3: Endpoint Status Register: Bit Allocation

Bit	7	6	5	4	3	2	1	0
Symbol	EPSTAL	EPFULL1	EPFULL0	DATA_PID	OVER WRITE	SETUPT	CPUBUF	reserved
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	R	R	R

Table 6-4: Endpoint Status Register: Bit Description

Bit	Symbol	Description
7	EPSTAL	This bit indicates whether the endpoint is stalled or not (1 = stalled, 0 = not stalled). Set to logic 1 by a Stall Endpoint command, cleared to logic 0 by an Unstall Endpoint command. The endpoint is automatically unstalled upon reception of a SETUP token.
6	EPFULL1	A logic 1 indicates that the secondary endpoint buffer is full.
5	EPFULL0	A logic 1 indicates that the primary endpoint buffer is full.
4	DATA_PID	This bit indicates the data PID of the next packet (0 = DATA PID, 1 = DATA1 PID).
3	OVERWRITE	This bit is set by hardware, a logic 1 indicating that a new Setup packet has overwritten the previous setup information, before it was acknowledged or before the endpoint was stalled. This bit is cleared by reading, if writing the setup data has finished. Firmware must check this bit before sending an Acknowledge Setup command or stalling the endpoint. Upon reading a logic 1 the firmware must stop ongoing setup actions and wait for a new Setup packet.
2	SETUPT	A logic 1 indicates that the buffer contains a Setup packet.
1	CPUBUF	This bit indicates which buffer is currently selected for CPU access (0 = primary buffer, 1 = secondary buffer).
0	-	reserved

6.5.6. Control IN Handler

After the Setup stage is complete, the host executes the Data phase. If the Device Controller of ISP1161x receives a control IN packet, it will go to the “control IN handler”. The microprocessor must first clear the control IN interrupt bit of the ISP1161x Device Controller by reading its Read Endpoint Status code (Code 0x51). Figure 6-11 shows a pseudocode to read the Endpoint Status register. This clears the corresponding endpoint interrupt. Using the Endpoint status, it can determine whether the IN buffer is empty or full. Figure 6-12 contains a pseudocode to check whether the IN endpoint is empty or not. After verifying that the Device Controller of ISP1161x is in the appropriate state, the microprocessor proceeds to send the data packet, see Figure 6-13.

Figure 6-14 shows the flowchart of the control IN handler. Since the Device Controller of the ISP1161x control endpoint has only 64 bytes FIFO, the microprocessor must control the amount of data during the transmission phase, if the requested length is more than 64 bytes. As indicated in the flowchart, the microprocessor must check its current and remaining data size to be sent to the host. If the remaining data size is greater than 64 bytes, the microprocessor will send the first 64 bytes and then subtract the reference length (requested length) by 64. When the next control IN token comes, the microprocessor determines whether the remaining byte is zero. If there is no more data to be sent, the microprocessor must send an empty packet to inform the host that there is no more data to be sent.

```
UCHAR Read_Endpoint_Status( UCHAR EPIndex)
{
        UCHAR c;
        outport(D13_COMMAND_PORT, READ_EP_ST + EPIndex);     /* READ_EP_ST = 0x50 */
        c = (UCHAR)(inport(D13_DATA_PORT) & 0x0ff);
        return c;
}
```

Figure 6-11: Code Example for Reading the Endpoint Status Register

```
EP_Status = Read_Endpoint_Status(0x01) /* Endpoint status of EP1 */
if(!(EP_Status & 0x20))              /* Check whether the primary buffer is empty or not */
{
       /* Proceed with the program flow */
}
```

Figure 6-12: Code Example to Check the Status of the IN Endpoint

```
USHORT Write_Endpoint (UCHAR EPIndex , USHORT* PTR , USHORT LENGTH)
{
USHORT  i;

/* Select the endpoint */
outport(D13_COMMAND_PORT , WRITE_EP+EPIndex);  /* WRITE_EP = 0x00 ; EPIndex = 0x01 */
outport (D13_DATA_PORT , LENGTH);  /* Write the length of the data into the IN buffer */

/* Write the buffer */
for(i=0 ; i<LENGTH ; i++)
outport(D13_DATA_PORT , *(PTR+i) );

/* Validate buffer */
outport(D13_COMMAND_PORT, EP_VALID_BUF+bEPIndex);  /* EP_VALID_BUF =0x60 ; EPIndex = 0x01 */

return j;
}
```

Figure 6-13: Code Example for Writing the Contents to an IN Buffer

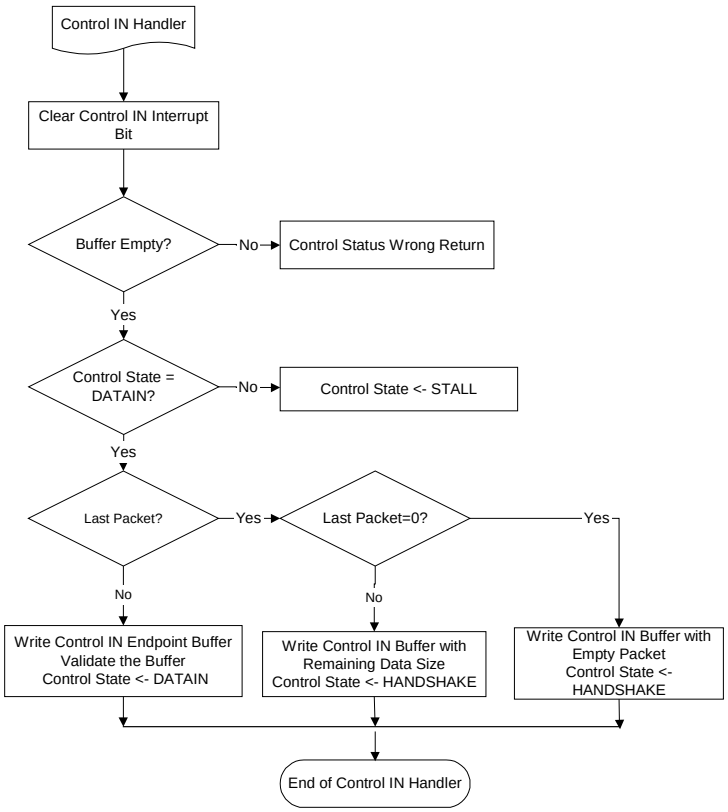


Figure 6-14: Flowchart of the Control IN Handler

Note: OUT and IN data transactions differ slightly in implementation. The control OUT handler and the control IN handler are called during a control OUT interrupt event and a control IN interrupt event, respectively. When the control OUT interrupt event occurs, it signifies that the host has already sent data to the control OUT endpoint. This OUT interrupt is the trigger to start reading from the buffer. However, for the control IN, the payload is first written in the IN endpoint, and then validated.

6.5.7. Bulk Endpoint Handler

The Device Controller of ISP1161x has 16 endpoints: control IN and OUT plus 14 configurable endpoints. The 14 endpoints can be individually defined as interrupt, Bulk or isochronous, IN or OUT. The size of the FIFO determines the maximum packet size that the hardware can support for a given endpoint. Table 6-5 shows the recommended register programming of the Endpoint Configuration register for a Bulk endpoint. The bit allocation and bit description of the Endpoint Configuration register are given in Table 6-6 and Table 6-7, respectively.

Table 6-5: Recommended Endpoint Configuration Register Programming for a Bulk Endpoint

Bit	Bit Setting	Description
7	1	Endpoint enable bit
6	0 for OUT 1 for IN	Endpoint direction
5	1	Enable double buffering
4	0	Bulk endpoint
3 to 0	0011	Size bits of an enabled endpoint: 64 bytes

Table 6-6: Endpoint Configuration Register: Bit Allocation

Bit	7	6	5	4	3	2	1	0
Symbol	FIFOEN	EPDIR	DBLBUF	FFOISO	FFOSZ[3:0]			
Reset	0	0	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Table 6-7: Endpoint Configuration Register: Bit Description

Bit	Symbol	Description
7	FIFOEN	A logic 1 indicates an enabled FIFO with allocated memory. A logic 0 indicates a disabled FIFO (no bytes allocated).
6	EPDIR	This bit defines the endpoint direction (0 = OUT, 1 = IN); it also determines the DMA transfer direction (0 = read, 1 = write).
5	DBLBUF	A logic 1 indicates that this endpoint has double buffering.
4	FFOISO	A logic 1 indicates an isochronous endpoint. A logic 0 indicates a bulk or interrupt endpoint.
3 to 0	FFOSZ[3:0]	Selects the FIFO size according to programmable FIFO size

An example on how to configure a Bulk OUT or Bulk IN endpoint is given in Figure 6-15.

```
#define EPCNFG_FIFO_EN            0x80
#define EPCNFG_DBLBUF_EN          0x20
#define EPCNFG_NONISOSZ_64        0x03
#define EPCNFG_IN_EN              0x40

/* Configuration of Bulk OUT */
SetEndpointConfig(EPCNFG_FIFO_EN\
               |EPCNFG_DBLBUF_EN\
               |EPCNFG_NONISOSZ_64\
                , Bulk_EPIndex\   /* Ranges from 0x00 – 0x0F, depending on which endpoint you */
                                  /* configure as Bulk OUT. */
                );

/* Configuration of Bulk IN */
SetEndpointConfig(EPCNFG_FIFO_EN\
               |EPCNFG_DBLBUF_EN\
               |EPCNFG_NONISOSZ_64\
               |EPCNFG_IN_EN\
```

```
                , Bulk_EPIndex\   /* Ranges from 0x00 – 0x0F, depending on which endpoint you */
                                  /* configure as Bulk IN. */
                );
```

Figure 6-15: Code Example for Configuring a Bulk OUT or Bulk IN Endpoint

The function definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is given in Figure 6-16.

```
void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)
{
        outport(D13_COMMAND_PORT, (USHORT)(WR_EP_CONFIG+bEPIndex)); /* WR_EP_CONFIG = 0x20 */
        outport(D13_DATA_PORT,(USHORT)bEPConfig);
}
```

Figure 6-16: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)

When the host is ready to transmit the Bulk data, it issues an OUT token packet followed by a data packet. The Device Controller of ISP1161x generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1161x Device Controller and verify the data length. The flowchart of the Bulk OUT handler is given in **F**igure 6-17.

