PHILIPS

Philips Semiconductors

Connectivity

June 2002

AN10008-01

ISP1362 Embedded Programming Guide

Rev: 0.9

Revision History:

Rev.	Date	Descriptions	Author
0.9	4/15/2002	Added content on OTG	Wang Zhong Wei
0.81	1/3/2002	Update on the Host Controller information	Ng Chee Yu
0.80	28/02/2002	Content on the Device Controller added	Alvin Lim
0.70	19/02/2002	Modification based on 9th Jan review	Ng Chee Yu
0.65	07/02/2002	OTG chapter added	Wang Zhong Wei
0.61	05/02/2002	Complete the Host Controller information added	Ng Chee Yu
0.60	29/01/2002	Three advanced features	Ng Chee Yu

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

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

DISCLAIMER

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

CONTENTS

ISP1362 Embedded Programming Guide Rev. 0.9

TABLES

FIGURES

1. Introduction

The ISP1362 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. They have the same data bus, but different I/O locations. The ISP1362 uses a flexible interrupt and DMA scheme, which allows the Host Controller and the Device Controller to use separate interrupt and DMA lines, or share a single interrupt and DMA line, if desired. Besides the Host Controller and the Device Controller, the ISP1362 also contains the On-The-Go (OTG) Controller. Devices with OTG Controller built-in can be a USB host or device. Therefore, these devices can communicate with each other without the need of a personal computer (PC).

There are two USB ports on the ISP1362: port 1 and port 2. Port 1 can be configured as a downstream port, an upstream port or an OTG port. Port 2 is a fixed downstream port.

The Host Controller is an advanced transfer-based USB Host Controller. It offers a very high efficiency in using the USB bandwidth and yet requires little CPU intervention. This is because its USB engine is mostly hardware-based.

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. The ISP1362 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. The ISP1362 brings high flexibility to the systems that have it built-in. For example, a system that has the ISP1362 built-in allows it not only to be connected to a PC or a USB hub that has a USB downstream port. But it can also be connected to a device that has a USB upstream port, such as USB printer, USB camera, USB keyboard, USB mouse, among others.

2. ISP1362 Programmer's Model

The ISP1362 can be viewed as a complete set of USB functionality—host, device and OTG—that is accessible to the programmer through four I/O ports. Writing and reading register sets control the functionality, and writing and reading the respective buffers access the USB traffic.

At the lowest level, the ISP1362 is accessed as four I/O ports. These are:

- Device Controller command port
- Device Controller data port
- Host Controller command port
- Host Controller data port.

The OTG Controller shares the same I/O ports as the Host Controller.

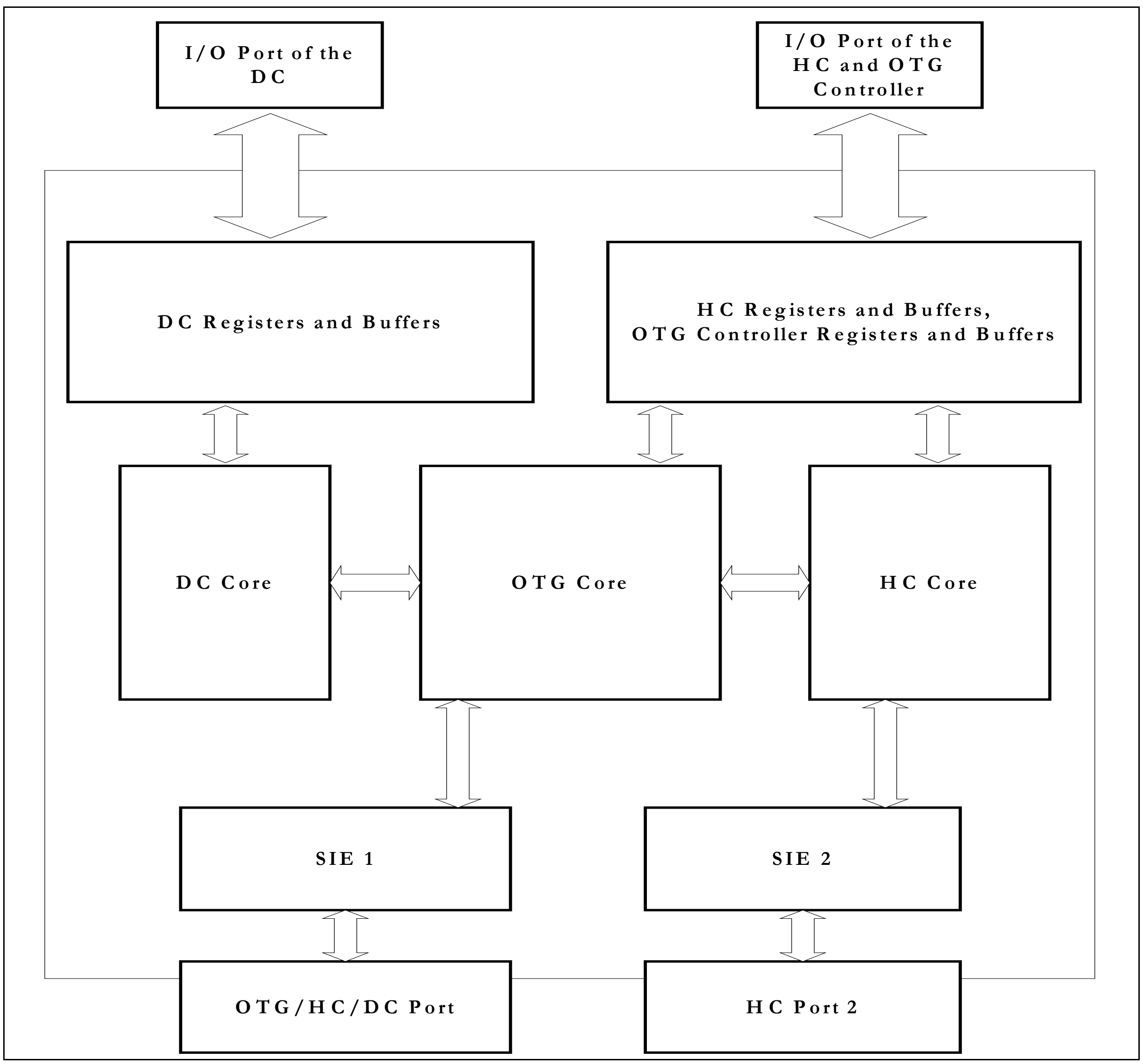

Figure 2-1: Programmer's Model of the ISP1362

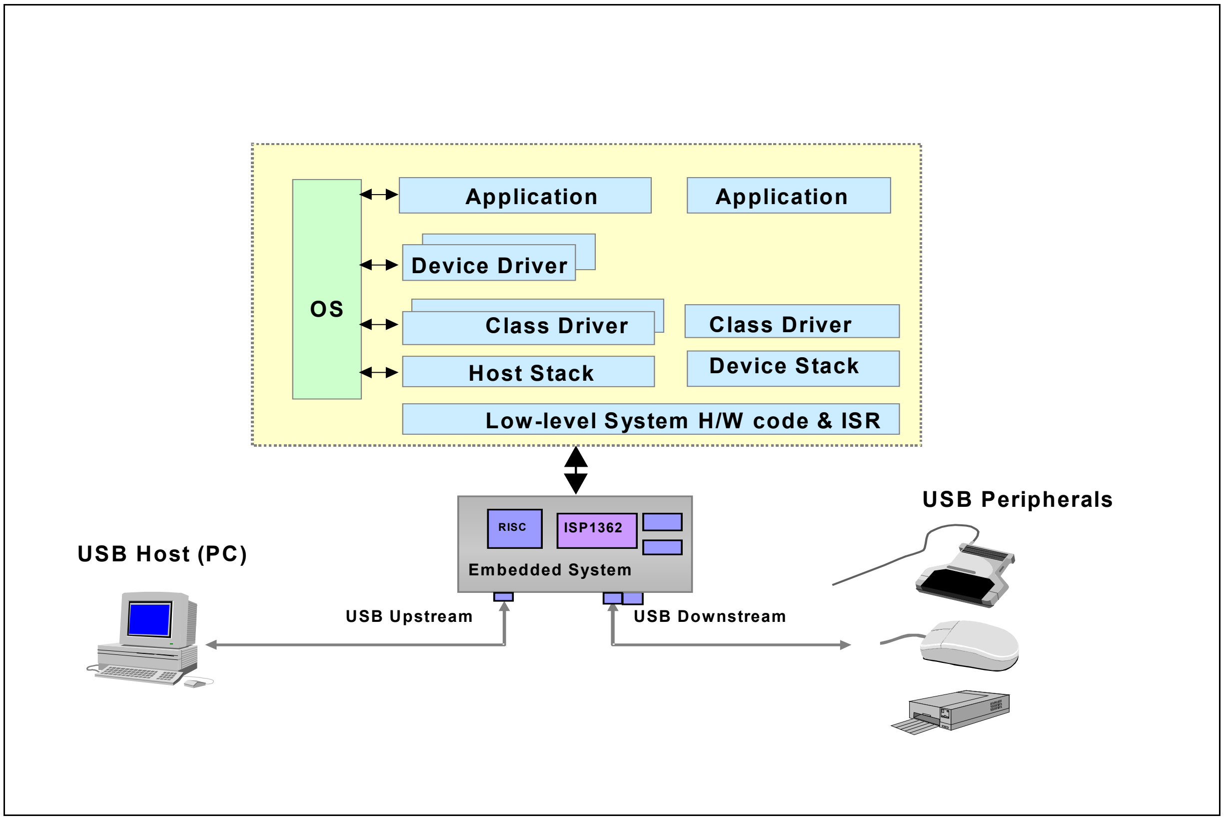

Figure 2-2: Software Model of the ISP1362

This programming guide contains the information and techniques required for writing the "low-level system hardware code and interrupt service routine (ISR)" as illustrated in Figure 2-2.

3. Accessing Registers

3.1. Software Accessible Hardware Components

The major hardware components of the Host Controller in the ISP1362 accessible by software are:

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

Details of these registers can be found in the ISP1362 datasheet. Three groups of registers in the ISP1362 control the functions of the chip. These registers are named according to their functional groups.

- **OTG Controller Registers** Start with Otg. For example, OtgControlStatus.
- **Host Controller Registers** Start with Hc. For example, HcBufferStatus.
- **Device Controller Registers** Start with Dc. For example, DcHardwareConfiguration.

This chapter will explain the basic routines that access registers and buffers.

3.2. I/O Ports of the ISP1362

The registers in the ISP1362 are accessible via four ports, specified by the combinations of A0 and A1. Depending on the type of platform that the ISP1362 is installed on, A0 and A1 can be mapped to different addresses. For example, the Philips ISP1362 evaluation board allows the ISP1362 to be mapped to 290/292/294/296 or 300/302/304/306 on the I/O port of the X86-PC. 0x290 and 0x300 are the base addresses of the ISP1362.

The ISP1362 has the I/O port architecture. However, the connected port for the CPU depends on platforms. For instance, some RISC CPUs (such as, MIPS, SH and PPC) do not has I/O address space. In such cases, the ISP1362 I/O port will be mapped on the memory space.

Table 3-1: Four Ports of the ISP1362

I/O Address	A0, A1	Description
Base address	A0 = 0, A1 = 0	Host Controller Data Port
Base address + 2	A0 = 1, A1 = 0	Host Controller Command Port
Base address + 4	A0 = 0, A1 = 1	Device Controller Data Port
Base address + 6	A0 = 1, A1 = 1	Device Controller Command Port

In this document, the ports will be referred to as follows:

- **Host Controller Data Port** hc_data
- **Host Controller Command Port** hc_com
- **Device Controller Data Port** dc_data
- **Device Controller Command Port** dc_com

In a PC-ISA system in which the ISP1362 is mapped to the base address of 0x290, the ports must be defined as:

```
#define        hc_data 0x290
#define        hc_com 0x292
#define        dc_data 0x294
#define        dc_com 0x296
```

All accesses through these ports for 8-bit registers are in the 16-bit mode. The valid data resides at the lower byte. For 32-bit registers, the first word to be written or read is the lower word.

3.3. Basic Register Accesses

The two basic types of access a programmer would require while programming with the ISP1362 Host Controller and OTG are:

- Read/Write, 8-bit or 16-bit
- Read/Write, 32-bit.

3.3.1. Reading and Writing of 8-Bit and 16-Bit Registers

This type of access has two phases: command and data. During the command phase, the index of the target register is written to the command port. During the data phase, the desired data is written to or read from the data port. The index of various registers can be found in the ISP1362 datasheet.

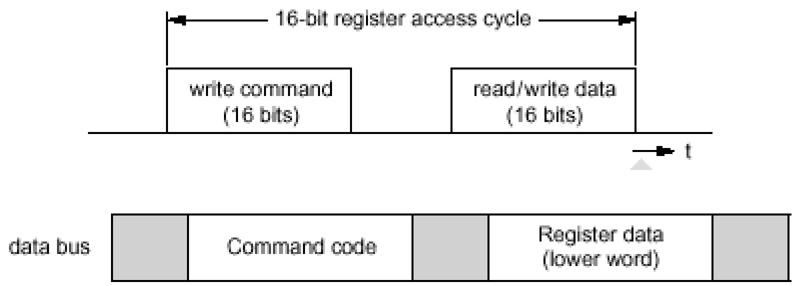

Figure 3-1: 16-Bit Register Access Cycle

Note that the indices of all the Host Controller and OTG registers follow the convention in which the MSB signifies whether it is a write or read operation. An index for writing to a register always has the seventh bit set to 1.

For example:

```
HcBufferStatus (Read)  => 0x2C
HcBufferStatus (Write) => 0x2C | 0x80
                       => 0xAC
```

For convenience, in any command-write operation, the index is ORed with 0x80, so that only one value for each register needs to be defined.

Note: The registers in the ISP1362 Device Controller do not follow this convention.

Figure 3-2 shows a sample code for a 16-bit register write.

```
void hc_write(unsigned int reg_index, unsigned int data_to_write
{
        outport(hc_com, reg_index|0x80);      // Writes the register index to the command port
        outport(hc_data, data_to_write);      // Writes data to the data port
}
```

Figure 3-2: Code Example for 16-Bit Register Write

The code example in Figure 3-3 reads data from a 16-bit register.

```
Unsigned int hc_read(unsigned int reg_index)
{
        unsigned int data_to_return;

        outport(hc_com, reg_index);           // Writes the register index to the command port
        data_to_return = inport(hc_data);     // Reads data from the data port

        return(data_to_return);
}
```

Figure 3-3: Code Example for 16-Bit Register Read

3.3.2. Reading and Writing of 32-Bit Registers

This type of access has three phases: command, first data and second data. During the command phase, the index of the target register is written to the command port. During data phases, two words (16-bit) are written to or read from the data port. The lower word must be accessed first, followed by the higher word.

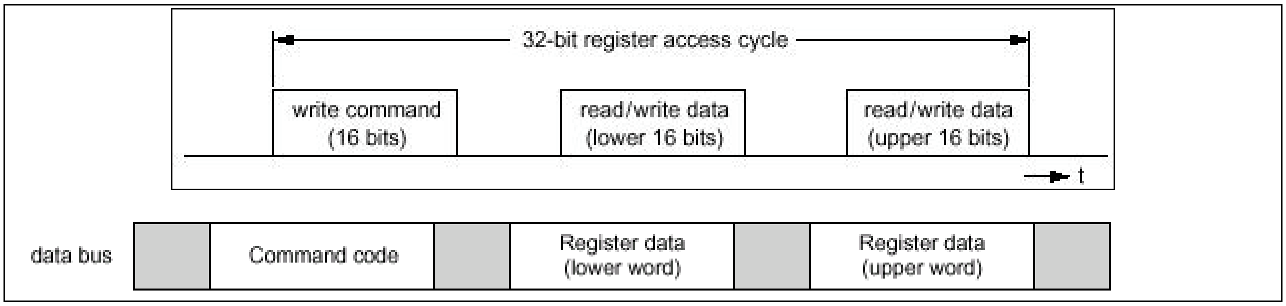

Figure 3-4: 32-Bit Register Access Cycle

Executing a 32-bit access without completing the second data phase will send the ISP1362 into an indeterminate state. This is because the ISP1362 expects two data phases following the command phase. A pseudo code for a 32-bit register read is given in Figure 3-5.

```
Unsigned int hc_read32(unsigned int reg_index)
{
        unsigned int data_low;
        unsigned int data_high;
        unsigned long data32;

        outport(hc_com, reg_index);           // Writes the register index to the command port
        data_low = inport(hc_data);           // Reads the LOW word data from the data port
        data_high= inport(hc_data);           // Reads the HIGH word data from the data port

        data32=data_high;
        data32=data32<<16;
        data32+=data_low;

        return(data32);
}
```

Figure 3-5: 32-Bit Register Read

3.3.3. I/O Functions

In the previous sections, a number of routines have been provided for basic register accesses. These routines are written in Turbo C and use a number of functions provided by Turbo C. For example, inport() and outport(). If you are using any other compiler, you may need to use a different set of I/O functions. In some embedded cases, you may even need to implement your own inport and outport functions. This section will provide a pseudo code for a general CPU or MCU by using the general-purpose I/O (GPIO) pins.

Assuming that the CPU or the MCU:

- Has at least three I/O ports of 8-bit width each (P1, P2, P3), and
- All ports are bi-directional.

The following declarations defines the I/O ports of the CPU or MCU:

```
#define lo_byte       P1                      // Port 1 is lower byte of the 16-bit data bus
#define hi_byte       P2                      // Port 2 is higher byte of the 16-bit data bus
#define CS            P3~0                    // Bit 0 of port 3 is CS
#define WR            P3~1                    // Bit 1 of port 3 is WR
#define RD            P3~2                    // Bit 2 of port 3 is RD
#define A0            P3~3                    // Bit 3 of port 3 is A0
#define A1            P3~4                    // Bit 4 of port 3 is A1
```

The pseudo code to access the ISP1362 by using GPIO pins is given in Figure 3-6.

```
void hc_write(unsigned int reg_index, unsigned int data_to_write)
{
 // Initialize the control signals
 CS=1;
 RD=1;
```

```
 WR=1;

 A1=0;  // Access the HC

 // Command phase of the access
 A0=1;  // Command phase

 // Output the data on the ports
 lo_byte=(reg_index&0x00FF);
 hi_byte=(reg_index&0xFF00)>>8;

 CS=0;  // Assert the CS
 WR=0;  // Assert the WR

 Wait_uS(50);  // Wait for 50 •s

 WR=1;  // De-assert WR
 CS=1;  // De-assert CS

 // Data phase of the access
 A0=0;  // Data phase

 // Output the data to ports
 lo_byte=(data_to_write&0x00FF);
 hi_byte=(data_to_write&0xFF00)>>8;

 CS=0;  // Assert the CS
 RD=0;  // Assert the RD

 Wait_uS(50);  // Wait for 50 •s

 RD=1;  // De-assert RD
 CS=1;  // De-assert CS
}
```

Figure 3-6: Code Example for Accessing the ISP1362 by Using GPIOs

3.3.4. Example: Reading the Chip ID

After installing the ISP1362 on your hardware platform, the simple program given in Figure 3-7 can be used to make sure that the physical connection between the microprocessor system and the ISP1362 is correct.

```
#include <stdio.h>
#include <dos.h>

#define hc_data 0x290
#define hc_com  0x292

#define HcChipID 0x27

unsigned int hc_read(unsigned int reg_index);

void main(void)
{
 unsigned int ChipID;

 ChipID = hc_read(HcChipID);

 printf("\nChipID is : %4X",ChipID);
}

unsigned int hc_read(unsigned int reg_index)
{
        unsigned int data_to_return;
        outport(hc_com, reg_index);           // Writes the register index to the command port
        data_to_return = inport(hc_data);     // Reads the data from the data port

        return(data_to_return);
}
```

Figure 3-7: Reading the Chip ID

3.3.5. Example: Testing the HcScratch Register

The program in Figure 3-8 is written to test the ISP1362 Write/Read cycle. The target register (HcScratch) does not have any specific use. The programmer may use this register for any purposes.

```
#include <stdio.h>

#define hc_data 0x290
#define hc_com  0x292
#define HcScratch 0x28

unsigned int hc_read(unsigned int reg_index);
void hc_write(unsigned int reg_index, unsigned int data_to_write);

void main(void)
{
 unsigned int cnt=0,error=0;
 unsigned int test_data;

 do
  {
   hc_write(HcScratch,cnt);
   test_data=hc_read(HcScratch);
   if(test_data!=cnt)
    {
     printf("\nError Encountered!!");
     printf("\nWrite:%4X, Read:%4X",cnt,test_data);
     error++;
    }
   cnt++;
  }
 while(cnt<0xFFFF);

 if(error==0)
  {
    printf("\nNo error!!");
  }
 else
  {
    printf("\nTotal error : %d",error);
  }
}

void hc_write(unsigned int reg_index, unsigned int data_to_write)
{
       outport(hc_com, reg_index|0x80);      // Writes the register index to the command port
       outport(hc_data, data_to_write);      // Writes the data to the data port
}

unsigned int hc_read(unsigned int reg_index)
{
       unsigned int data_to_return;

       outport(hc_com, reg_index);           // Writes the register index to the command port
       data_to_return = inport(hc_data);     // Reads the data from the data port

       return(data_to_return);
}
```

Figure 3-8: Testing the HcScratch Register

4. Accessing Host Controller Buffers

The programmed I/O (PIO) memory access in the ISP1362 is similar to the register access, except that the data phase is variable in length. The length of the data phase depends on the amount of data you wish to access. The PIO memory access can be done using two addressing mode: Indirect Addressing and Direct Addressing.

4.1. Indirect Addressing

This addressing method uses a dedicated register to access each of the four buffer areas. The access always starts from the location zero of the respective buffer area. The amount of data to be accessed must be specified in HcTransferCounter.

An example of a C code for reading from the ATL buffer is given in Figure 4-1. In this code example:

a_ptr is a pointer that points to the memory array to hold data from the ATL buffer.

data_size is the number of words to be read.

```
void read_atl(unsigned int *a_ptr, unsigned int data_size)
{
 int cnt;

 hc_write(HcTransferCounter,data_size*2);
 outport(hc_com,HcATLBufferPort);

 cnt=0;
 do
  {
   *(a_ptr+cnt)=inport(hc_data);
   cnt++;
  }
 while(cnt<(data_size));
}
```

*data_size is multiplied by two because HcTransferCounter is a **Byte** counter.*

Figure 4-1: Code Example for Reading from the ATL Buffer

Figure 4-2 contains a C code example for writing to the ATL buffer. In this C code example:

a_ptr is a pointer that points to the memory array that holds data to be written to the ATL buffer.

data_size is the number of words to be read.

```
void write_atl(unsigned int *a_ptr, unsigned int data_size)
{
 int cnt;

 hc_write(HcTransferCounter,data_size*2);
 outport(hc_com,HcATLBufferPort|0x80);

 cnt=0;
 do
  {
   outport(hc_data,*(a_ptr+cnt));
   cnt++;
  }
 while(cnt<(data_size));
}
```

*data_size is multiplied by two because HcTransferCounter is a **Byte** counter.*

Figure 4-2: Code Example for Writing to the ATL Buffer

4.2. *Direct Addressing*

This addressing method views the entire Host Controller buffer memory as a single linear array of 4096 bytes. Since the ISP1362 does not have dedicated memory address lines, the direct addressing memory access is performed using several special registers. These registers are:

- HcDirectAddressLength (32-bit)
- HcDirectAddressData (16-bit)

HcDirectAddressLength provides three sets of information necessary to perform a directly addressed access. These are:

- BufferStartAddress[14:0]
- Inc/DecBufferAddress

- DataByteCount[15:0].

Table 4-1: HcDirectAddressLength Register: Bit Allocation

Bit	31	30	29	28	27	26	25	24
Symbol	DataByteCount[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	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	DataByteCount[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	Inc/DecBufferAddress	BufferStartAddress[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	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	BufferStartAddress[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

Table 4-2: HcDirectAddressLength Register: Bit Description

Bit	Description
BufferStartAddress[14:0]	This field specifies the exact location where data would be written to or read from.
Inc/DecBufferAddress	This bit determines whether the access would be done in auto increasing or auto decreasing manner.
DataByteCount[15:0]	This 16-bit value takes the higher word of the HcDirectAddressLength register i.e. [31:16]. It specifies the exact number of bytes to be accessed.

HcDirectAddressData is a data access register.

