AN10005

ISP1161A1 embedded programming guide

Rev. 03 — 12 October 2009 Application note

Document information

Info	Content
Keywords	isp1161a1; host controller; usb; universal serial bus
Abstract	This application note describes the use of the ISP1161A1 in embedded systems.

ST ERICSSON

Revision history

Rev	Date	Description
03	20091012	Rebranded to the ST-Ericsson template.
02	20090303	Rebranded to the ST-NXP Wireless template.
01	20020401	First release

Contact information

For additional information, please visit: **http://www.stericsson.com**

For document related queries, please send an email to: **wired.support@stericsson.com**

1. Introduction

ISP1161A1 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.

ISP1161A1 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 VBUS detection input pin. ISP1161A1 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. ISP1161A1 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. ISP1161A1 provides high flexibility to the systems that have it built-in. For example, a system that has ISP1161A1 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. ISP1161A1 enables point-to-point connectivity between embedded systems. An interesting application example is to connect a ISP1161A1 host controller with a ISP1161A1 device controller.

2. ISP1161A1 software models

As ISP1161A1 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. Fig 1 shows the data flow and the call hierarchy of the software components in this software model.

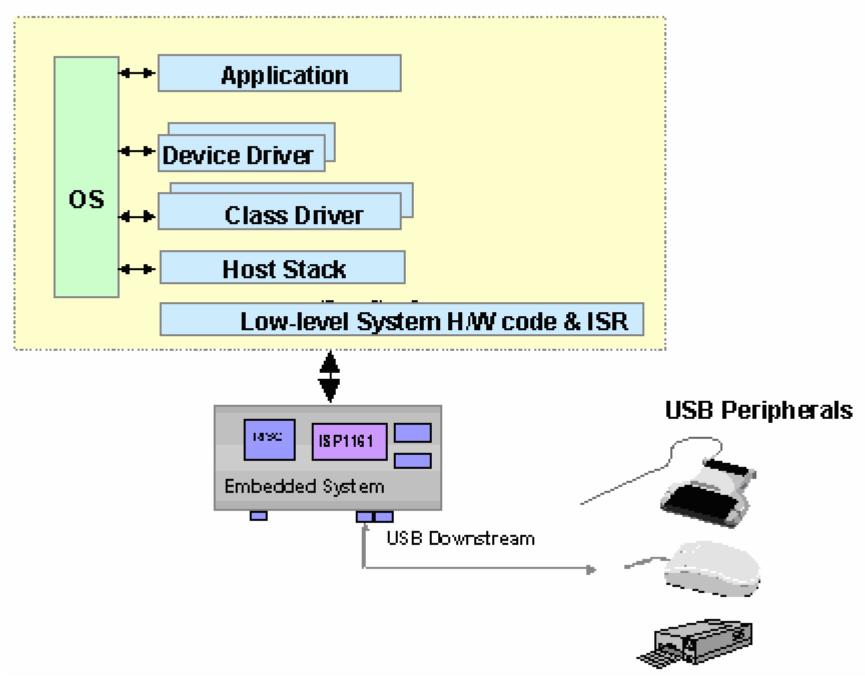


Fig 1. ISP1161A1 host-only mode software model

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 Fig 2.

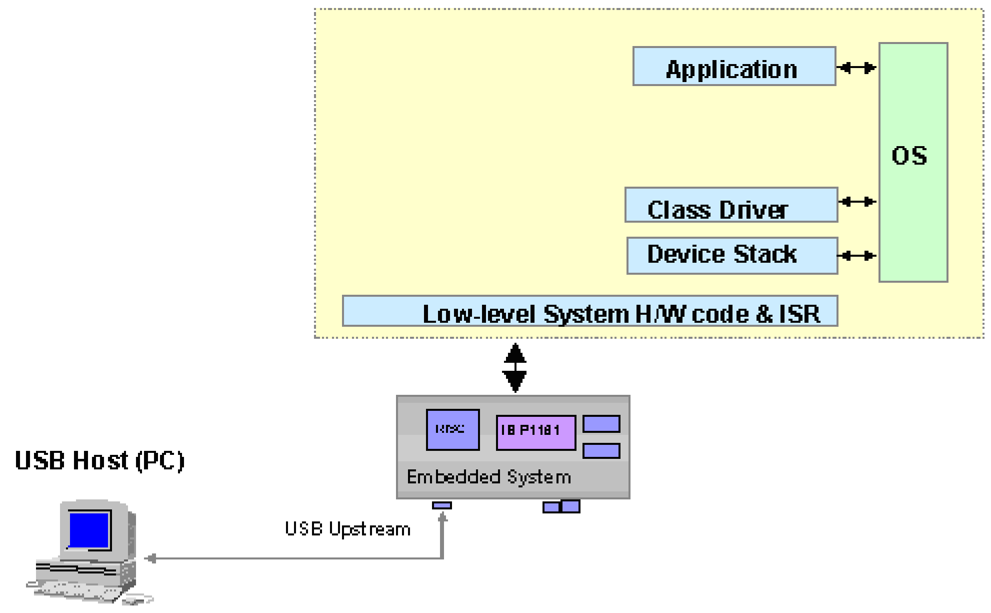


Fig 2. ISP1161A1 device-only mode software model

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 ISP1161A1 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 Fig 3.

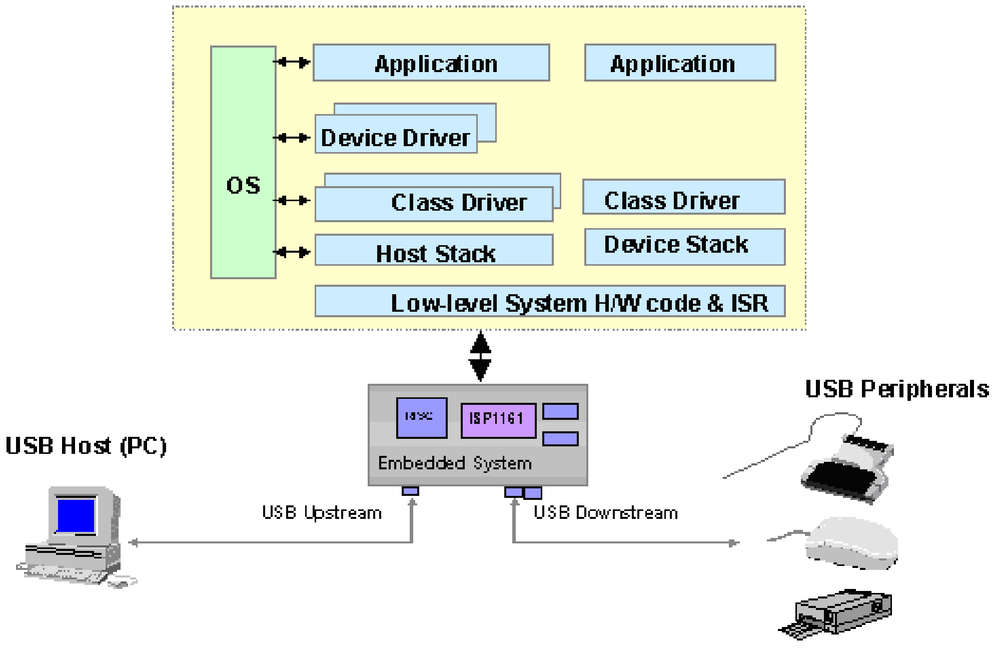


Fig 3. ISP1161A1 simultaneous host-and-device mode software model

In this mode, ISP1161A1 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. ISP1161A1 hardware models

3.1 Host controller hardware model

The major difference between the OHCI host controller and ISP1161A1 is that the OHCI host controller is a bus-master device whereas ISP1161A1 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 ISP1161A1, the microprocessor is responsible for moving the USB data packet between the system memory, and the ITL and ATL buffers inside ISP1161A1. An I/O bus interface and the bus master DMA are eliminated from ISP1161A1, and therefore, the term “slave host controller”. This is because ISP1161A1 is intended to be used for embedded applications in which cost and design simplicity are important design considerations for choosing a host controller IC.

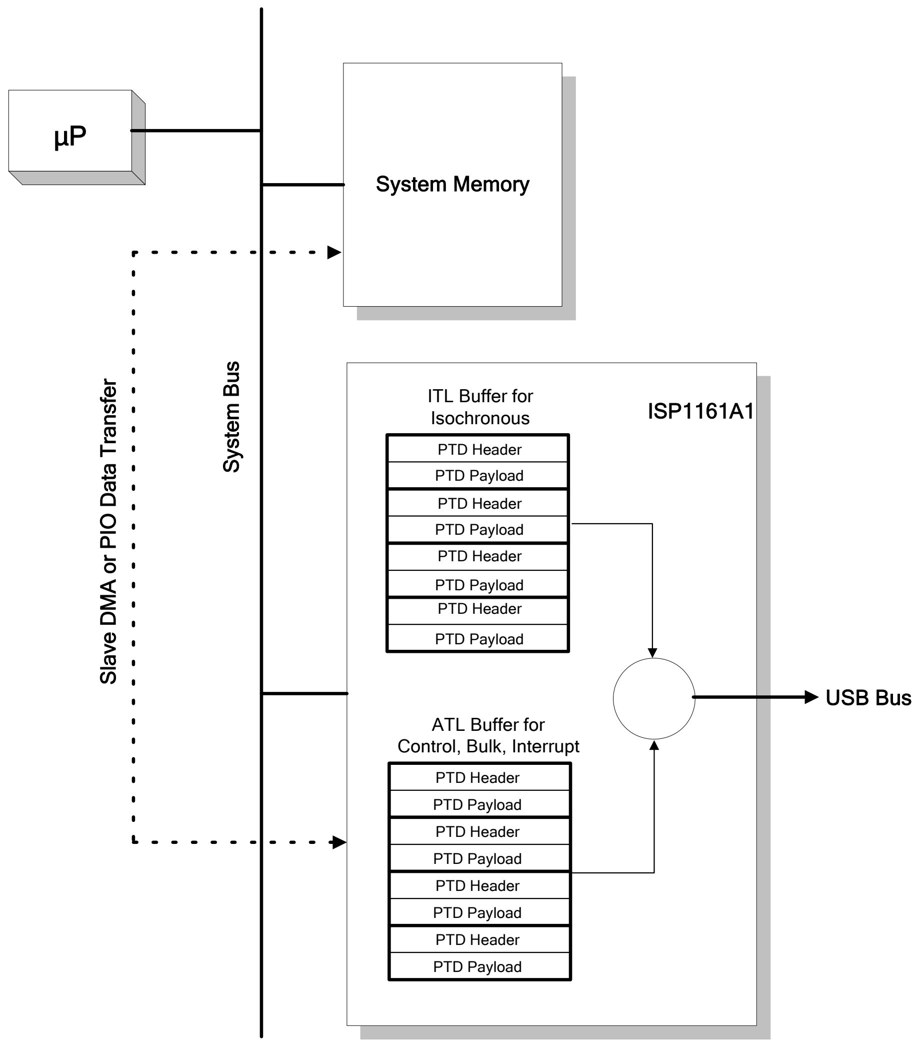


Fig 4. ISP1161A1 host controller hardware model

3.2 Device controller hardware model

When the device controller part of ISP1161A1 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 ISP1161A1. USB packets are sent from and received in endpoint FIFOs.

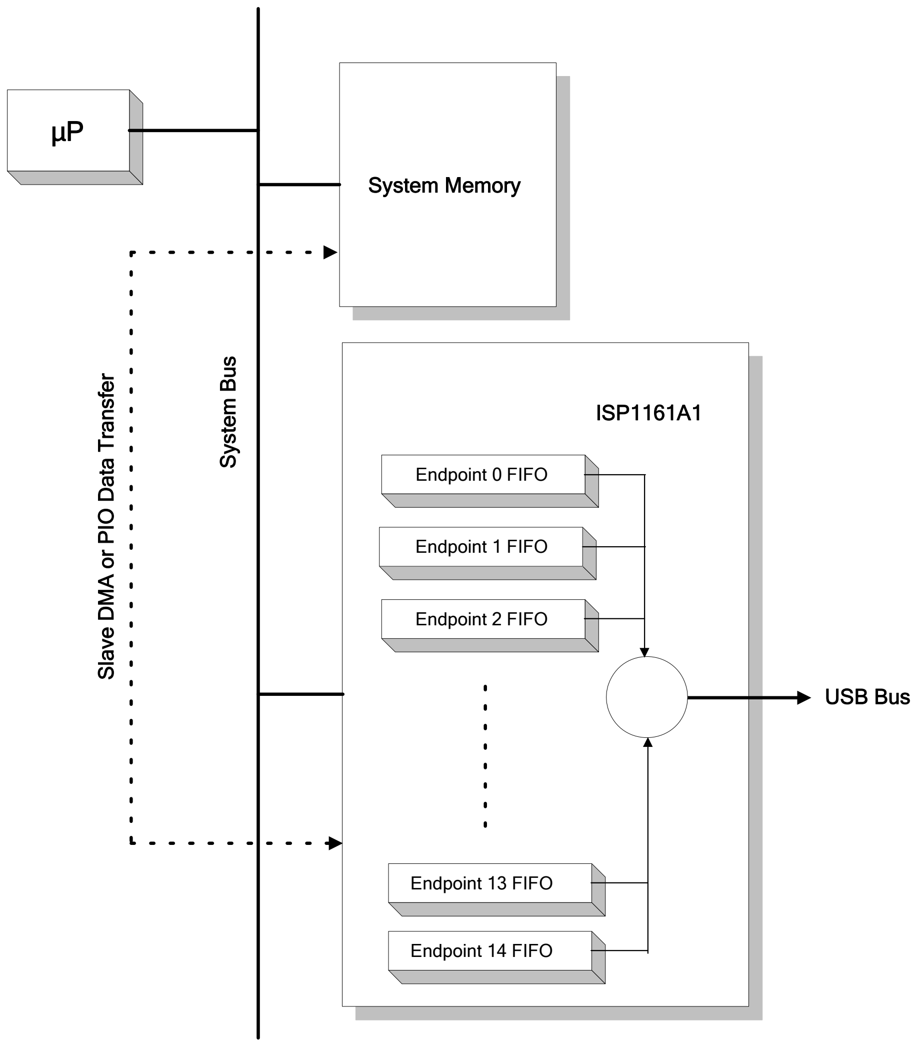


Fig 5. ISP1161A1 device controller hardware model

4. ISP1161A1 software architecture

4.1 USB host software architecture

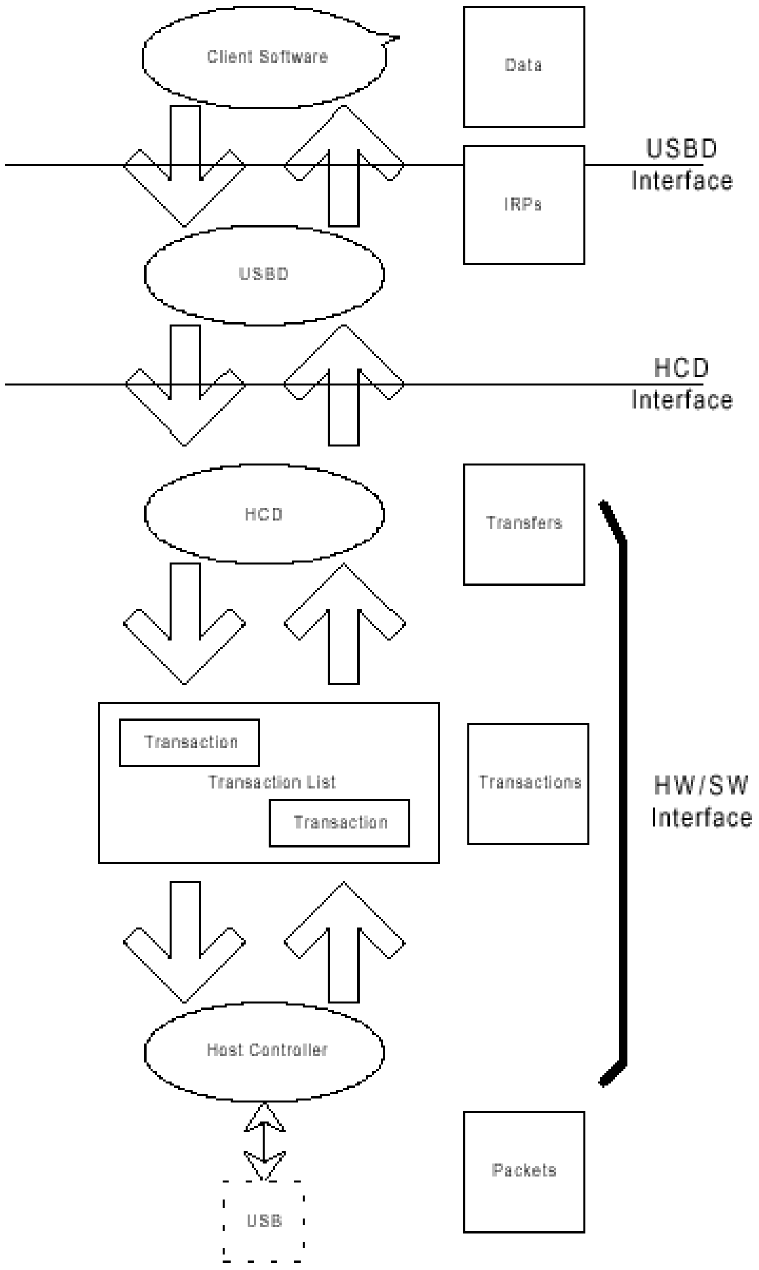


Fig 6. USB host software architecture

As can be seen in Fig 6, 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 ISP1161A1 to enable it to perform HCD functions when it runs as a USB host controller.

4.2 Host stack architecture

Fig 7 shows the major functions built in a USB host stack.