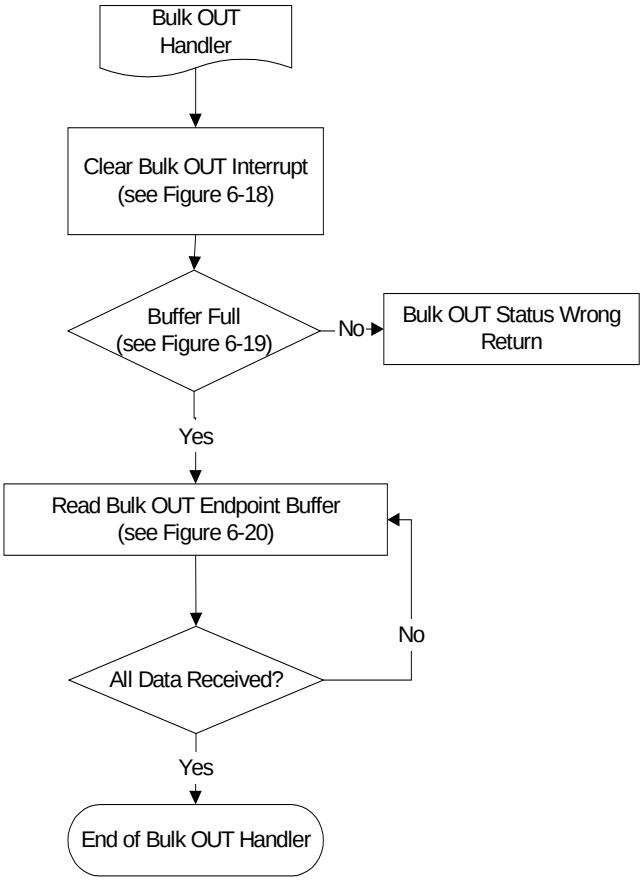


Figure 6-17: Flowchart of the Bulk OUT Handler

Figure 6-18 shows the code example for reading the Endpoint Status register. This clears the corresponding endpoint interrupt.

```
UCHAR Read_Endpoint_Status( UCHAR EPIndex)
{
        UCHAR c;
        outport(D13_COMMAND_PORT, READ_EP_ST + EPIndex);   /* READ_EP_ST = 0x50 */
        c = (UCHAR)(inport(D13_DATA_PORT) & 0x0ff);
        return c;
}
```

Figure 6-18: Code Example for Reading the Endpoint Status Register

```
/* Bulk_EPIndex ranges from 0x50 – 0x5F, depending on which endpoint you configure as Bulk */
EP_Status = Read_Endpoint_Status(BULK_EPIndex)
if(EP_Status & 0x20)            /* Check whether the primary buffer is full */
{
        /* Proceed with the program flow */
}
```

Figure 6-19: Code Example to Check the Status of the Bulk OUT Endpoint

```
USHORT Read_Endpoint (UCHAR EPIndex , USHORT* PTR , USHORT LENGTH)
{
        USHORT  j,i;
        /* Select endpoint */
        outport(D13_COMMAND_PORT , READ_EP+EPIndex);   /* READ_EP = 0x10 */
        j = inport(D13_DATA_PORT); // Read the length in bytes inside the OUT buffer
        if( j > LENGTH)
                j = LENGTH;
        /*Read the buffer */
        for(i=0 ; i<j ; i++)
                *(PTR+i) = inport(D13_DATA_PORT);

        /* Clear the buffer */
        outport(D13_COMMAND_PORT , CLEAR_BUFF+ EPIndex);  /* CLEAR_BUFF = 0x70 */
        return j;
}
```

Figure 6-20: Code Example for Reading the Contents of a Bulk OUT Buffer

When the host is ready to receive the Bulk data, it issues an IN token. The Device Controller of ISP1161x generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1161x Device Controller and return the data packet to be sent. The flowchart of the Bulk IN handler is given in Figure 6-21.

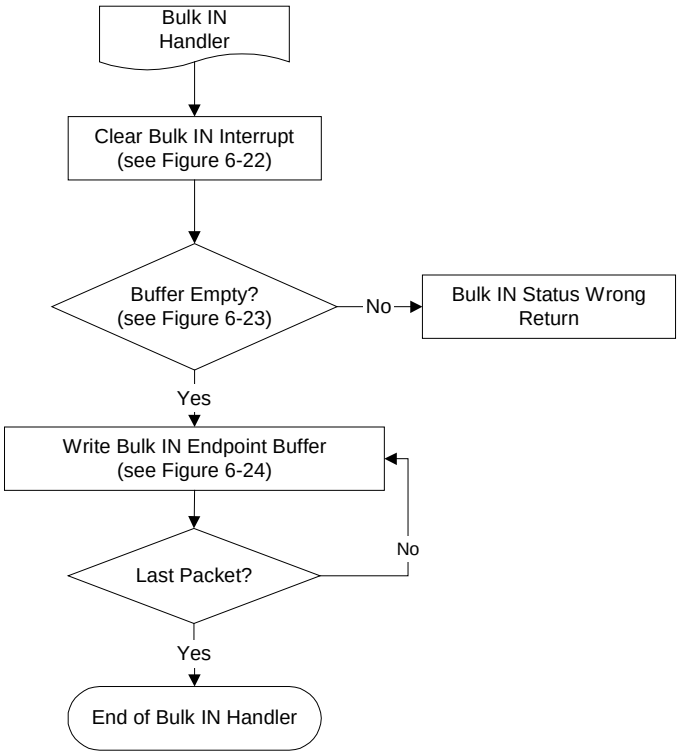


Figure 6-21: Flowchart of the Bulk IN Handler

A pseudocode for reading the Endpoint Status register is given in Figure 6-22. This clears the corresponding endpoint interrupts.

```
UCHAR Read_Endpoint_Status(UCHAR EPIndex)
{
        UCHAR c;
        outport(D13_COMMAND_PORT, READ_EP_ST + EPIndex); /* READ_EP_ST = 0x50 */
        c = (UCHAR)(inport(D13_DATA_PORT) & 0x0ff);
        return c;
}
```

Figure 6-22: Code Example for Reading the Endpoint Status Register

```
/* Bulk_EPIndex ranges from 0x50 – 0x5F, depending on which endpoint you configure as Bulk. */
EP_Status = Read_Endpoint_Status(BULK_EPIndex)
If( !(EP_Status & 0x20)) /* Check whether the primary buffer is full or not */
{
        /*Proceed with the program flow */
}
```

Figure 6-23: Code Example to Check the Status of the Bulk IN Endpoint

```
USHORT Write_Endpoint (UCHAR EPIndex , USHORT* PTR , USHORT LENGTH)
{
        USHORT  i;
        /* Select the endpoint */
        outport(D13_COMMAND_PORT , WRITE_EP+EPIndex);  /* WRITE_EP = 0x00 */
        outport (D13_DATA_PORT , LENGTH);       /* Write the length of data into the IN buffer */

        /* Write the buffer */
        for(i=0 ; i<LENGTH ; i++)
                outport(D13_DATA_PORT , *(PTR+I) );
```

```
        /* Validate the buffer */
        ?outport(D13_COMMAND_PORT, EP_VALID_BUF+bEPIndex); /* EP_VALID_BUF =0x60; */

        return j;
}
```

Figure 6-24: Code Example for Writing the Contents into a Bulk IN Buffer

6.5.8. ISO Endpoint Handler

Table 6-8 contains the recommended register programming in the Endpoint Configuration register for an ISO endpoint.

Table 6-8: Recommended Endpoint Configuration Register Programming for an ISO Endpoint

Bit	Bit Setting	Description
7	1	Endpoint enable bit
6	0 for OUT 1 for IN	Endpoint direction
5	1	Enable double buffering
4	1	ISO endpoint
3 to 0	1011	Size bits of an enabled endpoint: 512 bytes

Figure 6-25 contains an example on how to configure an ISO OUT or ISO IN endpoint.

```
#define EPCNFG_FIFO_EN                0x80
#define EPCNFG_DBLBUF_EN              0x20
#define EPCNFG_ISOSZ_512              0x0B
#define EPCNFG_IN_EN                  0x40
#define EPCNFG_ISO_EN                 0x10

/* Configuration of ISO OUT */
SetEndpointConfig(EPCNFG_FIFO_EN\
               |EPCNFG_DBLBUF_EN\
               |EPCNFG_ISOSZ_512\
               |EPCNFG_ISO_EN \
                , ISO_EPIndex\ /* Ranges from 0x00 – 0x0F, depending on which endpoint you */
                              /* configure as ISO OUT.*/
                );

/* Configuration of ISO IN */
SetEndpointConfig(EPCNFG_FIFO_EN\
               |EPCNFG_DBLBUF_EN\
               |EPCNFG_ISOSZ_512\
               |EPCNFG_ISO_EN \
               |EPCNFG_IN_EN\
                , ISO_EPIndex\   /* Ranges from 0x00 – 0x0F, depending on which endpoint you */
                                /* configure as ISO IN */
                );
```

Figure 6-25: Code Example for Configuring an ISO OUT or ISO IN Endpoint

The function definition of SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is given in Figure 6-26.

```
void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)
{
        outport(D13_COMMAND_PORT, (USHORT)(WR_EP_CONFIG+bEPIndex)); /* WR_EP_CONFIG = 0x20 */
        outport(D13_DATA_PORT,(USHORT)bEPConfig);
}
```

Figure 6-26: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)

Figure 6-27 and Figure 6-28 contains the flowcharts of the ISO OUT handler and the ISO IN handler, respectively.