4.2.1. Setting Up the HcDirectAddressLength Register

The sample code in Figure 4-3 sets up the HcDirectAddressLength register according to the three parameters passed from the calling function:

count number of bytes of data to access

flow 0 for incremental address and 1 for decremental address

addr starting address for data access.

```
void Set_DirAddrLen(unsigned int count, unsigned char flow, unsigned int addr)
{
 unsigned long addr2return;

 addr2return =(long)(addr&0x7FFF);
 addr2return|=((long)flow)<<15;
 addr2return|=(((long)count)<<16);

 hc_write32(HcDirAddrLen,addr2return);
}
```

Figure 4-3: Code Example for Setting Up the HcDirectAddressLength Register

4.2.2. Accessing Memory Using the Direct Addressing Mode

Figure 4-4 shows a sample code to read the ISP1362 Host Controller memory and store data in an array pointed to by the "*a_ptr" pointer. It reads a total of "data_size" words from the byte address "start_addr".

```
void random_read(unsigned int *a_ptr,unsigned int start_addr,unsigned int data_size)
{
 unsigned int test_size=data_size*2;
 unsigned long cnt=0;

 Set_DirAddrLen(test_size,0,start_addr);

 outport(hc_com,HcDirAddr_Port);

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

Figure 4-4: Code Example for a Direct Address Read

Note that HcTransferCounter is not used in the direct addressing mode.

5. Setting Up the ISP1362 Host Controller for USB Operations

5.1. Setting Up the Built-In Host Controller Buffer

The ISP1362 has a built-in buffer of 4096 bytes. By writing the desired buffer sizes into the HcATLBufferSize, HcISTLBufferSize and HcINTLBufferSize registers, this 4096 bytes will be divided into four areas of sizes specified by the corresponding registers.

For typical applications, the recommended buffer sizes are:

- **ATL** 1536 bytes
- **INTL** 512 bytes
- **ISTL** 1024 bytes.

ISTL is made up of ISTL0 and ISTL1. The value written into the HcISTLBufferSize register specifies the individual buffer size of both ISTL0 and ISTL1. Therefore, in this case, the total amount of buffer used is 4096 bytes (1536 + 512 + 1024 + 1024 = 4096).

The buffer size can be configured in any sequence. It is re-allocated whenever any of the Host Controller buffer size register is updated. Allocation of buffer memory follows the fixed sequence of ISTL0, ISTL1, INTL and ATL, irrespective of the sequence of in which values are written to Host Controller buffer size registers. For details on how the ISP1362 Host Controller allocates the buffer memory, refer to the ISP1362 datasheet.

5.2. Setting Up Registers

This section describes a number of registers, which must be initialized after powering on and before any USB transfer. At the end of the initialization process, the Host Controller will be ready to process PTDs in buffers. The initialization process can be sub-divided into a number of steps:

- Control and Status Setup
- Frame Counter Setup
- Root Hub Setup

- Interrupt Setup
- Hardware Configuration Setup.

Each section corresponds to a group of registers in the ISP1362 datasheet. This section does not provide complete information on all these registers. It covers the minimal information that is necessary to prepare the ISP1362 for USB operations.

5.2.1. Control and Status Setup

The register involved in this step of the set up is:

- HcControl register (see Table 5-1).

Table 5-1: HcControl Register: Bit Allocation

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

The RWE and RWC bits must be set to logic 1. The HCFS field controls the state of the USB Host Controller. In this programming guide, only two states are used: reset and operational. To enter the operational state, HCFS must be set to "10B". To enter the reset state, HCFS must be set to "00B".

Summary:

To start the USB operation, write 0x0680 to the HcControl register.

To reset the USB operation, write 0x0600 to the HcControl register.

5.2.2. Frame Counter Setup

The register involved in this step of the set up is:

- HcFmInterval register (see Table 5-2).

Table 5-2: HcFmInterval Register: Bit Allocation

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	-	-	1	0	1	1	1	0
Access	-	-	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

Write the value 0x27782EDF to this register. The FrameInterval field (FI[13:0]) specifies the number of bit time between two SOFs. The nominal value of FrameInterval is 11999. The FSLargestDataPacket field (FSMPS[31:16] 0x2778) specifies the largest amount of data in bits that can be sent or received by the Host Controller in a single frame. The value of 0x2778 can be derived from the following calculations:

Maximum number of bits in one frame	11999 bits
Bits used in overhead (SOF, EOP, etc.)	210 bits
Remaining available bits	11999 - 210 = 11789
Maximum bit stuffing	11790 x 6/7 = 10104 (0x2778).

5.2.3. Root Hub Setup

The registers involved in this step of the set up are:

- HcRhDescriptorA
- HcRhDescriptorB
- HcRhStatus
- HcRhPortStatus[1]
- HcRhPortStatus[2].

HcRhDescriptorA and HcRhDescriptorB select a number of features of the Host Controller. The details of these features can be found in the ISP1362 datasheet. The values used in this programming guide are:

HcRhDescriptorA	0x05000B01
HcRhDescriptorB	0x00000000.

HcRhPortStatus[1] and HcRhPortStatus[2] provide the control and status of the two downstream ports of the ISP1362.

The following section explains details how to experiment these two registers.

1. The Host Controller must be set to the operational mode. This step is explained in Section 5.2.1 Control and Status Setup.
2. Write 0x0000 0100 to both the HcRhPortStatus registers (see Table 5-3 for the bit allocation). This resets both the ports and sets them ready for operation. At this stage the value in the HcRhPortStatus register will be 0x0001 0100.
3. Connect a USB device to one of the ports. You will see that the bit 0 of the HcRhPortStatus register changes to logic 1 for the respective port. The LS bit will be set to logic 1, if the connected device is of the low-speed type.
4. Write 0x0000 0002 to HcRhPortStatus of the port that has a device connected to it. This enables the port.

Table 5-3: HcRhPortStatus[1:2] Register: Bit Allocation

Bit	31	30	29	28	27	26	25	24
Symbol	reserved							
Reset	-	-	-	-	-	-	-	-
Access	-	-	-	-	-	-	-	-
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	reserved			PRSC	OCIC	PSSC	PESC	CSC
Reset	-	-	-	0	0	0	0	0
Access	-	-	-	R/W	R/W	R/W	R/W	R/W
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved						LSDA	PPS
Reset	-	-	-	-	-	-	0	0
Access	-	-	-	-	-	-	R/W	R/W

5.2.4. Interrupt Setup

The registers involved in this step of the set up are:

- HcInterruptStatus
- HcInterruptEnable
- HcInterruptDisable
- HcµPInterrupt
- HcµPInterruptEnable.

The HcInterruptStatus register reflects a number of events that are closely related to the root hub operation. Each of these events may generate a hardware interrupt, if not masked off by using the HcInterruptDisable register. The OPR_Reg bit (bit 4) in the HcµPInterrupt register (see Table 5-4) reflects an interrupt generated by any of these events.

HcµPInterrupt is one of the most important registers in the ISP1362. It reflects the completion of data transfer, PTD processing in each of the buffer and the start-of-frame (SOF). These events generate a hardware interrupt unless they are masked-off by the HcµPInterruptEnable register.

Table 5-4: HcμPInterrupt Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved						OTG_IRQ	ATL_IRQ
Reset	-	-	-	-	-	-	0	0
Access	-	-	-	-	-	-	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	INT_IRQ	ClkReady	HC Suspended	OPR_Reg	AllEOT Interrupt	ISTL_1_ INT	ISTL_0_ INT	SOFISTL Int
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

5.2.5. Hardware Configuration Setup

The HcHardwareConfiguration register controls many aspect of the ISP1362 interface to the external circuitry. The bit allocation and bit description of the HcHardwareConfiguration register are given in Table 5-5 and Table 5-6.

Table 5-5: HcHardwareConfiguration Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved		Connect PullDown 15K_DS2	Connect PullDown 15K_DS1	Suspend ClkNotStop	AnalogOC Enable	OneINT	DACKMode
Reset	-	-	0	0	0	0	0	0
Access	-	-	R/W	R/W	R/W	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	OneDMA	DACKInput Polarity	DREQOut putPolarity	DataBusWidth[1:0]		InterruptOut putPolarity	Interrupt PinTrigger	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

Table 5-6: HcHardwareConfiguration Register: Bit Description

Bit	Description
15 to 14	reserved
13 to 12	Set to logic 1 if external 15 kΩ pull-down resistor is not available
11	Set to logic 1 if you wish to stop system clock in the suspend mode
10	Set to logic 1 for internal overcurrent detection
9	Set to logic 1 if you wish to use only one INT line for both the Host Controller and the Device Controller
8	Must be set to logic 0
7	Set to logic 1 if you wish to use only one DMA channel for both the Host Controller and the Device Controller
6 to 5	DMA setting, depends on the DMA controller requirements
4 to 3	Data bus width, must use 16-bit, i.e. “01B”
2 to 0	Controls the interrupt polarity, trigger type and enables or disables the interrupt pin

5.3. Host Controller in the Operational Mode

Once set in the operational mode, the Host Controller goes through a series of steps as shown in the flowchart in Figure 5-1. As can be seen in the figure, the ISTL buffer is the first to be processed, followed by the INTL buffer, and finally, the ATL buffer. Note that if the EOF timing is not reached by the end of the ATL processing, the Host Controller loops back to check ATL_Active. This feature allows a single-frame enumeration of USB devices.

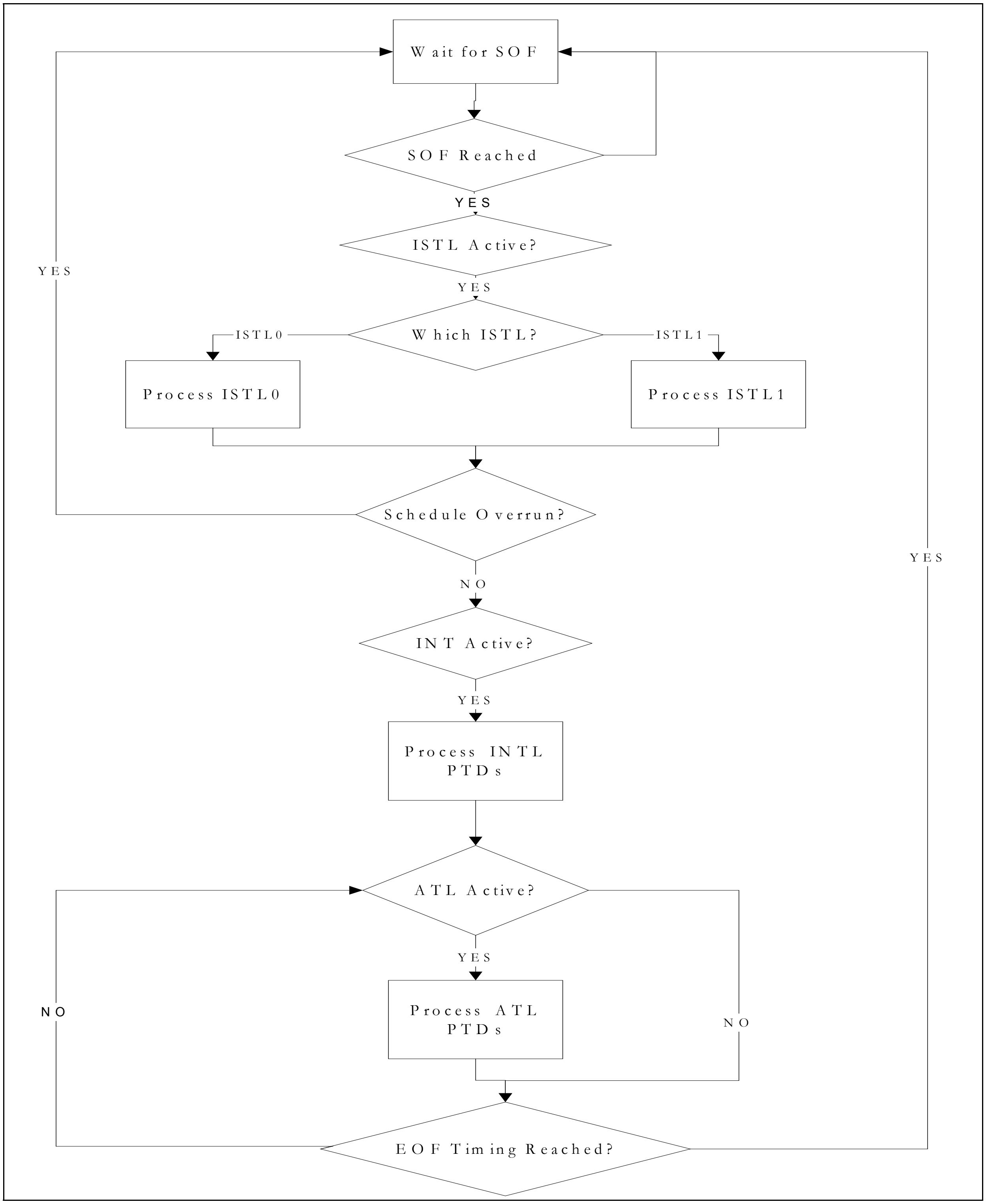

Figure 5-1: Flowchart of the Host Controller in the Operational State

6. Basic USB Transfer

When a USB device is connected to a USB host, "USB transfer" moves information between them. They are four types of USB transfers: control, bulk, isochronous and interrupt. Each transfer is made up of one or more "USB transactions". They are three types of transaction: setup, IN and OUT. Each transaction is in turn made up of several "USB packets". The ISP1362 is a transfer/transaction based Host Controller. Depending on the type of transfer involved, the number of PTDs required ranges from one to three as can be seen in Table 6-1.

Table 6-1: Number of PTDs required for a Transfer

Transfer Type	Number of PTD Per Transfer
Control	2-3
Bulk	1
Isochronous	1
Interrupt	1

A PTD is capable of generating more than one transaction. For example, a bulk PTD with 640 bytes of payload and maximum packet size of 64 bytes will generate 10 transactions of 64 bytes each. This is done without intervention of the CPU.

A control transfer requires 2-3 PTDs due to its complexity. A control transfer starts with a Setup phase, followed by an optional Data phase and a Status phase, in which a PTD is required for every phase.

A transaction is made up of two or three USB packets:

- ISO traffic (Direction Token, Data)
- Non-ISO traffic (Direction Token, Data, ACK)

The first packet is the Direction Token, which can be a setup, IN or OUT. The Host Controller sends this packet. The second packet is the data packet, which can be sent by either the Host Controller or the Device Controller, depending on the first packet. The last packet is the ACK, which can be sent by either the Host Controller or the Device Controller, depending on the second packet. Isochronous traffic does not require an ACK in the transaction.

This chapter provides a step-by-step guide to the process of executing a USB transaction using the ISP1362. The five basic steps are:

1. Preparing or formatting data into PTD
2. Copying data to the ISP1362
3. Activating the ISP1362
4. Checking transfer status
5. Post-transfer processing.

6.1. Preparing or Formatting Data in the PTD

Philips Transfer Descriptor (PTD) is an 8-byte data structure used to provide communication between the USB Host Controller and the microprocessor. PTD dictates how the Host Controller must handle or process the data in the buffer memory and reflects the status of the corresponding USB transaction.

A summary of the fields in PTD is given in Section 6.1.1. For a detailed description of these fields, refer to the ISP1362 datasheet.

6.1.1. Generic PTD fields

Table 6-2: Generic PTD: Bit Allocation

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	MaxPktSize[7:0]							
Byte 3	EndpointNumber[3:0]				B3_3	Speed	MaxPktSize[9:8]	
Byte 4	TotalBytes[7:0]							
Byte 5	B5_7	B5_6	B5_5	B5_4	DirToken[1:0]		TotalBytes[9:8]	
Byte 6	reserved	FunctionAddress[6:0]						
Byte 7	B7[7:0]							

[1] All reserved bits should be set to logic 0.

Table 6-3: Generic PTD: Bit Description

Name	Description
ActualBytes[9:0]	Actual amount of data transferred at the moment
MaxPktSize[9:0]	Maximum amount of data per packet
TotalBytes[9:0]	Total amount of data to be transferred
CompletionCode[3:0]	Reports success or error in a transaction
EndpointNumber[3:0]	Target endpoint number
DirToken[1:0]	Specifies setup, IN or OUT token
FunctionAddress[6:0]	Address of the target device
Active	Set to logic 1 by the firmware to enable execution of transactions by the Host Controller. When the transaction associated with this descriptor is completed, the Host Controller sets this bit to logic 0.
Toggle	This bit is used to generate or compare the data PID value (DATA0 or DATA1) for IN and OUT transactions.
Speed	Is set to logic 1 for low-speed device and logic 0 for high-speed device

The fields in Table 6-3 are generic PTD fields that are used in all USB transfers.

6.1.2. Traffic Specific Fields

There are a number of traffic specific fields that are specifically designed to enhance the bulk, isochronous and interrupt transfers as follows:

Bulk Paired (1 bit), Ping-Pong (1 bit)

ISO StartingFrame (7 bits), Last (1 bit)

INT PollingRate (3 bits), StartingFrame (5 bits)

6.2. Copying Data to the ISP1362

The data in the ATL buffer is accessed through the HcATLBufferPort or HcDirectAddressData register. Accessing through the HcDirectAddressData port is more efficient and flexible because it allows access to any location in the buffer area. However, you must be careful in calculating the address to write, as the direct addressing method allows writing to anywhere within the buffer. Incorrect address will certainly corrupt the buffer.

The PTD constructed using the method described in Section 5.1 is then copied into the ATL buffer, by PIO or DMA.

6.3. *Activating the ISP1362 and Checking Transfer Status*

After transferring the valid PTD into the ATL buffer, the Host Controller must be informed about whether it must process data, and also which part of the data is must processed, if it is to process data. There are two levels of buffer activation control: overall buffer level and block level.

6.3.1. Overall Buffer-Level Activation—HcBufferStatus Register

When the ISP1362 is in the operational mode, it checks the status of the various buffers by using the HcBufferStatus register. The first four bits of the HcBufferStatus register act as a "switch" that determine whether the Host Controller will process data in each buffer area.

Table 6-4: Bits in HcBufferStatus to Activate the Buffer Areas

Bit	Buffer Area
0	PTL0
1	PTL1
2	INT
3	ATL

When the bits in Table 6-4 are all logic 0 (inactive), the data in the buffer area will not be processed. When any of the bits is set to logic 1, the Host Controller will check a number of other registers and depending on the settings in those registers, it will take appropriate actions. This process will be explained in details in this chapter.

The bit 3 in the HcBufferStatus register must be set to logic 1 for the ISP1362 Host Controller to start processing PTD in the ATL buffer.

Table 6-5: HcBufferStatus Register: Bit Allocation

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol			reserved			PairedPTD PingPong	ISTL1 BufferDone	ISTL0 BufferDone
Reset	-	-	-	-	-	0	0	0
Access	-	-	-	-	-	R	R	R
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved	ISTL1_ Active Status	ISTL0_ Active Status	Reset_HW PingPong Reg	ATL_Active	INTL_ Active	ISTL1 BufferFull	ISTL0 BufferFull
Reset	-	0	0	0	0	0	0	0
Access	-	R	R	R/W	R/W	R/W	R/W	R/W

6.3.2. Block-Level Activation—HcATLSkipMap and HcATLLastPTD

The ATL buffer is separated into blocks of equal size (up to 32 blocks), as specified by the HcATLBlockSize register. If two blocks of 1000 bytes ATL buffers are required in an application, HcATLBufferSize must be at least 2016 bytes (1000 + 1000 + 8 + 8), due to the additional 8 bytes of header per PTD.

Each of these blocks can be individually activated or de-activated using HcATLSkipMap and HcATLLastPTD.

HcATLSkipMap allows the programmer to mask out any particular PTD in the ATL buffer. This 32-bit register is a bitmap representation of up to 32 blocks in the ATL buffer. If the ATL buffer has less than 32 blocks, the corresponding bits will be ignored. Logic 1 in the bit 7 of the HcATLSkipMap register, for example, will mask out the block 7 in the ATL buffer. (The first block in the ATL buffer is the block 0.)

HcATLLastPTD tells the Host Controller when to stop searching for active PTDs. This register is a bitmap representation to up to 32 blocks in the ATL buffer. Starting from bit 0, the Host Controller checks the register bits upward. When it encounters the first bit that is set to logic 1, it will be considered as the "LastPTD".

For example: Consider that HcATLLastPTD = 0x00300000.

Its binary equivalent is 0000 0000 0011 0000 0000 0000 0000 0000.

The first occurrence of 1 is at location 20. The Host Controller will search for active PTD from block 0 to block 20.

6.4. *Checking Status of the ATL Transfer*

After a PTD has been processed, the Host Controller will update the PTD and a number of registers. An interrupt may or may not be generated, depending on the setting that you have chosen.

6.4.1. Checking the PTD (Polling)

The Host Controller on processing the PTD updates a number of fields. These fields are given in Table 6-6.

Table 6-6: Fields Updated After PTD Processing

Name	**Description**
Active	Set to logic 0 upon completion of transfer
CompletionCode[3:0]	Reflects completion status
ActualBytes[9:0]	Actual amount of data transferred
Toggle	Is toggled by the Host Controller

6.4.2. Checking HcATLDoneMap (Polling)

The HcATLDoneMap register provides a complete and real-time status report of the PTDs in the ATL buffer. Each bit in the register represents the status of one of the blocks in the ATL buffer. If a block is processed, its corresponding bit in the HcATLDoneMap register will be set.

Note that this register is cleared on reading. It is recommended that you do not read this register, unless the ATL_IRQ interrupt has been received.

6.4.3. Interrupt Driven Checking

The ISP1362 can be programmed to generate an interrupt on completion of a number of ATL PTDs. This flexible interrupt generation method allows you to choose the optimum reaction time or system efficiency in the USB host stack.

The HcATLPTDDoneThresholdCount register controls a number of PTDs that must be processed before an interrupt can be generated. This register can be set to "1" if an interrupt must be generated for every PTD processed.

6.5. *Not Acknowledge (NAK)*

HcATLPTDDoneThresholdTimeout determines when an interrupt must be generated when the target device returns a Not Acknowledge (NAK). This register specifies the number of milliseconds of NAKs the Host Controller will get.

6.6. *Post-Transfer Processing*

Once the PTD is processed, the host software must check CompletionCode to see if there is any error. If CompletionCode is 0 (no error), 8 (Data Overrun) or 9 (Data Underrun), it will proceed to retrieve data if it is an IN data stage, or proceed to the next if it is an OUT data stage.

6.7. *Example: Sending OUT a Setup Token*

One of the basic transfers in the USB device enumeration is to get a device descriptor from the target device. In this example, a PTD will be constructed and the procedure of using the ISP1362 will be explained in detail.

Note: This in a case in which a PTD generates just one transaction in a Control Transfer.

6.7.1. Constructing the PTD

To get a device descriptor from the connected USB device, a "USB Device Request" must be constructed first. Format of the USB Device Request can be found in USB Specification 2.0 (Chapter 9.3.1). In this case, the 8-bytes request will be 0x80, 0x06, 0x00, 0x02, 0x00, 0x00, 0x08, 0x00 from byte 0 to byte 7, respectively. Since the ISP1362 uses 16-bit access, the four words to be written to the ATL buffer must be 0x0680, 0x0200, 0x0000 and 0x0008. Note that this is **not** the PTD, but the PTD payload.

For a payload size of 8 bytes, Table 6-7 shows the bit description of the PTD structure.

Table 6-7: PTD Bit Description

Name	Description
TotalBytes[9:0]	Must be set to 8; status field to be updated by the Host Controller
MaxPktSize[9:0]	Must be set 8 because all control endpoints support at least 8 bytes in a packet
EndpointNumber[3:0]	Is 0x00 (control endpoint)
DirToken[1:0]	Is 00B (setup)
FunctionAddress[6:0]	Is 0x00 because all un-enumerated devices respond to request on address 0
Active	Is set to logic 1 to indicate that this field is now **active**
Toggle	Is set to logic 0
Speed	Is set to logic 1 for low-speed device, or logic 0 for high-speed device
CompletionCode[3:0]	Status field to be updated by the Host Controller
Last	Is not used in the ATL and INT transactions

In this example the target device is a mouse. Therefore, "Speed" must be logic 1 (for low-speed).

The PTD will be 0x00, 0x08, 0x08, 0x08, 0x08, 0x00, 0x00, 0x00 from byte 0 to byte 7, respectively.

Now that both the PTD and the PTD payload are ready, they will be combined (PTD followed by the PTD payload) and copied into the ATL buffer. The write_atl() function developed in Section 3 can be used for this purpose.

6.7.2. Activating the PTD

Assuming that the Host Controller is in the operational mode, it must now be informed that data in the ATL buffer is ready for processing, i.e. the ATL_Active bit in HcBufferStatus must be set to logic 1.

Now that the ATL buffer is active, the block that you copied the PTD to must also be activated. This step involves the HcATLSkipMap and HcATLLastPTD registers. Once it is activated, the Host Controller will send out the setup token as specified by the PTD.

6.7.3. Looking at the Result

Once the PTD has been processed, it is then read back from the ATL buffer. You will notice that the fields given in Table 6-8 have been updated.

Table 6-8: Fields Updated after PTD Processing

Name	Modification
Active	Set to logic 0 on completion
CompletionCode[3:0]	Reflects completion status
ActualBytes[9:0]	Actual amount of data transferred
Toggle	Will be toggled by the Host Controller

Example: The PTD header before and after the transaction is 0x00, 0x08, 0x08, 0x08, 0x08, 0x00, 0x00, 0x00 from byte 0 to byte 7, respectively.

Table 6-9: Changes after the Transaction

Before	After	Remarks
0x00	0x08	First 8 bits of ActualBytes
0x08	0x0C	CompletionCode[7:4], Active[3], Toggle[2], Last 2 bits of ActualBytes
0x08	0x08	First 8 bits of MaxPktSize
0x08	0x08	EndpointNumber[7:4], Last[3], Speed[2], Last 2 bits of MaxPktSize
0x08	0x08	First 8 bits of TotalBytes
0x00	0x00	Special Bits[7:4], DirToken[3:2], Last 2 bits of TotalBytes
0x00	0x00	Special Bit[7], FunctionAddress[6:0]
0x00	0x00	Special Bits[7:0]

Note: The fields in blue changes after the transaction.

For actual meaning of the CompletionCode field, refer to the ISP1362 datasheet.

6.8. *Error Handling*

The Host Controller hardware reports any error occurred during 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. Table 6-10 lists the errors and the causes for the errors in the case of an OUT transaction and the treatment of the errors by the Host Controller in the case of an IN transaction.

Table 6-10: USB Transaction Error Codes

Fatal Errors	Error Code	IN Token	OUT Token
ERROR_CRC,	01	No ACK sent	Not applicable
ERROR_Bitstuffing	02	No ACK sent	Not applicable
ERROR_DatatTogglingMismatch	03	ACK sent	Not applicable
ERROR_Stall	04	No ACK sent	Host received Stall from device
ERROR_DeviceNotResponding	05	No ACK sent	Host did not received a hand shake reply within 18Bit time, or 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 send	Not applicable

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.

7. Case Study: USB Mouse

In this chapter, a complete program that enables the ISP1362 DOS evaluation kit to enumerate or use a standard USB mouse is presented. If you wish to get familiarized with the basic USB host programming technique, you can use this as a reference. The code is written entirely in C for IBM PC, using Turbo C ver. 3.0 and all header files are included. No special operating system support is required for this, and it will be fairly easy to port this program to any other platform.

The steps involved in this program are:

1. Configuring the ISP1362
2. Setting the Host Controller to the operational state
3. Enabling the port on detecting a connection
4. Assigning an address to the connected device

5. Getting the required descriptors
6. Setting configuration
7. Polling for mouse movement data.

The method given here is as simple as possible. It might not be the optimal way to enumerate a general USB device but it does illustrate the way in which the ISP1362 Host Controller can be used.

Four important functions that are widely used here are:

```
w16()          Writing to a 16-bit Host Controller register
r16()          Reading from a 16-bit Host Controller register
w32()          Writing to a 32-bit Host Controller register
r32()          Reading from a 32-bit Host Controller register
```

7.1.1. Configuring the ISP1362

First, the following three buffers must be configured to suitable sizes:

```
w16(HcATLBufferSize,          1536);
w16(HcINTLBufferSize,         1024);
w16(HcISTLBufferSize,         512);
```

As this program will not use any interrupts, all interrupts are disabled:

```
w16(HcuPInterruptEnable,      0);
w32(HcInterruptDisable,       0xFFFFFFFF);
```

Note that disabling the interrupt does not stop the corresponding bit in the Interrupt register to be set when the event occurs. Disabling the interrupt stops the interrupt pin from being asserted when the event occurs.
This program uses the bit in HcµPInterrupt to check for completion of transfer. It does not require the actual interrupt signal from the chip to activate an ISR.

Setting up the controls of the ATL buffer:

```
w32(HcATLSkip,         0xFFFF FFFE    );             // Disable all but the first ATL PTD
w32(HcATLLast,         1              );             // First PTD is the last
w16(HcATLBlkSize,      64             );             // Block size of 64 bytes
w16(HcATLThrsCnt,      1              );             // Generates interrupt for every PTD Done
w16(HcATLTimeOut,      5              );             // 5 ms before giving up on NAKs
```

Disable processing of all buffers:

```
w16(HcBufferStatus, 0);
```

7.1.2. Setting the Host Controller to the Operational State and Enabling the Port

In this step, two functions are used: set_operational() (see Figure 7-1) and enable_port() (see Figure 7-2). As the name suggests, the enable_port() function enables a port if a USB device is found to be connected to it. The set_operational function sets the Host Controller to the operational mode. Once in the operational mode, the bits in the HcBufferStatus register can request the Host Controller to start processing the data in buffers.