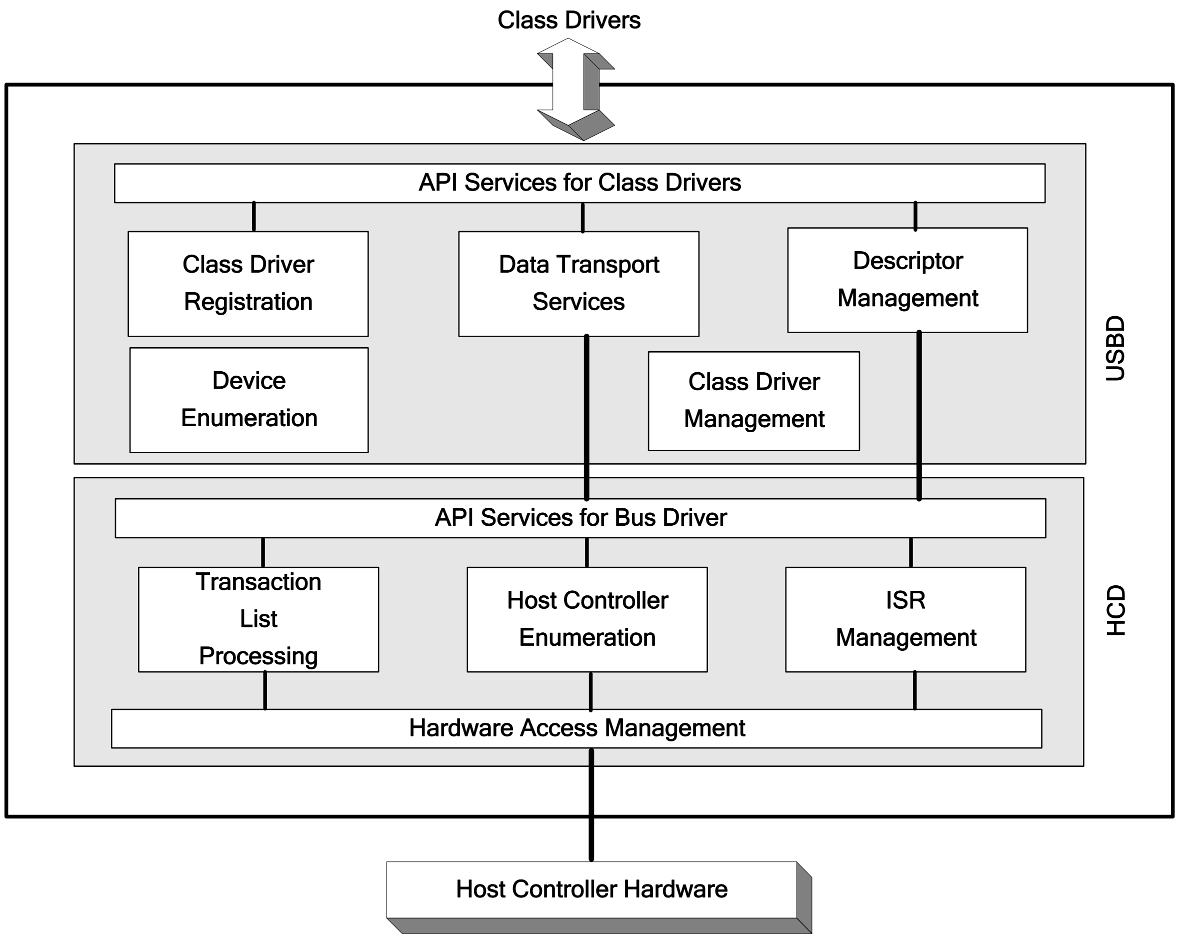

Class Drivers
API Services for Class Drivers
Class Driver Registration
Data Transport Services
Descriptor Management
Device Enumeration
Class Driver Management
USBD
API Services for Bus Driver
Transaction List Processing
Host Controller Enumeration
ISR Management
HCD
Hardware Access Management
Host Controller Hardware

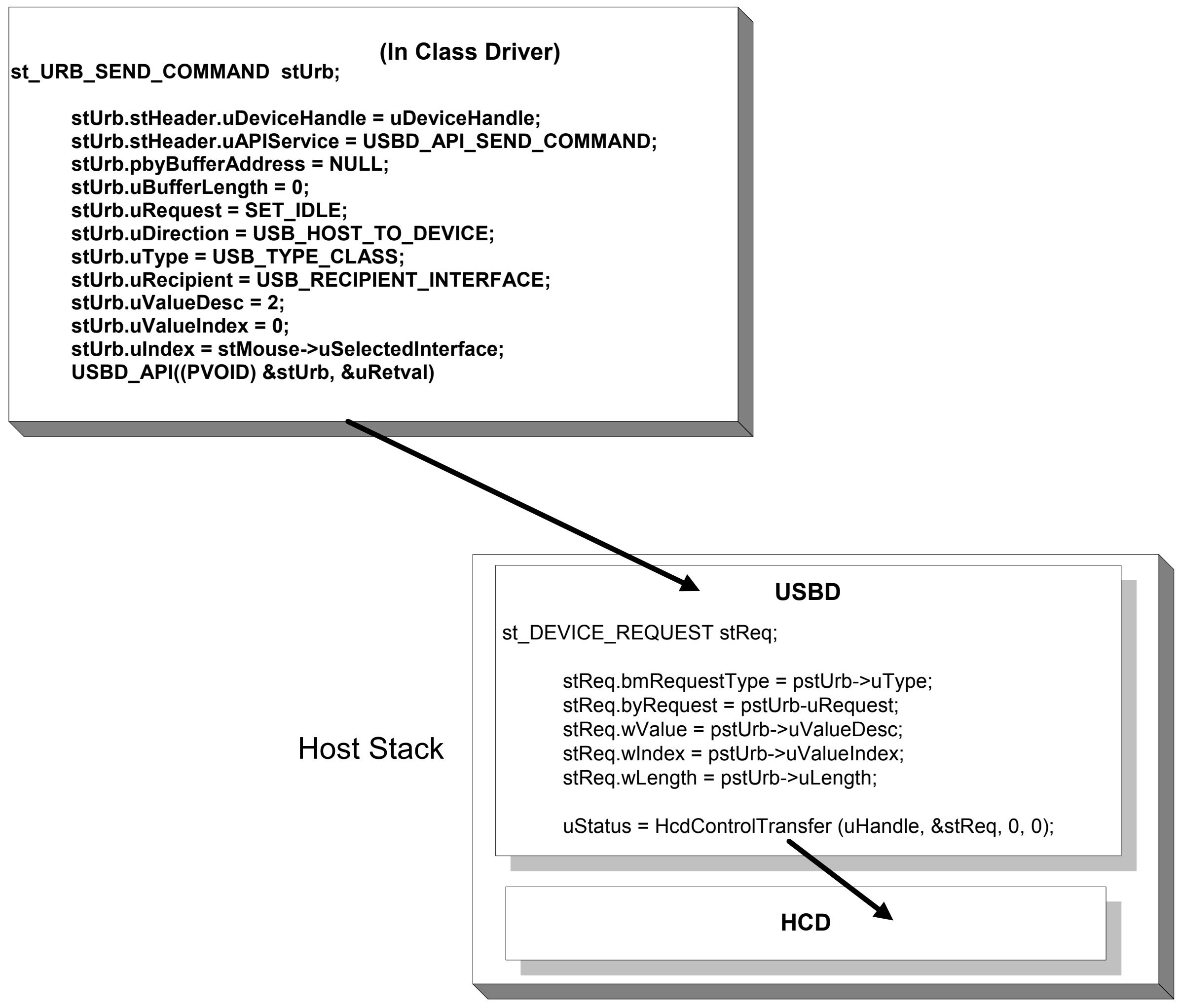
Fig 7. Host stack architecture

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.



```
st_URB_SEND_COMMAND  stUrb;

        stUrb.stHeader.uDeviceHandle = uDeviceHandle;
        stUrb.stHeader.uAPIService = USBD_API_SEND_COMMAND;
        stUrb.pbyBufferAddress = NULL;
        stUrb.uBufferLength = 0;
        stUrb.uRequest = SET_IDLE;
        stUrb.uDirection = USB_HOST_TO_DEVICE;
        stUrb.uType = USB_TYPE_CLASS;
        stUrb.uRecipient = USB_RECIPIENT_INTERFACE;
        stUrb.uValueDesc = 2;
        stUrb.uValueIndex = 0;
        stUrb.uIndex = stMouse->uSelectedInterface;
        USBD_API((PVOID) &stUrb, &uRetval)
```

```
st_DEVICE_REQUEST stReq;

        stReq.bmRequestType = pstUrb->uType;
        stReq.byRequest = pstUrb-uRequest;
        stReq.wValue = pstUrb->uValueDesc;
        stReq.wIndex = pstUrb->uValueIndex;
        stReq.wLength = pstUrb->uLength;

        uStatus = HcdControlTransfer (uHandle, &stReq, 0, 0);
```

Fig 8. 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

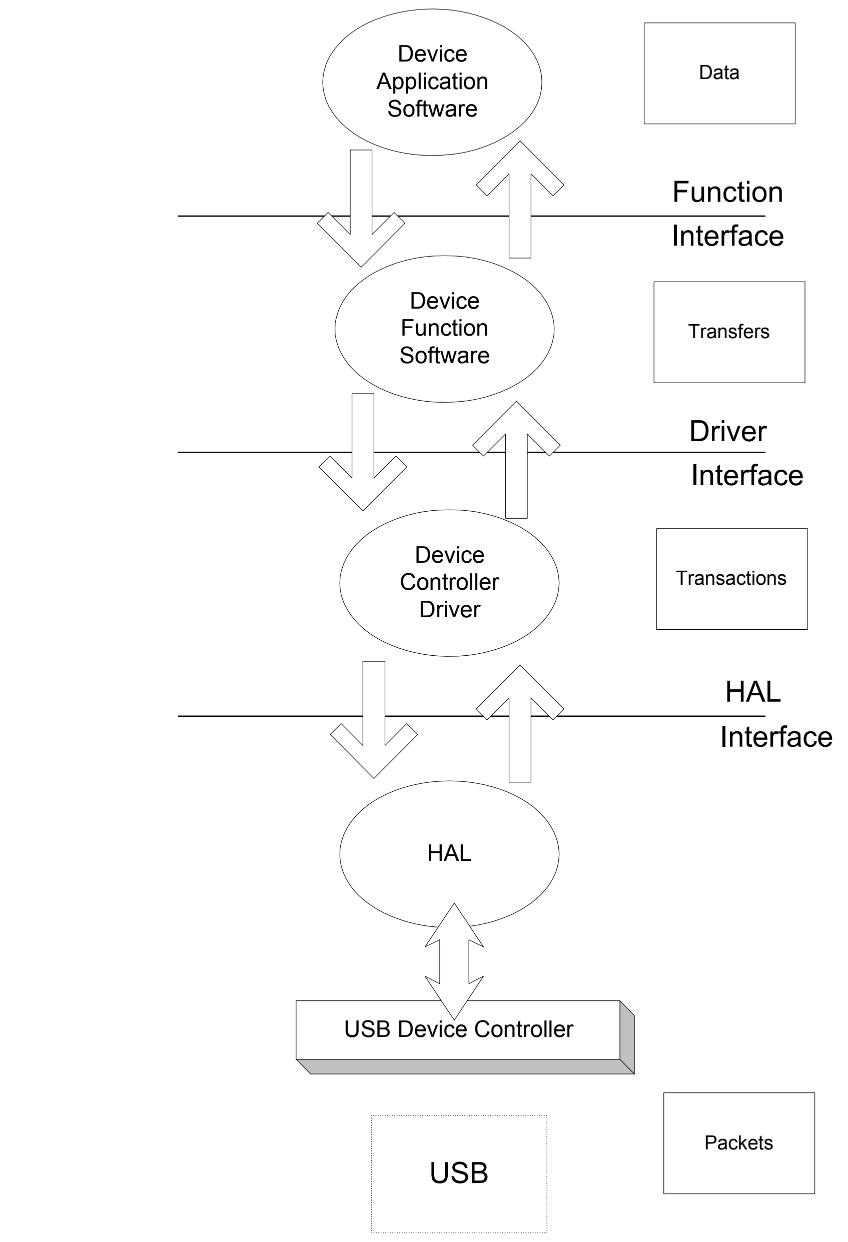


Fig 9. USB device software architecture

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 ISP1161A1

5.1 Software accessible hardware components

The major hardware components of the host controller in ISP1161A1 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 ISP1161A1 include a set of operational OHCI compliant registers (32-bit wide) and a set of ISP1161A1 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 1.

Table 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	8E	HcFmRemaining	32	
0F	8F	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	

Command (Hex)		Register	Width	Functionality
Read	Write			
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 ISP1161A1 is 16-bit wide, 32-bit registers are read or written in two data cycles. Fig 10 illustrates a 16-bit register access cycle.

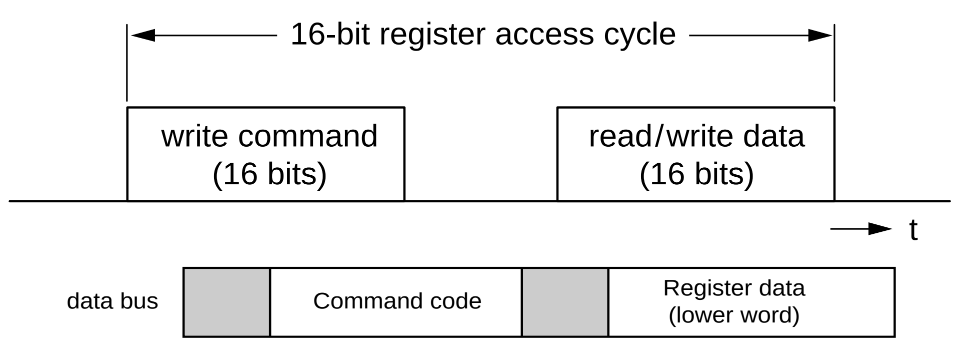

16-bit register access cycle
write command (16 bits)
read/write data (16 bits)
t
data bus
Command code
Register data (lower word)

Fig 10. 16-bit register access cycle

In Fig 10, 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.

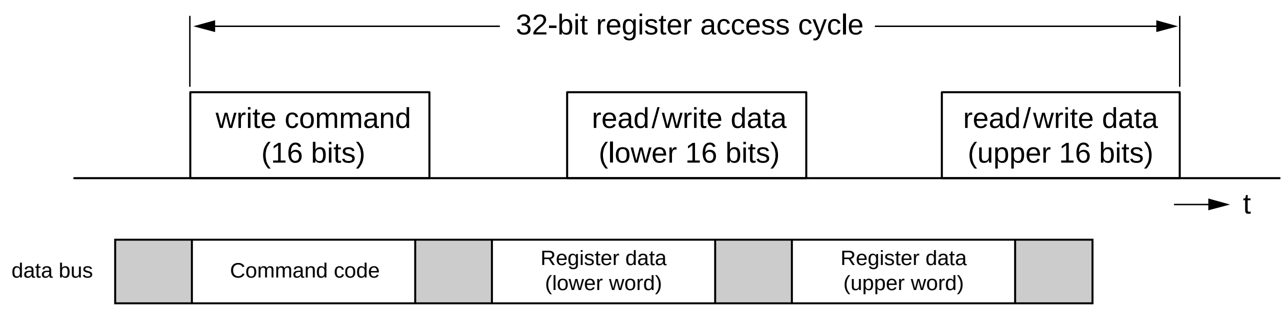

32-bit register access cycle
write command (16 bits)
read/write data (lower 16 bits)
read/write data (upper 16 bits)
t
data bus
Command code
Register data (lower word)
Register data (upper word)

Fig 11. 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 Fig 12 shows a 32-bit register access with ISP1161A1 connected to an ISA bus in the x86 platform with two ISA ports assigned to the host controller of ISP1161A1: 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);
```

Fig 12. Code example for 32-bit register write

In the above 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;
```

Fig 13. Code example for 32-bit register read

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

The code example in Fig 14 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);
```

Fig 14. Code example for 16-bit register read

The code example in Fig 15 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.
Write the 16-bit data to the data port.

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

Fig 15. 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 ISP1161A1 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 ISP1161A1. 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 Fig 16 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();
    }
```

Fig 16. Code example for writing to the ATL buffer

5.4 Typical hardware initialization sequence

When the ISP1161A1 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
        - DACK input polarity
        - DREQ output polarity
4. Configuring interrupts
    a. USB specific interrupts
        - Master interrupt enable
        - Root hub status change interrupt
        - Frame number overflow interrupt
        - Unrecoverable error interrupt
        - Resume detect interrupt
        - Start-of-Frame (SOF) interrupt

        - Scheduling overrun interrupt
    b. Host controller related interrupts
        - Clock ready interrupt
        - Host controller suspend interrupt
        - OPR register interrupt
        - All EOT interrupt
        - ATL done interrupt
        - 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 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 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	R/W	R/W	R/W	R/W	R/W	R/W	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	R/W	R/W	R/W	R/W	R/W	R/W	R/W

```
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 ISP1161A1.
      if (uData & 0xFF00) == 0x6100
            foundISP1161A1;
}
else
      NotFoundISP1161A1;
```

Fig 17. 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:

11. Resetting the host controller.
12. 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 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. The pseudocode for resetting the host controller is given below.

```
// 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));
```

Fig 18. Code example for resetting the host controller

Table 3. HcCommandStatus register: bit allocation
Read index—02H; Write index—82H

Bit	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
Access	R	R	R	R	R	R	R	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	R	R	R	R	R	R
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
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	reserved							HCR
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

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 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);
```

Fig 19. Code example for setting the host controller to the RESET state

Table 4. HcControl register: bit allocation
Read index—01H; Write index—81H

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	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	00H	00H	00H	00H	00H	00H	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 ISP1161A1. 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. HcHardwareConfiguration register: bit allocation
Read index—20H; Write index—A0H

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved			2_Down streamPort 15K resistorSel	Suspend ClkNotStop	AnalogOC Enable	reserved	DACKMode
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	EOTInput Polarity	DACKInput Polarity	DREQ Output Polarity	DataBusWidth		Interrupt Output Polarity	InterruptPin Trigger	InterruptPin Enable
Reset	0	0	0	0	1	0	0	0
Access	R/W	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 6.

Table 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 cannot 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	**01 —** 16 bits **Others —** reserved

Bit	Symbol	Description
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 ISP1161A1 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 ISP1161A1 evaluation board connected to a personal computer (PC) motherboard.