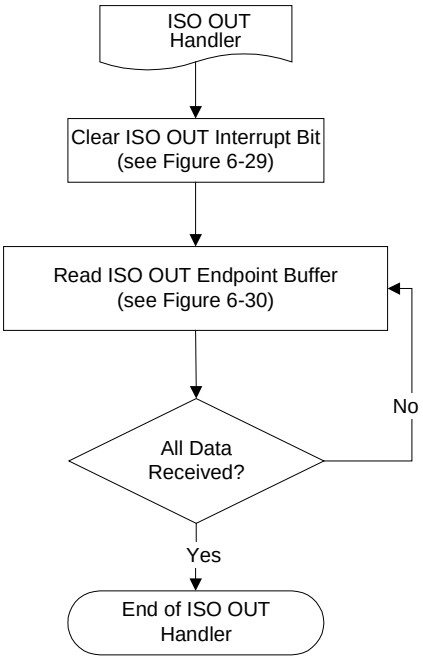


Figure 6-27: Flowchart of the ISO OUT Handler

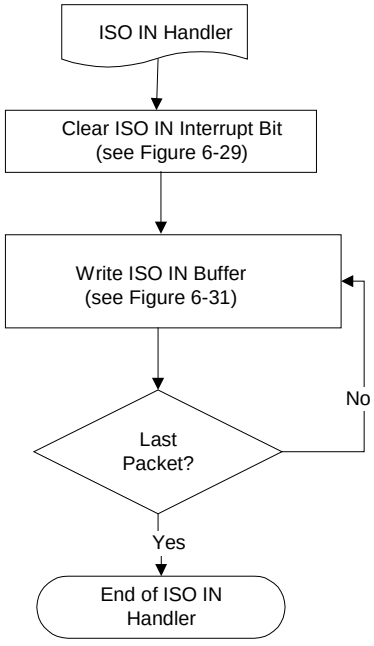


Figure 6-28: Flowchart of the ISO IN Handler

Time is a key element of an isochronous transfer. A typical example of the isochronous data is voice. All isochronous pipes move exactly one data packet in each frame, that is, every 1 ms.

A pseudocode for reading the Endpoint Status register is given in Figure 6-29. This clears the corresponding endpoint interrupts.

```
UCHAR Read_Endpoint_Status( UCHAR EPIndex)
{
        UCHAR c;
        outport(D13_COMMAND_PORT, READ_EP_ST + EPIndex); /* READ_EP_ST = 0x50 */
        c = (UCHAR)(inport(D13_DATA_PORT) & 0x0ff);
        return c;
}
```

Figure 6-29: Code Example for Reading the Endpoint Status Register

```
USHORT ReadISOEndpoint(UCHAR bEPIndex, USHORT* ptr, USHORT len)
{
        USHORT i, j;

        /* Select the endpoint */
        outport(D13_COMMAND_PORT, READ_EP+ bEPIndex);  /* READ-EP = 0x10 */
        j = inport(D13_DATA_PORT);                     /* Reading length of data in the buffer */

        if(j != len)
        j = len;

        /* Read the buffer */
        for(i=0; i<j; i++)
        *(ptr + i) = inport(D13_DATA_PORT);

        /* Clear the buffer */
        outport(D13_COMMAND_PORT, CLEAR_BUF+bEPIndex);  /* CLEAR_BUF = 0x70 */
        return j;
}
```

Figure 6-30: Code Example for Reading from an ISO Endpoint Buffer

```
USHORT WriteISOEndpoint(UCHAR bEPIndex, USHORT* ptr, USHORT len)
{
        USHORT i;
        static UCHAR j;

        /* Select the endpoint */
        outport(D13_COMMAND_PORT, WRITE_EP + bEPIndex); /* WRITE_EP = 0x00 */
        outport(D13_DATA_PORT, len);  /* Writing the length of data */

        /* Write the buffer */
        for(i=0; i<len; i=i+2)
                outport(D13_DATA_PORT, *(ptr+i) );
        /* Validate the buffer */
        outport(D13_COMMAND_PORT, VALID_BUF+bEPIndex);  /* VALID_BUF = 0x60 */
        return i;
}
```

Figure 6-31: Code Example for Writing to an ISO Endpoint Buffer

6.6. Main Loop

When power is switched on, the microprocessor must initialize its ports, memory, timer, and interrupt service routine handler. Then, the microprocessor reconnects USB, which involves setting the SOFTCT bit in the Mode register to ON. This procedure is important because it ensures that the ISP1161x Device Controller will not operate before the microprocessor is ready to serve the ISP1161x Device Controller.

The flowchart of the Main Loop is given in Figure 6-32. In the Main Loop routine, the microprocessor polls for any activity on the keyboard. If any of the specific keys is pressed, the handle key commands will execute the routine and then return to the Main Loop. This routine is added for debugging purposes only. A 1 ms timer is programmed to activate the routine to check for any key pressed on the evaluation board.

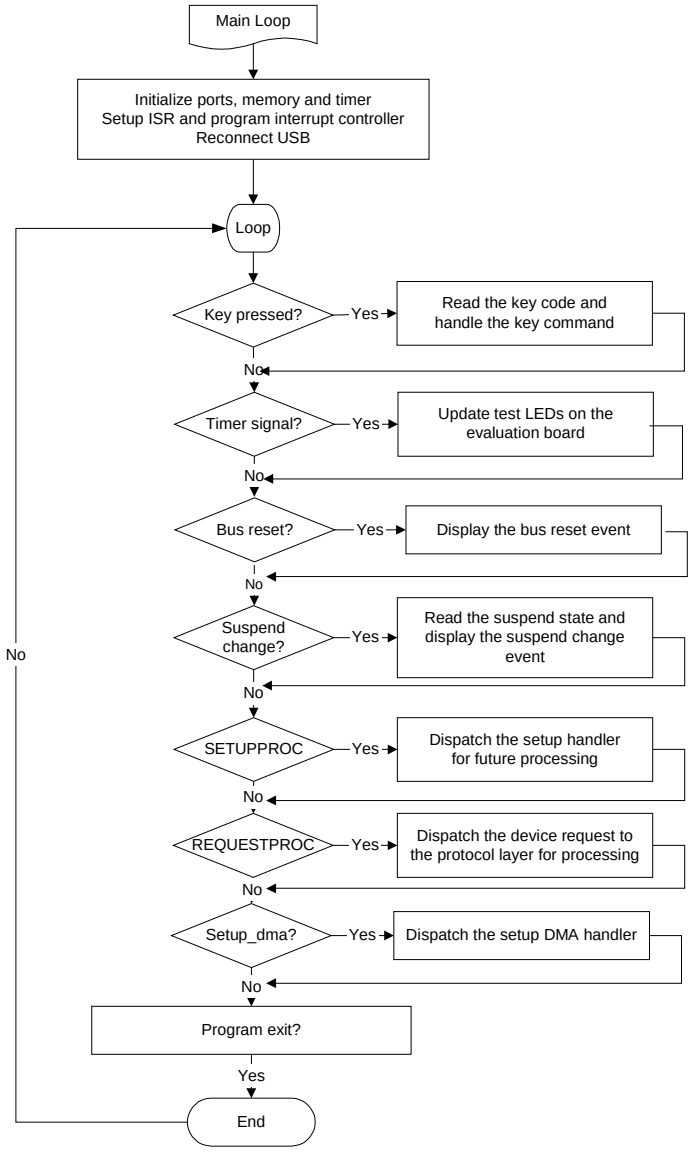


Figure 6-32: Flowchart of the Main Loop

Table 6-9: Mode Register: Bit Allocation

Bit	7	6	5	4	3	2	1	0
Symbol	DMAWD	reserved	GOSUSP	reserved	INTENA	DBGMOD	reserved	SOFTCT
Reset	0[1]	0	0	0	0[1]	0[1]	0[1]	0[1]
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

[1] Unchanged by a bus reset.

Table 6-10: Mode Register: Bit Description

Bit	Symbol	Description
7	DMAWD	A logic 1 selects 16-bit DMA bus width (bus configuration modes 0 and 2). A logic 0 selects 8-bit DMA bus width. Bus reset value: unchanged.
6	-	reserved
5	GOSUSP	Writing a logic 1 followed by a logic 0 will activate 'suspend' mode.
4	-	reserved
3	INTENA	A logic 1 enables all interrupts. Bus reset value: unchanged.
2	DBGMOD	A logic 1 enables debug mode. where all NAKs and errors will generate an interrupt. A logic 0 selects normal operation, where interrupts are generated on every ACK (bulk endpoints) or after every data transfer (isochronous endpoints). Bus reset value: unchanged.
1	-	reserved
0	SOFTCT	A logic 1 enables SoftConnect. This bit is ignored if EXTPUL = 1 in the Hardware Configuration Register. Bus reset value: unchanged.

Figure 6-33 contains a pseudocode for writing to the Mode register. An example on setting the SOFCT bit to enable SoftConnect is given in Figure 6-34.

```
void SetMode(UCHAR bMode)   // Function definition
{
       outport(D13_COMMAND_PORT, WRITE_MOD_REG); /* WRITE_MOD_REG = 0xB8 */
       outport(D13_DATA_PORT, bMode);
}
```

Figure 6-33: Code Example for Writing to the Mode Register

```
SetMode( MODE_INT_EN\          /* MODE_INT_EN = 0x08* enables all interrupts */
       |MODE_SOFTCONNECT\      /* MODE_SOFTCONNECT = 0x01 enables SoftConnect */
       |MODE_DMA16\            /* MODE_DMA16 = 0x80* selects 16-bit DMA bus width */
);
```

Figure 6-34: Code Example on Setting SoftConnect

When the polling reaches the check setup packet, the microprocessor verifies whether the current status is SETUPPROC. Then, it dispatches it to set up handler subroutines for processing. On reaching REQUESTPROC, it dispatches the device request to the protocol layer for processing.