```
void set_operational(void)
{
 w16(HcHWCfg           , 0x002D);
 w32(HcFmItv           , 0x25002EDF);
 w32(HcControl         , 0x00000680);
}
```

Figure 7-1: set_operational() Subroutine

```
void enable_port(void)
{
 unsigned long dat32;

 w32(HcRhP1,0x00000102);
 w32(HcRhP2,0x00000102);
```

```
 w32(HcRhA,0x05000B01);
 w32(HcRhB,0x00000000);

 w32(HcRhP1,0x00000102);
 w32(HcRhP2,0x00000102);

 dat32=r32(HcRhP2);

 if((dat32&0x00000001)==1)
 {
  set_port_speed(2,0);
  if(((dat32)&(0x00000200))!=0)
  {
   set_port_speed(2,1);
  }
 }

 dat32=r32(HcRhP1);

 if((dat32&0x00000001)==1)
 {
  set_port_speed(1,0);
  if(((dat32)&(0x00000200))!=0)
  {
   set_port_speed(1,1);
  }
 }
}
```

Figure 7-2: enable_port() Subroutine

7.1.3. Some Backgrounds

Before you proceed to the next step, which is assigning an address, you will learn about the two basic routines—make_ptd() (see Figure 7-3) and send_control() (see Figure 7-4)—used in this section to construct and process a PTD.

The make_ptd() routine is used to construct a data structure that conforms to the PTD. To call this function, the calling function must provide the following parameters:

***rptr** a pointer pointing to the array in which resultant PTD must be stored

token direction token, can be IN, OUT or setup

ep the target endpoint

max maximum packet size

tog toggle bit

addr function address

port which port is the target device connected to.

```
void make_ptd(int *rptr,char token,char ep,int max,char tog,char addr,char port)
{
 ptd2send.c_code=0x0F;
 ptd2send.active_bit=1;
 ptd2send.toggle=tog;
 ptd2send.actual_size=0;
 ptd2send.endpoint=ep;
 ptd2send.last_ptd=0;

 ptd2send.speed=port_speed;
 if(port==1) {ptd2send.speed=port1speed;}
 if(port==2) {ptd2send.speed=port2speed;}

 ptd2send.max_size=max;
 ptd2send.total_size=max;
 ptd2send.pid= token;
 ptd2send.format=0;
 ptd2send.fm=0;
 ptd2send.func_addr=addr;

 *(rptr+0)=    (ptd2send.c_code       &0x0000)<<12
                |(ptd2send.active_bit      &0x0001)<<11
```

```
                  |(ptd2send.toggle          &0x0001)<<10
                  |(ptd2send.actual_size   &0x03FF);

 *(rptr+1)=  (ptd2send.endpoint              &0x000F)<<12
                  |(ptd2send.last_ptd        &0x0001)<<11
                  |(ptd2send.speed                  &0x0001)<<10
                  |(ptd2send.max_size        &0x03FF);

 *(rptr+2)=  (0x0000                  &0x000F)<<12
                  |(ptd2send.pid             &0x0003)<<10
                  |(ptd2send.total_size      &0x03FF);

 *(rptr+3)=  (ptd2send.fm              &0x00FF)<<8
                  |(ptd2send.format          &0x0001)<<7
                  |(ptd2send.func_addr     &0x007F);
}
```

Figure 7-3: make_ptd() Subroutine

The send_control() routine copies the PTD and payload (if any) into the ATL buffer and activates the buffer. It then waits for it to be completed. Once the PTD has been processed (i.e. the transaction is completed), the routine terminates and returns a number greater than zero together with the processed PTD to the calling function.

If the target device NAKs continuously, this routine will return a "0" because of a time out in the polling loop.

```
unsigned int send_control(unsigned int *a_ptr,unsigned int *r_ptr)
{
 unsigned int cnt=retry;
 unsigned int active_bit;
 unsigned int abuf[128];
 unsigned int UpInt;
 unsigned int ccode;
 unsigned int timeout=9;

 do
 {
  cnt=retry;

  write_atl(a_ptr,8);  // Write 16 bytes
  w16(HcUpInt,0x100);
  r32(HcATLDone);      // Read and clear done map, enables ATL interrupt
  w16(HcBufStatus,0x08);

  do
   {
        UpInt=r16(HcUpInt);
        if( (UpInt&0x100)!=0) {active_bit=0;}
        else {active_bit=1;}

        poll(50);
        cnt--;
   }
  while((cnt!=0)   &&   (active_bit!=0));
  w16(HcBufStatus,0x00);
  read_atl(r_ptr,72);

  ccode=((*r_ptr)&(0xF000))>>12;

  timeout--;
 }
 while(  (ccode!=0) && (timeout!=0)  );

 return(cnt);
}
```

Figure 7-4: send_control() Subroutine

This routine polls the ATL_IRQ bit in the HcµPInterrupt register (see

Table 5-4) for the status of the PTD. Since there is only one PTD in the ATL buffer, you can be sure that a bit set at ATL_IRQ indicates that the PTD is done. If there is more than one PTD in the buffer, HcATLPTDDoneMap is used to determine which PTD is done. Note that even though there is only one PTD in the ATL buffer, HcATLPTDDoneMap must be read (by the Host Controller Driver) and cleared (by the hardware).

Enumeration:

Figure 7-5 shows the USB traffic in the enumeration stage and some mouse movement data transfers.

Figure 7-5: Enumeration Capture

ISP1362 Embedded Programming Guide Rev. 0.9

7.1.4. First Contact

Assuming that a mouse (and **only** one mouse) is connected to the ISP1362, after the set_operational() and enable_port() routines are run. This USB mouse must be connected and in the powered mode. In the very first contact with the USB mouse, you do a partial Get_Descriptor (device) to find out the MaxPktSize of the device. The traffic flow of the partial Get_Descriptor is shown in Figure 7-6.

Figure 7-6: Get_Descriptor Capture

Three PTDs are required for a control transfer: Setup, Data and Status.

Setup Stage:

Note that the device is not assigned an address yet, and it will respond to any USB traffic directed to function address 0. Since nothing, except that it is low speed as detected in HcPortStatus[N], is known about the USB mouse at this stage, the parameters in Table 7-1 are chosen.

Table 7-1: Values of Fields in the PTD

Name	Values Chosen
FunctionAddress[6:0]	0
TotalBytes[9:0]	8 bytes
DirToken[1:0]	setup
EndpointNumber[3:0]	0
Speed	low speed

For the device descriptor, the payload must be 0x0680, 0x0100, 0x0000, 0x8. Details can be found in the USB Specification Chapter 9.

A code example of the Setup stage is given in Figure 7-7.

```
make_ptd(cbuf,SETUP,0,8,0,0,port);    // Makes the PTD
array_app(cbuf+4,dev_req,4);          // Adds payload to the PTD
tout=send_control(cbuf,rbuf);         // Sends out the PTD
ccode|=(rbuf[0]&0xF000)>>12;          // Checks the completion code
```

Figure 7-7: Code Example of the Setup Stage

The code given in Figure 7-7 will produce the "Transaction 1 of Transfer 0" in the traffic capture as shown in Figure 7-6. You can see that a single PTD written into the ATL buffer has produced three USB data packets (106, 107, 108). This is done by the ISP1362 Host Controller hardware, without any CPU intervention.

Data Stage:

In this stage, the Host Controller expects 8 bytes of data from the USB mouse that it has just requested in the Setup stage. Therefore, a data IN is sent out. A pseudo code of the Data stage is given in Figure 7-8.

```
make_ptd(cbuf,IN,0,8,0,0,port);       // Makes the PTD
tout=send_control(cbuf,rbuf);         // Sends out the PTD
ccode|=(rbuf[0]&0xF000)>>12;          // Checks the completion code
```

Figure 7-8: Code Example of the Data Stage

Again, a single PTD produced three data packets on the USB line. 8 bytes of data are received from the USB mouse: 0x0112, 0x0110, 0x0000, 0x0800.

We can deduce from this that it is a USB 1.1 device, with a MaxPktSize of 8 bytes.

Status Stage:

For a Setup transfer with data IN, the Host Controller must conclude the transaction with an empty data OUT. Figure 7-9 shows a pseudo code of the Status stage.

```
make_ptd(cbuf,OUT,0,0,0,0,port);      // Makes the PTD
tout=send_control(cbuf,rbuf);         // Sends out the PTD
ccode|=(rbuf[0]&0xF000)>>12;          // Checks the completion code
```

Figure 7-9: Code Example of the Status Stage

7.1.5. Reset and SetAddress

Writing a logic 1 to the bit 4 of the HcPortStatus[N] register resets the corresponding port. This is usually done in the beginning of the enumeration, just after the detection of a connection on the port. After the reset, the Host Controller must allocate an available address to the newly connected USB device. The routine given in Figure 7-10 can be used for such a purpose.

```
unsigned int set_address(int old_addr, int new_addr, int port)
{
 unsigned int cbuf[128];
 unsigned int rbuf[128];
 unsigned int uni_req[4]={0x0500,0x0000,0x0000,0x0000};
 unsigned int mycode=0;
 unsigned int tcnt;
```

```
 uni_req[1]=new_addr;

 w16(HcUpInt,0x100);

 r32(HcATLDone);

 make_ptd(cbuf,SETUP,0,8,0,old_addr,port);                  // SETUP stage
 array_app(cbuf+4,uni_req,4);
 tcnt=send_control(cbuf,rbuf);
 mycode=(*rbuf&0xF000)>>12;

 if(tcnt==0)
 {
  mycode|=0xF000;                                           // tcnt=0 means time out
 }

 if(mycode==0)
 {
  // Send out data IN packet
  make_ptd(cbuf,IN,0,0,1,old_addr,port);                    // Status stage
  tcnt=send_control(cbuf,rbuf);
  mycode=(*rbuf&0xF000)>>12;
   if(tcnt==0)
        {
         mycode|=0xF000;
        }
 }

 r32(HcATLDone);

 return(mycode);
}
```

Figure 7-10: set_address () Subroutine

Note: Some operating systems perform a partial Get_Descriptor before resetting the device, whereas some operating systems reset the device on connection and start assigning an address thereafter.

The set_address() routine has only two stages: Setup and Status. If an error is encountered in the Setup stage, it will not proceed to the Status stage. Figure 7-11 shows the capture of the set_address() routine.

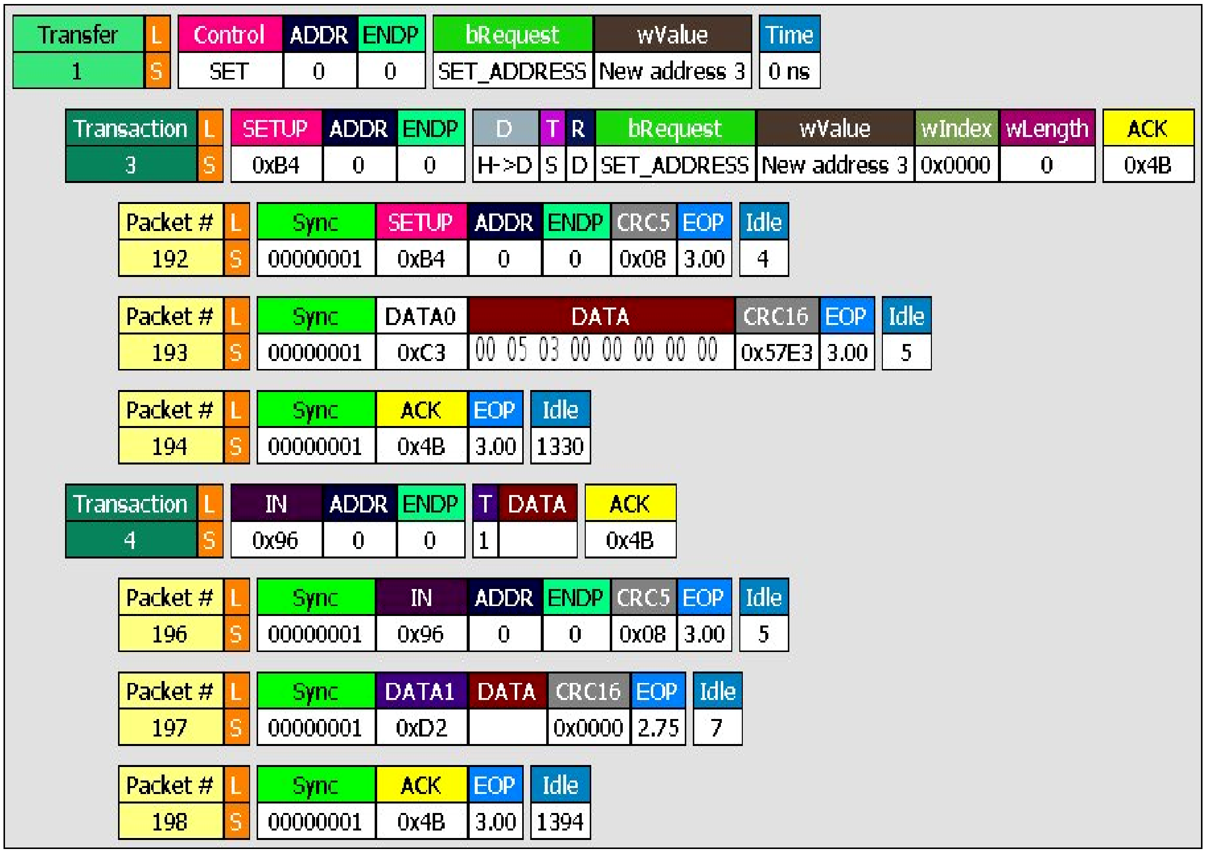

Figure 7-11: set_address() Capture

7.2. *get_control() Function*

After assigning an address to the USB device, you are now ready to proceed to find out more about what type of device is connected. You will get several descriptors from the USB device and from these descriptors you can deduce if a USB mouse is indeed connected, and then proceed to use the mouse, if it is connected.

You will request the following descriptors:

- Device Descriptor
- Configuration Descriptor
- Endpoint Descriptor
- String Descriptor
- Human Interface Device (HID) Descriptor.

The first problem in the above-mentioned requests is that you do not know the size of the descriptor until you receive it. However, it is required to include the correct value of data size in the TotalBytes field in the PTD. Therefore, to set the correct value before the transaction, you must request for a partial descriptor by requesting for just 8 bytes of data. The size of the descriptor is in these 8 bytes of data and you will be able to use the correct data size in the next transfer. The flowchart of the get_control() function is shown in Figure 7-12.

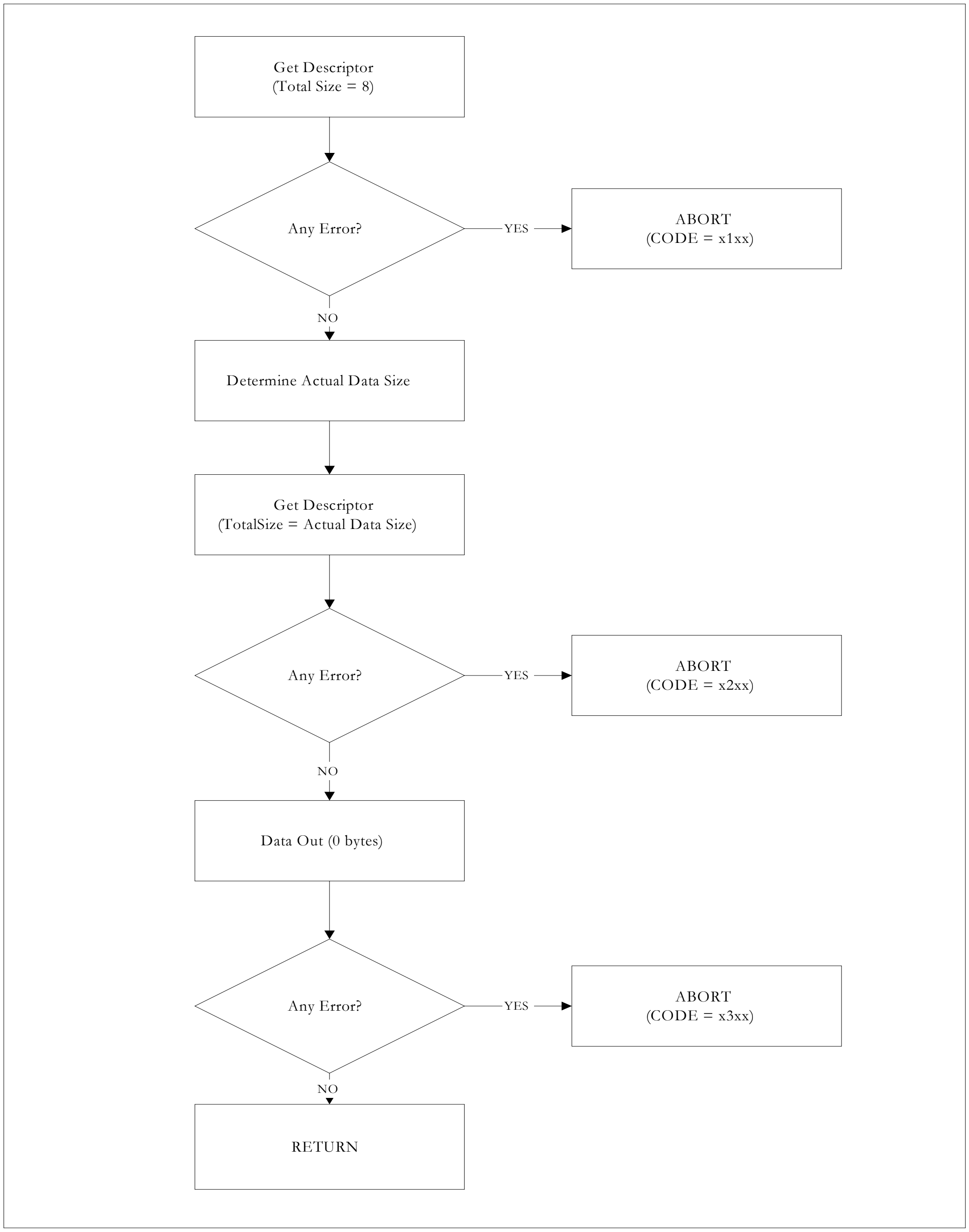

Figure 7-12: get_control() Flowchart

After getting the HID descriptor, the program checks to see if the device is indeed a mouse. Once the identity is confirmed, the program polls the mouse every 8 ms by using the following function.

```
unsigned int get_control(unsigned int
*rptr,unsigned int addr,char
control_type,unsigned int extra,int port)
{
 unsigned int cbuf[128];
 unsigned int rbuf[128];
 unsigned int cnt=0,lcnt=0;
 unsigned int toggle_cnt=0;
 unsigned int word_size;
 unsigned int DesSize,MaxSize,RemainSize;
 unsigned int LocalLimit;

 unsigned int dev_req[4]={0x0680,0x0100
,0x0000,0x8};
 unsigned int cfg_req[4]={0x0680,0x0200
,0x0000,0x8};
 unsigned int str_req[4]={0x0680,0x0300
,0x0000,0x8};
 unsigned int int_req[4]={0x0680,0x0400
,0x0000,0x8};
 unsigned int end_req[4]={0x0680,0x0500
,0x0000,0x8};
 unsigned int hid_req[4]={0x0681,0x2100
,0x0000,0x8};

 unsigned int ccode=0;
 unsigned int stage=1;
 unsigned int tout; // Timeout indicator

 // Stage 1: Send out first setup packet
 make_ptd(cbuf,SETUP,0,8,0,addr,port);
 if(control_type=='D')
{array_app(cbuf+4,dev_req,4);}
 if(control_type=='C')
{array_app(cbuf+4,cfg_req,4);}
 if(control_type=='S')
{array_app(cbuf+4,str_req,4);}
 if(control_type=='I')
{array_app(cbuf+4,int_req,4);}
 if(control_type=='E')
{array_app(cbuf+4,end_req,4);}
 if(control_type=='H')
{array_app(cbuf+4,hid_req,4);}

 if(control_type=='S')
 {
  cbuf[5]=cbuf[5]|extra;   // This is for
string processing
 }

 tout=send_control(cbuf,rbuf);
 if(tout==0) {ccode|=0xF000;}   // Indicates
timeout in transaction

 if(ccode==0)
 {
  toggle_cnt++;

make_ptd(cbuf,IN,0,8,toggle_cnt%2,addr,port)
;
  tout=send_control(cbuf,rbuf);
  ccode|=(rbuf[0]&0xF000)>>12;

  if(ccode==0x09)  // Descriptor size is
less than 8
  {
   ccode=0;
  }

  if(tout==0) {ccode|=0xF000;}   //
Indicates timeout in transaction

  if(control_type!='C')
  {
   DesSize=((rbuf[4]&0x00FF));
  }
```

```
  if(control_type=='C')
  {
   DesSize=rbuf[5];
  }

  if(control_type!='D')
  {
   MaxSize=addr_info(addr,'R','M',MaxSize);
  }

  if(control_type=='D')
  {
   MaxSize=(rbuf[7]&0xFF00)>>8;
   if(MaxSize<8) {MaxSize==8;}

   addr_info(addr,'W','M',MaxSize);
  }

  if(control_type=='H')
  {
   DesSize=(rbuf[7]&0xFF00)>>8;
   if(DesSize<8) {DesSize==8;}
  }

// printf("\nDesSize = %2d    MaxSize =
%2d",DesSize,MaxSize);
 }

 if(ccode==0)
 {
  // Send out data OUT packet

make_ptd(cbuf,OUT,0,0,toggle_cnt%2,addr,port
);
  tout=send_control(cbuf,rbuf);
  if(tout==0) {ccode|=0xF000;}   //
Indicates Timeout in transaction

  ccode|=(rbuf[0]&0xF000)>>12;
 }
 // Stage 1: END

 if(ccode==0)
 {
  stage=2;

  hid_req[1]=0x2200; // Change HID req into
HID report descriptor

  // Stage 2
  make_ptd(cbuf,SETUP,0,8,0,addr,port);
  if(control_type=='D')
{array_app(cbuf+4,dev_req,4);}
  if(control_type=='C')
{array_app(cbuf+4,cfg_req,4);}
  if(control_type=='S')
{array_app(cbuf+4,str_req,4);}
  if(control_type=='I')
{array_app(cbuf+4,int_req,4);}
  if(control_type=='E')
{array_app(cbuf+4,end_req,4);}
  if(control_type=='H')
{array_app(cbuf+4,hid_req,4);}

  if(control_type=='S')
  {
   cbuf[5]=cbuf[5]|extra;
  }

  cbuf[7]=DesSize;
  tout=send_control(cbuf,rbuf);
  if(tout==0) {ccode|=0xF000;}   //
Indicates Timeout in transaction

  word_size=(DesSize+1)>>1;
```

```
  RemainSize=DesSize;

  toggle_cnt=0;
  cnt=0;
  do
  {
   // Send out data IN packet
   toggle_cnt++;

   // The last transaction where remaining
data size < max pac size
   if(RemainSize<MaxSize)
   {
        make_ptd(cbuf,IN,0,RemainSize,toggle_
cnt%2,addr,port);
   }

   // Normal
   else
   {
        make_ptd(cbuf,IN,0,MaxSize,toggle_cnt
%2,addr,port);
   }

   tout=send_control(cbuf,rbuf);
   if(tout==0) {ccode|=0xF000;}   //
Indicates Timeout in transaction

   ccode|=(rbuf[0]&0xF000)>>12;

   RemainSize=RemainSize-MaxSize;

   LocalLimit=MaxSize>>1;

   if(ccode==0)// Data In is successful
   {
        lcnt=0;
        do
        {
         // Copy the data located right after
the 8 bytes PTD
         *(rptr+cnt)=rbuf[4+lcnt];


         cnt++;
         lcnt++;
        }
        while(lcnt<(LocalLimit));
}

}
  while((cnt<word_size)&&(ccode==0));
 // Stage 2: END
 }

 if(ccode==0)
 {
  stage=3;

  // Stage 3: Send out DATA OUT packet

make_ptd(cbuf,OUT,0,0,toggle_cnt%2,addr,port
);
  send_control(cbuf,rbuf);

  ccode=(rbuf[0]&0xF000)>>12;
  // Stage 3: END
 }

 return( (ccode)|(stage<<8) );

// Byte 0 indicates the error code
// Byte 2 indicates at which stage the error
was encountered
// Byte 3 is F if time-out, else 0
}
```

7.2.1. Getting Descriptors

In this section, the get_control() function is used to get a device descriptor from the mouse. In the code given in Figure 7-13, "status" is a variable returned by a subroutine that assigns address to devices attached to the ISP1362 host by using the set_address function.

Bit Structure of the "status" variable:

Bit 1: Port 1 active

Bit 8: Port 2 active.

This routine checks the status of port 1. If port 1 has a device attached and an address (In this case, address is 1 for port 1 and 2 for port 2) has been successfully assigned to it, status &0x0100 becomes TRUE and the routine will be run.

The variable "mycode" provides the result of the get_control() function. The lowest nibble (bits 3-0) shows the CompletionCode of the last executed transaction. The second highest nibble (bits 11-8) shows the last executed stage. For a completely successful get_control(), the value of "mycode" must be 0x0300, which means it has completed all the three stages without any error.

Two pieces of information—iManufacturer and iProduct—are extracted from the returned descriptor. These are required later in the HID request to obtain the name of the manufacturer and the name of product for this particular device, respectively. The iManufacturer and iProduct values are stored in a routine named "addr_info" that acts as a static data bank.

Next, the get_control() function is used to get a HID descriptor from the mouse. It then checks for the identity of the device by reading the second word of the descriptor. It must be 0x0209, if the connected device is a mouse.