```
#define    INTERRUPT_PIN_ENBLE        0x0001     // INT1 pin in ISP1161A1
#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);
```

Fig 20. Code example for initializing the HcHardwareConfiguration register

5.4.4 Configuring interrupts

The host controller in ISP1161A1 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 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	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	00H	00H	00H	00H	00H	00H	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							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
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	reserved	RHSC	FNO	UE	RD	SF	reserved	SO
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

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. Fig 21 shows the relationship between these two groups of interrupts.

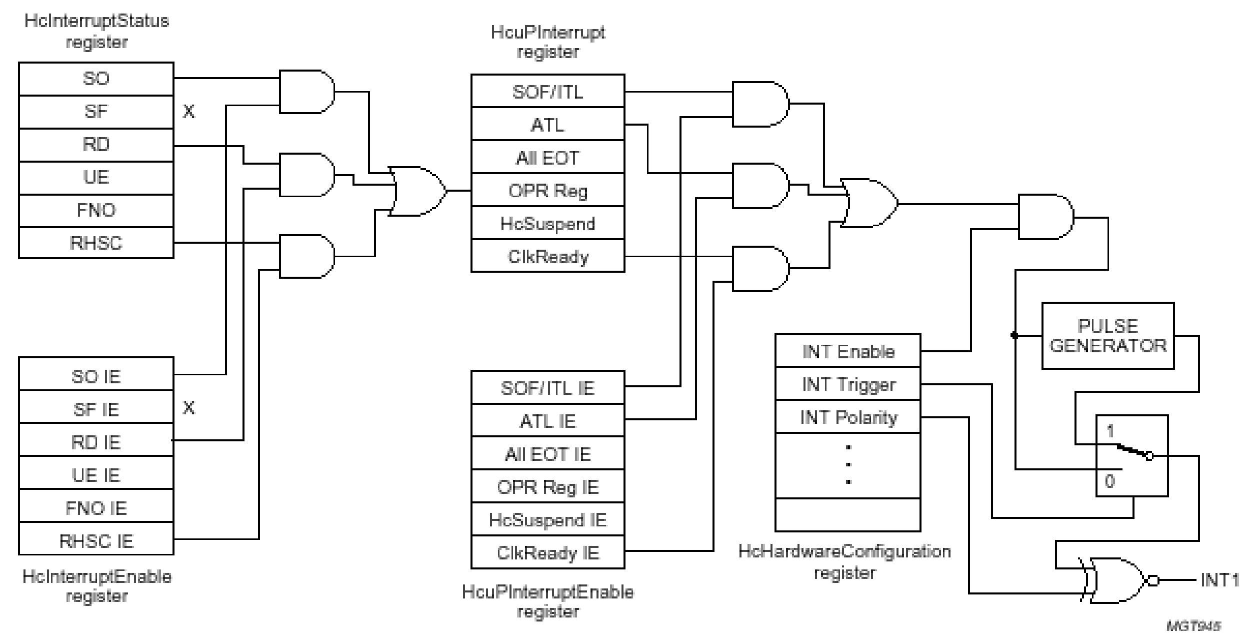


Fig 21. ISP1161A1 host controller interrupt logic

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 8).

Table 8. HcµPInterruptEnable register: bit allocation
Read index—25H; Write index—A5H

Bit	15	14	13	12	11	10	9	8
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
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	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	R/W	R/W	R/W	R/W	R/W	R/W	R/W

When initializing the interrupts available in the host controller of ISP1161A1, 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(HcµPInterrupt, 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);
```

Fig 22. 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 9). For example, the following code clears the root hub status change (RHSC) interrupt bit:

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

Table 9. HcInterruptStatus register: bit allocation
Read index—03H; Write index—83H

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	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	00H	00H	00H	00H	00H	00H	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							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
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	reserved	RHSC	FNO	UE	RD	SF	reserved	SO
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

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 10). For example, the following code clears the OPR_Reg interrupt:

```
WRITE_16BIT_REG(HcμPInterrupt, OPR_Reg);
```

Table 10. HcμPInterrupt register: bit allocation
Read index—24H; Write index—A4H

Bit	15	14	13	12	11	10	9	8
Symbol	reserved							
Reset	00H	00H	00H	00H	00H	00H	00H	00H
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	reserved	ClkReady	HC Suspended	OPR_Reg	reserved	AllEOT Interrupt	ATLInt	SOFITInt
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

Note: Since ISP1161A1 is a frame-based slave host controller, the microprocessor must update Proprietary 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 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	R/W	R/W	R/W	R/W	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	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		FI[13:8]					
Reset	0	0	1	0	1	1	1	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	FI[7:0]							
Reset	1	1	0	1	1	1	1	1
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	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 12), all bit fields except the DT bit are implementation specific (IS). For the ISA-based ISP1161A1 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 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	IS	IS	IS	IS	IS	IS	IS
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	00H	00H	00H	00H	00H	00H	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			NOCP	OCPM	DT	NPS	PSM
Reset	0	0	0	IS	IS	0	IS	IS
Access	R	R	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	IS
Access	R	R	R	R	R	R	R	R

The code example to initialize the HcRhDescriptorA register for the ISA-based ISP1161A1 evaluation board is given below.

```
#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);
```

Fig 23. Code example for initializing the HcDescriptorA register

With the HcRhDescriptorA register initialized, the LPSC (LocalPowerStatusChange) bit in the HcRhStatus register (see Table 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);
```

Fig 24. Code example for initializing the HcRhStatus register

Table 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	0	0	0	0	0	0
Access	W	R	R	R	R	R	R	R
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved						CCIC	LPSC
Reset	0	0	0	0	0	0	0	0
Access	R	R	R	R	R	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	R	R	R	R	R	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	R	R	R	R	R/W

For the ISA-based ISP1161A1 evaluation board, the PPCM field (see Table 14) must be set to logic 0 because the power switching mode is global power-on and the DR field (see Table 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 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	IS	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	IS	IS
Access	-	-	-	-	-	R/W	R/W	R/W

5.4.7 Setting the ITL and ATL buffer lengths

The host controller in ISP1161A1 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);
```

Fig 25. 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 ISP1161A1. The INT1 pin is usually connected to an interrupt controller through which the microprocessor receives an interrupt from

ISP1161A1. In the ISA-based ISP1161A1 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 ISP1161A1 evaluation board, the INT1 pin is usually set to IRQ10. When the ISP1161A1 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);
}
```

Fig 26. Code example to program the cascaded PICs on the PC motherboard

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.

```
Value = READ_32BIT_REG(HcControl);

// When writing a new value to the HcControl register, the state of the other bits
// in theregister 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);
```

Fig 27. 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 ISP1161A1 //
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 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 ISP1161A1 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.

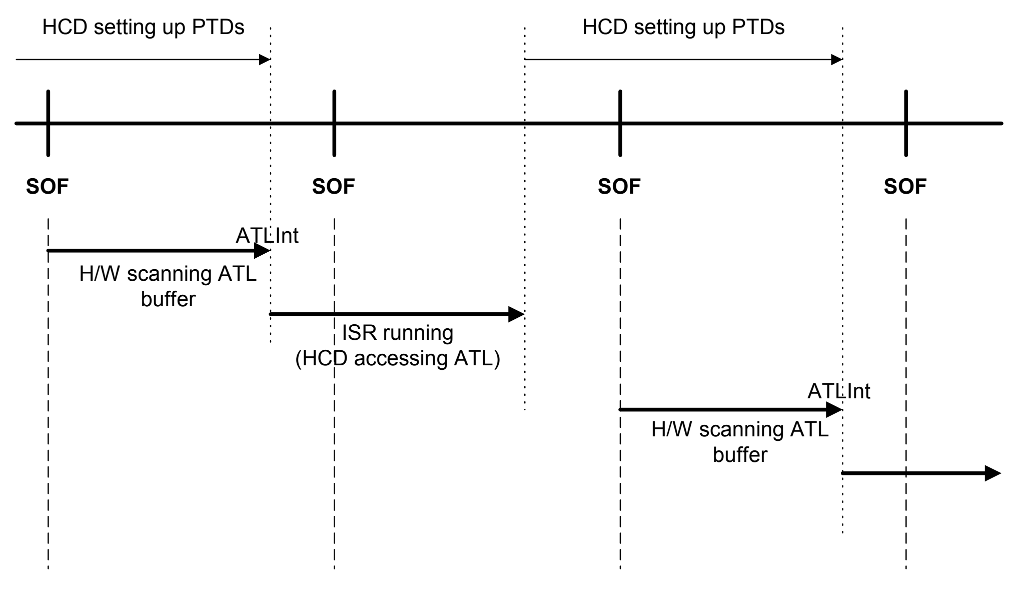


Fig 28. ATLInt interrupt flow

In the timing diagram in Fig 28, 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 Fig 28, 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 Fig 29 illustrates the case in which USB transactions occur in every frame.

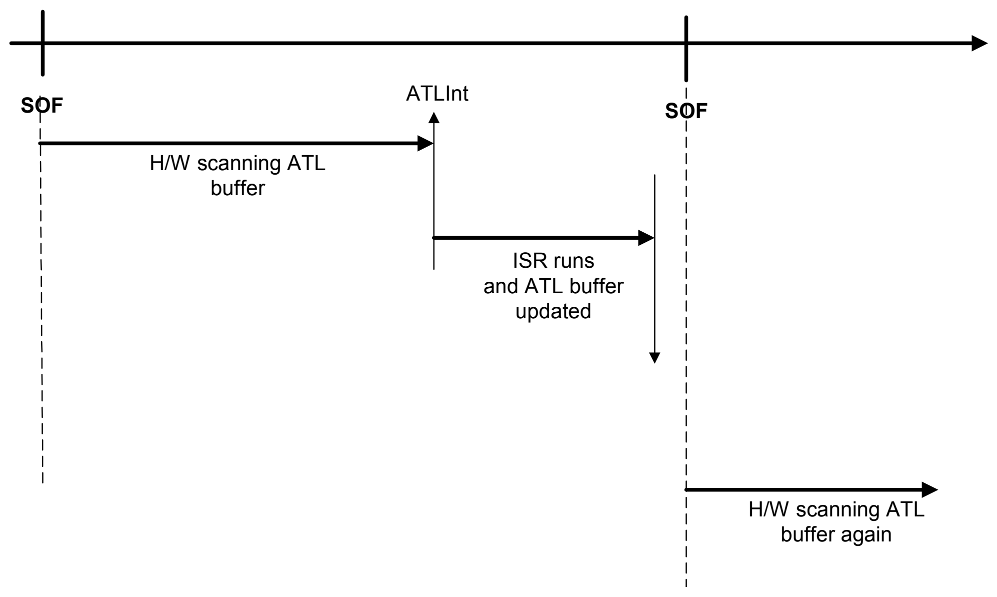

SOF
ATLInt
SOF
H/W scanning ATL buffer
ISR runs and ATL buffer updated
H/W scanning ATL buffer again

Fig 29. 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 Fig 30.

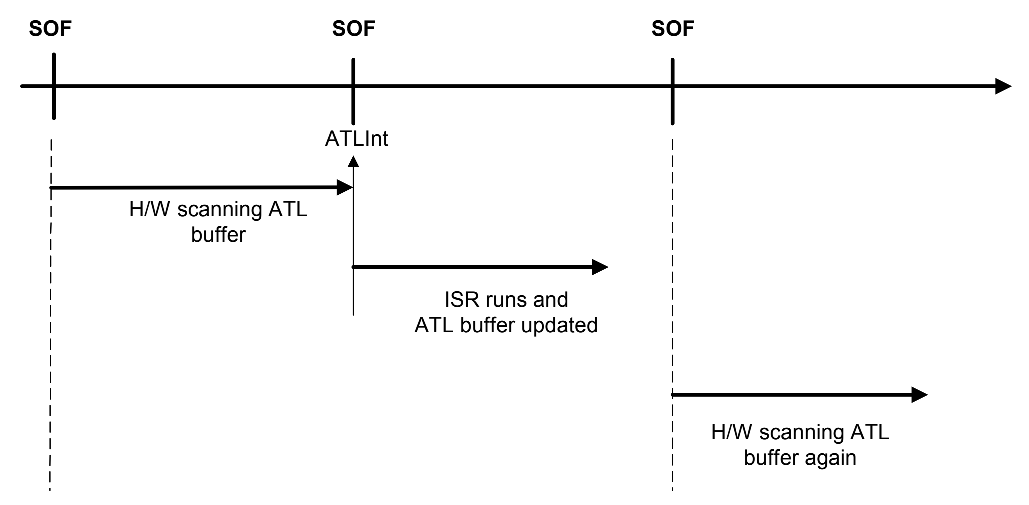

SOF
SOF
SOF
ATLInt
H/W scanning ATL buffer
ISR runs and ATL buffer updated
H/W scanning ATL buffer again

Fig 30. 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();
}
```

Fig 31. 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();

}
```

Fig 32. 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 Fig 33 illustrates how the ping pong buffer of the ITL buffer is accessed.

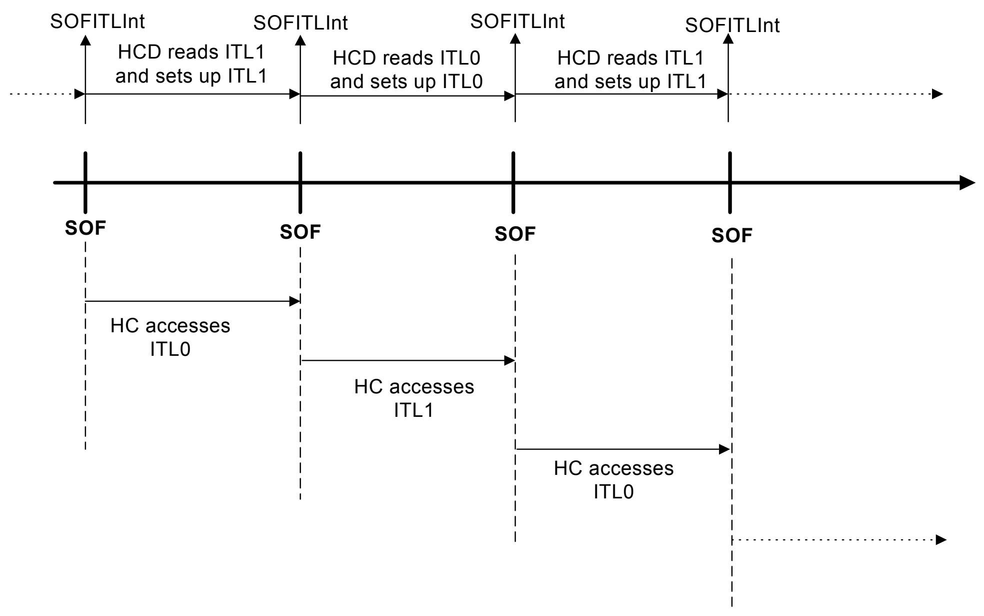


Fig 33. ITL buffer access flow

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();
}
```