6.7. Standard Device Requests

All USB devices must respond to a variety of requests called “standard” requests. These requests are used for configuring a device and controlling the state of its interface, along with other miscellaneous features. The host issues these device requests by using the control transfer mechanism. The three states—Default State, Address State and Configured State—must be taken care of. At a particular time, the device can be in only one of the states. For detailed information, refer to Chapter 9 of *USB Specification Rev. 2.0.*

6.7.1. Clear Feature Request

In the Clear Feature request, the microprocessor must clear or disable a specific feature of the device based on the three states. The flowchart of Clear Feature is given in Figure 6-35. In this case, the microprocessor determines whether the request is meant for the device, interface or endpoints. There will not be any support if the recipient is an interface. Feature selectors are used when enabling or setting features specific to the device or endpoint, such as remote wake-up. If the recipient is a device, the microprocessor must disable the remote wake-up function, if this function is enabled. If the recipient is an endpoint, the microprocessor must unstall the specific endpoint through the Write Endpoint Status command.

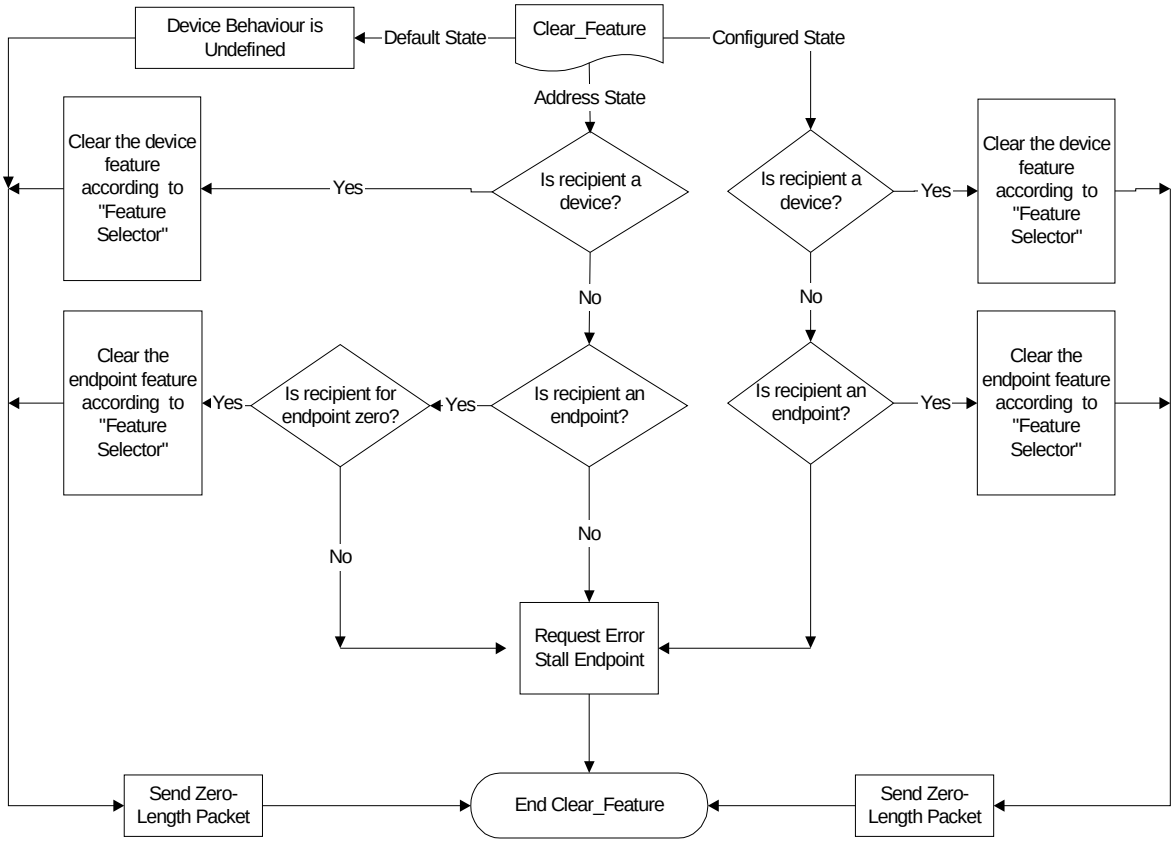


Figure 6-35: Flowchart of Clear Feature

Zero-Length Packet

A zero-length packet is a data packet with data length as zero. It is not the same as placing a 0x00 in the buffer and sending it out because this means a data length of 1 and a payload of 0x00. As can be seen in the pseudocode in Figure 6-13, sending a zero-length packet can be easily done by calling the Write_Endpoint() function with the arguments as given.

```
// This function call will send a zero-length packet to the host through the control IN endpoint.
Write_Endpoint (1 ,0 ,0)  // See Figure 6-13
```

Figure 6-36: Code Example to Send Zero-Length Packet

Request Error

When a control pipe request is not supported or the device is unable to transmit or receive data, a STALL must be returned in response to an IN Token. A stalled control endpoint is automatically unstalled when it receives a Setup token, regardless of the packet content. If the microcontroller wishes to unstall an endpoint, the Stall Endpoint or Unstall Endpoint command can be used.

```
void Write_EP_Status(UCHAR bEPIndex, UCHAR bStalled)
{
if(bStalled&0x01) // Check to stall or unstall the endpoint
outport(D13_COMMAND_PORT, STALL_EP + bEPIndex); /* STALL_EP = 0x40 */
else
outport(D13_COMMAND_PORT, UNSTALL_EP + bEPIndex); /* UNSTALL_EP = 0x80 */
}
```

Figure 6-37: Code Example to Stall or Unstall an Endpoint

6.7.2. Get Status Request

In the Get Status request, the microprocessor must return the status of the specific recipient based on the state of the device. The microprocessor must also determine the recipient of the request. If the request is to a device, the microprocessor must return the status of the device to the host, depending on the states. For a system having remote wake-up and self-powering capabilities, the returning data is 0x0003. Figure 6-38 shows the Get Status flowchart.

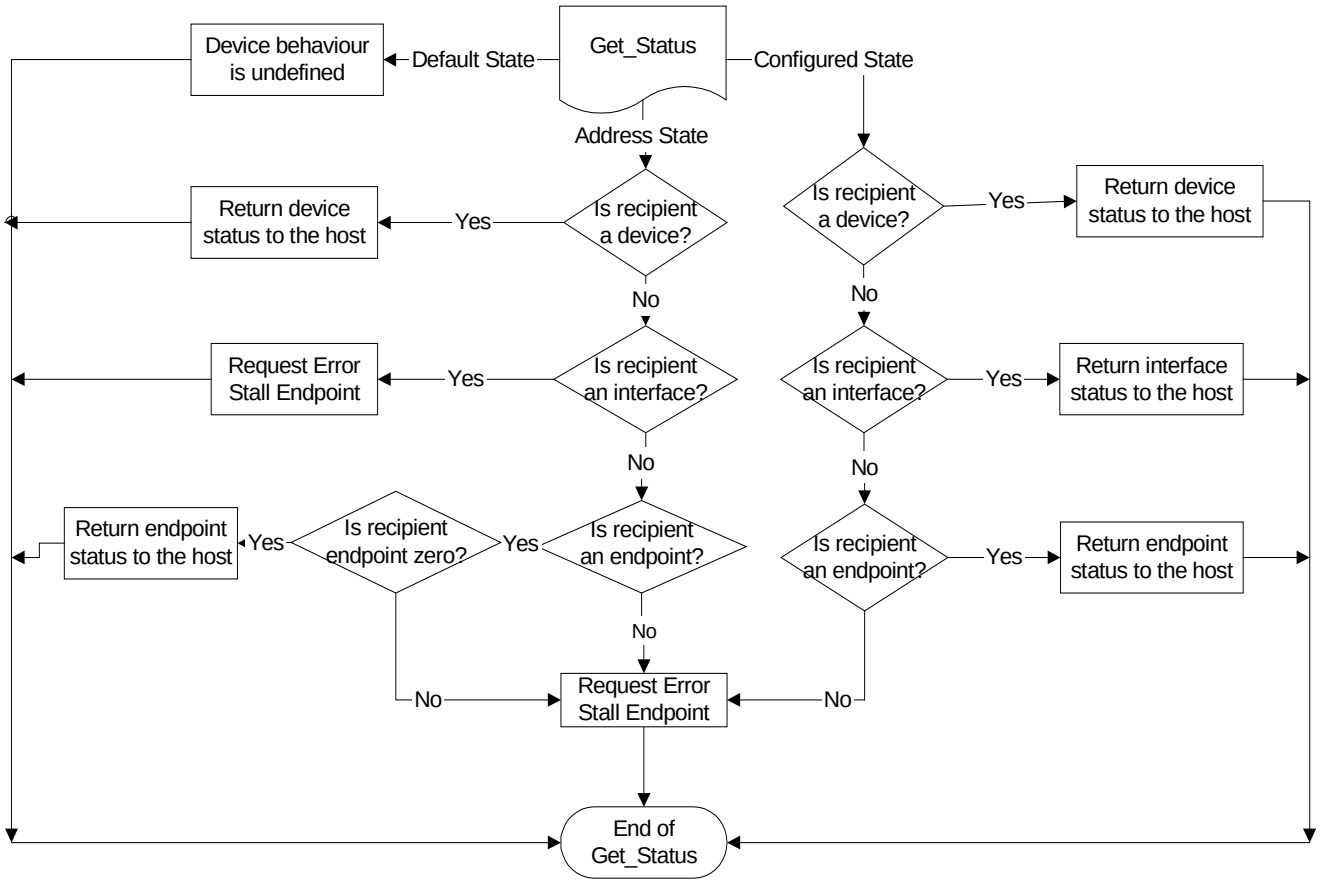


Figure 6-38: Flowchart of Get Status

6.7.3. Set Address Request

In the Set Address request (see Figure 6-39), the device gets the new address from the content of the Setup packet. Note that this Set Address request does not have a Data phase. Therefore, the microprocessor must write a zero-length data packet to the host at the acknowledgment phase.