```
if( (status&0x0100)!=0) // Port 1 active
 {
  // Check port 1 for mouse
  printf("\nGetting device descriptor for device at port 1... ");

  mycode=get_control(rbuf,1,'D',0,1);
  printf("%04X",mycode);

  if(mycode==0x0300)
  {
   iManufacturer = rbuf[7]&0xFF;
   iProduct = (rbuf[7]&0xFF00)>>8;

   addr_info(1,'W','O',iManufacturer);
   addr_info(1,'W','P',iProduct);

   mycode=get_control(rbuf,1,'H',addr_info(1,'R','P',0),1); // Getting HID descriptor

   if( *(rbuf+1)==0x0209  )
   {
        printf("\nMouse Detected @Port1!!! ");

        mouse01=1;
   }
  }
 }
```

Figure 7-13: Code Example for Checking whether a Mouse is Connected

7.3. *set_config Function*

After confirming that the device connected is indeed a mouse, the host must now choose a configuration on the device. The set_config() function (see Figure 7-14) allows the host to set a configuration on the mouse and once this is done, the mouse is ready to transmit movement data.

```
void set_config(int addr, int config)
{
 unsigned int cbuf[128];
 unsigned int rbuf[128];
 unsigned int uni_req[4]={0x0900,0x0000,0x0000,0x0000};
 unsigned int tcnt;
```

```
 unsigned int mycode=0;

 uni_req[1]=config;

 w16(HcUpInt,0x100);

 r32(HcATLDone);
 r32(HcATLDone);

 make_ptd(cbuf,SETUP,0,8,0,addr,addr);
 array_app(cbuf+4,uni_req,4);
 tcnt=send_control(cbuf,rbuf);

 if(tcnt==0) { mycode|=0xF000;}
 mycode=mycode | (*rbuf&0xF000)>>12;

 if(mycode==0)
 {
 // Send out DATA IN packet
  make_ptd(cbuf,IN,0,0,1,addr,addr);
  tcnt=send_control(cbuf,rbuf);

  if(tcnt==0) { mycode|=0xF000;}
  mycode=mycode | (*rbuf&0xF000)>>12;
 }

 r32(HcATLDone);
 r32(HcATLDone);

 return(mycode);
}
```

Figure 7-14: set_config() Function

To call set_config(), the code given in Figure 7-15 must be used.

```
mycode=set_config(1,1);
  printf("\nSetting config of device 1 to config 1... %04X",mycode);