Fig 34. Code example for writing to the ITL buffer

The following code example 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();
}
```

Fig 35. 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 Fig 36. The ITL buffer is processed first, followed by the interrupt and the ATL buffer.

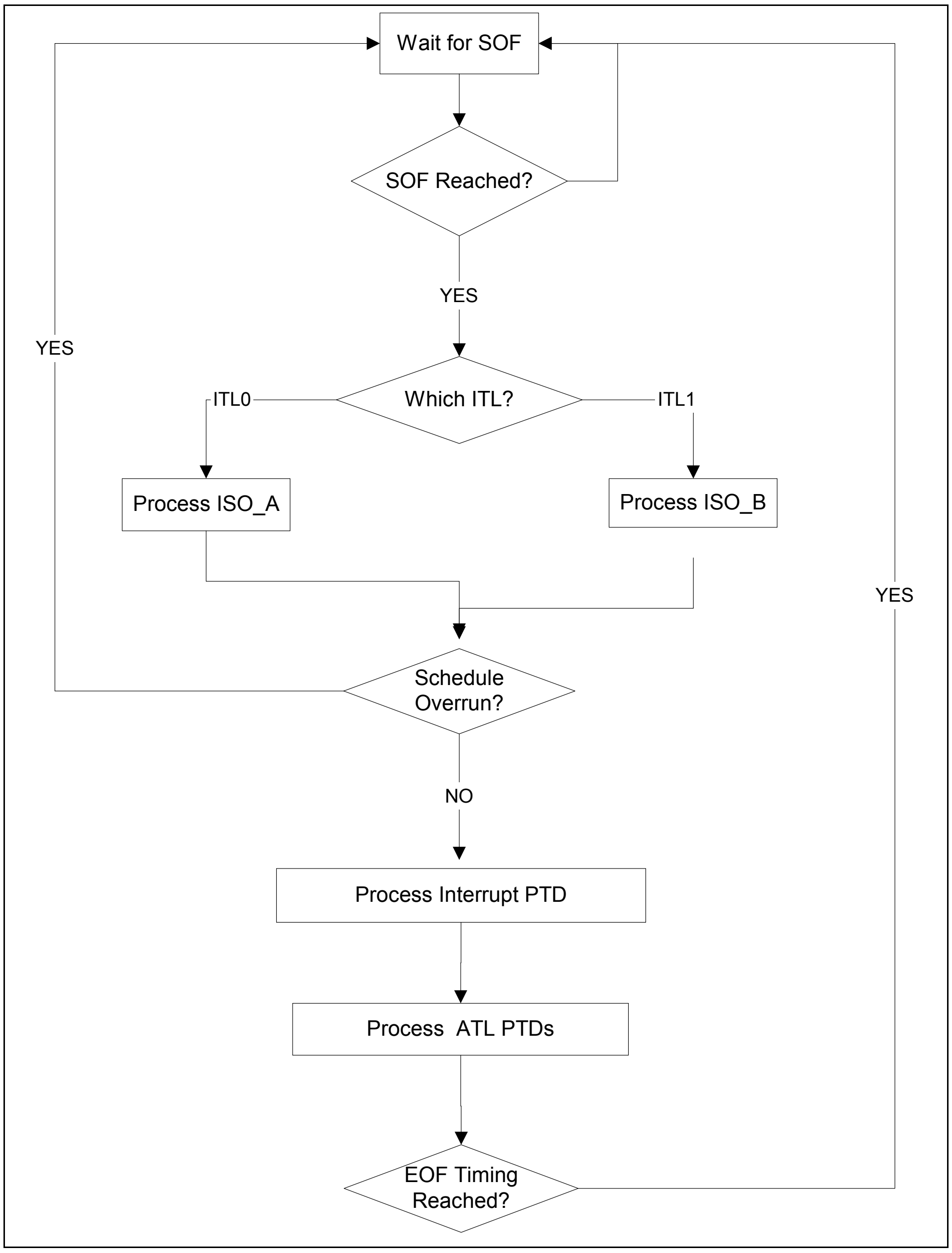


Fig 36. Host controller in the operational state flow chart

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 Table 15 and Table 16.

Table 15. PTD header fields

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							

Table 16. PTD header field descriptions

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

Symbol	Access	Description		
		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 the 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.		
DirectionPID[1:0]	R	**00** — SETUP **01** — OUT **10** — IN **11** — reserved		

Symbol	Access	Description
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	This is the USB address of the function containing the endpoint that this PTD refers to.

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 ISP1161A1 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. Fig 37 illustrates the PTD flow for a control transfer.

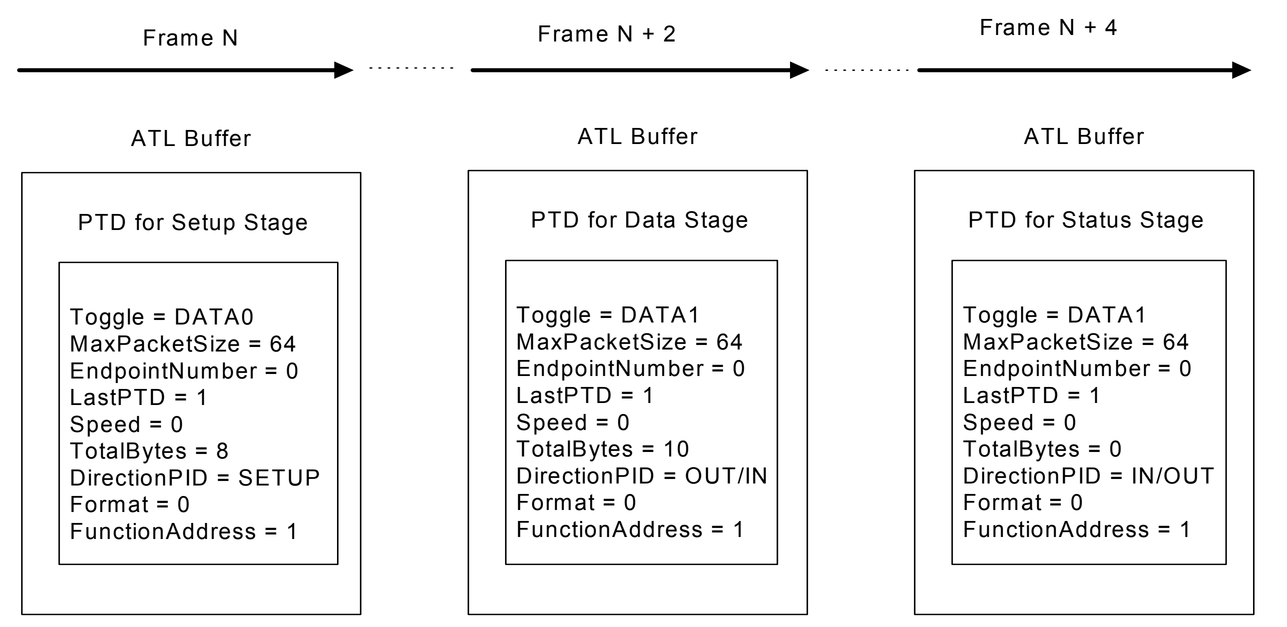


Fig 37. PTD flow for the control transfer

- 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

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 Fig 37). 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. illustrates the setting of the data toggle bit field across multiple PTDs.

Fig 38 illustrates the setting of the data toggle bit field across multiple PTDs.

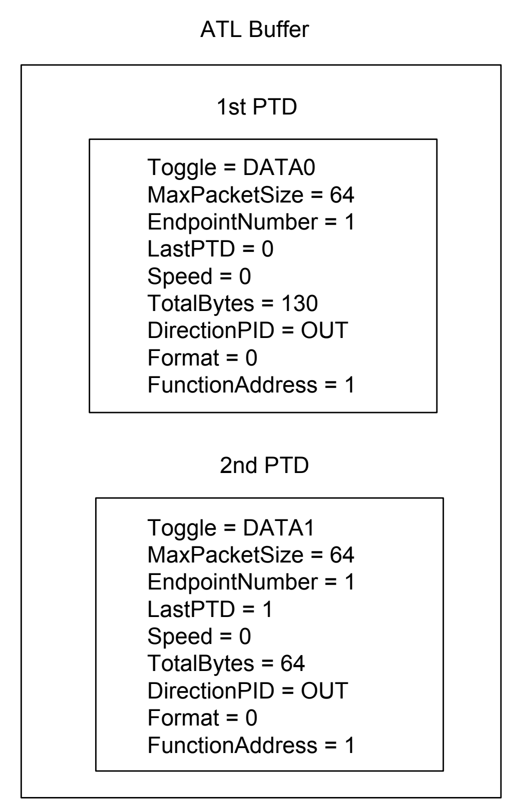


Fig 38. Data toggle bit setting example across multiple PTDs

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 Fig 39.

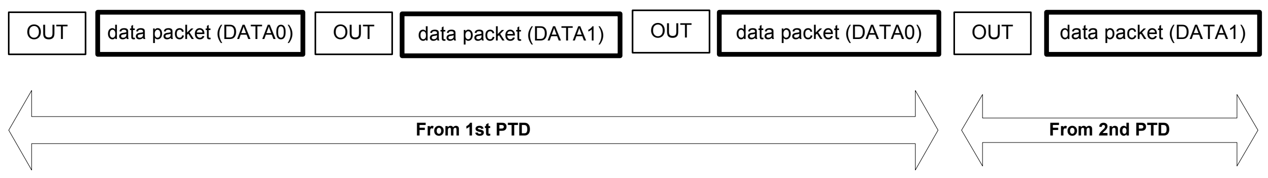


Fig 39. Data toggle bit setting in multiple PTD data packets

As shown in Fig 39, 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*). 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 Fig 40). 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.

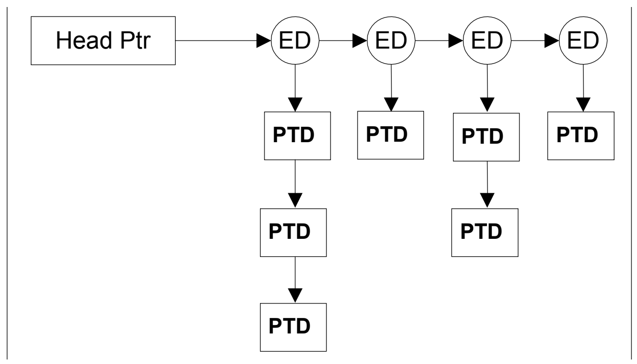


Fig 40. Typical list structure

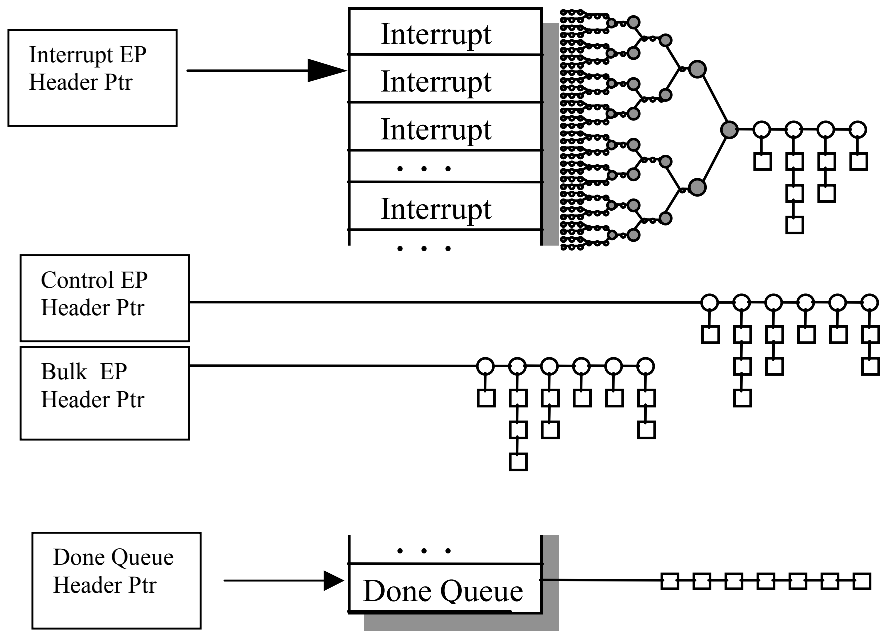


Fig 41. List processing data structure

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 17. USB transaction error codes

Error	Error Code	IN Token	OUT Token
Fatal errors			
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 *ISP1161A1 Frequently Asked Questions* document.

6. Programming the device controller of the ISP1161A1

The Device Controller (DC) of ISP1161A1 is a core based on ST-Ericsson ISP1181 device controller, which is a full-speed USB interface device with up to 14 configurable endpoints. You can access the device controller of ISP1161A1 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 ISP1161A1 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 ISP1161A1 is based on the firmware code of the ISP1161A1 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, see Fig 42.

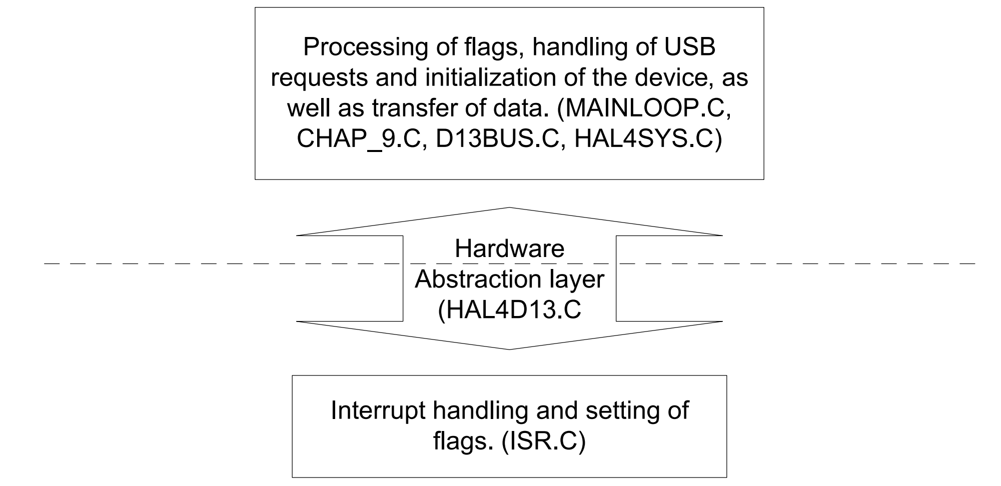

Processing of flags, handling of USB requests and initialization of the device, as well as transfer of data. (MAINLOOP.C, CHAP_9.C, D13BUS.C, HAL4SYS.C)
Hardware Abstraction layer (HAL4D13.C
Interrupt handling and setting of flags. (ISR.C)

Fig 42. Firmware structure of the ISP1161A1 device controller

As can be seen in Fig 42, 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 ISP1161A1, 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 ISP1161A1, the firmware defines a set of command interfaces that encapsulate all the functions used to access the device controller of ISP1161A1. 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 ISP1161A1. It retrieves data from the ISP1161A1 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 18 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 18. 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 polling mode

To develop the firmware in 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 the ISP1161A1

The following functions are defined as the device controller command interface of ISP1161A1 to simplify the device programming. These are implementations of the ISP1161A1 device controller command set, which is defined in the ISP1161A1 data sheet.

```
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 ISP1161A1 firmware is fully interrupt-driven. The flowchart of Interrupt Service Routine (ISR) is given in Fig 43.

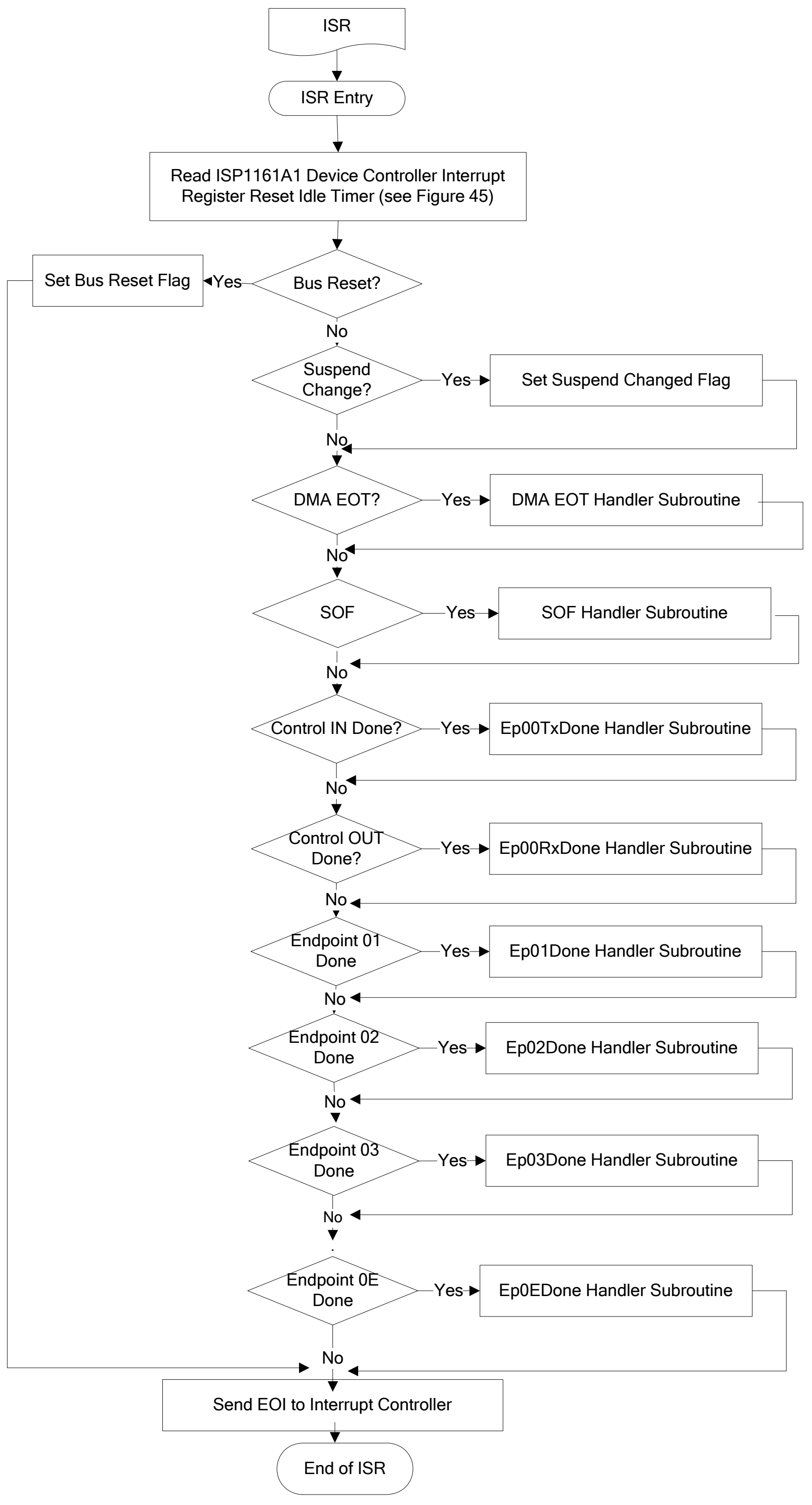

ISR
ISR Entry
Read ISP1161A1 Device Controller Interrupt Register Reset Idle Timer (see Figure 45)
Bus Reset?
Yes
Set Bus Reset Flag
No
Suspend Change?
Yes
Set Suspend Changed Flag
No
DMA EOT?
Yes
DMA EOT Handler Subroutine
No
SOF
Yes
SOF Handler Subroutine
No
Control IN Done?
Yes
Ep00TxDone Handler Subroutine
No
Control OUT Done?
Yes
Ep00RxDone Handler Subroutine
No
Endpoint 01 Done
Yes
Ep01Done Handler Subroutine
No
Endpoint 02 Done
Yes
Ep02Done Handler Subroutine
No
Endpoint 03 Done
Yes
Ep03Done Handler Subroutine
No
Endpoint 0E Done
Yes
Ep0EDone Handler Subroutine
No
Send EOI to Interrupt Controller
End of ISR

Fig 43. Flowchart of ISR

Table 19. 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.

Fig 44 contains the pseudocode of a typical interrupt service routine.

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

    i_st = ReadInterruptRegister();  /* See Fig 45 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 */
            }
        }
    }
```

Fig 44. Code example of a typical ISR

A pseudocode to read the Interrupt register is given in Fig 45.

```
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 */
}
```

Fig 45. Pseudocode 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;
```

Fig 46. Control flags

The task splitting between ISR and the main loop is that ISR collects data from the internal buffer of the ISP1161A1 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 ST-Ericsson.