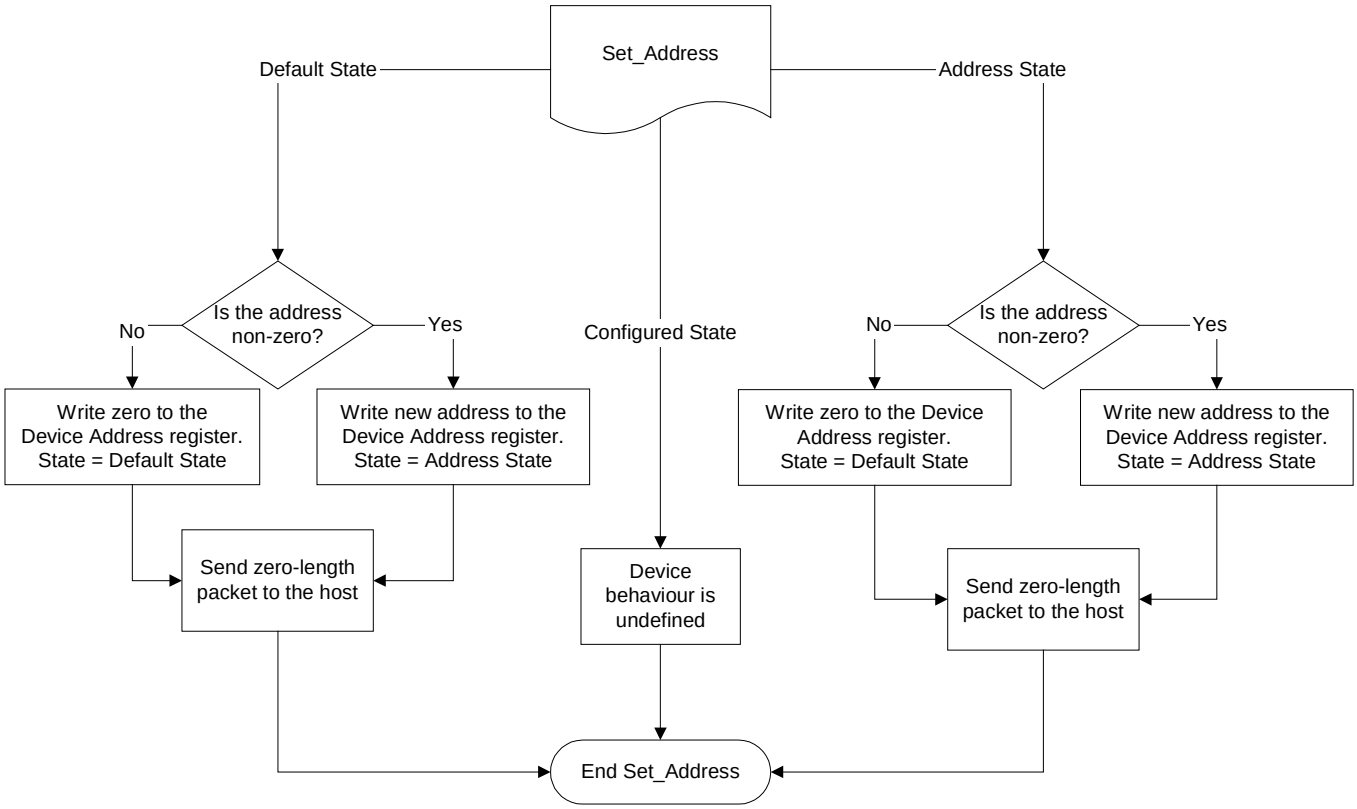


Figure 6-39: Flowchart of Set Address

Figure 6-40 shows a pseudocode of the Set Address routine.

```
void SetAddress(UCHAR bAddress, UCHAR bEnable)
{
        outport(D13_COMMAND_PORT, WR_DEV_ADD); // WR_DEV_ADD = 0xB6
        if(bEnable) // Enables or disables the address
                bAddress |= ADDR_EN;          /* ADDR_EN = 0x80 */
        else
                bAddress &= ADDR_MASK;       /* ADDR_MASK = 0x7F */
        outport(D13_DATA_PORT, bAddress);
}
```

Figure 6-40: Code Example of the Set Address Routine

Table 6-11: Device Address Register: Bit Allocation

Bit	7	6	5	4	3	2	1	0
Symbol	DEVEN	DEVADR[6:0]						
Reset	0	0	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Table 6-12: Device Address Register: Bit Description

Bit	Symbol	Description
7	DEVEN	A logic 1 enables the device.
6 to 0	DEVADR[6:0]	This field specifies the USB device address.

6.7.4. Get Configuration Request

In the Get Configuration request (see the flowchart in Figure 6-41), the microprocessor must return the current configuration value. The microprocessor first determines what state the device is in. Depending on the state, the microprocessor will either send a zero or the current non-zero configuration value back to the host.

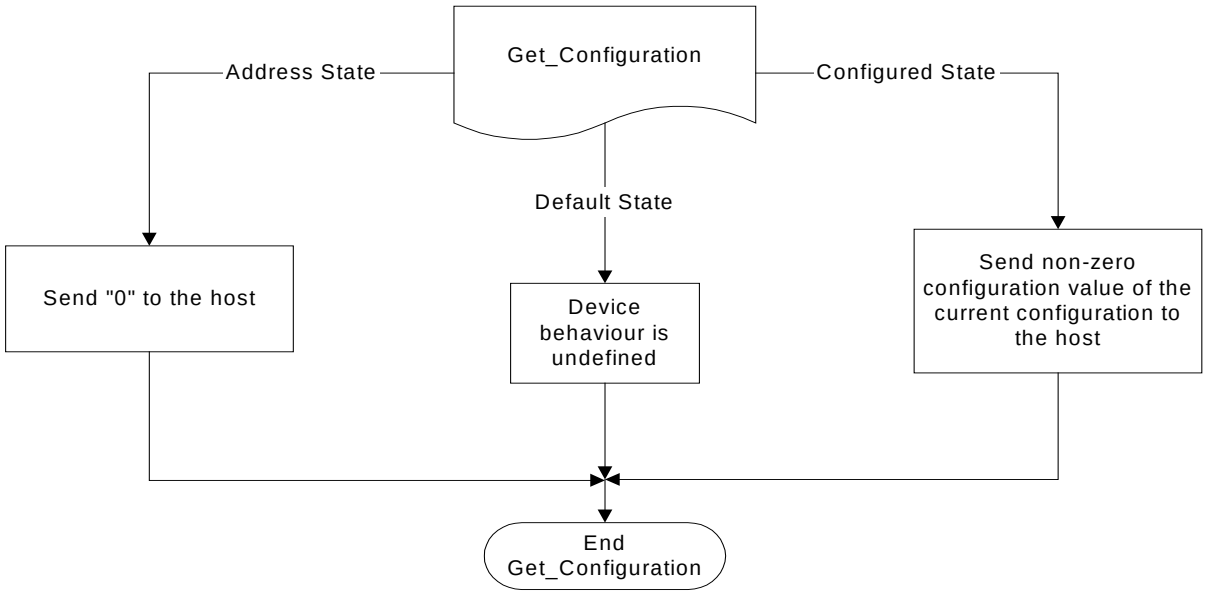


Figure 6-41: Flowchart of Get Configuration

6.7.5. Get Descriptor Request

For the Get Descriptor request, the microprocessor must return the specific descriptor, if the descriptor exists. First, the microprocessor determines whether the descriptor type request is for a device or configuration. It then sends the first 64 bytes of the device descriptor, if the descriptor type is for a device. The reason for controlling the size of returning bytes is that the control buffer has only 64 bytes of memory. The microprocessor must set a register to indicate the location of the transmitted size. The Get Descriptor request is a valid request for Default State, Address State and Configured State. Figure 6-42 shows the flowchart of Get Descriptor.