  if(mycode==0)
  {
   play_mouse(1);
  }
```

Figure 7-15: Code for Calling the set_config() Function

If the set_control() function returns a "0", it means the execution is successful and you can now proceed to get the movement data from the mouse.

Getting the Mouse Movement Data

Mouse movement data can be obtained by sending a DataIn to the mouse endpoint using an Interrupt transfer. This is accomplished using the make_int_ptd() and send_int() functions, which are equivalent to make_ptd() and send_control() used earlier.

The 4 bytes (2 words) of data that the mouse returned follow this format:

Word[0], Upper Byte: signed 8-bit integer for X-direction movement

Word[1], Lower Byte: signed 8-bit integer for Y-direction movement

Word[0], Bit 0: left button pressed

Word[0], Bit 1: right button pressed

Word[0], Bit 2: middle button pressed.

A snapshot consisting of two USB mouse movement data transfer is given in Figure 7-16.

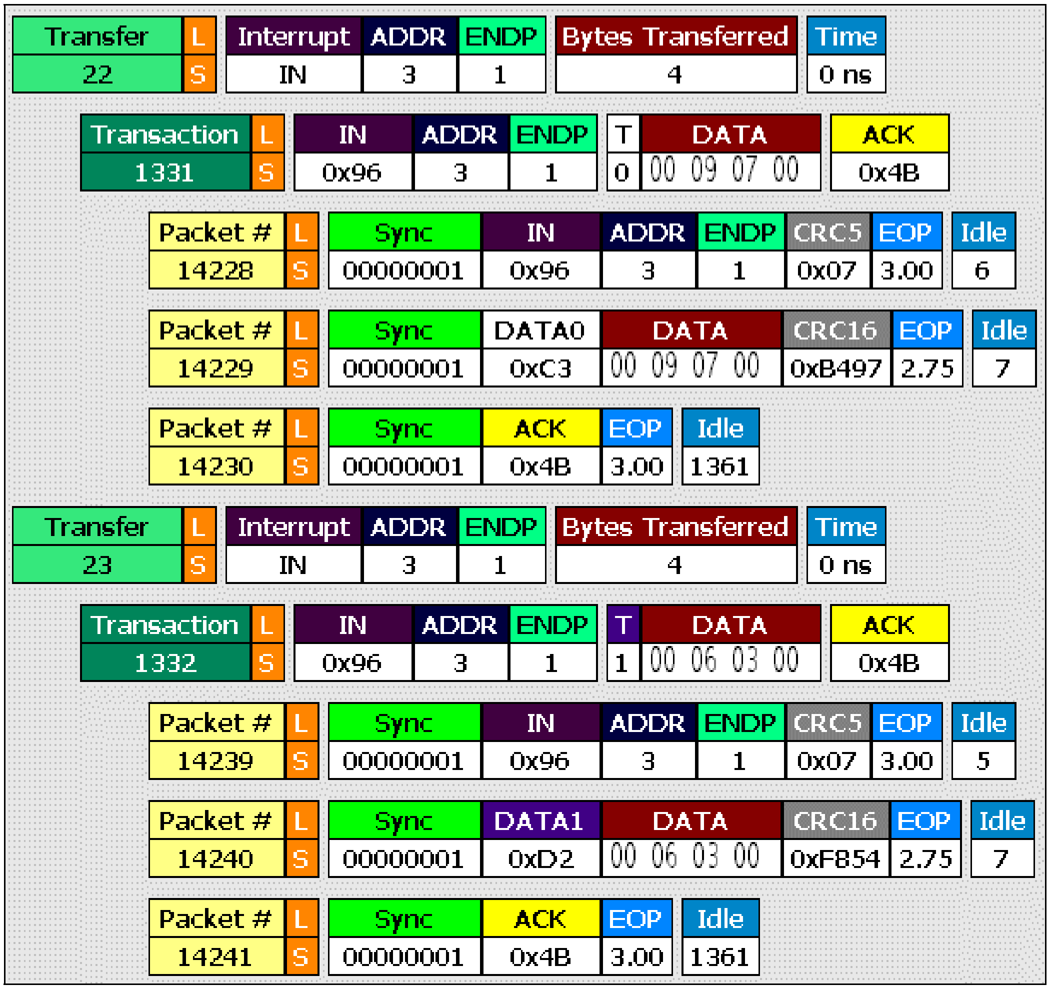

Figure 7-16: Snapshot of the Mouse Movement Data on the USB Bus

8. Advanced Feature 1: Multi-frame Buffering of the ISO Transfer

An isochronous (ISO) transfer is a periodic, fixed-length data stream that is normally used for audio and video data. ISO transfer requires timely delivery and it tolerates occasionally missed data. ISO transfer requires no acknowledgement. The sender does not know about the success of the transfer and therefore, it will never be retried.

One major difficulty in using USB to deliver ISO data stream is the stringent requirement of timely delivery. A typical 44.1 kHz stereo 16-bit USB speaker requires 44.1 x 2 x 2 = 176.4 bytes of data to be delivered every millisecond. Failure in meeting this time limit results in distortion of video or audio. However, many CPUs in embedded system cannot meet this 1 ms time limit either because of the scheduling of operating system or because the CPU is busy handling other time critical applications.

The ISP1362 is designed keeping this problem in mind. The "Multi-frame Buffering" mechanism (patent pending) allows the Host Controller Driver to update the ISO buffer for a relatively longer time period (up to 15 ms) and yet deliver data to the device at a regular 1 ms interval. This chapter explains this very powerful feature of the ISP1362.

8.1. *Configuration of the ISO Buffer*

The ISP1362 has a total of 4096 bytes of buffer memory. In a typical operation scenario with a mixture of the bulk, isochronous and interrupt traffic, it is recommended that you set the ISO buffer at 1024 bytes each (i.e. 2048 bytes in total for ISO).

8.2. *ISO PTD Format*

The PTD structure for an ISO transfer is given in Figure 8-1.

Table 8-1: ISO PTD Structure: Bit Allocation

Bytes 1, 3, 5, 7								Bytes 0, 2, 4, 6							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CompletionCode				Active	Toggle	ActualBytes									
EndpointNumber				Last	Speed	MaxPktSize									
B5-7	B5-6	B5-5	B5-4	DirToken		TotalBytes									
StartingFrame								R[1]	FunctionAddress						

[1] R—denotes reserved.

Table 8-2: ISO PTD Structure: Bit Description

Name	Description
ActualBytes[9:0]	Actual amount of data transferred at the moment
MaxPktSize[9:0]	Maximum amount of data per packet
TotalBytes[9:0]	Total amount of data to be transferred
CompletionCode[3:0]	Reports success or errors in transaction
EndpointNumber[3:0]	Target endpoint number
DirToken[1:0]	Specifies IN, OUT or setup token
FunctionAddress[6:0]	Address of target device
Active	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.
Toggle	This bit is used to generate or compare the data PID value (DATA0 or DATA1) for IN and OUT transactions.
Speed	Is set to logic 1 for low-speed device, or logic 0 for high-speed device
Last	This indicates that it is the last PTD of a list (ITL or ATL).
StartingFrame	This field is used specifically for the ISO transfer. It determines when the Host Controller will process the PTD.

8.3. *Multi-Frame Buffering Control Registers*

The registers given in Table 8-3 are involved in the control of the ISP1362 multi-frame buffering mechanism.

Table 8-3: Registers Related to the ISO Transfer

Register	Remarks
HcISTLLength	ISO buffer size
HcISTLToggleRate	Controls the ISO buffer toggle rate
HcBufferStatus	Bits 0, 1, 5, 6, 8 and 9
HcµPInterrupt	Bits 1 and 2.
HcISTL0BufferPort	Data access
HcISTL1BufferPort	Data access

8.3.1. Multi-Frame Buffering Mechanism

Once set in the operational mode, the ISP1362 Host Controller checks the bits 0 (ISTL0BufferFull) and 1 (ISTL1BufferFull) in the HcBufferStatus register. The Host Controller Driver sets these two bits once it has completed writing ISO PTD in the ISO buffer. The Host Controller will start processing the ISO buffer only when ISTL0BufferFull is set to logic 1.

After detecting that ISTL0BufferFull is logic 1, the Host Controller resets the internal toggle counter to 0 and scans the ISTL0 buffer to check if any of the PTDs has a frame number that matches the current frame number. In case there is a match, the PTD will be processed and sent out. After sending, the Active bit in the PTD is set to logic 0.

The Host Controller will continue processing the ISTL0 buffer for a number of milliseconds, depending on the value in the HcISTLToggleRate register. The Host Controller keeps track of this by incrementing the internal toggle counter by one at every SOF. When the internal toggle counter reaches HcISTLToggleRate, the Host Controller toggles to the ISTL1 buffer, if ISTL1BufferFull is set to logic 1.

Table 8-4 is an example of multi-frame buffering, with HcISTLToggleRate = 3. At frame number < 3, the ISTL0 buffer is filled with three PTDs with the starting frame set to 3, 4 and 5. The ISTL1 buffer is filled with three PTDs with the starting frame set to 6, 7 and 8. ISTL0BufferFull and ISTL1BufferFull are set to logic 1, when the frame number is 2.

Table 8-4: Isochronous Buffering Mechanism

Frame Number	ISTL0	ISTL1	Action
3	Send the PTD with SF = 3	Idle, time for refilling	Nothing because both buffers are filled
4	Send the PTD with SF = 4		—
5	Send the PTD with SF = 5		—
6	Idle, time for refilling	Send the PTD with SF = 6	ISTL0BufferFull is set to logic 0 by the HCD
7		Send the PTD with SF = 7	—
8		Send the PTD with SF = 8	Must finish refilling ISTL0 (SF = 9, 10, 11)
9	Send the PTD with SF = 9	Idle, time for refilling	ISTL1BufferFull is set to logic 0 by the HCD
10	Send the PTD with SF = 10		—
11	Send the PTD with SF = 11		Must finish refilling ISTL1 (SF = 12, 13, 14)

By using toggling rate of 15 ms (maximum), the Host Controller Driver will need to refill the ISO buffer at a slow rate of 15 ms period. This can be easily achieved even in relatively slow embedded systems.

8.4. *Traffic, Host Controller and CPU Activities*

Figure 8-1 shows the typical traffic, Host Controller and CPU activities when Toggle is 3 isochronous transfer.

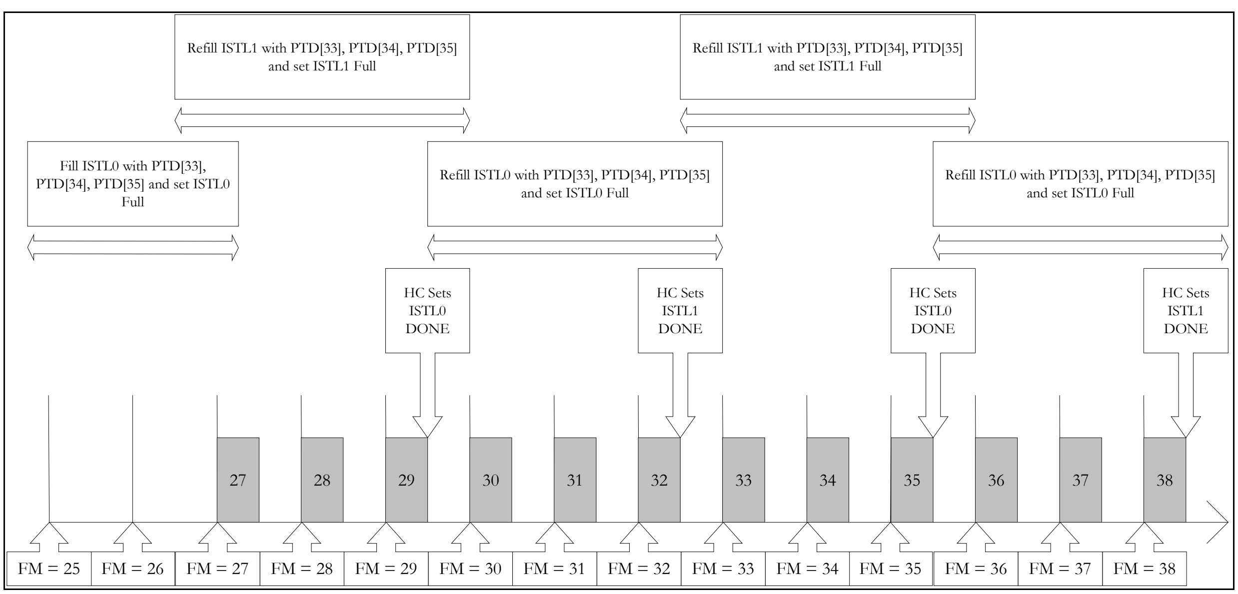

Figure 8-1: Traffic, Host Controller and CPU Activities

9. Advanced Feature 2: Paired-PTD for the Bulk Transfer

A bulk transfer is an aperiodic, non-time critical, data-integrity sensitive transfer. It is important that **all** data is transferred **correctly** but the time taken for the transfer is not that important. A typical use of this type of transfer is an external hard disk drive.

The ISP1362 uses the "paired-PTD" (patent-pending) mechanism to enhance the transfer speed of the bulk data. In an ideal environment in which there is no other USB device connected to the host, the ISP1362 can transfer 18 x 64 bytes of data to a single bulk endpoint in 1 ms, giving a data bandwidth of 1.152 Mbyte/s.

9.1. *Configuration of the ATL Buffer*

The ISP1362 has a total of 4096 bytes of buffer memory. In a typical operation scenario with a mixture of bulk, isochronous and interrupt traffic, it is recommended that you set the ATL buffer to 1536 bytes.

The ATL buffer in the ISP1362 uses a blocked architecture. The entire ATL buffer area is separated into blocks of equal sizes. HcATLBlockSize can determine the size of the blocks. Note that value in HcATLBlockSize does not include the 8 bytes taken by the PTD header. Therefore, a block of 128 bytes, for example, actually takes up 136 bytes (128 + 8) in the ATL buffer.

9.2. *PTD Format of Paired PTD*

The PTD structure for a paired-PTD transfer is given in Table 9-1.

Table 9-1: Paired-PTD Structure: Bit Allocation

Bytes 1, 3, 5, 7								Bytes 0, 2, 4, 6							
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
CompletionCode				**Active**	**Toggle**	**ActualBytes**									
EndpointNumber				B5-3	**Speed**	**MaxPktSize**									
Paired	**Ping Pong**	B5-5	B5-4	**DirToken**		**TotalBytes**									
B7								R[1]	**FunctionAddress**						

[1] R—denotes reserved.

Table 9-2: Paired-PTD Structure: Bit Description

Name	Description
ActualBytes[9:0]	Actual amount of data transferred at the moment
MaxPktSize[9:0]	Maximum amount of data per packet
TotalBytes[9:0]	Total amount of data to be transferred
CompletionCode[3:0]	Reports success or errors in transaction
EndpointNumber[3:0]	Target endpoint number
DirToken[1:0]	Specifies IN, OUT, or setup token
FunctionAddress[6:0]	Address of the target device
Active	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.
Toggle	This bit is used to generate or compare the data PID value (DATA0 or DATA1) for IN and OUT transactions.
Speed	Is set to logic 1 for low-speed device, or logic 0 for high-speed device
Paired (Bit 7 of byte 5)	This bit determines whether this PTD is a normal bulk PTD or a paired-PTD.
Ping-Pong (Bit 6 of byte 5)	This bit informs the Host Controller if this PTD is a PingPTD or a PongPTD. Ping and pong are two buffers in the buffering scheme.

9.3. Registers for Paired-PTD Mechanism Control

Table 9-3 provides the registers that are involved in the control of the ISP1362 paired-PTD buffering mechanism.

Table 9-3: Registers Related to the Paired-PTD Bulk Transfer

Register	Remarks
HcATLBufferLength	ATL buffer size
HcATLBufferPort	Data access
HcATLBlockSize	Size of each ATL block
HcATLPTDDoneMap	Bitmap to show the PTD that is done
HcATLPTDSkipMap	Bitmap to determine which PTD to skip
HcATLLastPTD	Bitmap to indicate the last valid PTD
HcATLCurrentActivePTD	Indicates the PTD that the Host Controller is processing
HcATLPTDDoneThresholdCount	Determines how many PTDs are processed per interrupt
HcATLPTDDoneThresholdTimeout	Determines the number of ms to retry if device NAKs
HcBufferStatus	**Bit 3 (ATL_Active)—**behaves like a switch to start and stop the Host Controller from processing the ATL buffer **Bit 4 (Reset_HWPingPongReg)—**allows the HCD to reset the PingPong toggling sequence, if required **Bit 10 (PairedPTDPingPong)—**indicates whether the Host Controller is processing the ping buffer or the pong buffer
HcμPInterrupt	**Bit 8 (ATL_IRQ)**—determines if an interrupt is to be generated

9.4. Done, Skip, Last

HcATLPTDDoneMap, HcATLPTDSkipMap and HcATLLastPTD are used with the blocked memory architecture for the bulk and interrupt transfers. These registers make the control and monitoring of PTDs much simpler and efficient.

SkipMap is used to individually enable or disable the PTDs in the ATL buffer. If the corresponding bit in SkipMap is skipped, the Host Controller will ignore this PTD and proceed to the next. For example, if the value 0x3301 (binary: 0011 0011 0000 0001) is written into HcATLPTDSkipMap, the 1st, 9th, 10th, 13th, 14th PTDs will be ignored.

HcATLLastPTD is used to notify the Host Controller about the location of the last valid PTD in the buffer. This increases the Host Controller processing speed because it does not have to check the whole buffer.

The HcATLPTDDone register is a 32-bit bitmap representation of the status of the ATL buffer blocks, in which each block may contain one PTD. For every ATL PTD done, the Host Controller sets the corresponding bit to logic 1. This block is disabled until HcATLPTDDone is read, and is therefore, cleared automatically by the Host Controller. If a bit in HcATLPTDDone is set to logic 1 and is not cleared by reading, the corresponding block will **not** be processed, even if it is not skipped and is set active.

9.5. Paired-PTD Buffering Mechanism

In the bulk transfer, the maximum packet size is 64 bytes as defined in the USB specification. The ISP1362 uses a single PTD to execute a transfer of up to 1023 bytes of data by splitting the data into chunks of 64 bytes and sending them out in sequence, all without the intervention of the Host Controller Driver. However, the ATL buffer is **not** double-buffered and there will be a gap between streams of bulk data because of the time required to refill the ATL buffer. Making the ATL buffer a double-buffered system will increase the hardware required and reduces memory usage efficiency because not all aperiodic transfers require high speed.

The ISP1362 strikes a balance between the two options by providing a mechanism to execute double buffering in a single buffer environment. This method uses two PTDs to serve the same endpoint alternatively. Therefore, one PTD can be refilled while the Host Controller is processing the other.

In the example in Table 9-4, the ISP1362 uses the paired-PTD mechanism to send out a stream of bulk data.

Table 9-4: Example of Register Values in a Bulk Transfer

Register	Remarks
HcATLBufferLength	1536 bytes
HcATLBufferPort	Not applicable
HcATLBlockSize	640 (for 10 x 64 bytes packet)
HcATLPTDDoneMap	For status monitoring
HcATLPTDSkipMap	0xFFFF FFFC (only the first two PTD are valid)
HcATLLastPTD	0x0000 0002 (The second PTD is the last PTD)
HcATLCurrentActivePTD	For status monitoring
HcATLPTDDoneThresholdCount	1 (generates an interrupt for every PTD done)
HcATLPTDDoneThresholdTimeout	3 (optional)
HcBufferStatus	For a block size of 640, the two PTDs must be copied into address of offset 0 and 648 from the ATL buffer starting address.
HcµPInterrupt	

A simplified flow chart of the paired-PTD mechanism is given in Figure 9-1.

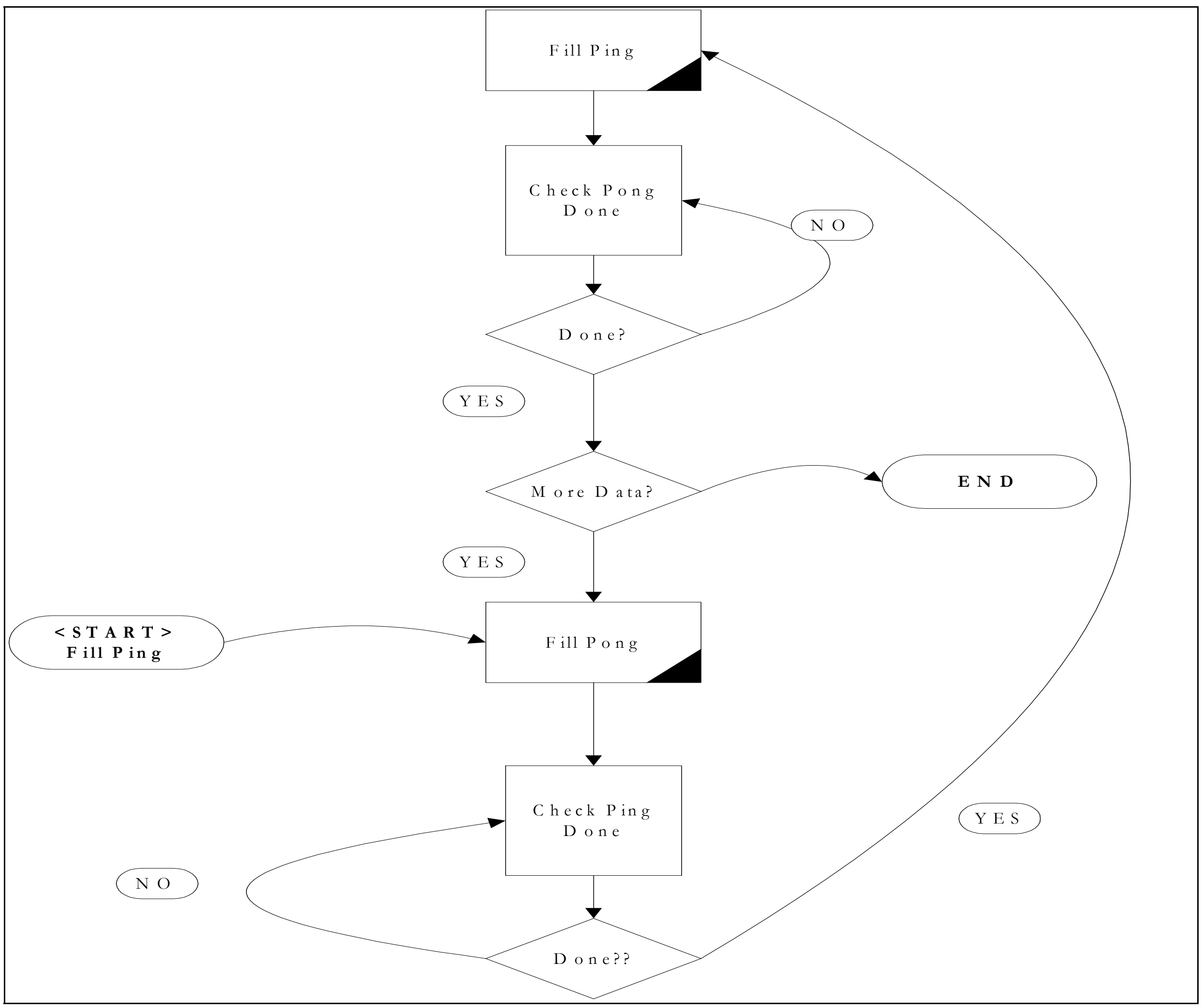

Figure 9-1: Paired-PTD Flowchart

10. Advanced Feature 3: Automatic Polling for the Interrupt Endpoint

In the USB protocol, interrupt endpoints are usually polled at a regular interval of time. The most commonly used interrupt device is the mouse. A mouse is usually polled every 8 ms for the movement data. It returns data if there has been any movements in the past 8 ms. If there has not been any movements, it NAKs. While not a major issue in most systems to handle this regular polling requirement, it will be helpful if the USB hardware takes over this responsibility. The ISP1362 uses a unique scheduling method to handle this regular polling requirement. The method will be described in details in this section.

10.1. Configuring the INTL Buffer

The ISP1362 has a total of 4096 bytes of buffer memory. In a typical operation scenario with a mixture of bulk, isochronous and interrupt traffic, it is recommended that you set the INTL buffer to 512 bytes.

The INTL buffer in the ISP1362 uses a blocked architecture. The entire INTL buffer area is separated into blocks of equal sizes. The size of the blocks can be determined by using HcINTLBlockSize. Note that value in HcINTLBlockSize does not include the 8 bytes taken by the PTD header. Therefore, a block of 64 bytes, for example, actually will take up 72 bytes (64 + 8) in the INTL buffer.

10.2. Interrupt PTD Format

Table 10-1: Interrupt PTD Structure: Bit Allocation

Bytes 1, 3, 5, 7								Bytes 0, 2, 4, 6							
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
CompletionCode				**Active**	**Toggle**	**ActualBytes**									
EndpointNumber				B5-3	**Spd**	**MaxPktSize**									
B5-7	B5-6	B5-5	B5-4	**DirToken**		**TotalBytes**									
Polling Rate			**Starting Frame**					R[1]	**FunctionAddress**						

[1] R—denotes reserved.

Table 10-2: Interrupt PTD Structure: Bit Description

Name	Description
ActualBytes[9:0]	Actual amount of data transferred at the moment
MaxPktSize[9:0]	Maximum amount of data per packet
TotalBytes[9:0]	Total amount of data to be transferred
CompletionCode[3:0]	Reports success or errors in a transaction
EndpointNumber[3:0]	Target endpoint number
DirToken[1:0]	Specifies IN, OUT or setup token
FunctionAddress[6:0]	Address of target device
Active	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.
Toggle	This bit is used to generate or compare the data PID value (DATA0 or DATA1) for IN and OUT transactions.
Speed	Is set to logic 1 for low-speed device, or logic 0 for high-speed device
Polling Rate Starting Frame	Special fields used to control the automatic polling. These fields will be explained in the next section.

10.3. Registers for the Interrupt Automatic Polling Control

The registers involved in the control of the ISP1362 interrupt automatic polling control are given in Table 10-3.

Table 10-3: Registers Related to the Interrupt Automatic Polling Control

Register	Remarks
HcINTLBufferSize	INTL buffer size
HcINTLBufferPort	Data access
HcINTLBlockSize	Size of each INTL block
HcINTLPTDDoneMap	Bitmap to show the PTD that is done
HcINTLPTDSkipMap	Bitmap to determine which PTD to skip
HcINTLLastPTD	Bitmap to indicate the last valid PTD
HcINTLCurrentActivePTD	Indicates the PTD that the Host Controller is processing
HcBufferStatus	**Bit 2 (INTL_Active)—**behaves like a switch to start or stop the Host Controller from processing the INTL buffer.
HcµPInterrupt	**Bit 7 (INT_IRQ)**—determines if an interrupt is to be generated

10.4. Done, Skip, Last

HcINTLPTDDoneMap, HcINTLPTDSkipMap, HcINTLLastPTD are used in conjunction with the blocked memory architecture for the bulk and interrupt transfers. These registers make the control and monitoring of the PTDs much simpler and efficient.

SkipMap is used to individually enable or disable the PTDs in the INTL buffer. If the corresponding bit in SkipMap is skipped, the Host Controller will ignore this PTD and proceed to the next. For example, if the value 0x3301 (binary: 0011 0011 0000 0001) is written into HcINTLPTDSkipMap, the 1^{st}, 9^{th}, 10^{th}, 13th, 14^{th} PTDs will be ignored.

HcINTLLastPTD is used to notify the Host Controller about the location of the last valid PTD in the buffer. This increases the Host Controller processing speed because it does not have to check the whole buffer.

The HcINTLPTDDone register is a 32-bit bitmap representation of the status of ATL buffer blocks. For every INTL PTD done, the Host Controller sets the corresponding bit to logic 1 and this block is disabled until HcATLPTDDone is read, and is therefore, cleared automatically by the Host Controller. If a bit in HcINTLPTDDone is set to logic 1 and is not cleared by reading, the corresponding block will **not** be processed, even if it is not skipped and is set active.

10.5. Interrupt Automatic Polling Control

In the interrupt PTD, there are two special fields to control the automatic polling mechanism: Starting Frame (SF) and Polling Rate (PR). The Host Controller sends out the PTD based on these two parameters, as well as the current frame number (Fm).

Algorithm:

The byte 7 of the interrupt PTD is used as the Reference Byte (RB). If the Polling Rate is N, the Host Controller compares the first N bits of RB and the first N bits of Fm. If the result is TRUE, the PTD is sent. If the result is FALSE, the PTD will be ignored until the next frame, in which the comparison will be done again.

For example, two interrupt PTDs are put into the interrupt buffer.

An example of the automatic polling scheduling is given in Table 10-4.

Table 10-4: Example of Automatic Polling Scheduling

Interrupt PTD	Polling Rate	Polling Interval	Starting Frame
1	2	4	6
2	2	4	5
3	4	16	6

The PTDs are copied into the Host Controller interrupt buffer at Fm = 4.

PTD 1 will be sent out at Fm = 10

PTD 2 will be sent out at Fm = 9

PTD 3 will be sent out at Fm = 22.

11. On-The-Go—HNP and SRP

11.1. Introduction

The ISP1362 is a single-chip On-The-Go (OTG) Controller when used in the OTG mode. It is designed to meet all the requirements defined in the *On-The-Go Supplement to the USB 2.0 Specification Rev. 1.0.* It supports Host Negotiation Protocol (HNP) and Session Request Protocol (SRP) for dual-role devices. This section describes how to implement HNP and SRP in software by using proper hardware resources support (OTG registers, interrupts, etc.).

11.2. OTG Registers

11.2.1. Register Sets

The ISP1362 defines a set of OTG related registers that allow you to implement HNP and SRP by means of software. Table 11-1 is a summary of OTG registers.

Table 11-1: OTG Registers

Register	Width	Description
OtgControl	16	Provides control to V_{BUS} driving and charging, data line pull-up and pull-down, SRP detection method, the Host Controller and Device Controller switching, etc.
OtgStatus	16	Provides status of the ID pin, V_{BUS} voltage levels, rmt_conn, etc.
OtgInterrupt	16	Provides interrupt on the OTG status change, bus events (suspend, resume, se0), SRP detection and OtgTimer timeout
OtgInterruptEnable	16	Provides interrupt enable and disable
OtgTimer	32	Provides 0.01 ms base programmable timer for use in the OTG state machine (timer range is 0.01 to 167772.15 ms)
OtgAltTimer	32	Provides 0.01 ms base hardware timer to measure the response time of remote device (timer range is 0.01 to 167772.15 ms)

11.2.2. Register Access

OTG registers use the same access method as that of Host Controller registers. For details, see Section 3.3.

11.3. Programming SRP

In the OTG system, only the A-device is allowed to drive V_{BUS}. To conserve power, the A-device can drop V_{BUS} when the bus is not in use. In this case, if the B-device wants to use the bus, it must initiate SRP to wake-up the A-device. Meanwhile, the A-device must be ready to detect and respond to the SRP event. With the help of SRP, any one of the two connected OTG devices can start the session when V_{BUS} is down.

11.3.1. B-Device Initiating SRP

The B-device initiates SRP by using dataline pulsing and V_{BUS} pulsing. When the ISP1362 is used as the B-device, the following steps are used to generate SRP:

1. Detect initial conditions (ID_REG, B_SESS_END and SE0_2MS (bits 0, 2 and 9) of the OtgStatus register (see Table 11-2) are logic 1).

Table 11-2: OtgStatus Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved						SE0_2MS	reserved
Reset	-	-	-	-	-	-	0	-
Access	-	-	-	-	-	-	R	-
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	reserved		RMT_CONN	B_SESS_VLD	A_SESS_VLD	B_SESS_END	A_V_{BUS}_VLD	ID_REG
Reset	-	-	0	0	0	1	0	1
Access	-	-	R	R	R	R	R	R

2. Start data line pulsing (set LOC_CONN (bit 4) of the OtgControl register to logic 1; see Table 11-3)

Table 11-3: OtgControl Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved				OTG_SE0_EN	A_SRP_DET_EN	A_SEL_SRP	SEL_HC_DC
Reset	-	-	-	-	0	0	0	1
Access	-	-	-	-	R/W	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	LOC_PULLDN_DM	LOC_PULLDN_DP	A_RDIS_LCON_EN	LOC_CONN	SEL_CP_EXT	DISCHRG_V_{BUS}	CHRG_V_{BUS}	DRV_V_{BUS}
Reset	1	1	0	0	0	0	0	0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

3. Wait for 5 to 10 ms (recommended 8 ms, you can use OtgTimer; see Table 11-4)

Table 11-4: OtgTimer Register: Bit Allocation

Bit	31	30	29	28	27	26	25	24
Symbol	START_TMR	reserved						
Reset	0	-	-	-	-	-	-	-
Access	R/W	-	-	-	-	-	-	-
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	TMR_INIT_VALUE[23:16]							
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	TMR_INIT_VALUE[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

4. Stop dataline pulsing (set LOC_CONN (bit 4) of the OtgControl register to logic 0; see Table 11-3)
5. Start V_{BUS} pulsing (set CHRG_V_{BUS} (bit 1) of the OtgControl register to logic 1; see Table 11-3)
6. Wait for 20 to 60 ms (recommended 30 ms, you can use OtgTimer; see Table 11-4)

7. Stop V_{BUS} pulsing (set CHRG_V_{BUS} (bit 1) of the OtgControl register to logic 0; see Table 11-3).
8. Discharge V_{BUS} for about 30 ms (using DISCHRG_V_{BUS} (bit 2) of the OtgControl register). This step is optional.

11.3.2. A-Device Detecting SRP

When the ISP1362 is used as the A-device, it can choose to detect either V_{BUS} pulsing SRP or dataline pulsing SRP. For V_{BUS} pulsing SRP, if voltage on V_{BUS} is more than VA_SESS_VLD, the a_srp_det bit will be set. For dataline pulsing SRP, if either the DP line or the DM line goes high, the a_srp_det bit will be set. In both cases, an interrupt will generate on INT1.

If the ISP1362 is in the idle state and does not want to respond to SRP, the SRP detection function can be disabled.

Steps for enabling the SRP detection by the V_{BUS} pulsing

1. Set A_SEL_SRP (bit 9) of the OtgControl register (see Table 11-3) to logic 0
2. Set A_SRP_DET_EN (bit 10) of the OtgControl register (see Table 11-3) to logic 1.

Steps for enabling the SRP detection by the dataline pulsing

1. Set A_SEL_SRP (bit 9) of the OtgControl register (see Table 11-3) to logic 1
2. Set A_SRP_DET_EN (bit 10) of the OtgControl register (see Table 11-3) to logic 1.

Steps for disabling the SRP detection

1. Set A_SRP_DET_EN (bit 10) of the OtgControl register (see Table 11-3) to logic 0.

11.4. Programming HNP State Machine

HNP allows two connected dual-role devices to exchange the host role back and forth without exchanging the two ends of the cable. The state diagram for dual-role device given in the OTG supplement offers one possible implementation of HNP. Not all of the state transitions are mandatory. Any other implementation that exhibits an equivalent behavior as observed at USB connector pins is considered as compliant to OTG specification. The ISP1362 allows software implementation of HNP, which gives the flexibility to meet different requirements from various applications. The ISP1362 OTG registers provide necessary inputs, outputs and timers for the HNP state machine.

11.4.1. HNP State Machine (OTG_FSM)

An example of the OTG_FSM is given in Section 11.5. This code is derived from the dual-role state diagram in the OTG supplement.

11.4.2. Procedures for Handling HNP

When there is an HNP event (OTG interrupt or application request), OTG_FSM is called. The result (id, current state, error codes) will pass to the application program.

For the purpose of illustration of HNP procedures, assume that there are two dual-role devices built with the ISP1362. The two devices are connected by a mini-A to mini-B cable. The application running on the device wants to send a file to the remote device, neglecting it is an A-device or a B-device. Both devices have the proper driver to support the file transfer.

Application initiating session on the A-device

Initially, V_{BUS} is off and both devices are in the Idle state. The application sends a bus_request to the ISP1362 OTG driver. The OTG driver calls OTG_FSM and finally, goes to the A_HOST state. The remote device goes to the B_PERIPHERAL state.

For A-device: *A_IDLE -> (application asserts bus_request) -> A_WAIT_VRISE -> A_WAIT_BCON -> A_HOST ->*

For B-device: *B_IDLE -> B_PERIPHERAL*

The application knows that it is in the A_HOST state. Therefore, it enumerates the B-device and starts to send the file. On completion, the application will de-assert the bus_request and the device will go back to the A_IDLE state.

For A-device: *A_HOST -> (application de-asserts bus_request) -> A_SUSPEND ->A_WAIT_VFALL -> A_IDLE*

For B-device: *B_PERIPHERAL -> B-IDLE*

Application initiating session on the B-device

Initially, V_{BUS} is off and both devices are in the Idle state. The application sends a bus_request to the ISP1362 OTG driver. The OTG driver calls OTG_FSM and goes to the B_PERIPHERAL state. The remote device goes to the A_HOST state.

For B-device: *B_IDLE -> (application asserts bus_request) -> B_SRP_INIT -> B_IDLE -> B_PERIPHERAL*

For A-device: *A_IDLE -> (detection SRP) -> A_WAIT_VRISE -> A_WAIT_BCON -> A_HOST ->*

The A-device enumerates the B-device, enables the HNP handoff by set_feature (b_hnp_en) and goes to the A-SUSPEND state. The B-device acknowledges and goes to the B_HOST state.

For B-device: *B_PERIPHERAL -> (b_hnp_en & bus_suspend) -> B_WAIT_ACON -> B_HOST*

For A-device: *A_HOST -> A_SUSPEND -> A_PERIPHERAL*

The application knows that it is in the B_HOST state. Therefore, it enumerates the A-device and starts to send the file. On completion, the application will de-assert the bus_request and the device will go back to the B_IDLE state.

For B-device: *B_HOST -> (application de-asserts bus_request) -> B_PERIPHERAL -> B_IDLE*

For A-device: *A_PERIPHERAL -> A_WAIT_VFALL -> A_IDLE*

[Alternatively, the A-device can transit by *A_PERIPHERAL -> A_WAIT_BCON -> A_HOST -> A_SUSPEND -> A_WAIT_VFALL -> A_IDLE*]

11.4.3. OTG Interrupt

The OTG interrupt is generated on the INT1 pin. It is shared with the Host Controller interrupt.

Enabling the OTG interrupt

The procedure to enable the OTG interrupts is as follows:

1. Set InterruptPinTrigger and InterruptOutputPolarity (bits 1 and 2) in the HcHardwareConfiguration register (see Table 5-5)
2. Program the OtgInterruptEnable register (see Table 11-5) depending on your application.

Table 11-5: OtgInterruptEnable Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved					OTG_TMR_IE	B_SE0_SRP_IE	A_SRP_DET_IE
Reset	-	-	-	-	-	0	0	0
Access	-	-	-	-	-	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	OTG_RESUME	OTG_SUSPND_IE	RMT_CONN_IE	B_SESS_VLD_IE	A_SESS_VLD_IE	B_SESS_END_IE	A_V_{BUS}_VLD_IE	ID_REG_IE
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

3. Set OTG_IRQ_InterruptEnable (bit 9) in the HcµPInterruptEnable register (see Table 11-6)

Table 11-6: HcµPInterruptEnable Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved						OTG_IRQ_ Interrupt Enable	ATL_IRQ_ Interrupt Enable
Reset	-	-	-	-	-	-	0	0
Access	-	-	-	-	-	-	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	INT_IRQ_ Interrupt Enable	ClkReady	HC Suspended Enable	OPR Interrupt Enable	EOT Interrupt Enable	ISTL_1 Interrupt Enable	ISTL_0 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

4. Set InterruptPinEnable (bit 0) in the HcHardwareConfiguration register (see Table 5-5).

Servicing the OTG interrupt

The interrupt service routine for INT1 will check if the interrupt is caused by an OTG event. The procedure is:

1. Hardware interrupt is generated on the INT1 pin
2. Read the HcµPInterrupt register (see Table 5-4). If the bit OTG_IRQ (bit 9) is logic 1, then
3. Read the OtgInterrupt register (see Table 11-7). If one of the bits ID_REG_C, A_V_{BUS}_VLD_C, B_SESS_END_C, A_SESS_VLD_C or B_SESS_VLD_C (bits 0 to 4) is set, then

Table 11-7: OtgInterrupt Register: Bit Allocation

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	reserved					OTG_TMR_IE	B_SE0_SRP_IE	A_SRP_DET_IE
Reset	-	-	-	-	-	0	0	0
Access	-	-	-	-	-	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	OTG_RESUME	OTG_SUSPND_IE	RMT_CONN_IE	B_SESS_VLD_IE	A_SESS_VLD_IE	B_SESS_END_IE	A_V_{BUS}_VLD_IE	ID_REG_IE
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

4. Read the OtgStatus register (see Table 11-2).

11.4.4. Using OtgTimer and OtgAltTimer

The ISP1362 OtgTimer and OtgAltTimer registers are used to program the on-chip timer. The timer resolution is 0.01 ms and the timer range is 0.01 to 167772.15 ms.

The OtgTimer register is used to program the timeout value for the HNP timers, such as TA_WAIT_VRISE, TA_WAIT_BCON, TA_BDIS_ACON, TB_ASE0_BRST and TB_SRP_FAIL. It can also be used to timer the pulse width of dataline pulsing and V_{BUS} pulsing SRP. The timer is started by software and can be stopped either by using software or when the timeout value is reached. If the timeout value is reached, the OTG_TMR_TMOUT bit in the OtgInterrupt register will be set and a hardware interrupt will be generated, if enabled.

The OtgAltTimer register is for debugging purposes. It can be started when the device transitions to a specific HNP state. It can be stopped by software or by any OTG related interrupt (such as, connect and disconnect).

11.4.5. Using Auto Connect

When the A-device is in the A_SUSPEND state and detects a disconnect event, the ISP1362 is required to enable its pull-up resistor on the DP line within 3 ms. Some systems may have problems to meet this requirement. To resolve this, the ISP1362 has a feature that allows automatic connection of the pull-up resistor on the DP line on detecting a remote disconnected event. Setting the A_RDIS_LCON_EN bit in the OtgControl register can enable this feature. Note that this bit can only be set when the device enters the A_SUSPEND state and **must** be cleared when the device leaves the A_PERIPHERAL state.

11.4.6. Using Auto Bus Reset

When the B-device is in the B_WAIT_ACON state and detects a connect event, the ISP1362 is required to send a bus reset (SE0) within 1 ms. Some systems may have problems to meet this requirement. To resolve this, the ISP1362 has a feature that allows automatic bus reset on detecting a connect event. This feature can be enabled by set the

OTG_SE0_EN bit of the OtgControl register. Note that this bit can only be set when the device enters the B_WAIT_ACON state and **must** be cleared after the device enters the B_HOST state.

11.4.7. Using OtgInterrupt to Wake-up the Chip

When the ISP1362 is in the Idle state (A_IDLE or B_IDLE), the chip can be put in power saving mode in which the Host Controller and the Device Controller are suspended and the PLL and oscillator are stopped. However, as a dual-role device, the ISP1362 is required to wake-up and respond to some bus events, such as ID change and SRP detection. To do this, the OtgInterruptEnable register must be programmed properly before the chip is put in the power save mode. Three interrupt events that are allowed to wake-up the chip are:

- ID_REG_C
- A_SRP_DET
- B_SESS_VLD_C.

11.5. OTG HNP State Machine Pseudo Code

11.5.1. Dual-Role A-Device State Machine

```
STATE a_idle
        Inputs
        id
                a_srp_det
                a_bus_req
                a_bus_drop/
                Outputs
                drv_vbus/
                loc_conn/
                loc_sof/
IF (a_srp_det | a_bus_req) & a_bus_drop/ THEN
                drv_vbus
                goto a_wait_vrise
        ELSE IF id THEN
                goto b_idle
        END IF
END a_idle

STATE a_wait_vrise
        Inputs
        Id
a_bus_drop
a_vbus_vld

        Timers
        a_wait_vrise_tmr
                Outputs
        drv_vbus
        loc_conn/
                loc_sof/
On Entry
        start a_wait_vrise_tmr
        IF id | a_bus_drop | a_vbus_vld | a_wait_vrise_tmout THEN
                goto a_wait_bcon
        END IF
END a_wait_vrise

STATE a_wait_bcon
                Inputs
                        id
                        a_bus_drop
                        a_vbus_vld/
                        b_conn
                Timers
                        a_wait_bcon_tmr
                Outputs
                        drv_vbus
                        loc_conn/
```

```
                        loc_sof/
                On Entry
                        start a_wait_bcon_tmr
                On Exit
                        stop a_wait_bcon_tmr
                IF id | bus_drop | a_wait_bcon_tmout THEN
                drv_vbus/
                goto a_wait_vfall
        ELSE IF b_conn THEN
                loc_sof
                goto a_host
        ELSE IF a_vbus_vld/ THEN
                drv_vbus/
                goto a_vbus_err
END IF
END a_wait_bcon

STATE a_host
                Inputs
                        id
                        a_bus_drop
                        b_conn/
                        a_bus_req/
                        a_suspend_req
                        a_vbus_vld/
                Outputs
                        drv_vbus
                        loc_conn/
                        loc_sof
        IF id | bus_drop | b_conn/ THEN
                loc_sof/
                goto a_wait_bcon
        ELSE IF a_bus_req/ | a_suspend_req THEN
                loc_sof/
                goto a_suspend
        ELSE IF a_vbus_vld/ THEN
                loc_sof/
                drv_vbus/
                goto a_vbus_err
END IF
END a_host

STATE a_suspend
                Inputs
                        id
                        a_bus_drop
                        b_conn/
                        a_bus_req
                        b_bus_resume
                        a_vbus_vld/
                Timers
                        a_aidl_bdis_tmr
                Outputs
                        drv_vbus
                        loc_conn/
                        loc_sof/
                On Entry:
                        start a_aidl_bdis_tmr
                On Exit
                        stop a_aidl_bdis_tmr

        IF id | bus_drop | a_aidl_bdis_tmout THEN
                drv_vbus/
                goto a_wait_vfall
        ELSE IF b_conn/ & a_set_b_hnp_en THEN
                loc_conn
                goto a_peripheral
        ELSE IF b_conn/ & a_set_b_hnp_en/ THEN
                goto a_wait_bcon
        ELSE IF a_bus_req | b_bus_resume THEN
                loc_sof
                goto a_host
        ELSE IF a_vbus_vld/ THEN
                drv_vbus/
                goto a_vbus_err
        END IF
END a_suspend
```

```
STATE a_peripheral
               Inputs
                      id
                      a_bus_drop
                      b_bus_suspend
                      a_vbus_vld/
               Outputs
                      drv_vbus
                      loc_conn
                      loc_sof/
        IF id | a_bus_drop THEN
drv_vbus/
        loc_conn/
               goto a_wait_vfall
        ELSE IF b_bus_suspend THEN
               loc_conn/
               goto a_wait_bcon
        ELSE IF a_vbus_vld/ THEN
               loc_conn/
               drv_vbus/
               goto a_vbus_err
        END IF
END a_peripheral