6.5.4 Control endpoint handler

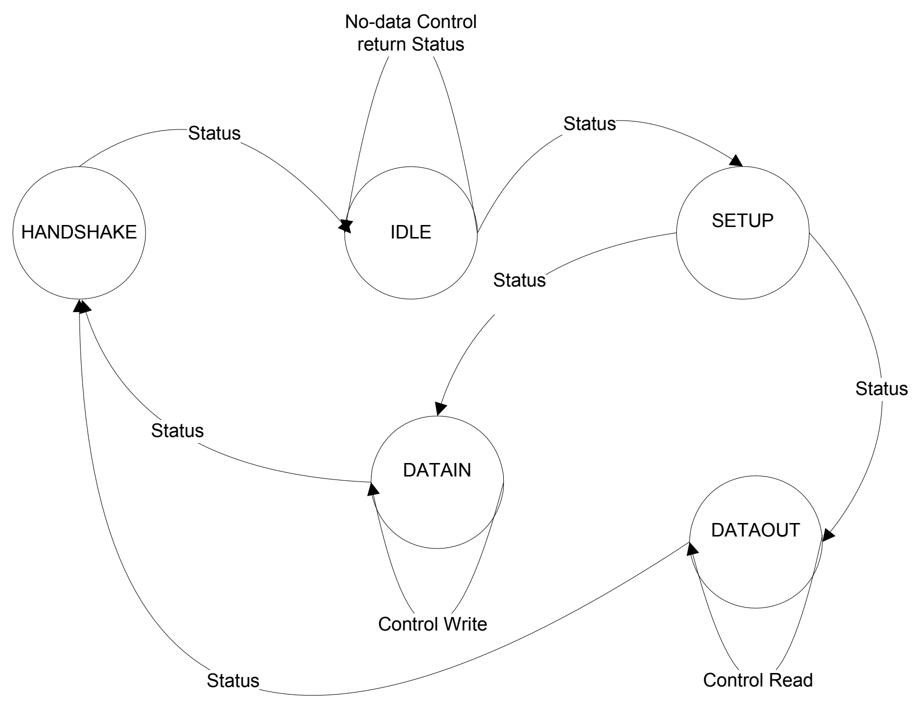


Fig 47. State machine of the control transfer

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. Fig 47 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

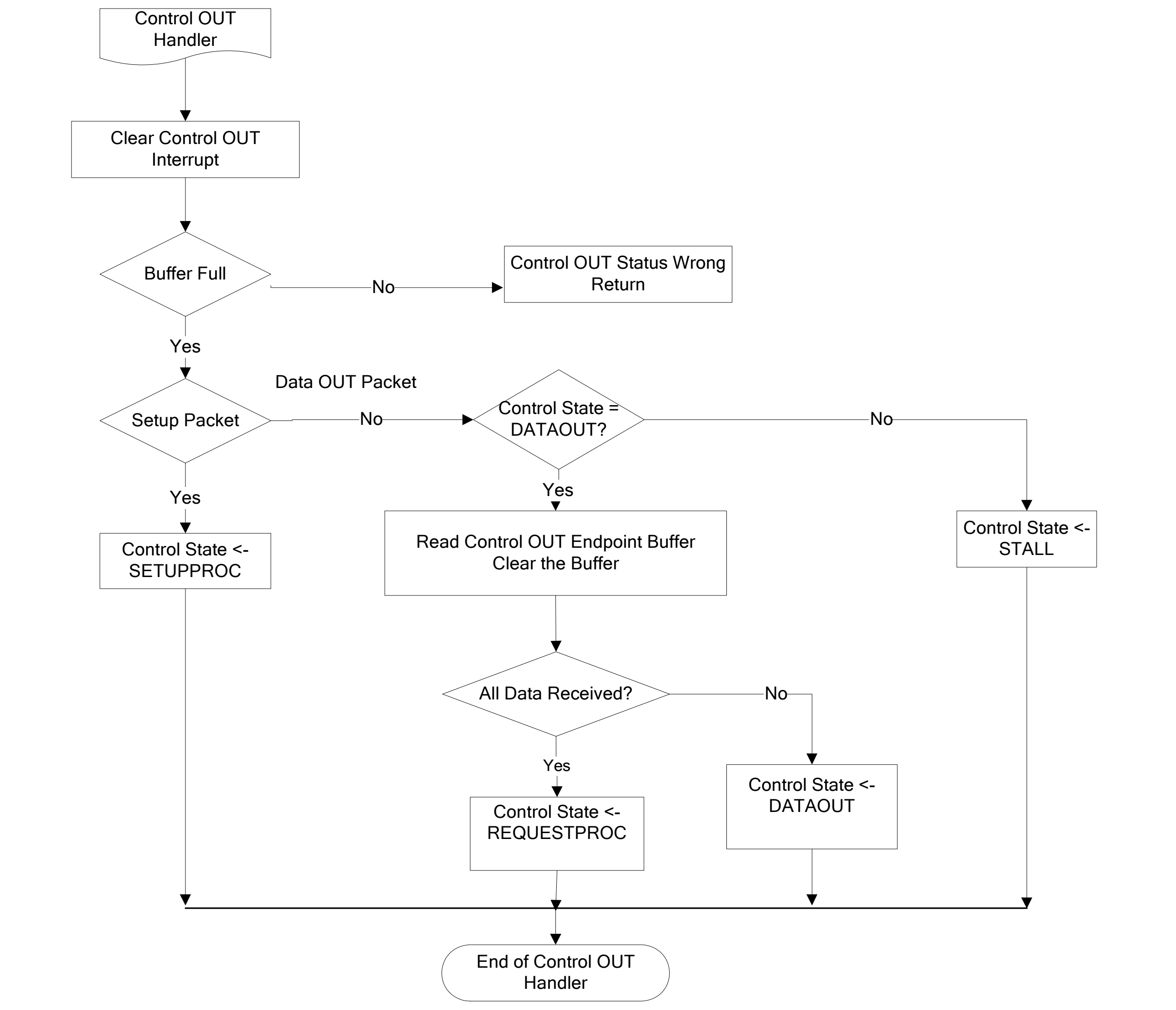

Control OUT Handler
Clear Control OUT Interrupt
Buffer Full
No
Control OUT Status Wrong Return
Yes
Data OUT Packet
Setup Packet
No
Control State = DATAOUT?
No
Yes
Yes
Control State <- SETUPPROC
Read Control OUT Endpoint Buffer Clear the Buffer
Control State <- STALL
All Data Received?
No
Yes
Control State <- REQUESTPROC
Control State <- DATAOUT
End of Control OUT Handler

Fig 48. Flowchart of the control OUT handler

The microprocessor must clear the control OUT interrupt bit on the device controller of ISP1161A1 and verify whether this endpoint is full. Fig 49 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. Fig 50 shows a pseudocode to read the Endpoint Status register (see Table 20 and Table 21). 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. Fig 51 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 ISP1161A1 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 */
}
```

Fig 49. 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;
}
```

Fig 50. Code example for reading the Endpoint Status register

A typical pseudocode to read the contents of an OUT buffer is given in Fig 51.

```
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 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;
}
```

Fig 51. Code example for reading the contents of an OUT buffer

Table 20. 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 21. 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 on receiving 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. Logic 1 indicates 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 set-up data has finished. Firmware must check this bit before sending an Acknowledge Setup command or stalling the endpoint. Upon reading a logic 1, 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 ISP1161A1 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 ISP1161A1 device controller by reading its Read Endpoint Status code (Code 0x51). Fig 52 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. Fig 53 contains a pseudocode to check whether the IN endpoint is empty or not. After verifying that the device controller of ISP1161A1 is in the appropriate state, the microprocessor proceeds to send the data packet, see Fig 54.

Fig 55 shows the flowchart of the control IN handler. Since the device controller of the ISP1161A1 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;
}
```

Fig 52. 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 */
}
```

Fig 53. 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;
}
```

Fig 54. Code example for writing the contents to an IN buffer

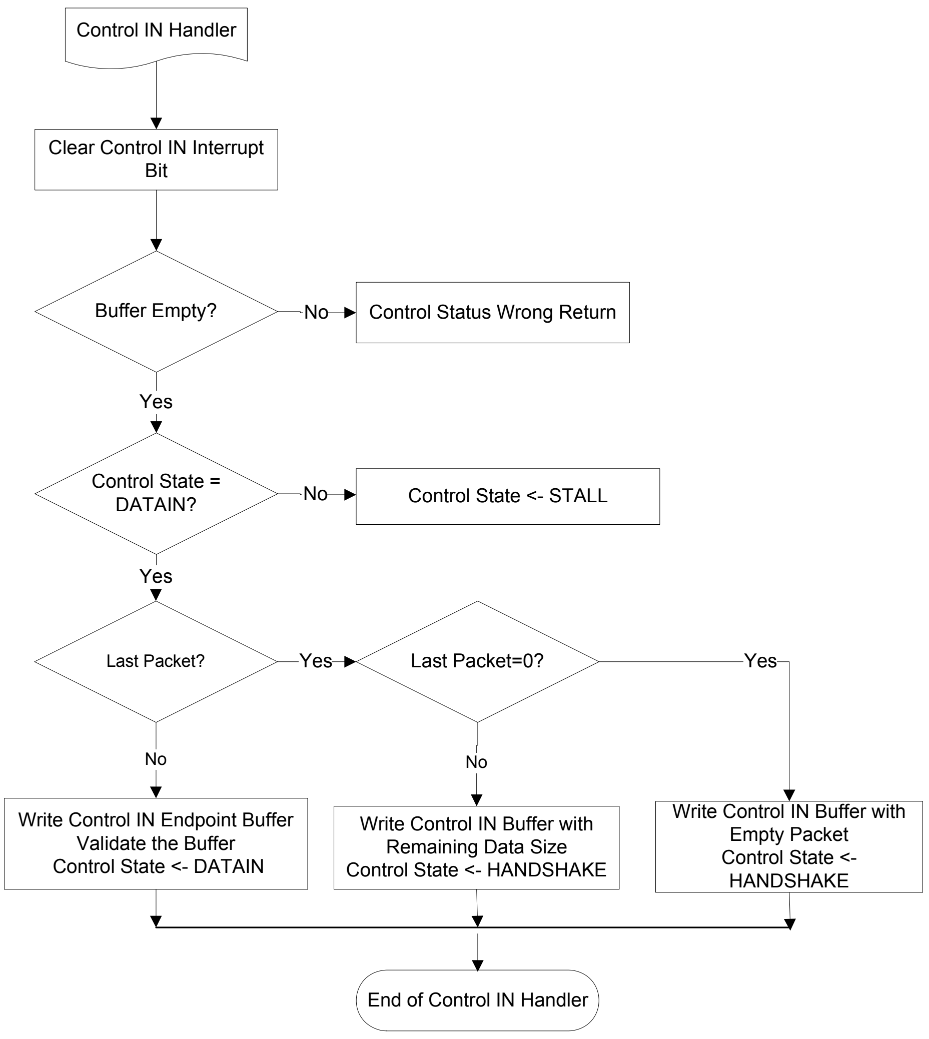

Control IN Handler
Clear Control IN Interrupt Bit
Buffer Empty?
No
Control Status Wrong Return
Yes
Control State = DATAIN?
No
Control State <- STALL
Yes
Last Packet?
Yes
Last Packet=0?
Yes
No
No
Write Control IN Endpoint Buffer Validate the Buffer Control State <- DATAIN
Write Control IN Buffer with Remaining Data Size Control State <- HANDSHAKE
Write Control IN Buffer with Empty Packet Control State <- HANDSHAKE
End of Control IN Handler

Fig 55. 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 ISP1161A1 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 22 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 23 and Table 24, respectively.

Table 22. 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 23. 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 24. 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 Fig 56.

```
#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. */
        );
```

Fig 56. Code example for configuring a bulk OUT or bulk IN endpoint

The function definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is as follows:

```
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);
}
```

Fig 57. 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 ISP1161A1 generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1161A1 device controller and verify the data length. The flowchart of the bulk OUT handler is given in Fig 58.

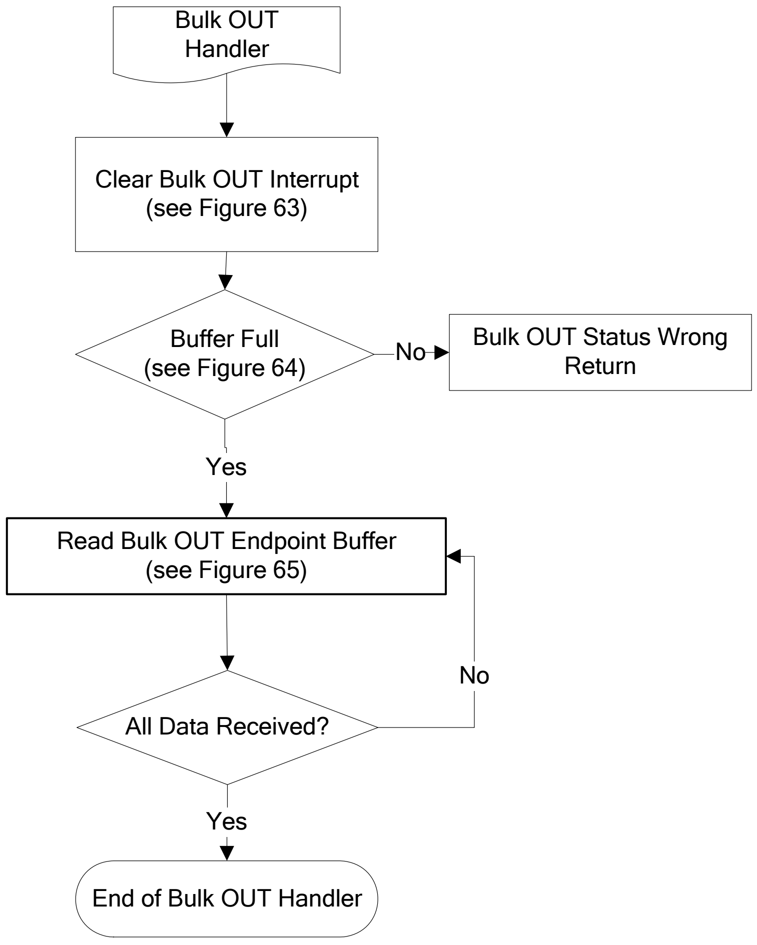

Bulk OUT Handler
Clear Bulk OUT Interrupt (see Figure 63)
Buffer Full (see Figure 64)
No
Bulk OUT Status Wrong Return
Yes
Read Bulk OUT Endpoint Buffer (see Figure 65)
No
All Data Received?
Yes
End of Bulk OUT Handler

Fig 58. Flowchart of the bulk OUT handler

Below is 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;
}
```

Fig 59. 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 */
}
```

Fig 60. 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;
}
```

Fig 61. 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 ISP1161A1 generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1161A1 device controller and return the data packet to be sent. The flowchart of the bulk IN handler is given in Fig 62.

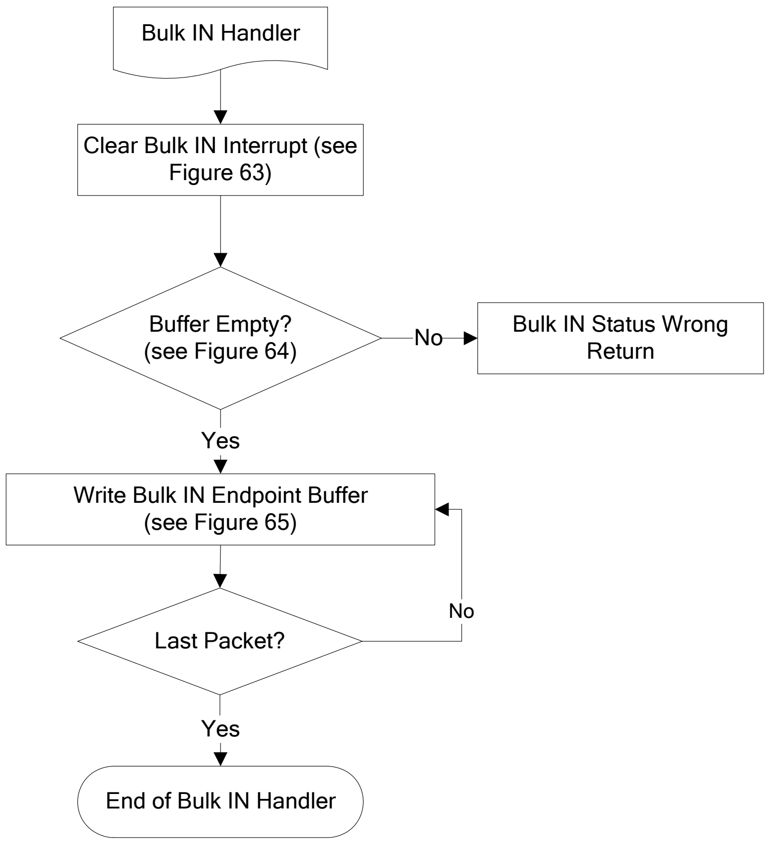

Bulk IN Handler
Clear Bulk IN Interrupt (see Figure 63)
Buffer Empty? (see Figure 64)
No
Bulk IN Status Wrong Return
Yes
Write Bulk IN Endpoint Buffer (see Figure 65)
Last Packet?
No
Yes
End of Bulk IN Handler

Fig 62. Flowchart of the bulk IN handler

A pseudocode for reading the Endpoint Status register is given below. 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;
}
```

Fig 63. 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 */
}
```

Fig 64. 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;
}
```

Fig 65. Code example for writing the contents into a bulk IN buffer

6.5.8 ISO endpoint handler

Table 25 contains the recommended register programming in the Endpoint Configuration register for an ISO endpoint.

Table 25. 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

Fig 66 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 */
        );
```

Fig 66. Code example for configuring an ISO OUT or ISO IN endpoint

The function definition of SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is given in Fig 67.

```
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);
}
```

Fig 67. Function definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)

Fig 68 and Fig 69 contain the flowcharts of the ISO OUT handler and the ISO IN handler, respectively.