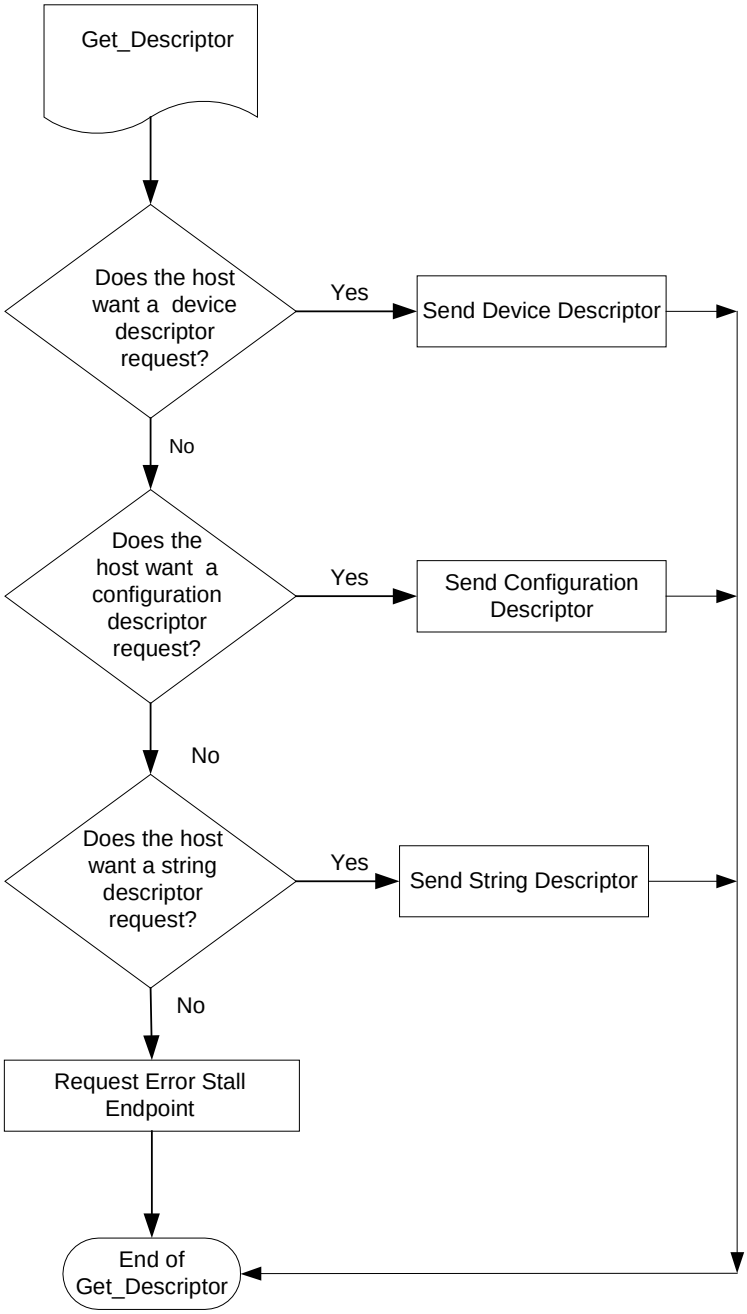


Figure 6-42: Flowchart of Get Descriptor

6.7.6. Set Configuration Request

For the Set Configuration request (see Figure 6-44), the microprocessor determines the configuration value from the Setup packet. If the value is zero, the microprocessor must clear the configuration flag in its memory and disable the endpoint. If the value is one, the microprocessor must set the configuration flag. Once the flag is set, the microprocessor must also send the zero-data packet to the host at the acknowledgment phase.

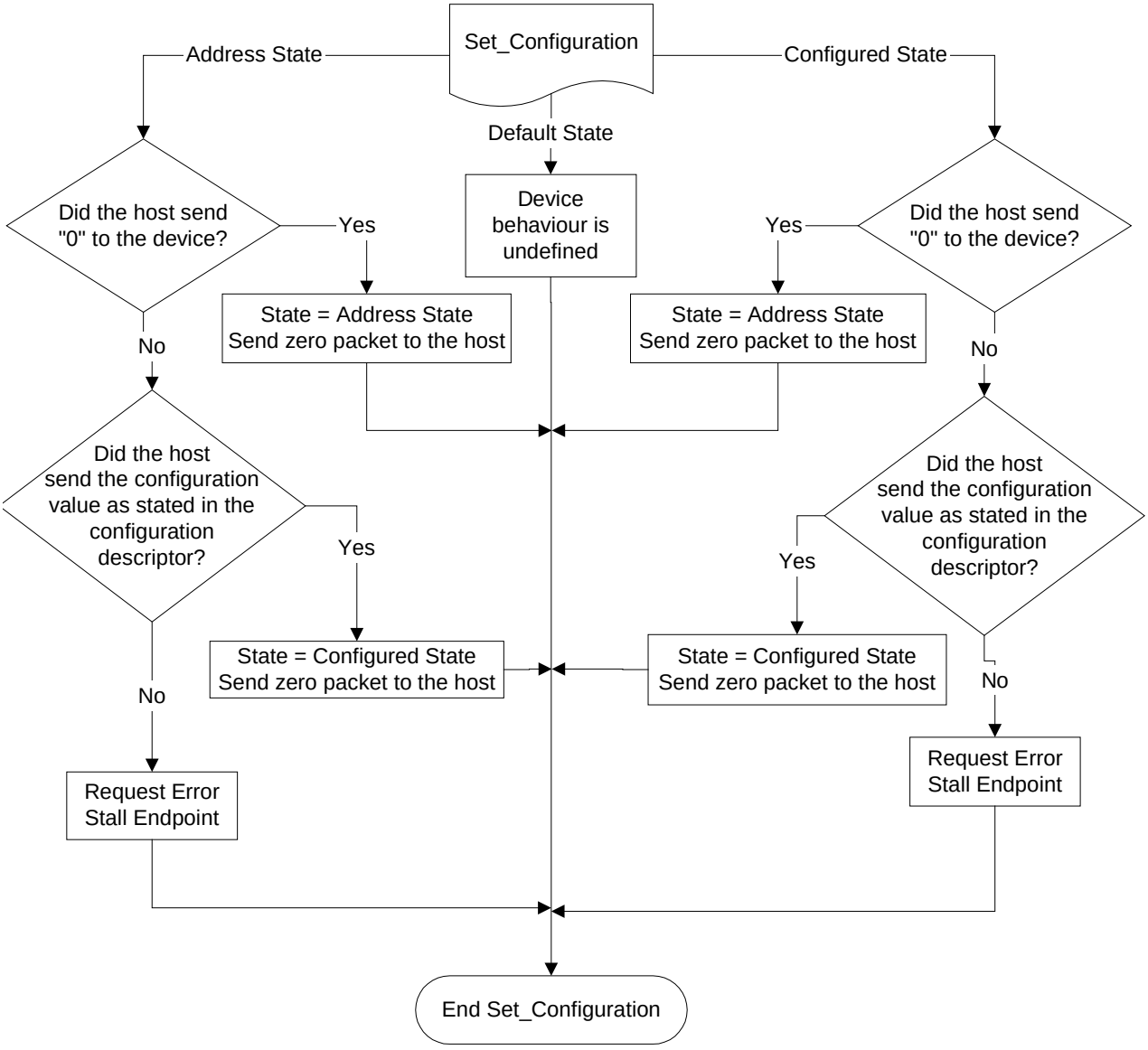


Figure 6-43: Flowchart of Set Configuration

6.7.7. Get and Set Interface Requests

For the Get and Set Interface requests (see flowcharts in Figure 6-44 and Figure 6-45), the microprocessor just needs to send one zero-data packet to the host because the Philips evaluation board only supports one type of interface. For the Set Interface request on the Philips evaluation board, the microprocessor need not do anything except to send one zero data packet to the host as the acknowledgment phase.

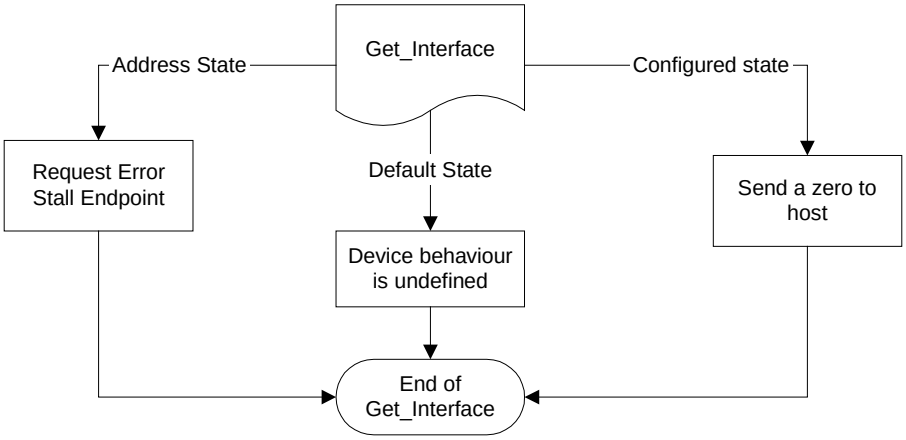


Figure 6-44: Flowchart of Get Interface

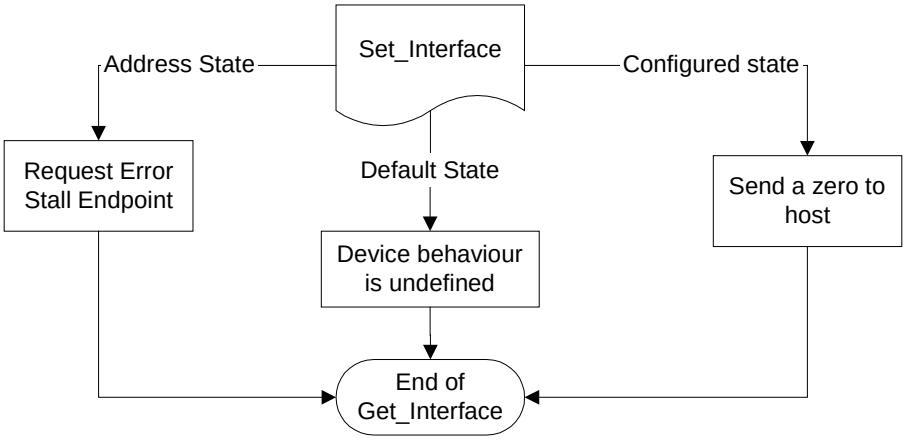


Figure 6-45: Flowchart of Set Interface

6.7.8. Set Feature Request

The Set Feature request is just the opposite of the Clear Feature request. Figure 6-46 contains the flowchart of Set Feature. If the recipient is a device, the microprocessor must set the feature of the device according to the feature selector in the Setup packet. Again, there is no support for the Interface recipient. For example, if the feature selector is 0 (which means enabling endpoint), the Device Controller of the ISP1161x specific endpoint must be stalled through the Write Endpoint Status command.