STATE a_vbus_err
               Inputs
                      id
                      a_bus_drop
               Outputs
        drv_vbus/
               loc_conn/
        loc_sof/
        IF id | a_bus_drop THEN
               goto a_wait_vfall
        END IF
END a_vbus_err

STATE a_wait_vfall
               Inputs
                      id
                      a_bus_req
                      a_sess_vld/
                      b_conn/
               Outputs
                      drv_vbus/
                      loc_conn/
                      loc_sof/
        IF id | a_bus_req | (a_sess_vld/ & b_conn/ THEN
               goto a_idle
        END IF
END a_wait_vfall
```

11.5.2. Dual-Role B-Device State Machine

```
STATE b_idle
        Inputs
               id/
               b_bus_req
               b_sess_end
               b_se0_srp
               b_sess_vld
        Outputs
               drv_vbus/
chrg_vbus/
               loc_conn/
               loc_sof/
        IF b_bus_req & b_sess_end & b_se0_srp THEN
               goto b_srp_init
        ELSE IF id/ THEN
               goto a_idle
        ELSE IF b_sess_vld THEN
               loc_conn
               goto b_peripheral
        END IF
END b_idle
```

```
STATE b_srp_init
        Inputs
                id/
                b_srp_done
        Outputs
                chrg_vbus (pulse)
                loc_conn (pulse)
                loc_sof/
        IF (id/ | b_srp_done) THEN
                chrg_vbus/
                loc_conn/
                goto b_idle
        END IF
END b_srp_init

STATE b_peripheral
        Inputs
                id/
                b_bus_req
                a_bus_suspend
                b_sess_vld/
        Outputs
                chrg_vbus/
                loc_conn/
                loc_sof/
        IF (id/ | b_sess_vld/) THEN
                loc_conn/
                goto b_idle
        ELSE IF (b_bus_req & a_bus_suspend & b_hnp_en) THEN
                loc_conn/
                goto b_wait_acon
        END IF
END b_peripheral

STATE b_wait_acon
        Inputs
                id/
                b_sess_vld/
                a_conn
                a_bus_resume
        Timers
                b_ase0_brst_tmr
        Outputs
                chrg_vbus/
                loc_conn/
                loc_sof/
        on entry
                start b_ase0_brst_tmr
        on exit
                stop b_ase0_brst_tmr
        IF (id/ | b_sess_vld/) THEN
                goto b_idle
        ELSE IF a_conn THEN
                loc_sof
                goto b_host
        ELSE IF b_ase0_brst_tmout | a_bus_resume THEN
                loc_conn
                goto b_peripheral
        END IF
END b_wait_acon

STATE b_host
        Inputs
                id/
                b_sess_vld/
                b_bus_req/
                a_conn/
        Outputs
                chrg_vbus/
                loc_conn/
                loc_sof
        IF (id/ | b_sess_vld/) THEN
                loc_sof/
                goto b_idle
        ELSE IF (b_bus_req/ | a_conn/) THEN
                loc_sof/
                loc_conn
                goto b_peripheral
```

```
        END IF
END b_host
```

11.6. Power Saving and Chip Wake-up

To save power when no session is active, the ISP1362 can be put in the power saving mode. In the power saving mode, the Host Controller and the Device Controller are suspended. The internal PLL and oscillator can be stopped. The charge-pump can be disabled and USB transceivers can be suspended. Typically, the power current can be reduced to below 100 μA when the whole chip is in the power saving-mode.

During a session, it is possible to suspend the Host Controller and the Device Controller individually. When the chip acts as a host, the Device Controller can be suspended. When the chip acts as a peripheral, the Host Controller can be suspended (if the port 2 is not in use).

This section will discuss suspend and wake-up issues when the ISP1362 is configured in the OTG mode. For the Host Controller only or the Device Controller mode, similar steps can be used.

11.6.1. Suspending the Host Controller

If the system does not need the USB host function (no session or no device connected or low power), you can suspend the Host Controller, i.e., put the Host Controller in the USB SUSPEND state.

The steps to suspend the Host Controller are:

1. Set the SuspendClkNotStop bit to logic 0 (bit 11 of the HcHardwareConfiguration register; see Table 5-5)
2. Enable OTG wake-up event (bits 0, 4, 8 of the OtgInterruptEnable register; see Table 11-5)
3. Enable interrupt on ClkReady, if you want to wake-up the chip after the clock is stopped (bit 6 of the HcμPInterruptEnable register; see Table 11-6)
4. Enable interrupt on the INT1 pin (bit 0 of the HcHardwareConfiguration register; see Table 5-5)
5. Set the Host Controller to the USB SUSPEND state (bits 7 and 6 of the HcControl register; see Table 5-1).

Note that steps 1 through 4 are application dependent. Some application may wish to keep the clock running while others may not want to respond to the OTG event when the ISP1362 is in the suspend state.

After performing step 5, the Host Controller will stop generating SOF immediately and go to suspend within 5 ms. The H_SUSPEND/$\overline{\text{H\_WAKEUP}}$ pin will go HIGH, indicating that the Host Controller is currently in the USB suspend state. If the Device Controller is also in suspend then the PLL and oscillator will stop running.

When the Host Controller is in the suspend state and the clock is stopped, the Host Controller and OTG registers are not accessible. If the clock is not stopped (Device Controller is not suspended or the SuspendClkNotStop bit is set to logic 1 in step 1), all the Host Controller and OTG registers can be accessed.

11.6.2. Suspending the Device Controller

When the Device Controller detects any of the following events, an interrupt will be generated indicating that the Device Controller will go to suspend.

- DP and DM have been idle for 3 ms
- V_{BUS} low (b_sess_vld bit in the OtgStatus register is logic 0)
- Device Controller is disconnected from DP and DM of the OTG port (i.e., the Host Controller is connected).

However, the hardware will not go to suspend automatically until software issues the GoSuspend command.

The steps to suspend the Device Controller are:

1. Detect the suspend interrupt (bit 2 of the DcInterrupt register; see Table 12-2)

2. Set the WKUPCS (0/1) and CLKRUN (0) bits properly (bits 3 and 12 of the DcHardwareConfiguration register; see Table 11-8).

Table 11-8: DcHardwareConfiguration Register: Bit Allocation

Bit	15	14	13	12	11	10	9	8
Symbol	reserved	EXTPUL	NOLAZY	CLKRUN	CKDIV[3:0]			
Reset	-	0	1	0	0	0	1	1
Access	-	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	DAKOLY	DRQPOL	DAKPOL	reserved	WKUPCS	reserved	INTLVL	INTPOL
Reset	0	1	0	0	0	1	0	0
Access	R/W	R/W	R/W	-	R/W	R/W	R/W	R/W

3. Write logic 1 followed by logic 0 to the GOSUSP bit of the DcMode register (bit 5 of the DcMode register; see Table 12-9).

After Device Controller has gone into suspend, none of the Device Controller registers can be accessed, irrespective of whether the clock is running or not. A wake-up event is expected before reading or writing to any Device Controller register.

11.6.3. Resuming the Host Controller

If the Host Controller is suspended and the clock is stopped, it can be resumed by any of the following ways:

- A low pulse on the H_SUSPEND/$\overline{\text{H\_WAKEUP}}$ pin
- A low pulse on the $\overline{\text{CS}}$ pin
- Remote wake-up (resume) signal on the USB bus, if the Host Controller is connected to the transceiver
- An OTG event (i.e., ID change, a_srp_det and/or b_sess_vld), if enabled.

The above resume event will trig the oscillator to start. After clock is stable, an interrupt will be generated on INT1 pin, indicating ClkReady. The software must write to set the Host Controller in USB OPERATION mode within 5ms, otherwise the oscillator will stop again.

11.6.4. Resuming the Device Controller

If the Device Controller is suspended and the clock is stopped, it can be resumed by any of the following ways:

- A low pulse on the D_SUSPEND/$\overline{\text{D\_WAKEUP}}$ pin
- A low pulse on the $\overline{\text{CS}}$ pin, if enabled
- A resume or SE0 signal on the USB bus, if the Device Controller is connected to the transceiver.

After resume, an Unlock command must be issued before any register read/write.

11.6.5. ISP1362 in Minimum Power Current State and Wake-Up Method

To put the whole chip in minimum power current state, all of the following must be done:

1. Make sure no session is running (i.e., the device is in the A-IDLE or B-IDLE state)
2. Disable the charge pump (bit 0 of the OtgControl register; see Table 11-3)

3. Disable the on-chip overcurrent (OC) detection module (bit 14 of the HcHardwareConfiguration register; see Table 5-5)
4. Suspend the Device Controller with clock stop option
5. Suspend the Host Controller with clock stop option.

The wake-up event can come from either hardware or software. The typical wake-up scenarios for an OTG dual-role device are discussed here.

1. In the A_IDLE or B_IDLE state, the application asserts bus_req, indicating it wants to use the USB bus.
    1. Software accesses any ISP1362 register (assert $\overline{CS}$)
    2. Wait for the ClkReady interrupt on the INT1 pin
    3. Get the ClkReady interrupt (typically within 1 ms from asserting $\overline{CS}$)
    4. Set the Host Controller to the USB OPERATION mode
    5. Go to A_WAIT_VRISE (for the A-Device) or B_SRP_INIT (for the B-Device) state.
2. In the A_IDLE state, the remote device initiates SRP.
    1. Get the ClkReady interrupt
    2. Set the Host Controller to the USB OPERATION mode
    3. Read the HcµPInterrupt and OtgInterrupt registers and get the A_SRP_DET bit set
    4. Go to the A_WAIT_VRISE state.
3. In the B_IDLE state, the remote device drives $V_{BUS.}$
    1. Get the ClkReady interrupt
    2. Set the Host Controller to the USB OPERATION mode
    3. Read the HcµPInterrupt and OtgInterrupt registers and get the B_SESS_VLD bit set
    4. Resume the Device Controller
    5. Go to the B_PERIPHERAL state.

12. Device Controller of the ISP1362

The Device Controller (DC) of the ISP1362 is a core based on Philips ISP1181 Device Controller, which is a full-speed USB interface device with up to 14 configurable endpoints. You can access the Device Controller of the ISP1362 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 the ISP1362 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 the ISP1362 is based on the firmware code of the ISP1362 ISA evaluation kit. The operating system used is DOS. Therefore, the Hardware Abstraction layer focuses on the ISA bus access.

12.1. *Firmware Structure of the Device Controller*

The firmware for the evaluation board consists of two major portions: the processing of information and the interrupt service routine. The Hardware Abstraction layer just moves data from hardware to memory space to be processed by the Main Loop as shown in Figure 12-1.

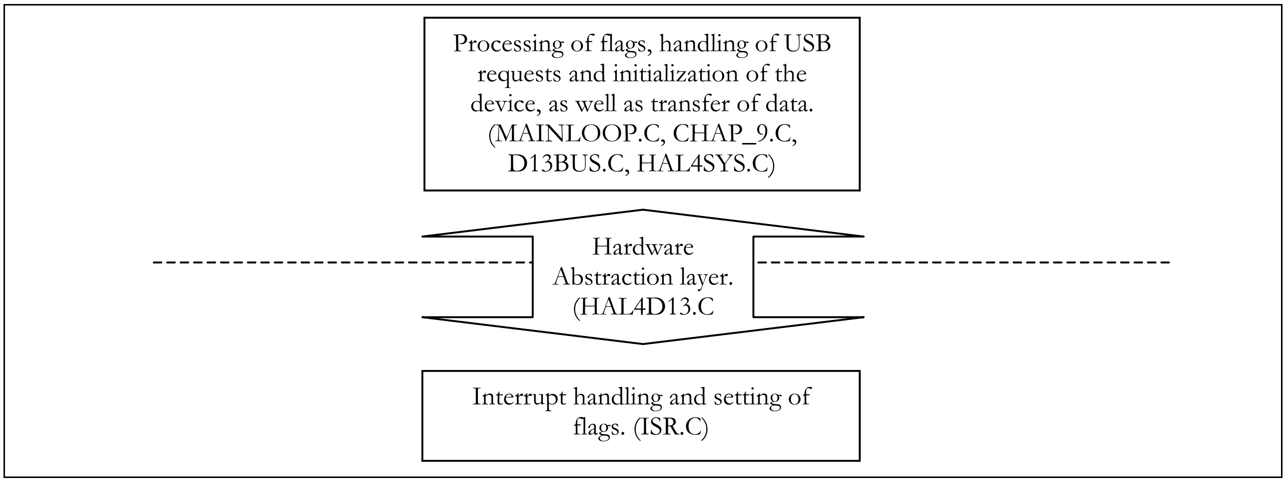

Figure 12-1: Firmware Structure of the Device Controller of the ISP1362

As can be seen in Figure 12-1, the firmware structure can be divided into the following six building blocks:

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

12.1.1. 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 the ISP1362, 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.

12.1.2. Hardware Abstraction Layer—HAL4D13.C

To further simplify programming with the Device Controller of the ISP1362, the firmware defines a set of command interfaces that encapsulate all the functions used to access the Device Controller of the ISP1362. When porting the firmware to other operation systems, this portion of the code must be modified.

12.1.3. Interrupt Service Routine—ISR.C

This part of the code handles interrupt generated by the Device Controller of the ISP1362. It retrieves data from the ISP1362 Device Controller's internal FIFO to CPU memory and sets up proper event flags to inform the Main Loop program to process.

ISP1362 Embedded Programming Guide Rev. 0.9

12.1.4. 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 details in Section 12.7.

12.1.5. Protocol Layer—D13BUS.C

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

12.1.6. Main Loop—MAINLOOP.C

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

12.2. Porting the Firmware to Other CPU Platform

Table 12-1 shows the modifications to building blocks that must be done. There are two levels of porting. The first level is the Standard Device Request, i.e. USB Chapter 9 only, which is just to make the firmware pass enumeration by supporting standard USB requests. The second level is the full product development. This involves product specific firmware code, i.e. Vendor Request.

Table 12-1: Building Blocks Modifications

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

12.3. Developing the Firmware in the Polling Mode

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

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

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

12.4. Hardware Abstraction Layer

12.4.1. Hardware Abstraction Layer for the System

This layer contains the lowest-layer functions that must be changed on different CPU platforms. The function prototypes present 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");
}
```

12.4.2. Hardware Abstraction Layer for the Device Controller of the ISP1362

The following functions are defined as the Device Controller command interface of the ISP1362 to simplify the device programming. These are implementations of the ISP1362 Device Controller command set, which is defined in the ISP1362 datasheet.

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

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

Hal4D13_SetMode(UCHAR bMode);
Hal4D13_GetMode(void);

Hal4D13_SetDevConfig(USHORT wDevCnfg);
Hal4D13_GetDevConfig(void);

Hal4D13_SetIntEnable(ULONG dIntEn);
Hal4D13_GetIntEnable(void);

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

Hal4D13_ResetDevice(void);

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

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

Hal4D13_AcknowledgeSETUP(void );

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

Hal4D13_ReadChipID(void);
Hal4D13_ReadCurrentFrameNumber(void);

Hal4D13_ReadInterruptRegister(void);
```

ISP1362 Embedded Programming Guide Rev. 0.9

12.5. Interrupt Service Routine

The Device Controller of the ISP1362 firmware is fully interrupt-driven. The flowchart of Interrupt Service Routine (ISR) is given in Figure 12-2.

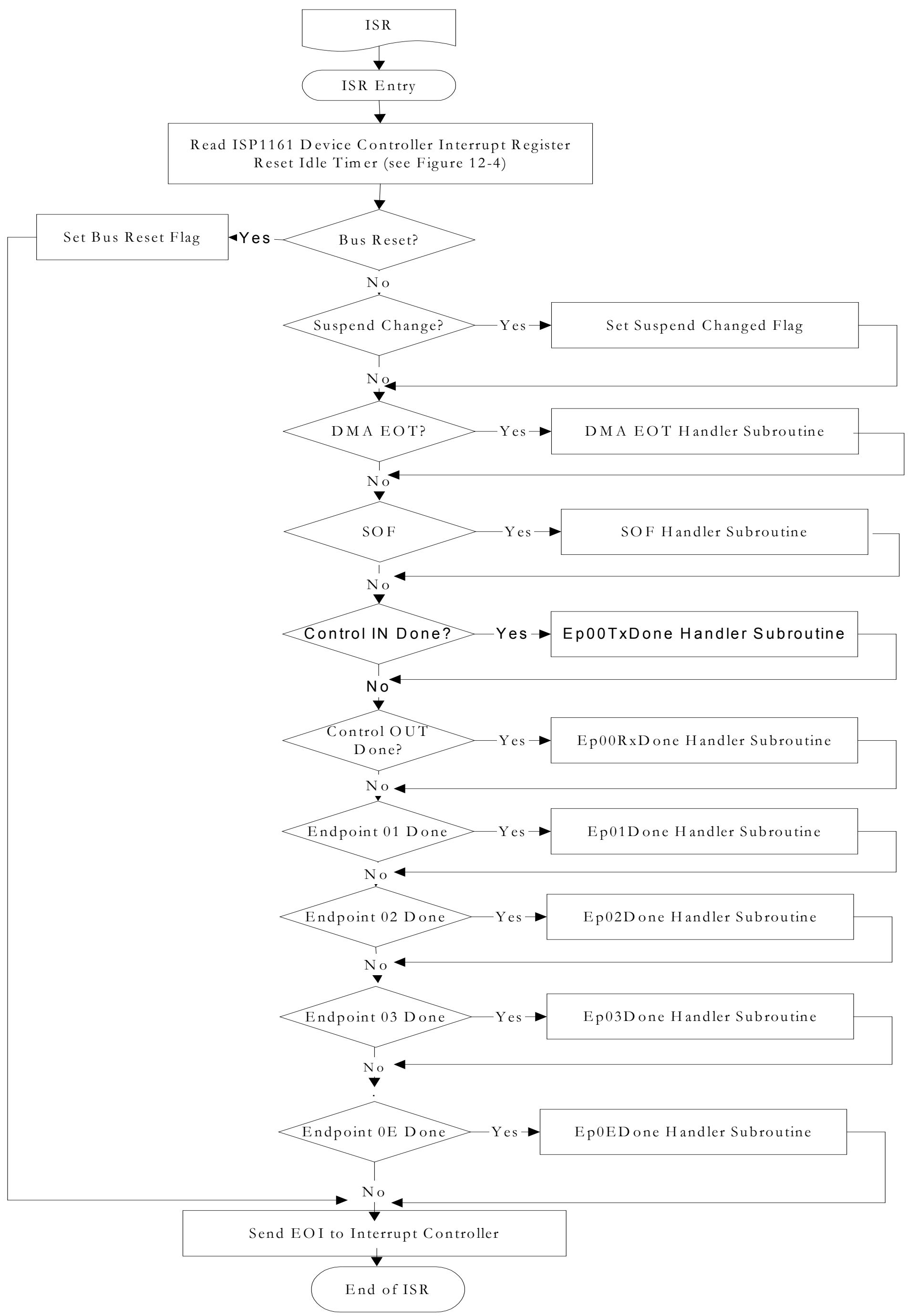

Figure 12-2: Flowchart of ISR

Table 12-2: DcInterrupt Register: Bit Allocation

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

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

Figure 12-3 contains the pseudo code of a typical Interrupt Service Routine.

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

        i_st = ReadInterruptRegister();  /* See Figure 12-4 on reading the Interrupt register */
        if(i_st != 0) {

                        if(i_st & D13REG_INTSRC_BUSRESET)
                                Isr_BusReset();

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

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

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

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

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

Figure 12-3: Code Example of a Typical ISR

A pseudo code to read the Interrupt register is given in Figure 12-4.

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

Figure 12-4: Code Example to Read the DcInterrupt Register

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

```
        typedef union _D13FLAGS
        {
                struct _D13FSM_FLAGS
                {

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

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

        } CONTROL_XFER, * PCONTROL_XFER;

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

Figure 12-5: Control Flags

The task splitting between ISR and the Main Loop is that ISR collects data from the internal buffer of the ISP1362 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.

12.5.1. Bus Reset

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

12.5.2. Suspend Change

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

12.5.3. EOT Handler

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

12.5.4. Control Endpoint Handler

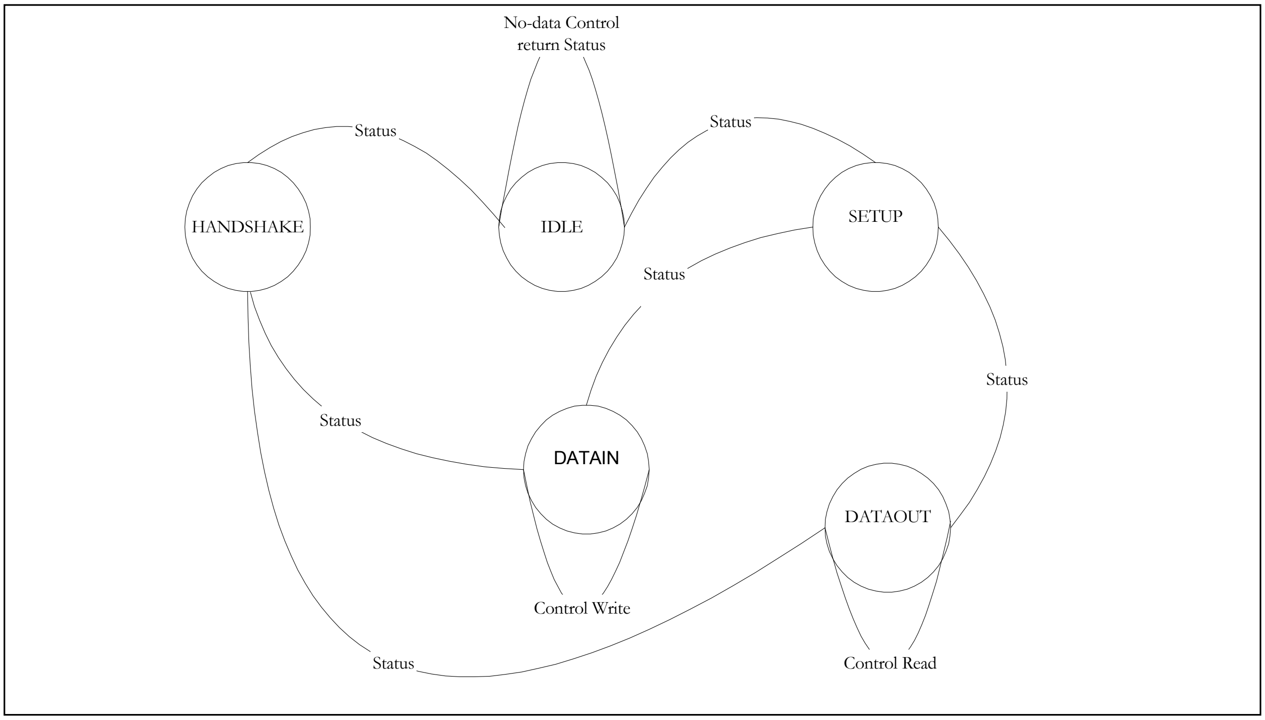

Figure 12-6: State Machine of the Control Transfer

The control transfer always begins with the Setup stage and is followed by an optional Data stage. The Data stage can be one or more IN or OUT transactions. Finally, it ends with the Status stage, i.e. HANDSHAKE. Figure 12-6 shows the various states of transitions on control endpoints. The firmware uses these five states to handle the control transfer correctly.