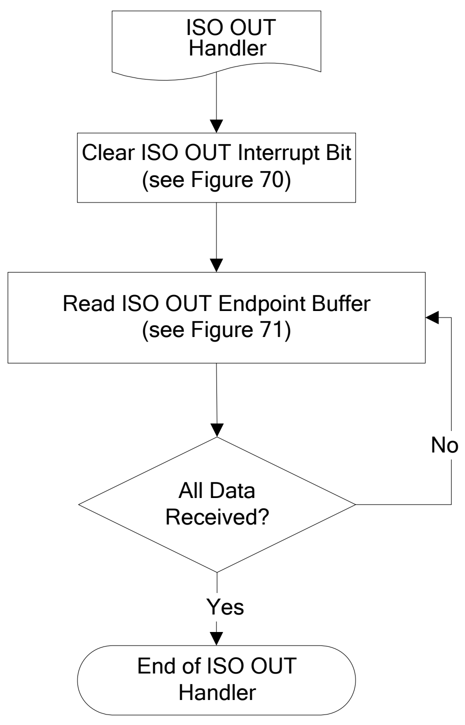

ISO OUT Handler
Clear ISO OUT Interrupt Bit (see Figure 70)
Read ISO OUT Endpoint Buffer (see Figure 71)
All Data Received?
No
Yes
End of ISO OUT Handler

Fig 68. Flowchart of the ISO OUT handler

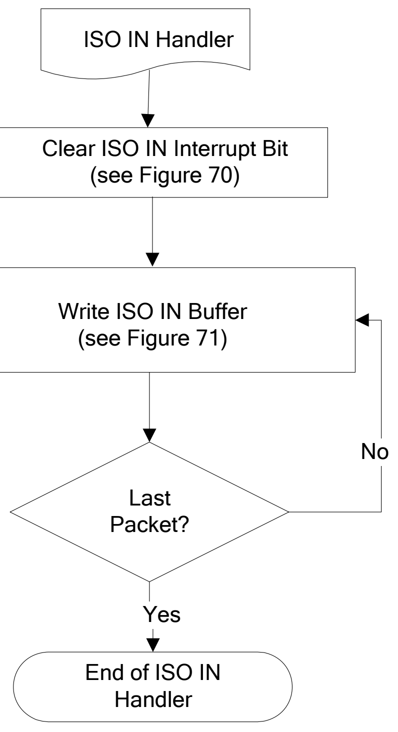

ISO IN Handler
Clear ISO IN Interrupt Bit (see Figure 70)
Write ISO IN Buffer (see Figure 71)
Last Packet?
No
Yes
End of ISO IN Handler

Fig 69. 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 Fig 70. 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;
}
```

Fig 70. 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;
}
```

Fig 71. 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;
}
```

Fig 72. 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 ISP1161A1 device controller will not operate before the microprocessor is ready to serve the ISP1161A1 device controller.

The flowchart of the main loop is given in Fig 73. 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.

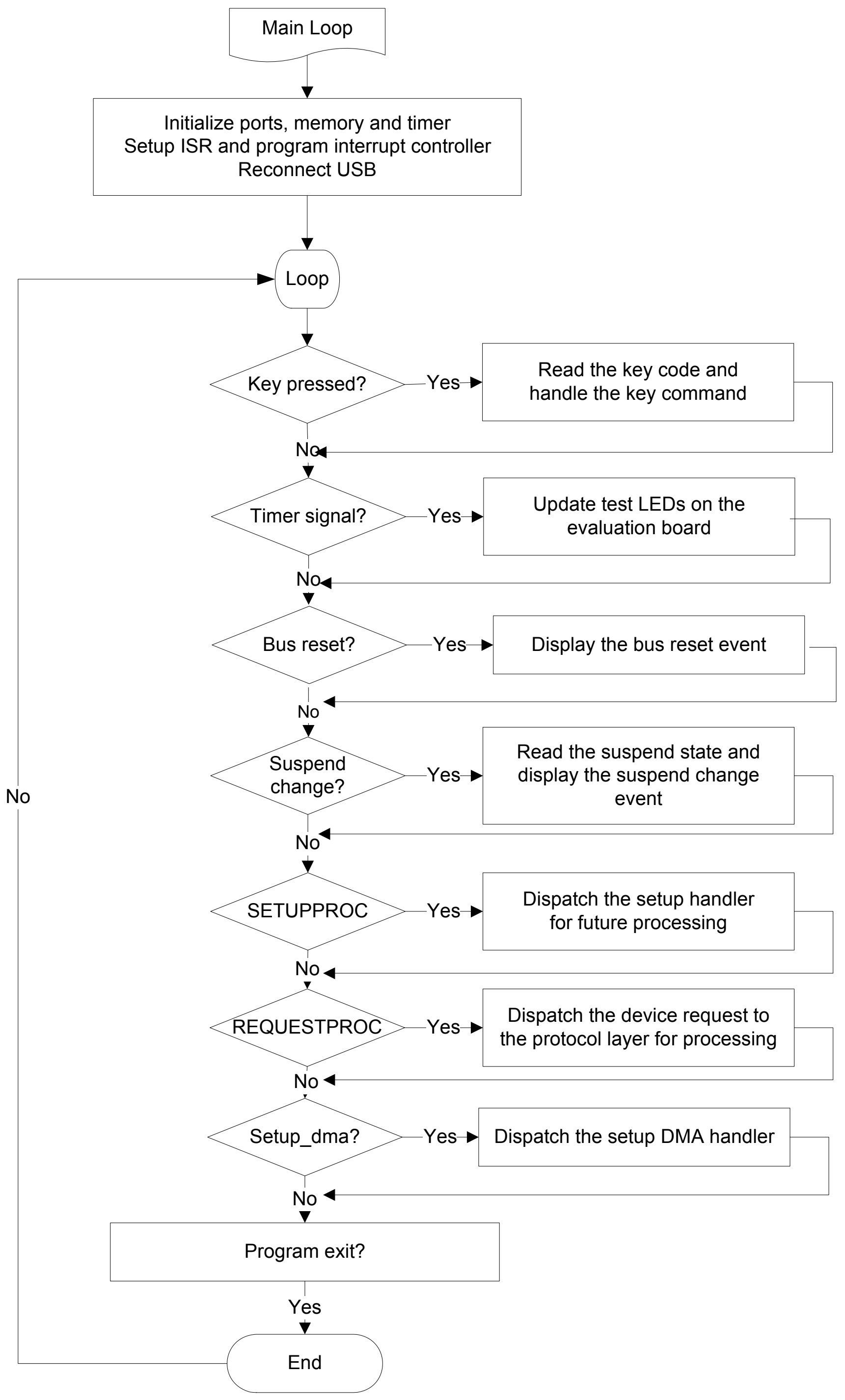

Main Loop
Initialize ports, memory and timer
Setup ISR and program interrupt controller
Reconnect USB
Loop
Key pressed?
Yes
Read the key code and handle the key command
No
Timer signal?
Yes
Update test LEDs on the evaluation board
No
Bus reset?
Yes
Display the bus reset event
No
Suspend change?
Yes
Read the suspend state and display the suspend change event
No
SETUPPROC
Yes
Dispatch the setup handler for future processing
No
REQUESTPROC
Yes
Dispatch the device request to the protocol layer for processing
No
Setup_dma?
Yes
Dispatch the setup DMA handler
No
Program exit?
Yes
End
No

Fig 73. Flowchart of the main loop

Table 26. 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 27. 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[1]. This bit is ignored if EXTPUL = 1 in the HardwareConfiguration register. Bus reset value: unchanged.

Fig 74 contains a pseudocode for writing to the Mode register. An example on setting the SOFCT bit to enable SoftConnect is given in Fig 75.

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

Fig 74. 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
*/
);
```

Fig 75. Code example on setting the SOFCT bit to enable SoftConnect

[1] SoftConnect is a trademark of ST-Ericsson.

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 *Universal Serial Bus 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 Fig 76. 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.

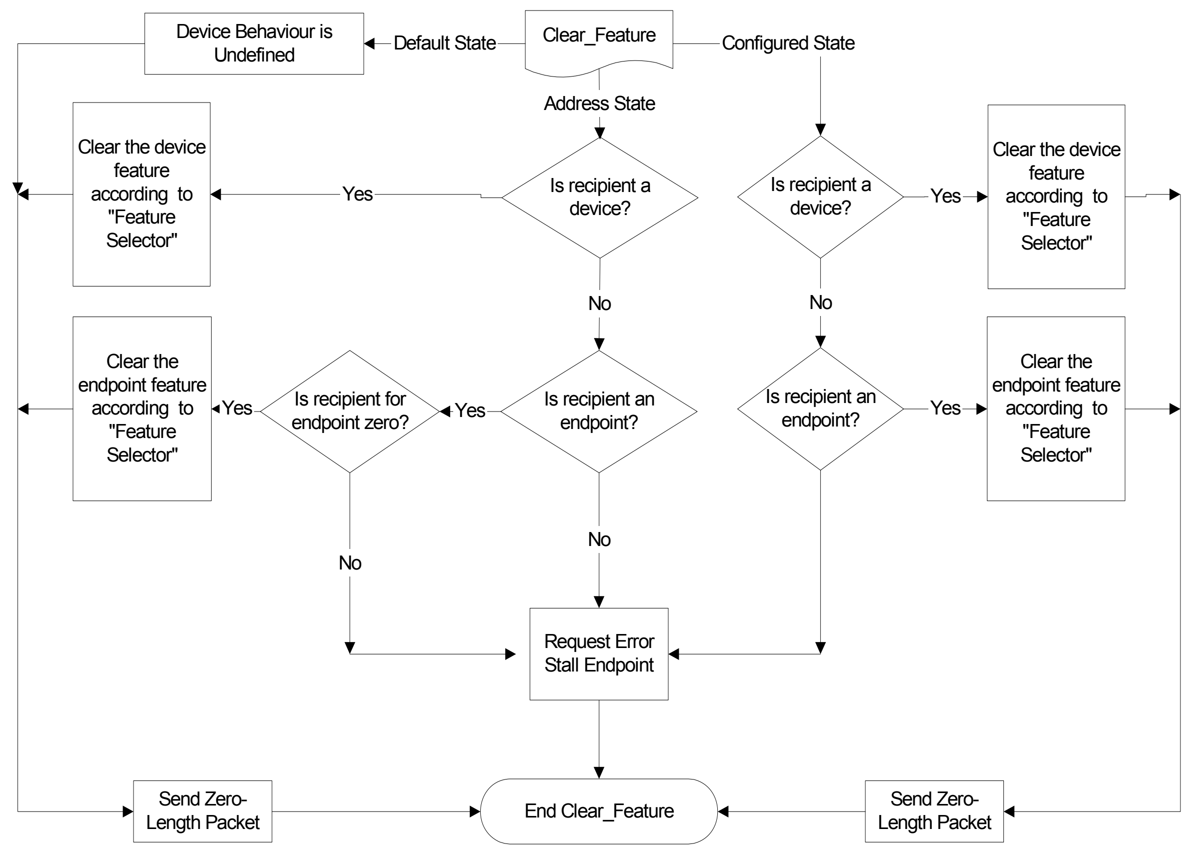

Device Behaviour is Undefined
Default State
Clear_Feature
Configured State
Address State
Is recipient a device?
Yes
No
Clear the device feature according to "Feature Selector"
Is recipient a device?
Yes
No
Clear the device feature according to "Feature Selector"
Clear the endpoint feature according to "Feature Selector"
Yes
Is recipient for endpoint zero?
Yes
Is recipient an endpoint?
Is recipient an endpoint?
Yes
Clear the endpoint feature according to "Feature Selector"
No
No
Request Error Stall Endpoint
Send Zero-Length Packet
End Clear_Feature
Send Zero-Length Packet

Fig 76. 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 Fig 54, 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 Fig 54
```

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 */
}
```

Fig 77. 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. Fig 78 shows the Get Status flowchart.

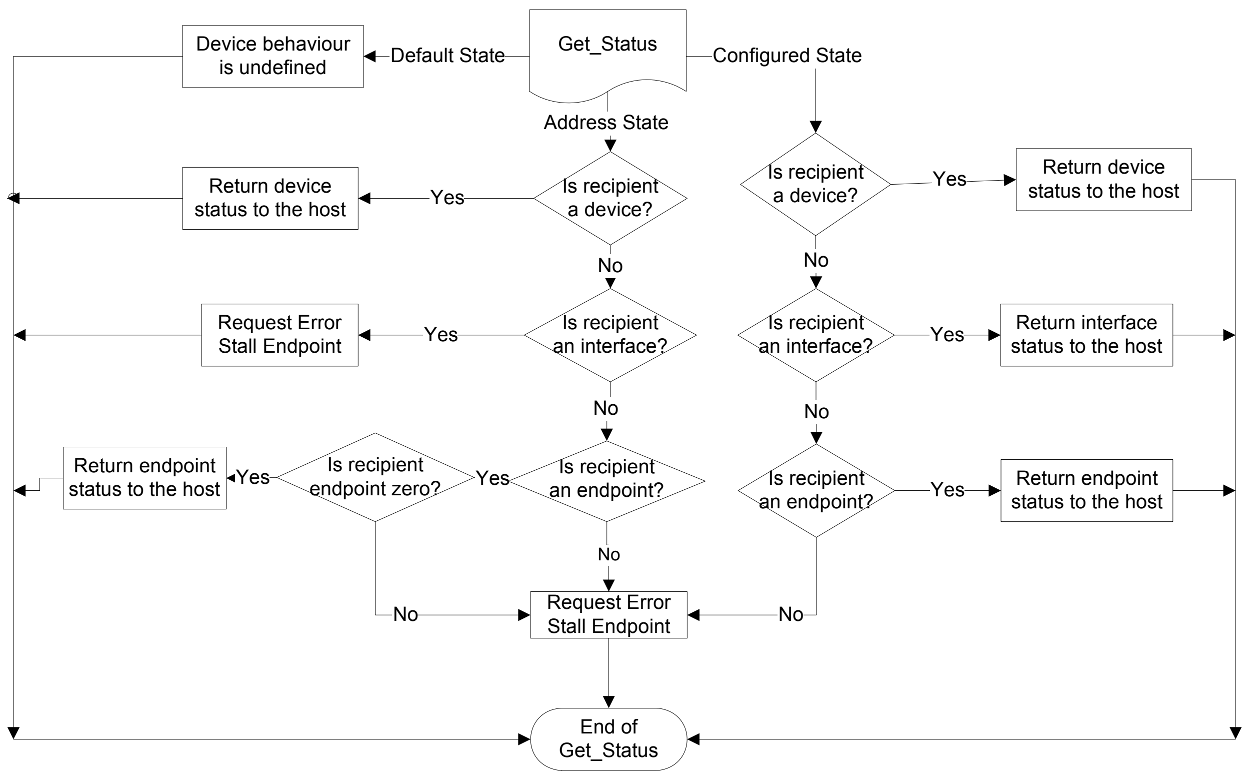

Get_Status
Default State
Device behaviour is undefined
Configured State
Address State
Is recipient a device?
Yes
Return device status to the host
No
Is recipient an interface?
Request Error Stall Endpoint
Return interface status to the host
Is recipient an endpoint?
Is recipient endpoint zero?
Return endpoint status to the host
Request Error Stall Endpoint
End of Get_Status

Fig 78. Flowchart of Get Status

6.7.3 Set Address request

In the Set Address request (see Fig 79), 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.

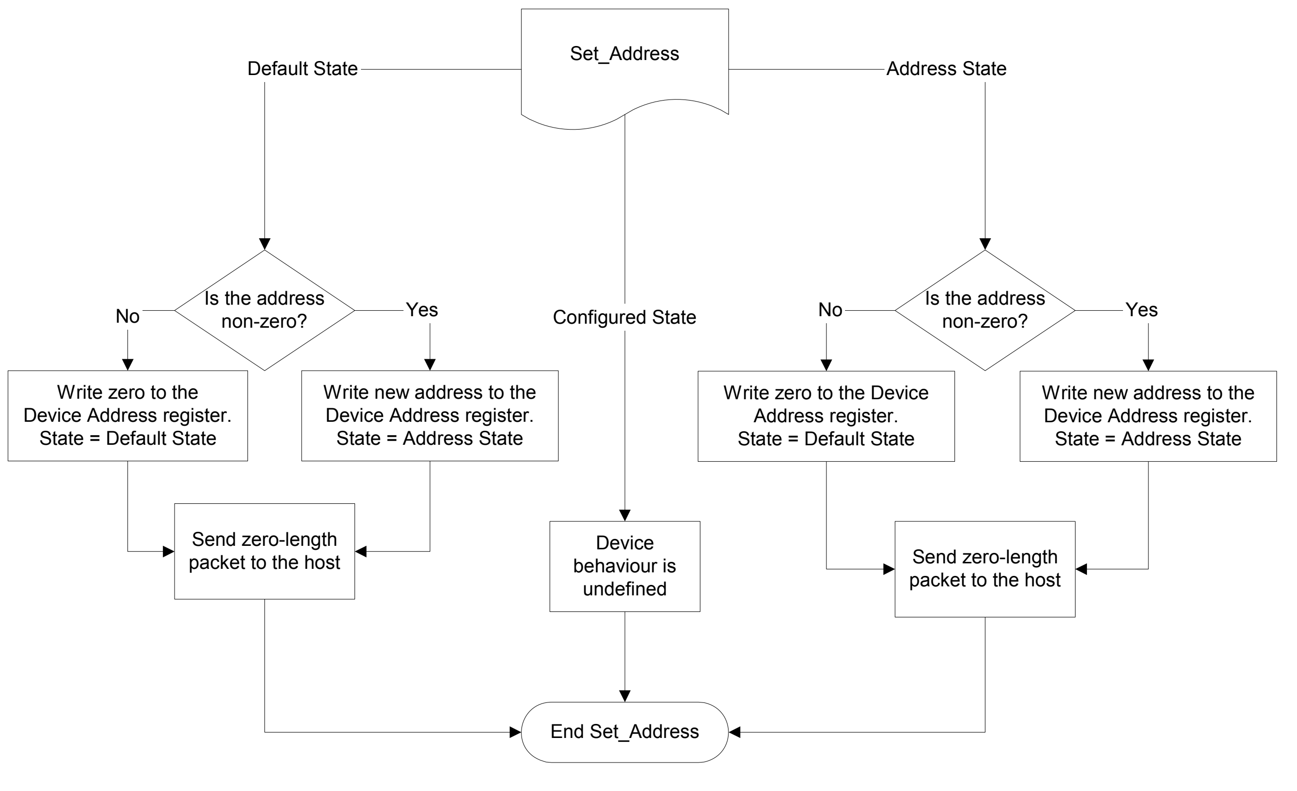

Set_Address
Default State
Address State
Configured State
Is the address non-zero?
No
Yes
Write zero to the Device Address register. State = Default State
Write new address to the Device Address register. State = Address State
Send zero-length packet to the host
Device behaviour is undefined
Is the address non-zero?
No
Yes
Write zero to the Device Address register. State = Default State
Write new address to the Device Address register. State = Address State
Send zero-length packet to the host
End Set_Address