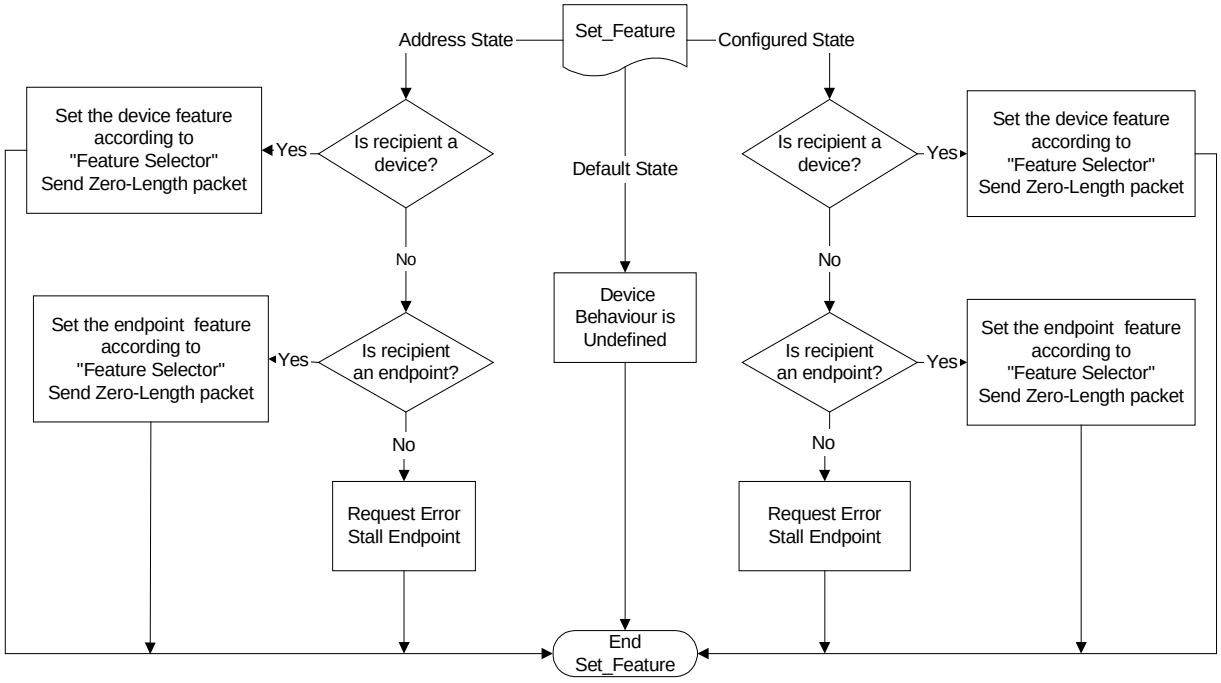


Figure 6-46: Flowchart of Set Feature

6.7.9. Class Request

Support for class requests is not included in the Device Controller of the ISP1161x sample firmware.

6.8. Vendor Request

In the ISP1161x Device Controller sample firmware and applet, the vendor request sets up the Bulk transfer or the isochronous transfer. This request is sent through the control pipe that is done by IOCTL_WRITE_REGISTER. IOCTL_WRITE_REGISTER is defined by Microsoft® Still Image USB Interface in Windows® 98 DDK. A device vendor may also define requests supported by the device.

6.8.1. Vendor Request for the Bulk Transfer

The device request is defined in Table 6-13.

Table 6-13: Device Request

Offset	Field	Size	Value	Comments
0	BmRequestType	1	0x40	Vendor request, host to device
1	Brequest	1	0x0C	Fixed value for IOCTL_WRITE_REGISTER
2	Wvalue	2	0	Offset, set to zero
4	Windex	2	0x0471	Fixed value of Setup Bulk transfer
6	Wlength	2	6	Data length of Setup Bulk transfer

The details requested by the Bulk transfer operation are sent in the Data phase after the Setup Token phase of the device request. The sample firmware and applet use a proprietary definition, which is given in Table 6-14.

Table 6-14: Proprietary Definition of the Sample Firmware and Applet

Offset	Field	Comments
0	Address[7:0]	The start address of the requested Bulk transfer.
1	Address[15:8]	—
2	Address[23:16]	—
3	Size[7:0]	Size of the transfer.
4	Size[15:8]	—
5	Command	Bit 7: 1—start Bulk transfer by DMA; 0—start Bulk transfer by PIO Bit 0: 1—IN token; 0—OUT token.