12.5.5. Control OUT Handler

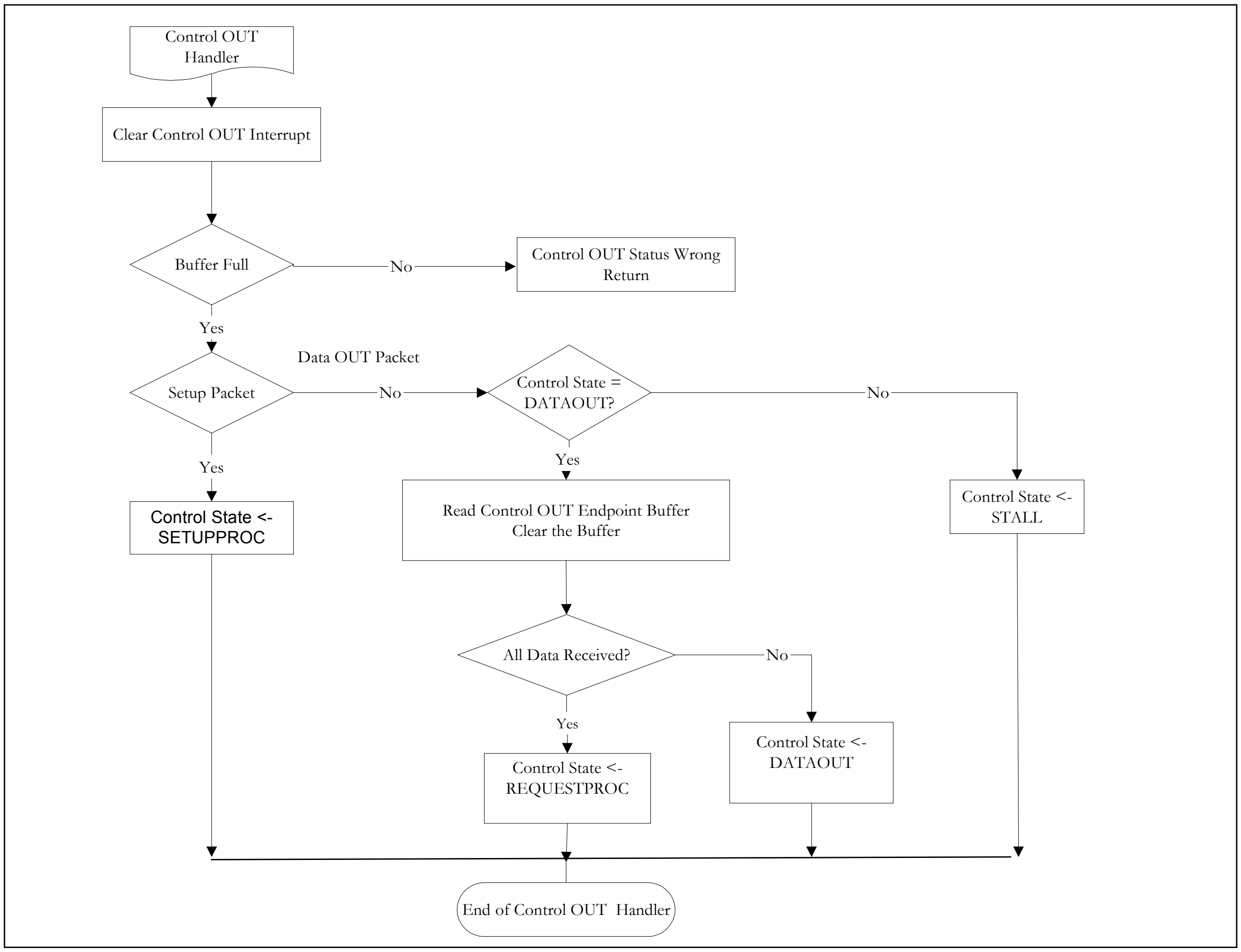

Figure 12-7: Flowchart of the Control OUT Handler

The microprocessor must clear the control OUT interrupt bit on the Device Controller of the ISP1362 and verify if this endpoint is full. Figure 12-8 contains a pseudo code to check if the OUT endpoint is full. This is done by issuing a Read Endpoint Status command (code 0x50) that clears the control OUT interrupt bit of the Interrupt register, and at the same time returns status information. Figure 12-9 shows a pseudo code to read the DcEndpointStatus register (see Table 12-3 and Table 12-4). This clears the corresponding endpoint interrupt. If the status information reports a Setup packet (SETUPT bit (bit 2) of the DcEndpointStatus register), the "SETUPPROC" state will be set for the Main Loop to process. Otherwise, the microprocessor extracts the content of the data OUT packet buffer by reading the control endpoint. Figure 12-10 contains a pseudo code to read the contents of an OUT buffer. After making sure all the data is received, the handler sets the Device Controller of the ISP1362 to the "REQUESTPROC" state.

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

Figure 12-8: Code Example for Checking Status of the OUT Endpoint

```
UCHAR Read_Endpoint_Status( UCHAR EPIndex)
```

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

Figure 12-9: Code Example for Reading the DcEndpointStatus Register

A typical pseudo code to read the contents of an OUT buffer is given in Figure 12-10.

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

Figure 12-10: Code Example for Reading the Contents of an OUT Buffer

Table 12-3: DcEndpointStatus 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 12-4: DcEndpointStatus Register: Bit Description

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

12.5.6. Control IN Handler

After the Setup stage is complete, the host executes the Data phase. If the Device Controller of the ISP1362 receives a control IN packet, it will go to the “control IN handler”. Again, the microprocessor must first clear the control IN interrupt bit of the ISP1362 Device Controller by reading its Read Endpoint Status code (Code 0x51). Figure 12-11 shows a pseudo code to read the DcEndpointStatus register. This clears the corresponding endpoint interrupt. Using the Endpoint status, it can determine whether the IN buffer is empty or full. Figure 12-12 contains a pseudo code to check whether the IN endpoint is empty or not. After verifying that the Device Controller of the ISP1362 is in the appropriate state, the microprocessor proceeds to send the data packet, see Figure 12-13.

Figure 12-14 shows the flowchart of the control IN handler. Since the Device Controller of the ISP1362 control endpoint has only 64 bytes FIFO, the microprocessor must control the amount of data during the transmission phase, if the requested length is more than 64 bytes. As indicated in the flowchart, the microprocessor must check its current and remaining data size to be sent to the host. If the remaining data size is greater than 64 bytes, the microprocessor will send the first 64 bytes and then subtract the reference length (requested length) by 64. When the next control IN token comes, the microprocessor determines whether the remaining byte is zero. If there is no more data to be sent, the microprocessor must send an empty packet to inform the host that there is no more data to be sent.

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

Figure 12-11: Code Example for Reading the DcEndpointStatus Register

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

Figure 12-12: Code Example for Checking the Status of the IN Endpoint

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

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

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

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

return j;
}
```

Figure 12-13: Code Example for Writing the Contents to an IN Buffer

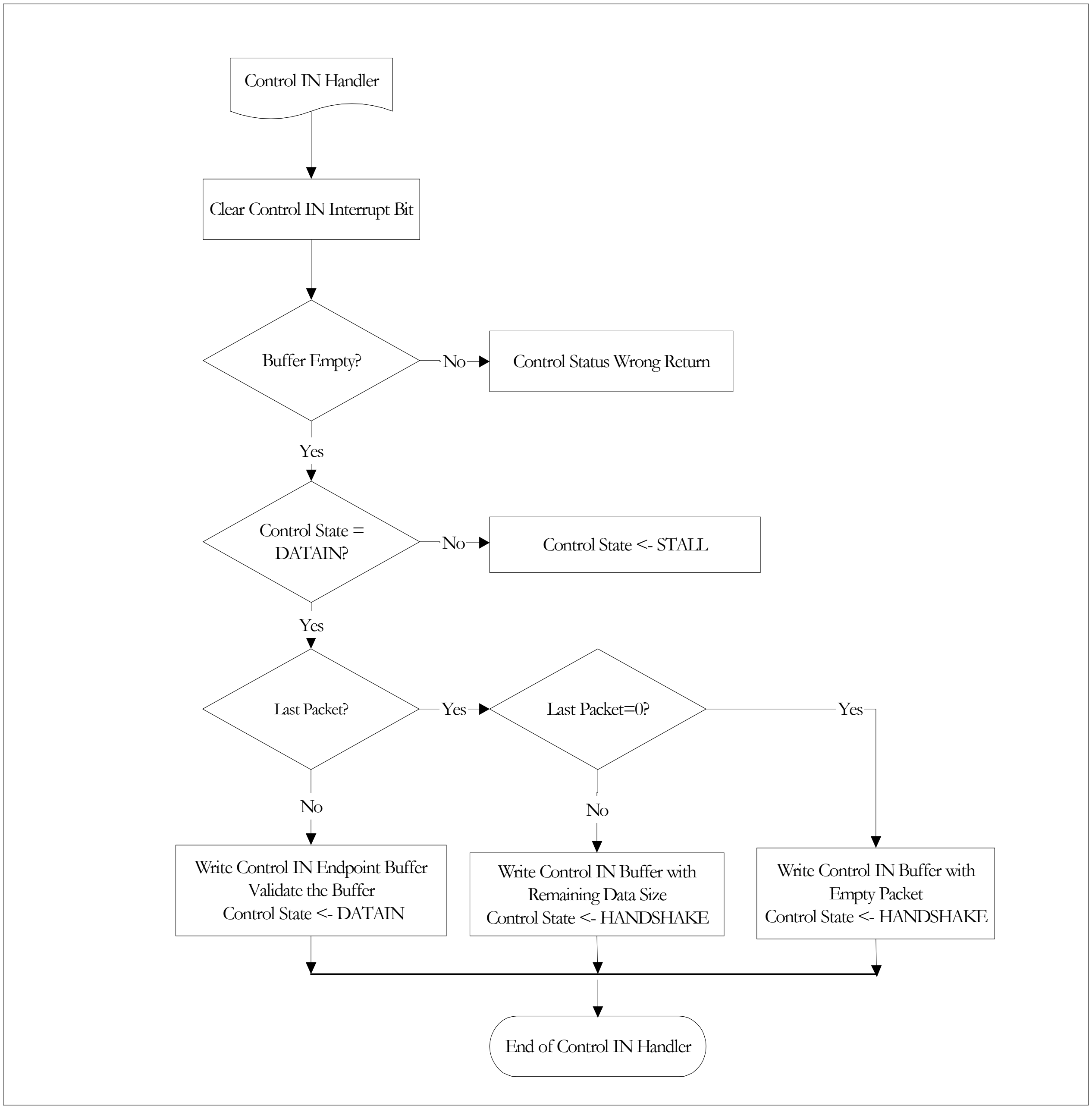

Figure 12-14: Flowchart of the Control IN Handler

Note: OUT data transactions and IN data transactions are slightly different 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.

12.5.7. Bulk Endpoint Handler

The Device Controller of the ISP1362 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 12-5 shows the recommended register programming of the DcEndpointConfiguration register for a bulk endpoint. The bit allocation and bit description of the DcEndpointConfiguration register are given in Table 12-6 and Table 12-7, respectively.

Table 12-5: Recommended DcEndpointConfiguration 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 12-6: DcEndpointConfiguration 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 12-7: DcEndpointConfiguration Register: Bit Description

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

An example on how to configure a bulk OUT or bulk IN endpoint is given in Figure 12-15.

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

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

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

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

Figure 12-15: Code Example for Configuring a Bulk OUT or Bulk IN Endpoint

The function definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is given in Figure 12-16.

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

Figure 12-16: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)

When the host is ready to transmit the bulk data, it issues an OUT token packet followed by a data packet. The Device Controller of the ISP1362 generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1362 Device Controller and verify the data length. The flowchart of the bulk OUT handler is given in Figure 12-17.

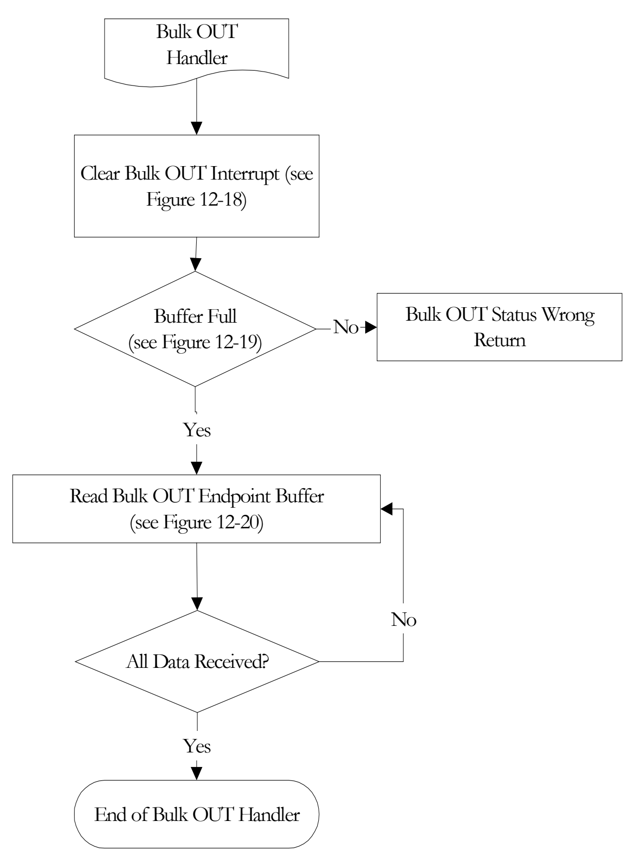

Figure 12-17: Flowchart of the Bulk OUT Handler

Figure 12-18 shows the code example for reading the DcEndpointStatus register. This clears the corresponding endpoint interrupt.

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

Figure 12-18: Code Example for Reading the DcEndpointStatus 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 if the primary buffer is full */
{
        /* Proceed with the program flow */
}
```

Figure 12-19: Code Example for Checking the Status of the Bulk OUT Endpoint

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

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

Figure 12-20: Code Example for Reading the Contents of a Bulk OUT Buffer

When the host is ready to receive the bulk data, it issues an IN token. The Device Controller of the ISP1362 generates an interrupt to inform the microprocessor. The microprocessor must clear the interrupt bit of the ISP1362 Device Controller and return the data packet to be sent. The flowchart of the bulk IN handler is given in Figure 12-21.

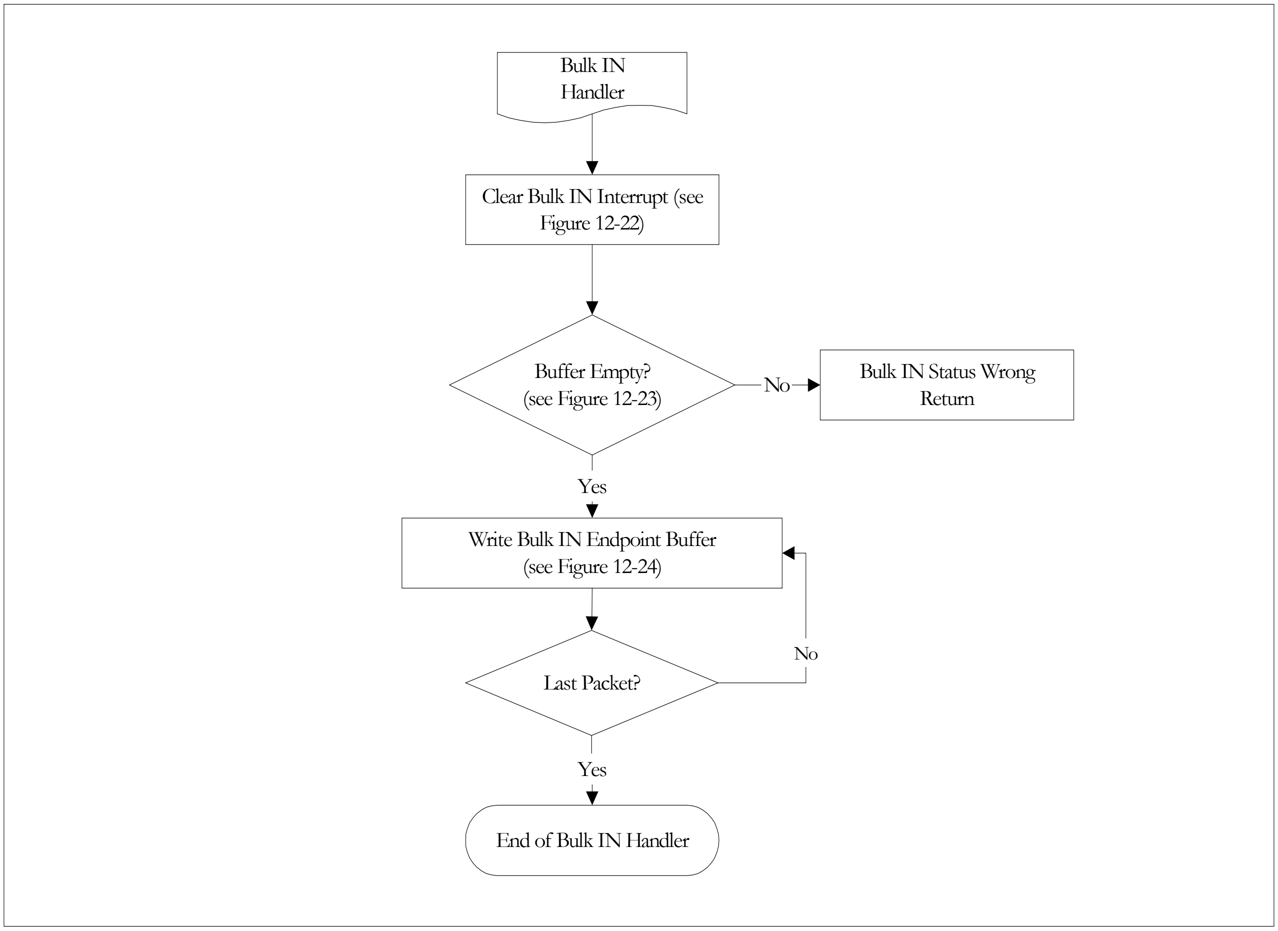

Figure 12-21: Flowchart of the Bulk IN Handler

A pseudo code for reading the DcEndpointStatus register is given in Figure 12-22. This clears the corresponding endpoint interrupts.

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

Figure 12-22: Code Example for Reading the DcEndpointStatus Register

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

Figure 12-23: Code Example for Checking the Status of the Bulk IN Endpoint

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

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

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

        return j;
}
```

Figure 12-24: Code Example for Writing the Contents into a Bulk IN Buffer

12.5.8. ISO Endpoint Handler

Table 12-8 contains the recommended register programming in the DcEndpointConfiguration register for an ISO endpoint.

Table 12-8: Recommended DcEndpointConfiguration Register Programming for an ISO Endpoint

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

Figure 12-25 contains an example on how to configure an ISO OUT or ISO IN endpoint.

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

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

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

Figure 12-25: Code Example for Configuring an ISO OUT or ISO IN Endpoint

The function definition of SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex) is given in Figure 12-26.

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

Figure 12-26: Function Definition of void SetEndpointConfig(UCHAR bEPConfig, UCHAR bEPIndex)

Flowcharts of the ISO OUT handler and the ISO IN handler are given in Figure 12-27 and Figure 12-28, respectively.

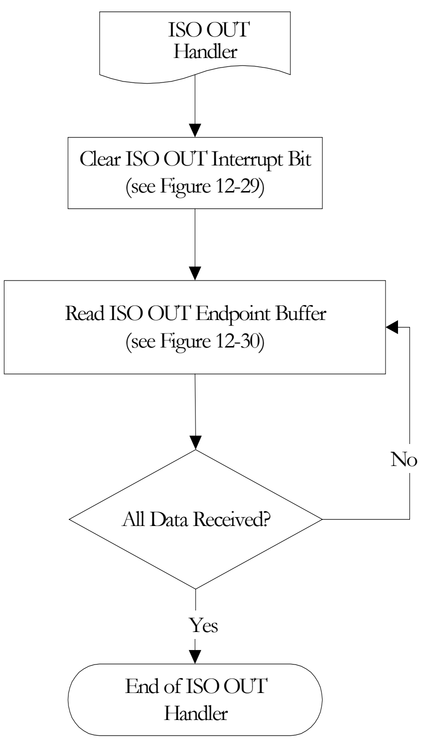

Figure 12-27: Flowchart of the ISO OUT Handler

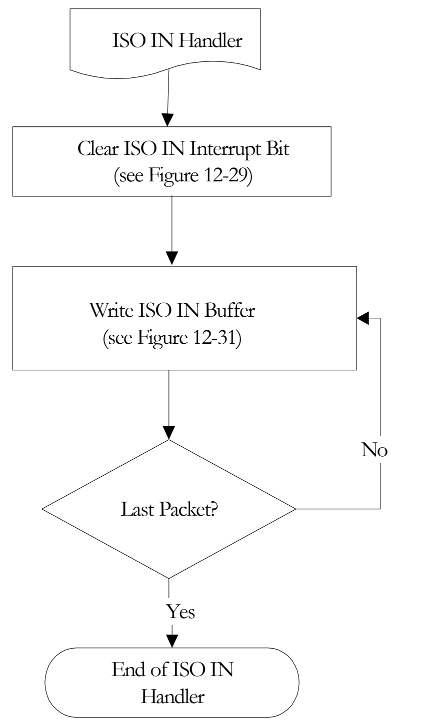

Figure 12-28: Flowchart of the ISO IN Handler

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

A pseudo code for reading the DcEndpointStatus register is given in Figure 12-29. This clears the corresponding endpoint interrupts.

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

Figure 12-29: Code Example for Reading the DcEndpointStatus Register

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

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

        if(j != len)
        j = len;

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

        /* Clear the buffer */
```

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

Figure 12-30: Code Example for Reading from an ISO Endpoint Buffer

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

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

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

Figure 12-31: Code Example for Writing to an ISO Endpoint Buffer

12.6. Main Loop

Upon powered 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 DcMode register to ON. This procedure is important because it ensures that the ISP1362 Device Controller will not operate before the microprocessor is ready to serve the ISP1362 Device Controller.

The flowchart of the Main Loop is given in Figure 12-32. In the Main Loop routine, the microprocessor polls for any activity on the keyboard. If any of the specific keys is pressed, the handle key commands will execute the routine and then return to the Main Loop. This routine is added for debugging purposes only. A 1 ms timer is programmed to activate the routine to check for any key pressed on the evaluation board.

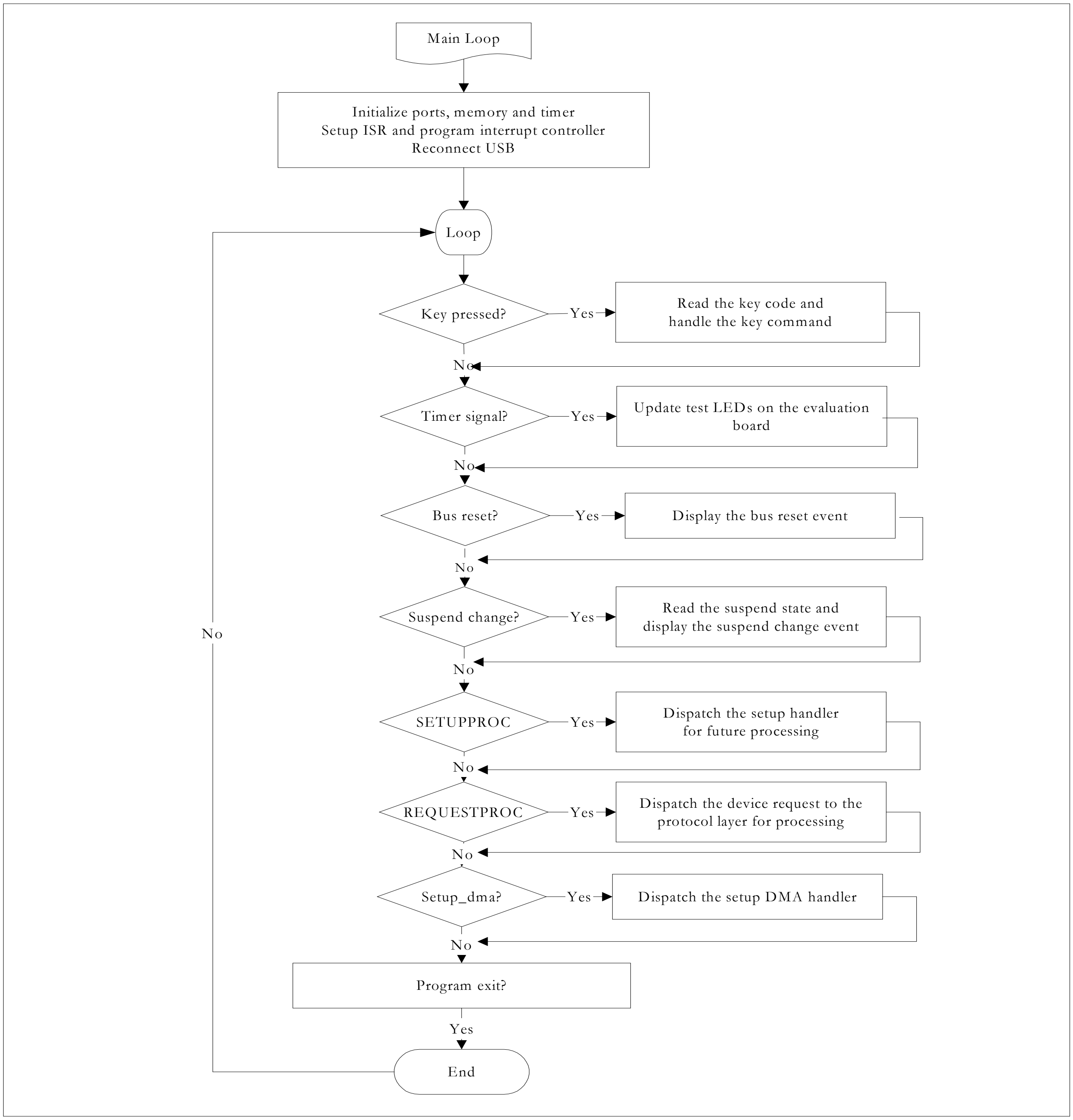

Figure 12-32: Flowchart of the Main Loop

Table 12-9: DcMode Register: Bit Allocation

Bit	7	6	5	4	3	2	1	0
Symbol	reserved		GOSUSP	reserved	INTENA	DBGMOD	reserved	SOFTCT
Reset	1[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 12-10: DcMode Register: Bit Description

Bit	Symbol	Description
7 to 6	-	reserved
5	GOSUSP	Writing a logic 1 followed by a logic 0 will activate the 'suspend' mode.
4	-	reserved
3	INTENA	A logic 1 enables all interrupts. Bus reset value: unchanged.
2	DBGMOD	A logic 1 enables debug mode where all NAKs and errors will generate an interrupt. A logic 0 selects normal operation, where interrupts are generated on every ACK (bulk endpoints) or after every data transfer (isochronous endpoints). Bus reset value: unchanged.
1	-	reserved
0	SOFTCT	A logic 1 enables SoftConnect. This bit is ignored if EXTPUL = 1 in the DcHardwareConfiguration Register (see Table 114). Bus reset value: unchanged. **Remark:** In the OTG mode, this bit is ignored. The LOC_CONN bit of the OtgControl register controls the pull-up resistor on the OTG_DP1 pin.

Figure 12-33 shows a pseudo code for writing to the DcMode register. An example on setting the SOFCT bit to enable SoftConnect is given in Figure 12-34.

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

Figure 12-33: Code Example for Writing to the DcMode Register

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

Figure 12-34: Code Example on Setting SoftConnect

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

12.7. 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 the USB specification rev. 2.0.

12.7.1. Clear Feature Request

In the Clear Feature request, the microprocessor must clear or disable a specific feature of the device based on the three states. The flowchart of Clear Feature is given in Figure 12-35. In this case, the microprocessor determines whether the request is meant for the device, interface or endpoints. There will not be any support if the recipient is an interface. Feature selectors are used when enabling or setting features specific to the device or endpoint, such as remote wake-up. If the recipient is a device, the microprocessor must disable the remote wake-up function, if this function is enabled. If the recipient is an endpoint, the microprocessor must unstall the specific endpoint through the Write Endpoint Status command.