Fig 79. Flowchart of Set Address

Fig 80 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);
}
```

Fig 80. Code example of the Set Address routine

Table 28. 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 29. Device Address register: bit description

Bit	Symbol	Description
7	DEVEN	A logic 1 enables the device.
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 Fig 81), 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.

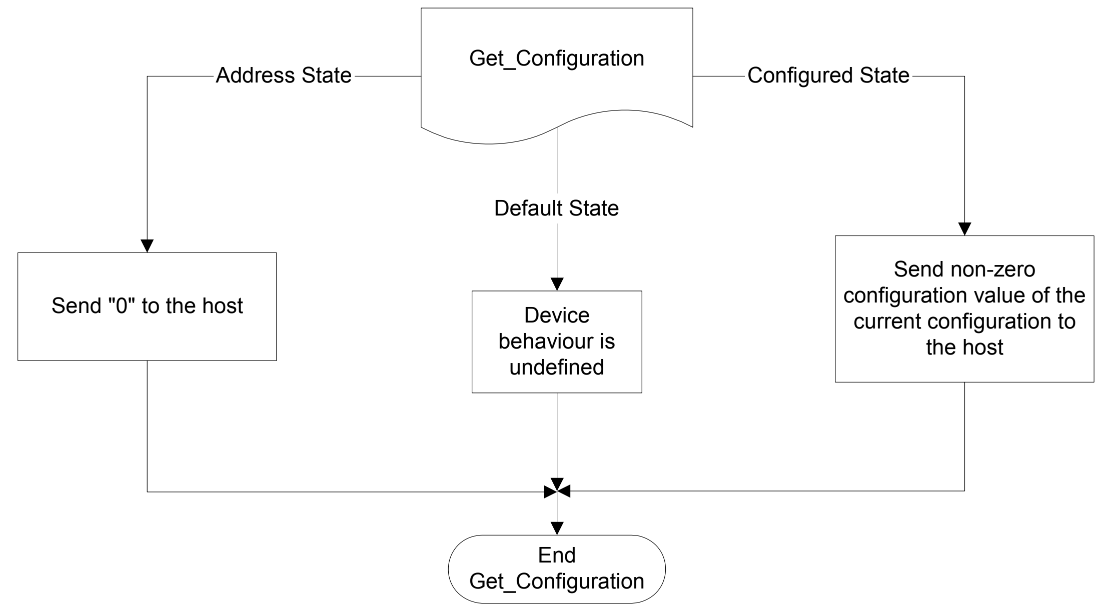

Get_Configuration
Address State
Configured State
Default State
Send "0" to the host
Device behaviour is undefined
Send non-zero configuration value of the current configuration to the host
End Get_Configuration

Fig 81. 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. Fig 82 shows the flowchart of Get Descriptor.

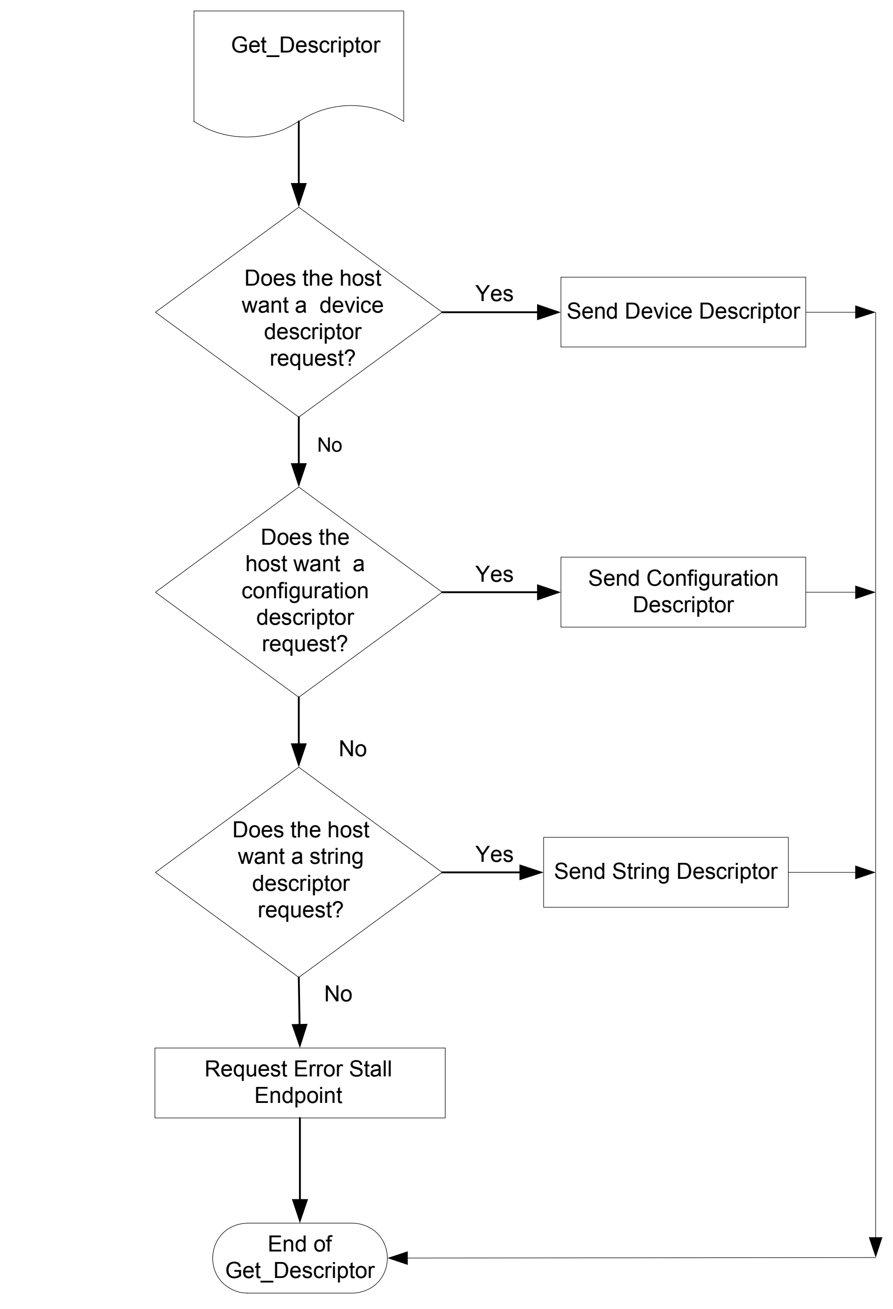

Get_Descriptor
Does the host want a device descriptor request?
Yes
Send Device Descriptor
No
Does the host want a configuration descriptor request?
Yes
Send Configuration Descriptor
No
Does the host want a string descriptor request?
Yes
Send String Descriptor
No
Request Error Stall Endpoint
End of Get_Descriptor

Fig 82. Flowchart of Get Descriptor

6.7.6 Set Configuration request

For the Set Configuration request (see Fig 83), 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.

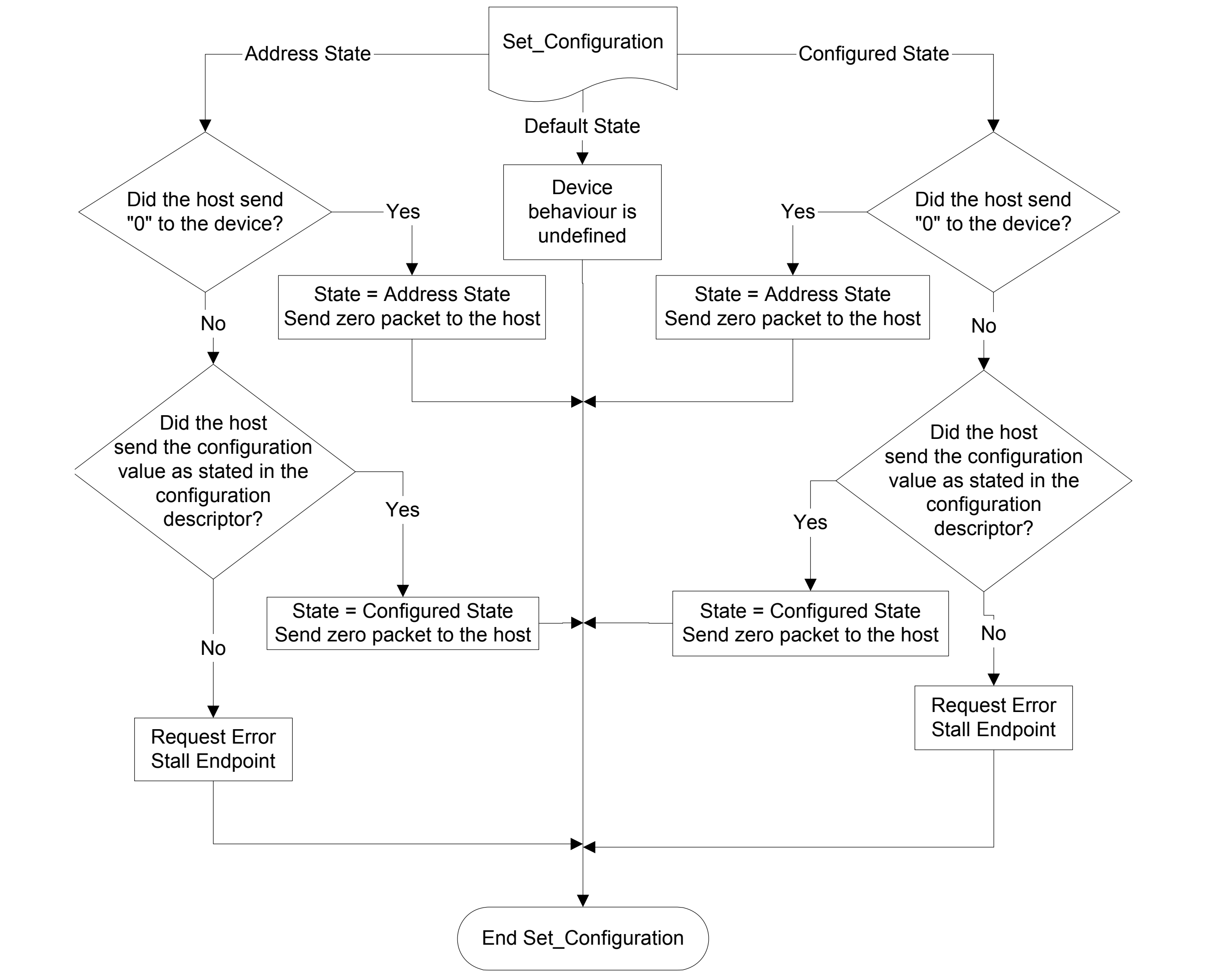

Set_Configuration
Address State
Configured State
Default State
Did the host send "0" to the device?
Yes
Device behaviour is undefined
Yes
Did the host send "0" to the device?
State = Address State Send zero packet to the host
State = Address State Send zero packet to the host
No
No
Did the host send the configuration value as stated in the configuration descriptor?
Yes
Did the host send the configuration value as stated in the configuration descriptor?
Yes
State = Configured State Send zero packet to the host
State = Configured State Send zero packet to the host
No
No
Request Error Stall Endpoint
Request Error Stall Endpoint
End Set_Configuration

Fig 83. Flowchart of Set Configuration

6.7.7 Get and Set Interface requests

For the Get and Set Interface requests (see flowcharts in Fig 84 and Fig 85), the microprocessor just needs to send one zero-data packet to the host because the ST-Ericsson evaluation board only supports one type of interface. For the Set Interface request on the ST-Ericsson evaluation board, the microprocessor need not do anything except to send one zero data packet to the host as the acknowledgment phase.

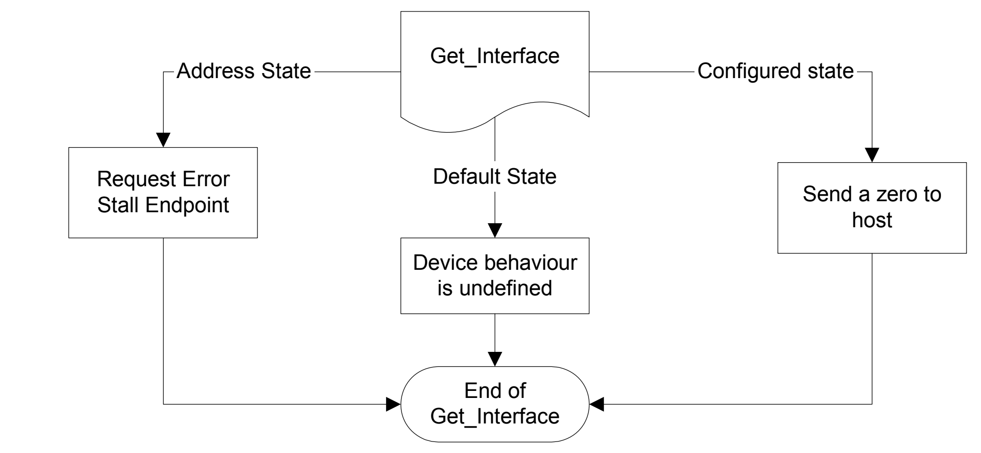


Fig 84. Flowchart of Get Interface

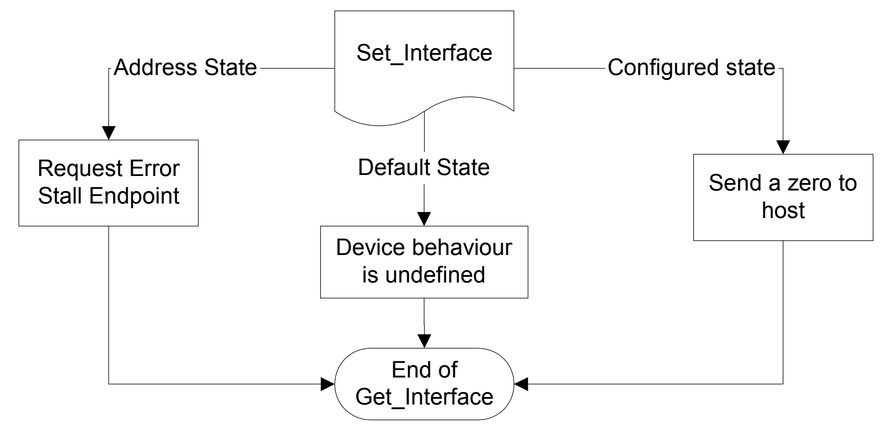


Fig 85. Flowchart of Set Interface

6.7.8 Set Feature request

The Set Feature request is just the opposite of the Clear Feature request. Fig 86 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 ISP1161A1 specific endpoint must be stalled through the Write Endpoint Status command.

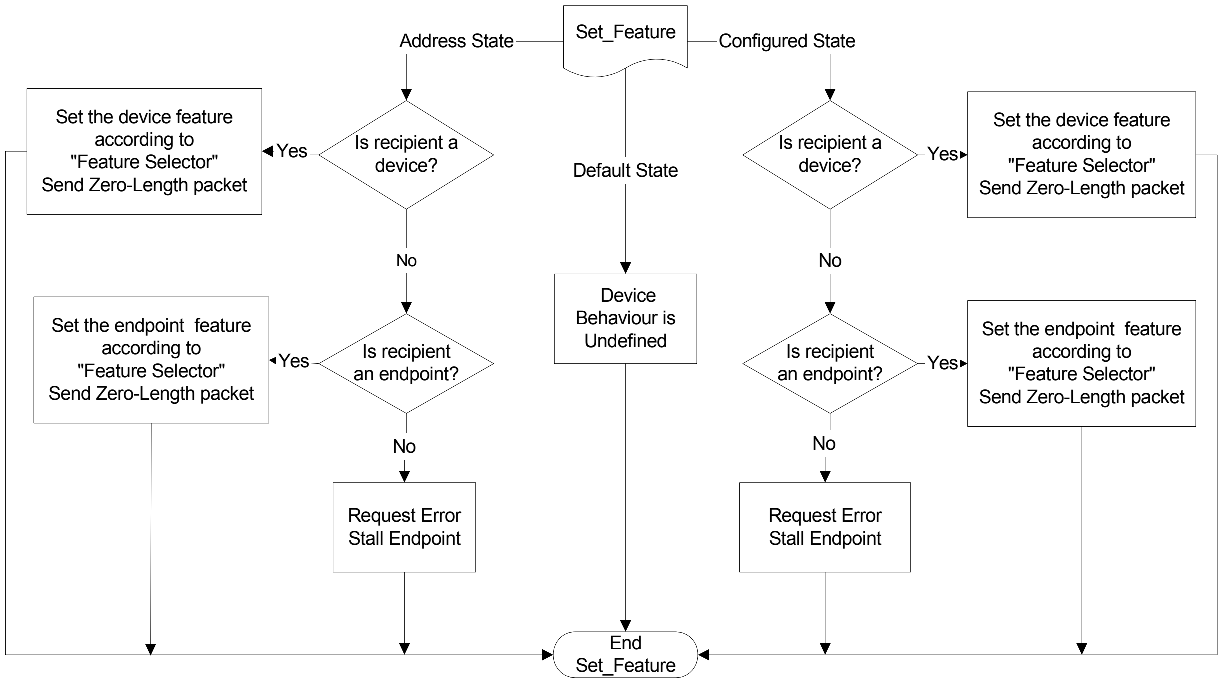

Set_Feature
Address State
Configured State
Default State
Is recipient a device?
Yes
No
Is recipient an endpoint?
Set the device feature according to "Feature Selector" Send Zero-Length packet
Set the endpoint feature according to "Feature Selector" Send Zero-Length packet
Device Behaviour is Undefined
Request Error Stall Endpoint
End Set_Feature

Fig 86. Flowchart of Set Feature

6.7.9 Class request

Support for class requests is not included in the device controller of the ISP1161A1 sample firmware.

6.8 Vendor request

In the ISP1161A1 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 30.

Table 30. Device request

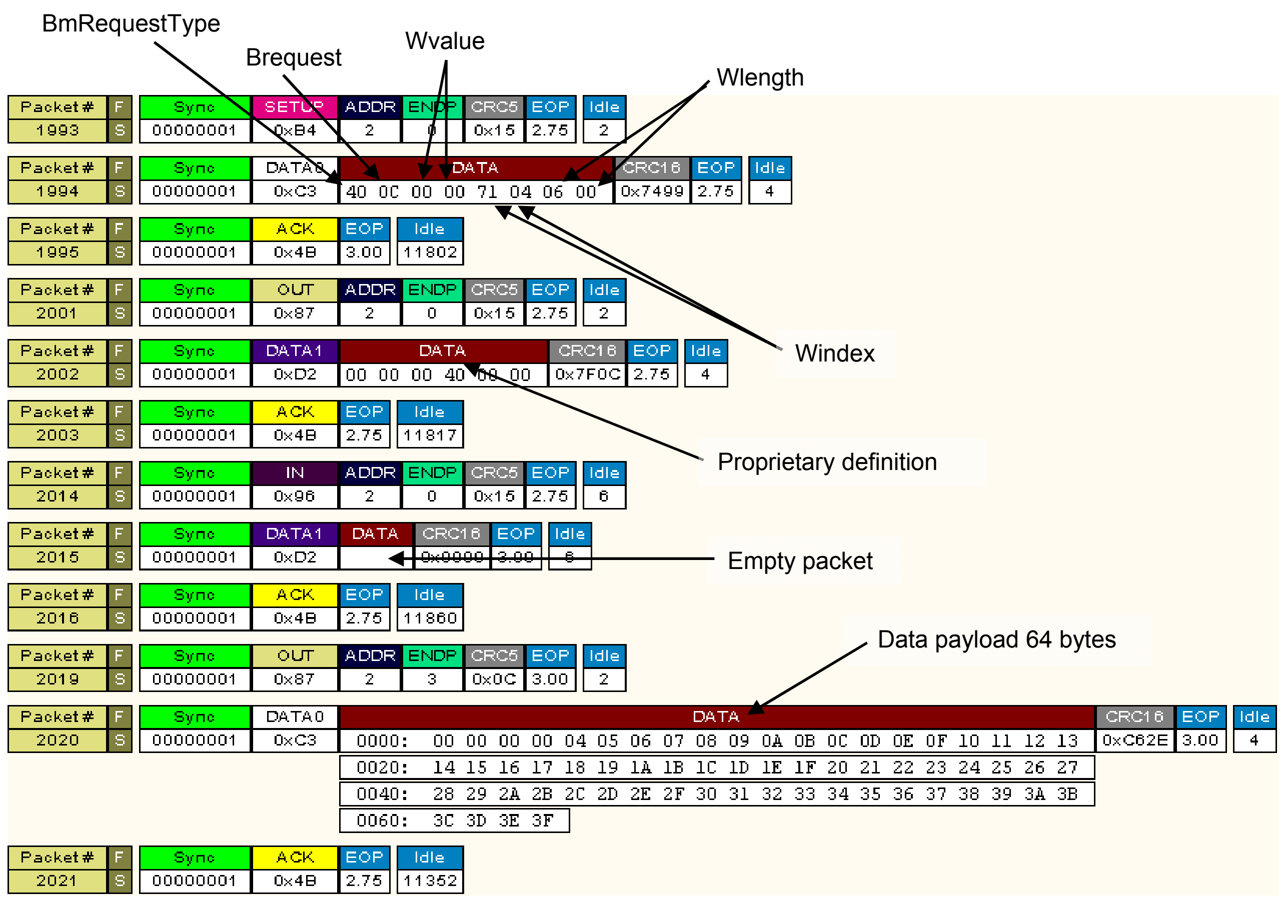
Offset	Field	Size	Value	Comments
0	BmRequestType	1	0x40	Vendor request, host to device
1	Brequest	1	0x0C	Fixed value for IOCTL_WRITE_REGISTER
2	Wvalue	2	0	Offset, set to zero
4	Windex	2	0x0471	Fixed value of setup bulk transfer
6	Wlength	2	6	Data length of setup bulk transfer

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 31.

Table 31. 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

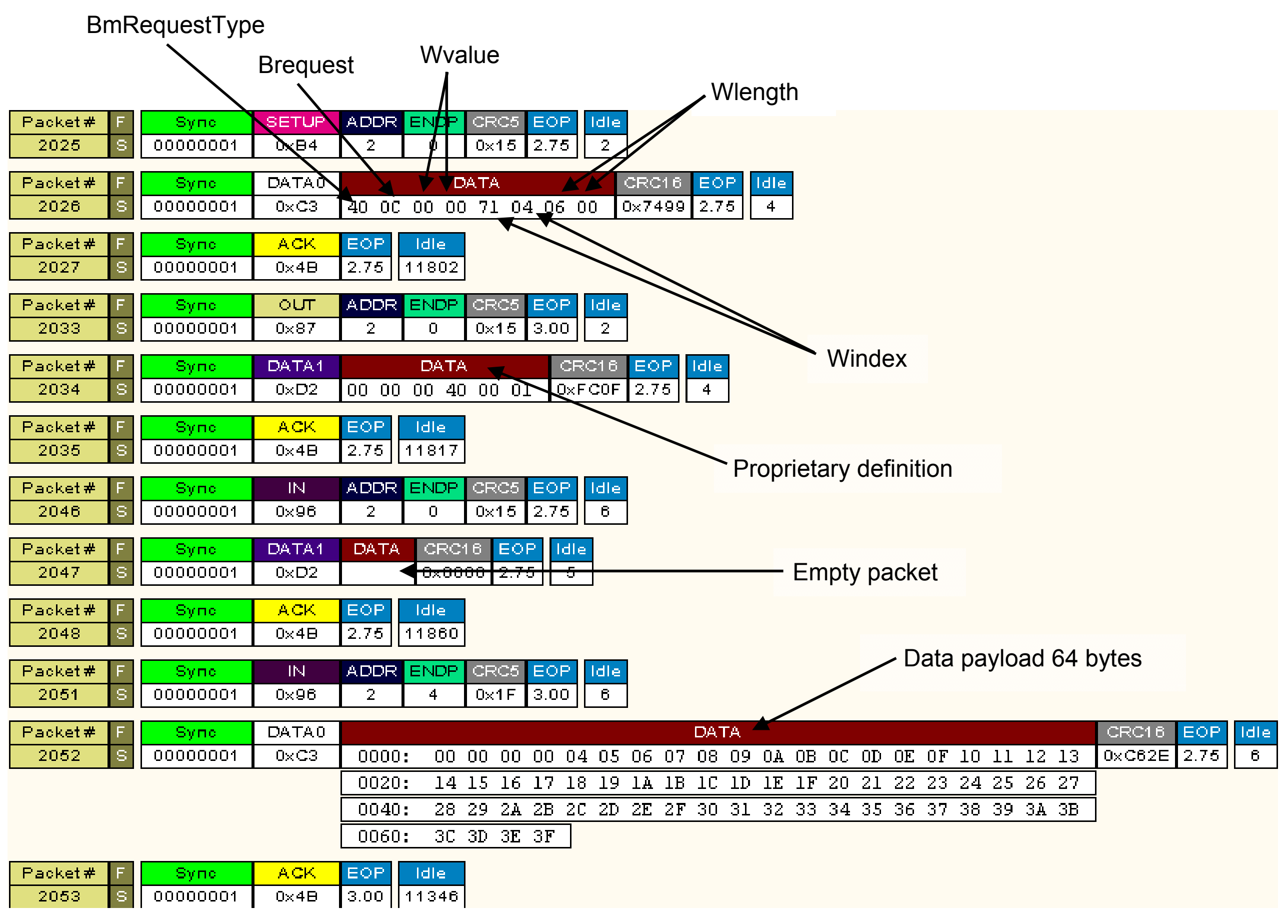
6.8.2 USB analyzer capture of a PIO OUT transfer