6.8.2. CATC Capture of a PIO OUT Transfer

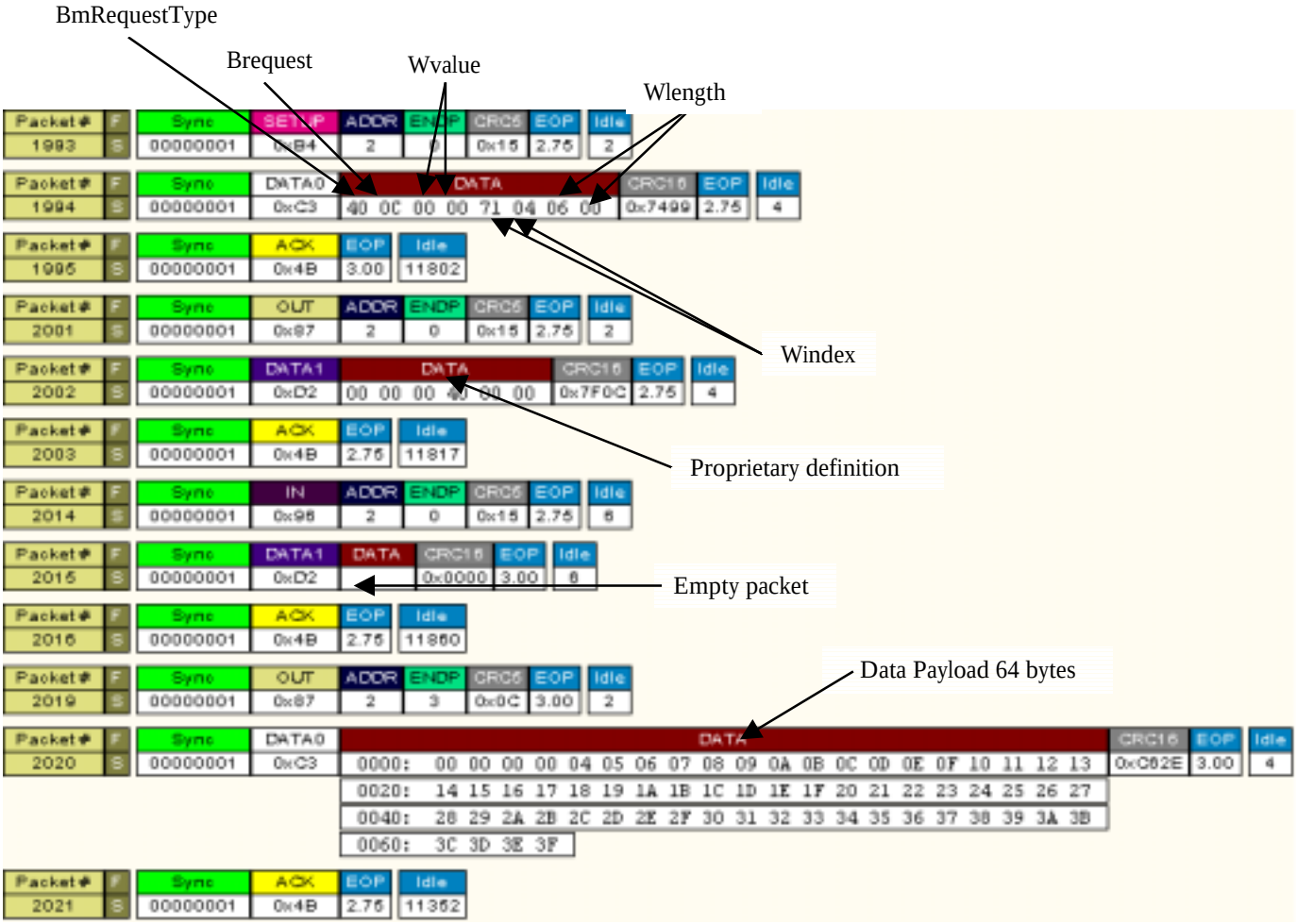


Figure 6-47: CATC Capture of a PIO OUT Transfer

6.8.3. CATC Capture of a PIO IN Transfer

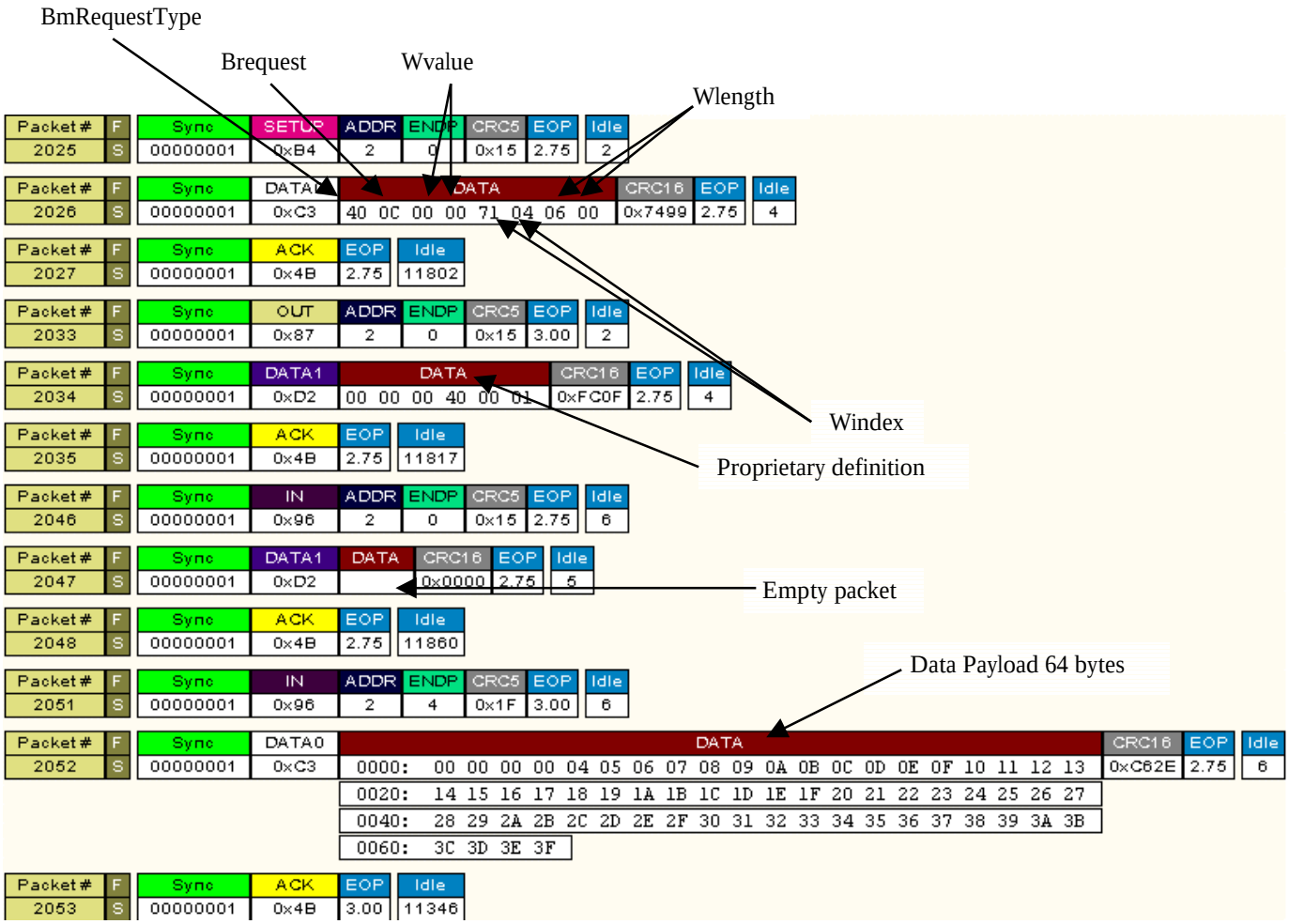


Figure 6-48: CATC Capture of a PIO IN Transfer

6.8.4. Vendor Request for the ISO Transfer

The device request is defined in Table 6-15.

Table 6-15: Device Request

Offset	Field	Size	Value	Comments
0	BmRequestType	1	0x40	Vendor request, host to device
1	Brequest	1	0x00	Fixed value for IOCTL_WRITE_REGISTER
2	Wvalue	2	—	0x0002 = ISO OUT; 0x0001 = ISO IN
4	Windex	2	—	0x0002 = ISO OUT; 0x0001 = ISO IN
6	Wlength	2	0x00	Data length of Setup ISO transfer

For the ISO transfer, the applet and the firmware must pre-arrange the size of the transfer before the transfer can be completed successfully. This is because the vendor request does not give any transfer size information to the firmware. Therefore, if you want to transfer 512 bytes of data, the ISO endpoint must be set to 512 bytes, which is the default size set by the firmware.

6.8.5. CATC Capture of an ISO OUT Transfer

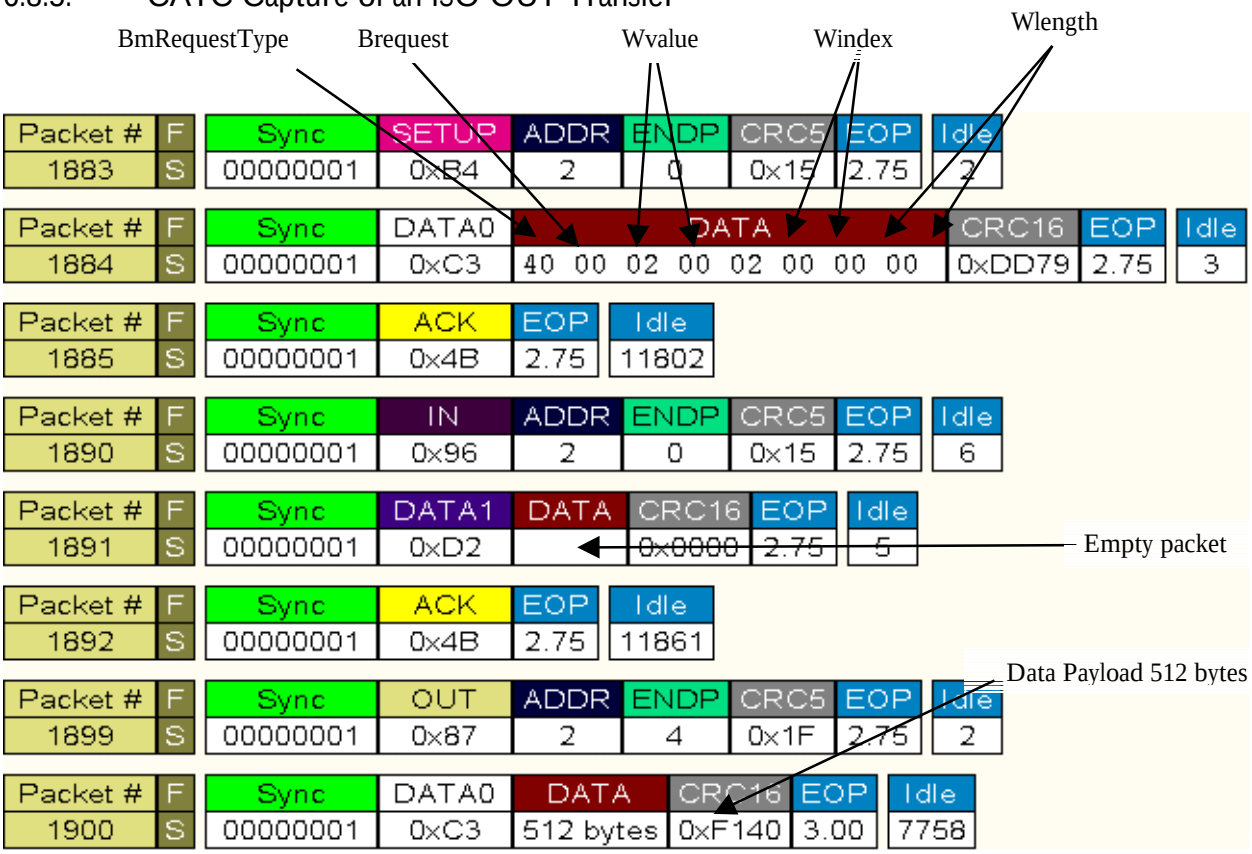


Figure 6-49: CATC Capture of an ISO OUT Transfer

6.8.6. CATC Capture of an ISO IN Transfer

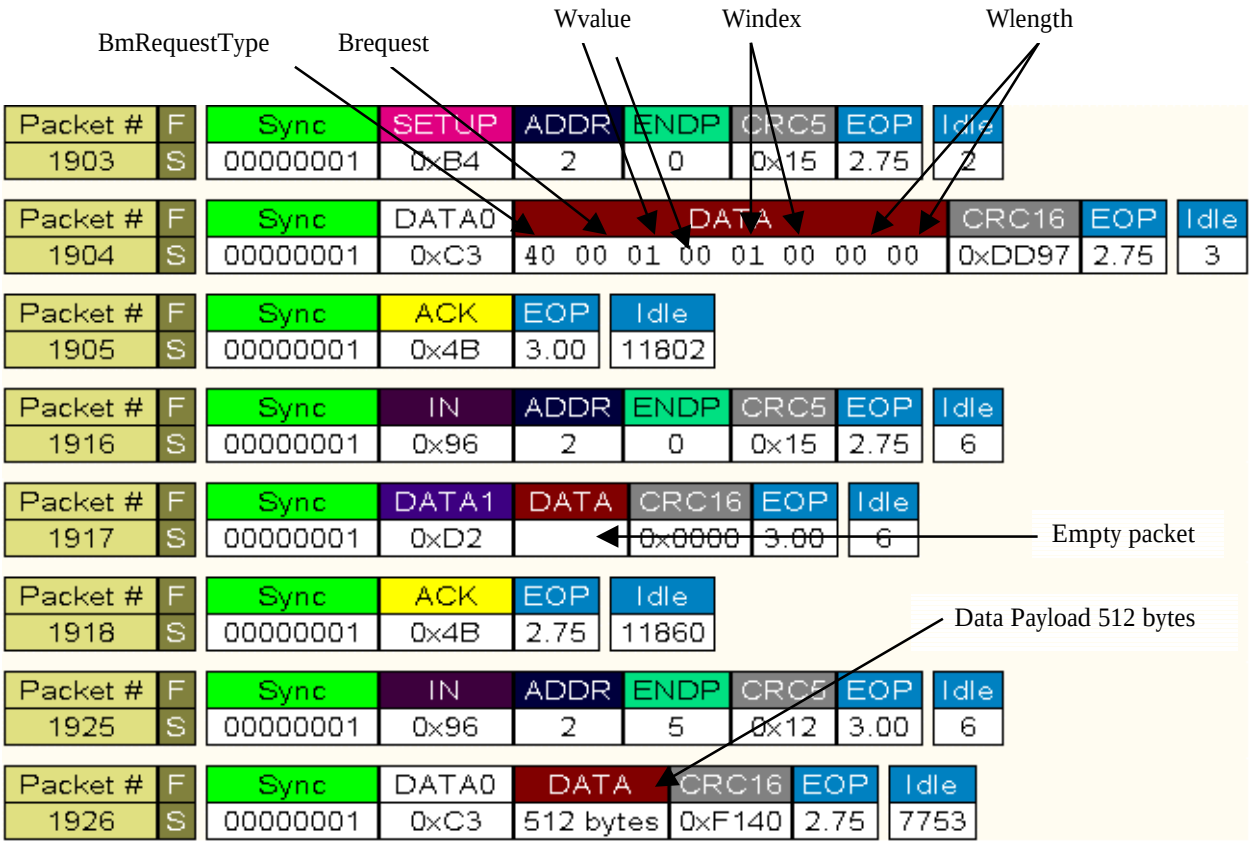


Figure 6-50: CATC Capture of an ISO IN Transfer

7. References

- *ISP1161x Full-speed Universal Serial Bus single-chip host and device controller datasheet*
- *Universal Serial Bus Specification Rev. 2.0 (full-speed section)*
- *Open Host Controller Interface Specification for USB, Release: 1.0a.*