```
Packet# F  Sync      SETUP  ADDR  ENDP  CRC5  EOP   Idle
1993    S  00000001  0xB4   2     0     0x15  2.75  2

Packet# F  Sync      DATA0  DATA                      CRC16   EOP   Idle
1994    S  00000001  0xC3   40 0C 00 00 71 04 06 00   0x7499  2.75  4

Packet# F  Sync      ACK    EOP   Idle
1995    S  00000001  0x4B   3.00  11802

Packet# F  Sync      OUT    ADDR  ENDP  CRC5  EOP   Idle
2001    S  00000001  0x87   2     0     0x15  2.75  2

Packet# F  Sync      DATA1  DATA                CRC16   EOP   Idle
2002    S  00000001  0xD2   00 00 00 40 00 00   0x7F0C  2.75  4

Packet# F  Sync      ACK    EOP   Idle
2003    S  00000001  0x4B   2.75  11817

Packet# F  Sync      IN     ADDR  ENDP  CRC5  EOP   Idle
2014    S  00000001  0x96   2     0     0x15  2.75  6

Packet# F  Sync      DATA1  DATA  CRC16   EOP   Idle
2015    S  00000001  0xD2         0x0000  3.00  6

Packet# F  Sync      ACK    EOP   Idle
2016    S  00000001  0x4B   2.75  11860

Packet# F  Sync      OUT    ADDR  ENDP  CRC5  EOP   Idle
2019    S  00000001  0x87   2     3     0x0C  3.00  2

Packet# F  Sync      DATA0  DATA                                                                   CRC16   EOP   Idle
2020    S  00000001  0xC3   0000:  00 00 00 00 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13   0xC62E  3.00  4
                            0020:  14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
                            0040:  28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B
                            0060:  3C 3D 3E 3F

Packet# F  Sync      ACK    EOP   Idle
2021    S  00000001  0x4B   2.75  11352
```

Fig 87. USB analyzer capture of a PIO OUT transfer

6.8.3 USB analyzer capture of a PIO IN transfer



```
Packet# F  Sync      SETUP  ADDR  ENDP  CRC5  EOP   Idle
2025    S  00000001  0xB4   2     0     0x15  2.75  2

Packet# F  Sync      DATA0  DATA                              CRC16   EOP   Idle
2026    S  00000001  0xC3   40 0C 00 00 71 04 06 00           0x7499  2.75  4

Packet# F  Sync      ACK    EOP   Idle
2027    S  00000001  0x4B   2.75  11802

Packet# F  Sync      OUT    ADDR  ENDP  CRC5  EOP   Idle
2033    S  00000001  0x87   2     0     0x15  3.00  2

Packet# F  Sync      DATA1  DATA                  CRC16   EOP   Idle
2034    S  00000001  0xD2   00 00 00 40 00 01     0xFC0F  2.75  4

Packet# F  Sync      ACK    EOP   Idle
2035    S  00000001  0x4B   2.75  11817

Packet# F  Sync      IN     ADDR  ENDP  CRC5  EOP   Idle
2046    S  00000001  0x96   2     0     0x15  2.75  6

Packet# F  Sync      DATA1  DATA  CRC16   EOP   Idle
2047    S  00000001  0xD2         0x0000  2.75  5

Packet# F  Sync      ACK    EOP   Idle
2048    S  00000001  0x4B   2.75  11860

Packet# F  Sync      IN     ADDR  ENDP  CRC5  EOP   Idle
2051    S  00000001  0x96   2     4     0x1F  3.00  6

Packet# F  Sync      DATA0  DATA                                                                 CRC16   EOP   Idle
2052    S  00000001  0xC3   0000:  00 00 00 00 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13   0xC62E  2.75  6
                            0020:  14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
                            0040:  28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B
                            0060:  3C 3D 3E 3F

Packet# F  Sync      ACK    EOP   Idle
2053    S  00000001  0x4B   3.00  11346
```

Fig 88. USB analyzer capture of a PIO IN transfer

6.8.4 Vendor request for the ISO transfer

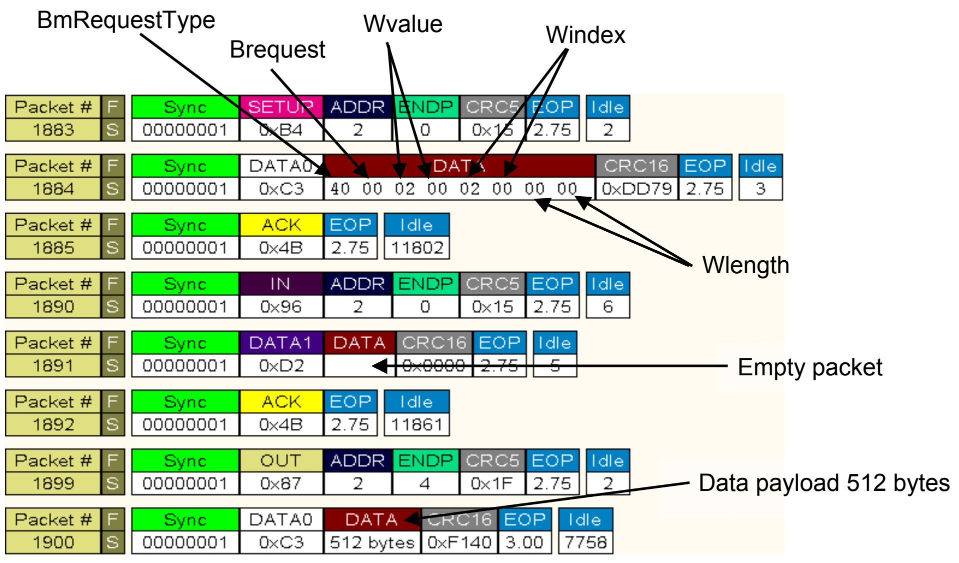
The device request is defined in Table 32.

Table 32. 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.

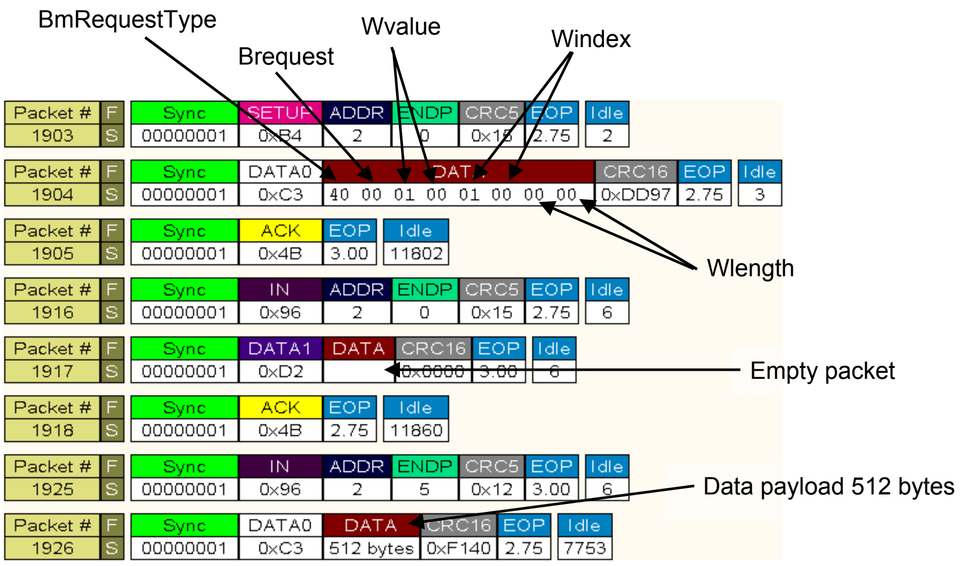
6.8.5 USB analyzer capture of an ISO OUT transfer



```
Packet # F | Sync     | SETUP | ADDR | ENDP | CRC5 | EOP  | Idle
1883     S | 00000001 | 0xB4  | 2    | 0    | 0x15 | 2.75 | 2

Packet # F | Sync     | DATA0 | DATA                    | CRC16  | EOP  | Idle
1884     S | 00000001 | 0xC3  | 40 00 02 00 02 00 00 00 | 0xDD79 | 2.75 | 3

Packet # F | Sync     | ACK   | EOP  | Idle
1885     S | 00000001 | 0x4B  | 2.75 | 11802

Packet # F | Sync     | IN    | ADDR | ENDP | CRC5 | EOP  | Idle
1890     S | 00000001 | 0x96  | 2    | 0    | 0x15 | 2.75 | 6

Packet # F | Sync     | DATA1 | DATA | CRC16  | EOP  | Idle
1891     S | 00000001 | 0xD2  |      | 0x0000 | 2.75 | 5

Packet # F | Sync     | ACK   | EOP  | Idle
1892     S | 00000001 | 0x4B  | 2.75 | 11861

Packet # F | Sync     | OUT   | ADDR | ENDP | CRC5 | EOP  | Idle
1899     S | 00000001 | 0x87  | 2    | 4    | 0x1F | 2.75 | 2

Packet # F | Sync     | DATA0 | DATA      | CRC16  | EOP  | Idle
1900     S | 00000001 | 0xC3  | 512 bytes | 0xF140 | 3.00 | 7758
```

Fig 89. USB analyzer capture of an ISO OUT transfer

6.8.6 USB analyzer capture of an ISO IN transfer



Packet #	F S	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
1903	S	00000001	0xB4	2	0	0x15	2.75	2

Packet #	F S	Sync	DATA0	DATA	CRC16	EOP	Idle
1904	S	00000001	0xC3	40 00 01 00 01 00 00 00	0xDD97	2.75	3

Packet #	F S	Sync	ACK	EOP	Idle
1905	S	00000001	0x4B	3.00	11802

Packet #	F S	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
1916	S	00000001	0x96	2	0	0x15	2.75	6

Packet #	F S	Sync	DATA1	DATA	CRC16	EOP	Idle
1917	S	00000001	0xD2		0x0000	3.00	6

Packet #	F S	Sync	ACK	EOP	Idle
1918	S	00000001	0x4B	2.75	11860

Packet #	F S	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
1925	S	00000001	0x96	2	5	0x12	3.00	6

Packet #	F S	Sync	DATA0	DATA	CRC16	EOP	Idle
1926	S	00000001	0xC3	512 bytes	0xF140	2.75	7753

Fig 90. USB analyzer capture of an ISO IN transfer

7. References

[1] ISP1161A1 Full-speed USB single-chip host and device controller data sheet

[2] Universal Serial Bus Specification Rev. 2.0 (full-speed section)

[3] Open Host Controller Interface Specification for USB, Release: 1.0a

8. Legal information

9. Contents

Date of release: 12 October 2009
Document identifier: AN10005_3