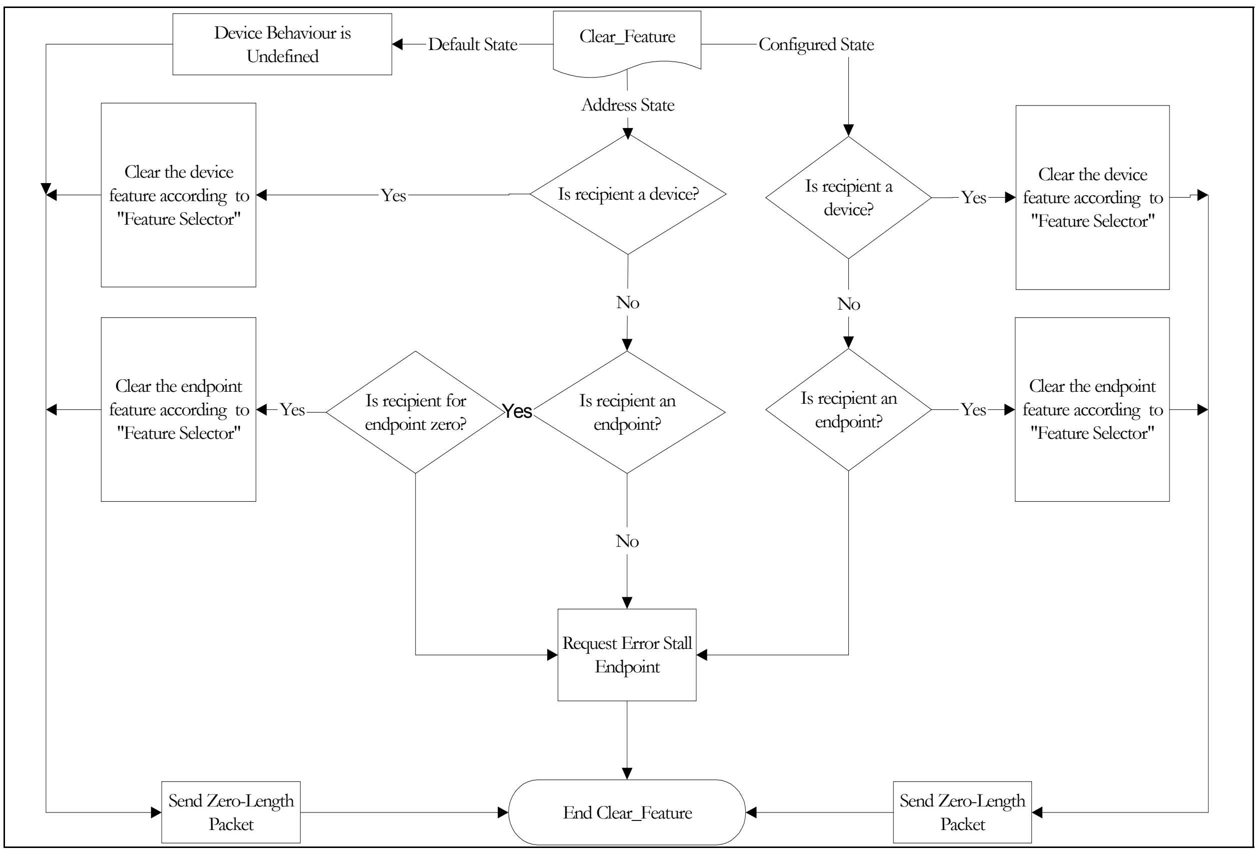

Figure 12-35: Flowchart of Clear Feature

Zero-Length Packet

A zero-length packet is a data packet with data length as zero. It is not the same as placing a 0x00 in the buffer and sending it out because this means a data length of 1 and a payload of 0x00. As can be seen in the pseudo code in Figure 12-13, sending a zero-length packet can be easily done by calling the Write_Endpoint() function with the following arguments in it.

```
// This function call will send a zero-length packet to the host through the control IN endpoint.
Write_Endpoint (1 ,0 ,0)  // See Figure 12-13
```

Figure 12-36: Code Example for Sending Zero-Length Packet

Request Error

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

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

Figure 12-37: Code Example to Stall or Unstall an Endpoint

12.7.2. Get Status Request

In the Get Status request, the microprocessor must return the status of the specific recipient based on the state of the device. The microprocessor must also determine the recipient of the request. If the request is to a device, the microprocessor must return the status of the device to the host, depending on the states. For a system having remote wake-up and self-powering capabilities, the returning data is 0x0003. Figure 12-38 shows the Get Status flowchart.

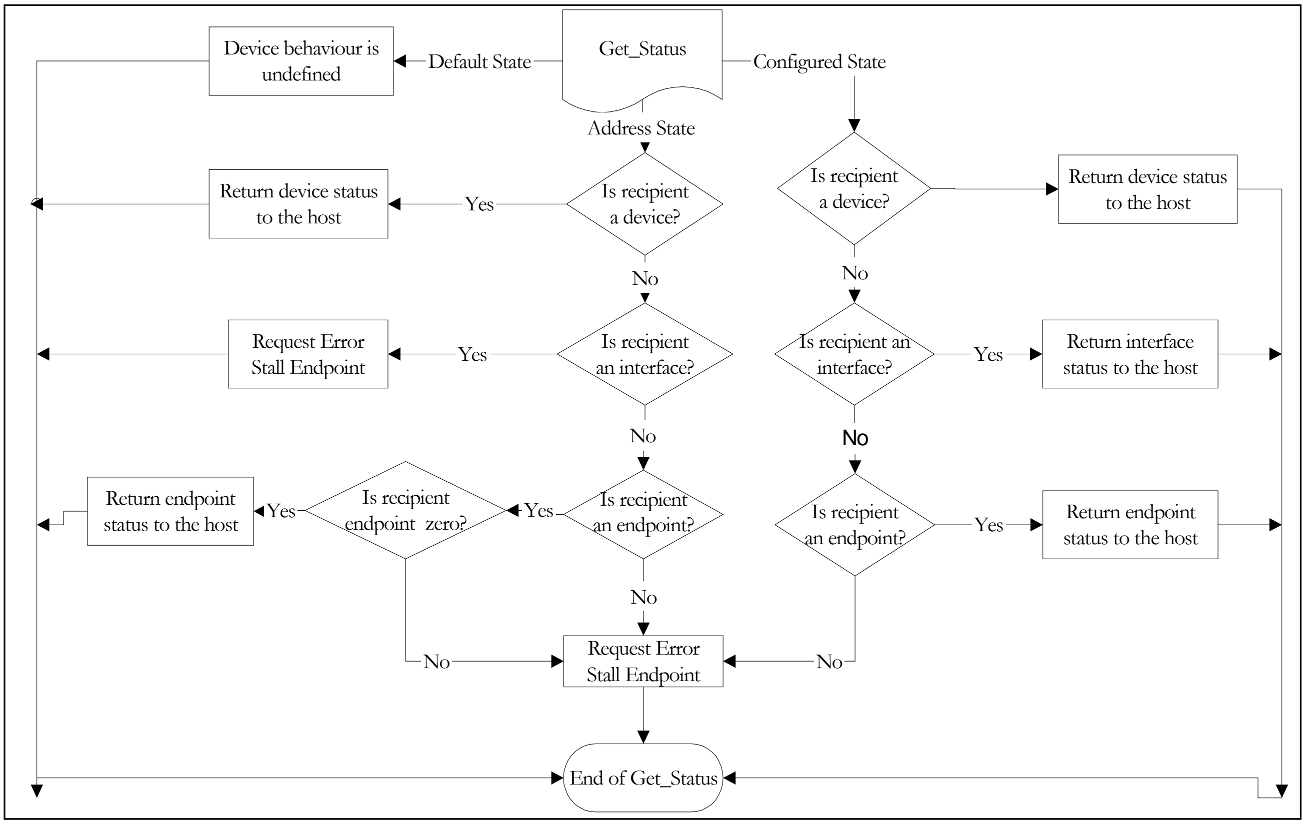

Figure 12-38: Flowchart of Get Status

12.7.3. Set Address Request

In the Set Address request (see Figure 12-39), the device gets the new address from the content of the Setup packet. Note that this Set Address request does not have a Data phase. Therefore, the microprocessor must write a zero-length data packet to the host at the acknowledgment phase.

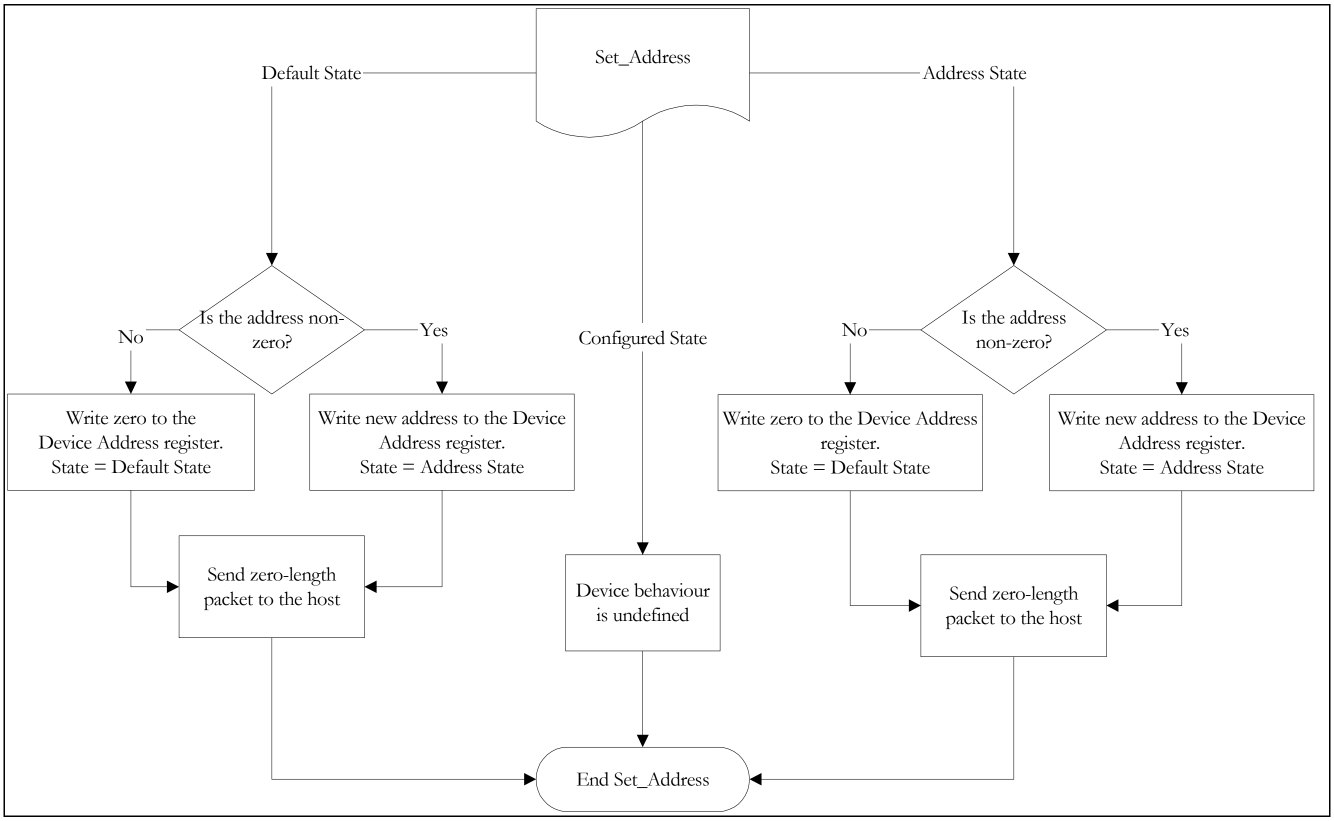

Figure 12-39: Flowchart of Set Address

Figure 12-40 shows a pseudo code of the Set Address routine.

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

Figure 12-40: Code Example of the Set Address Routine

Table 12-11: DcAddress 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 12-12: DcAddress Register: Bit Description

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

12.7.4. Get Configuration Request

In the Get Configuration request (see the flowchart in Figure 12-41), the microprocessor must return the current configuration value. The microprocessor first determines what state the device is in. Depending on the state, the microprocessor will either send a zero or the current non-zero configuration value back to the host.

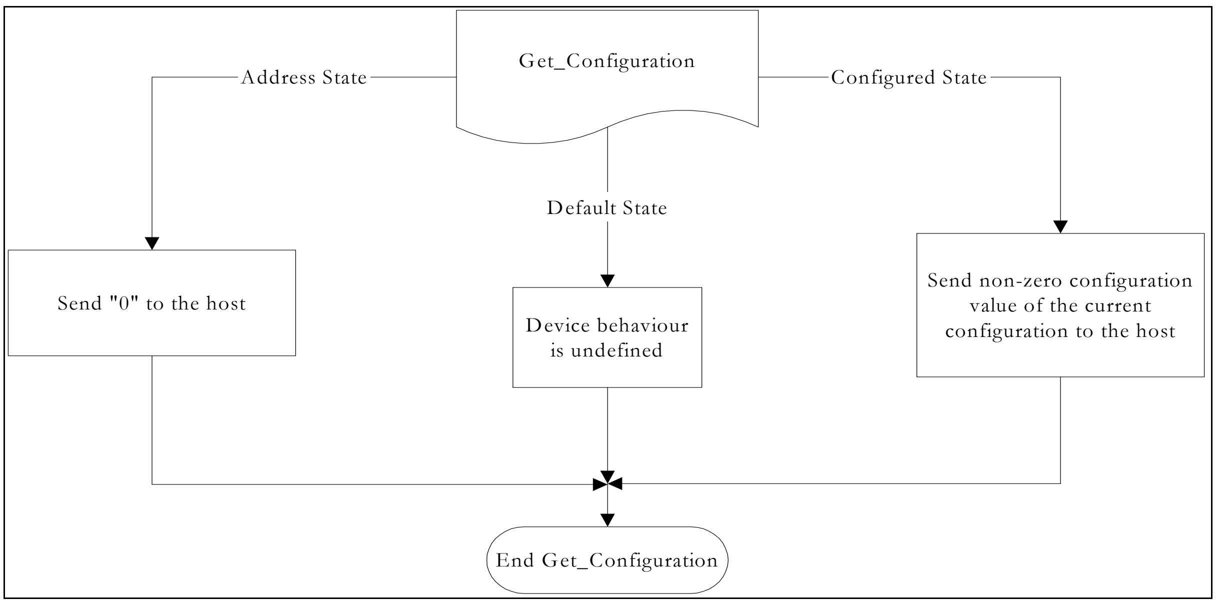

Figure 12-41: Flowchart of Get Configuration

12.7.5. Get Descriptor Request

For the Get Descriptor request, the microprocessor must return the specific descriptor, if the descriptor exists. First, the microprocessor determines whether the descriptor type request is for a device or configuration. It then sends the first 64 bytes of the device descriptor, if the descriptor type is for a device. The reason for controlling the size of returning bytes is that the control buffer has only 64 bytes of memory. The microprocessor must set a register to indicate the location of the transmitted size. The Get Descriptor request is a valid request for Default State, Address State and Configured State. Figure 12-42 shows the flowchart of Get Descriptor.

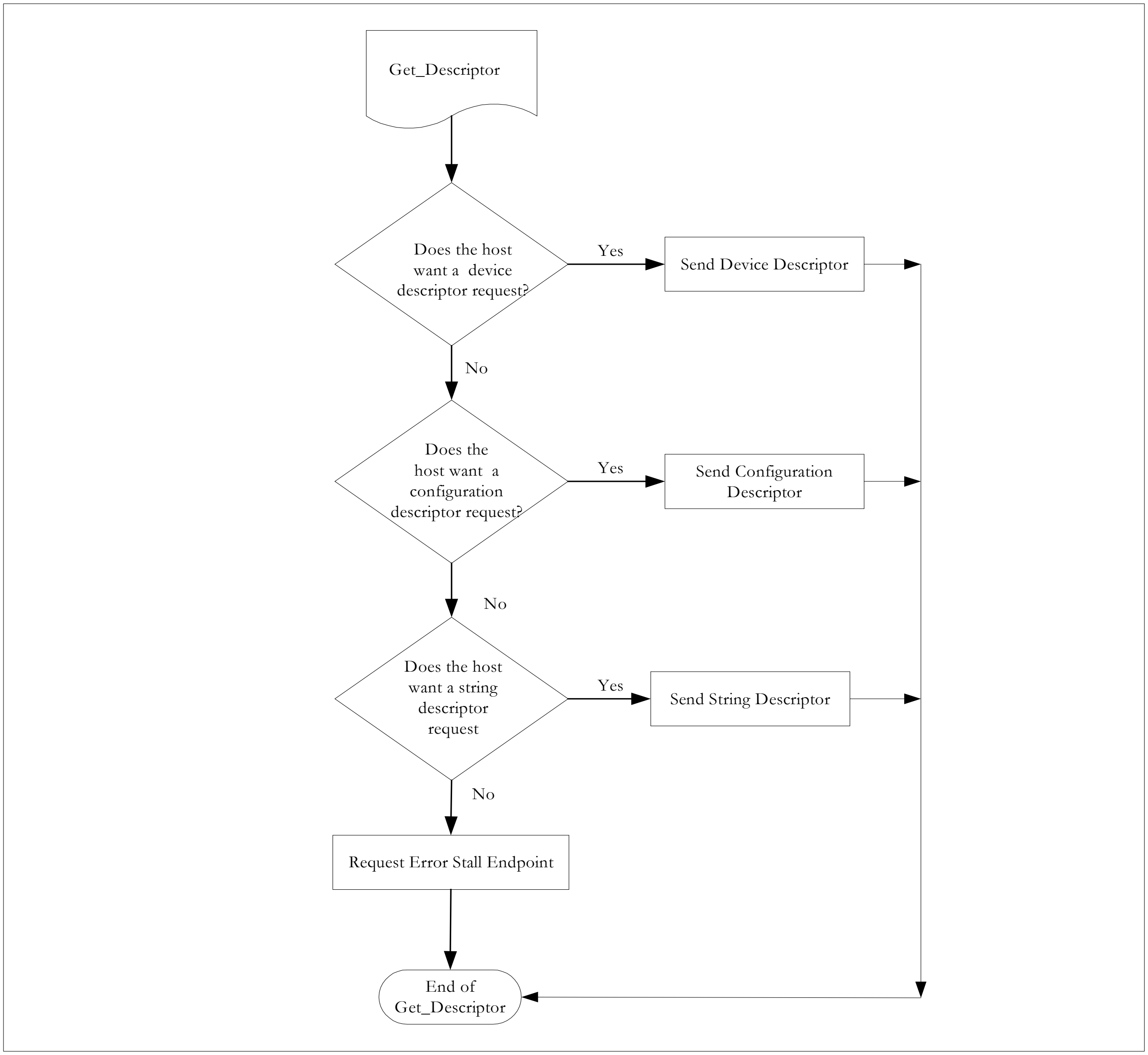

Figure 12-42: Flowchart of Get Descriptor

12.7.6. Set Configuration Request

For the Set Configuration request (see Figure 12-44), the microprocessor determines the configuration value from the Setup packet. If the value is zero, the microprocessor must clear the configuration flag in its memory and disable the endpoint. If the value is one, the microprocessor must set the configuration flag. Once the flag is set, the microprocessor must also send the zero-data packet to the host at the acknowledgment phase.

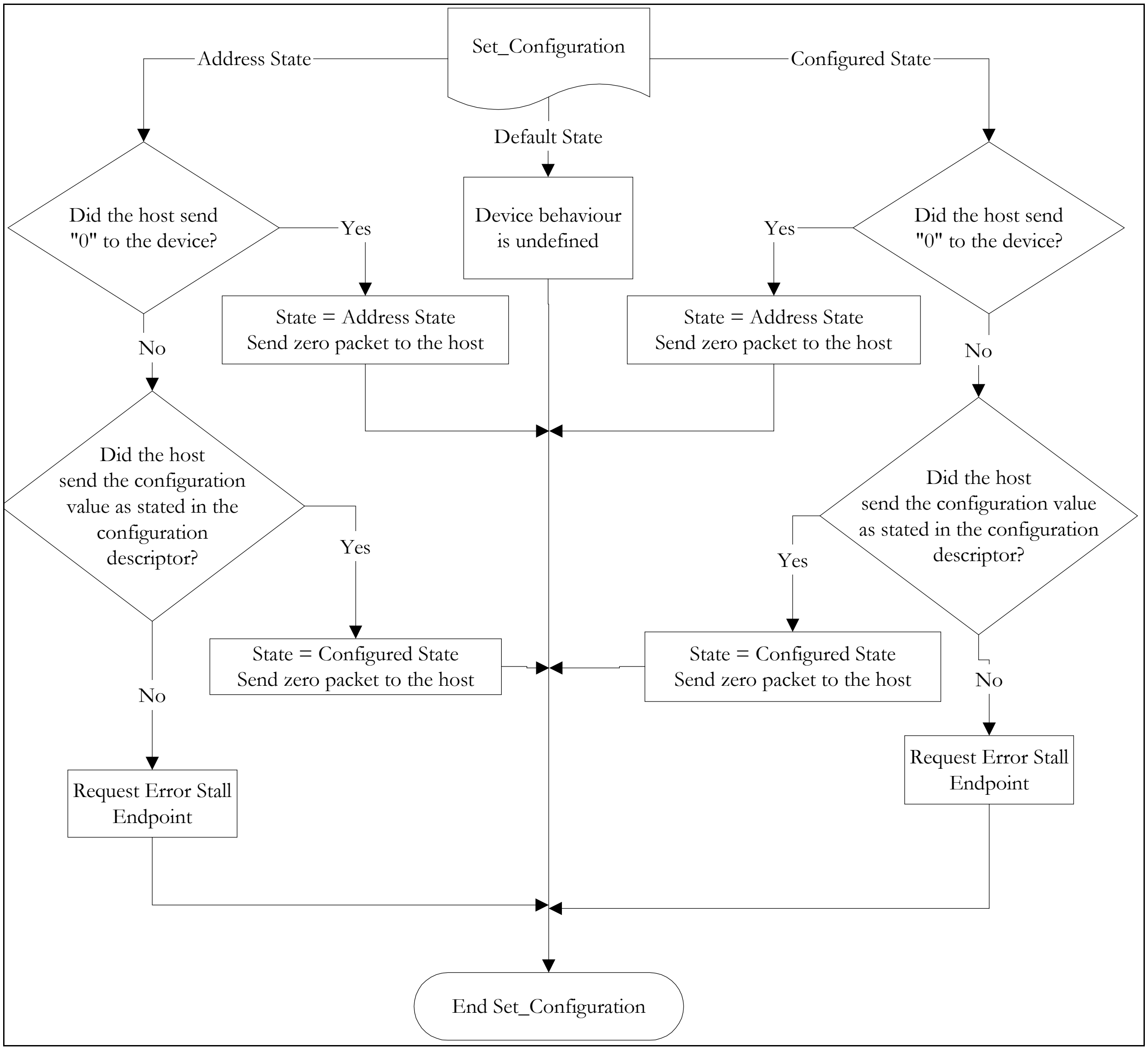

Figure 12-43: Flowchart of Set Configuration

12.7.7. Get and Set Interface Requests

For the Get and Set Interface requests (see flowcharts in Figure 12-44 and Figure 12-45), the microprocessor just needs to send one zero-data packet to the host because the Philips evaluation board only supports one type of interface. For the Set Interface request on the Philips evaluation board, the microprocessor need not do anything except to send one zero data packet to the host as the acknowledgment phase.

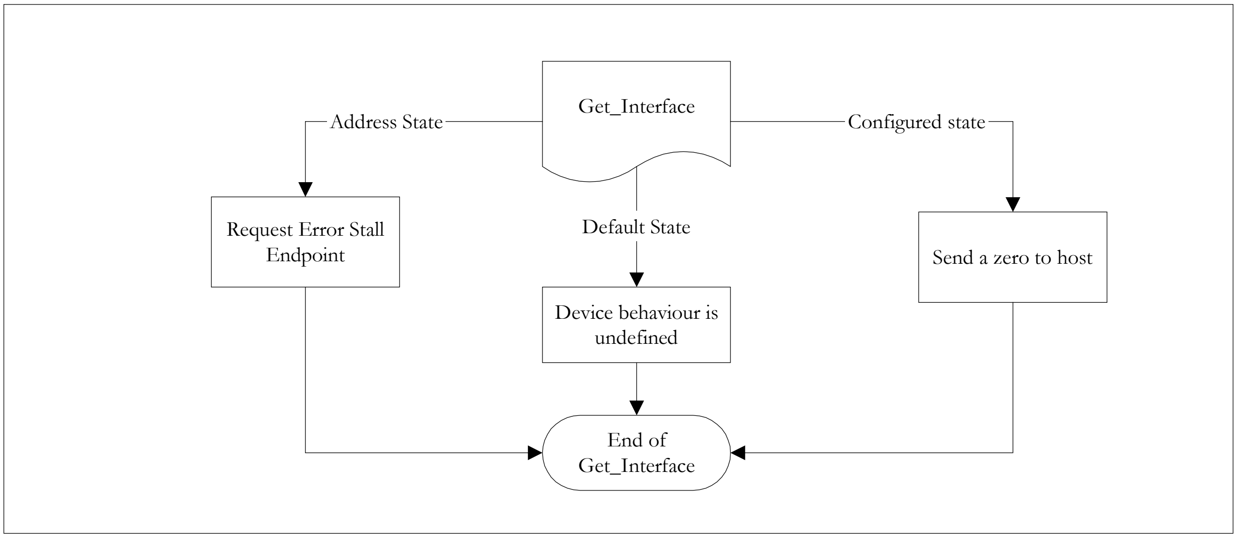

Figure 12-44: Flowchart of Get Interface

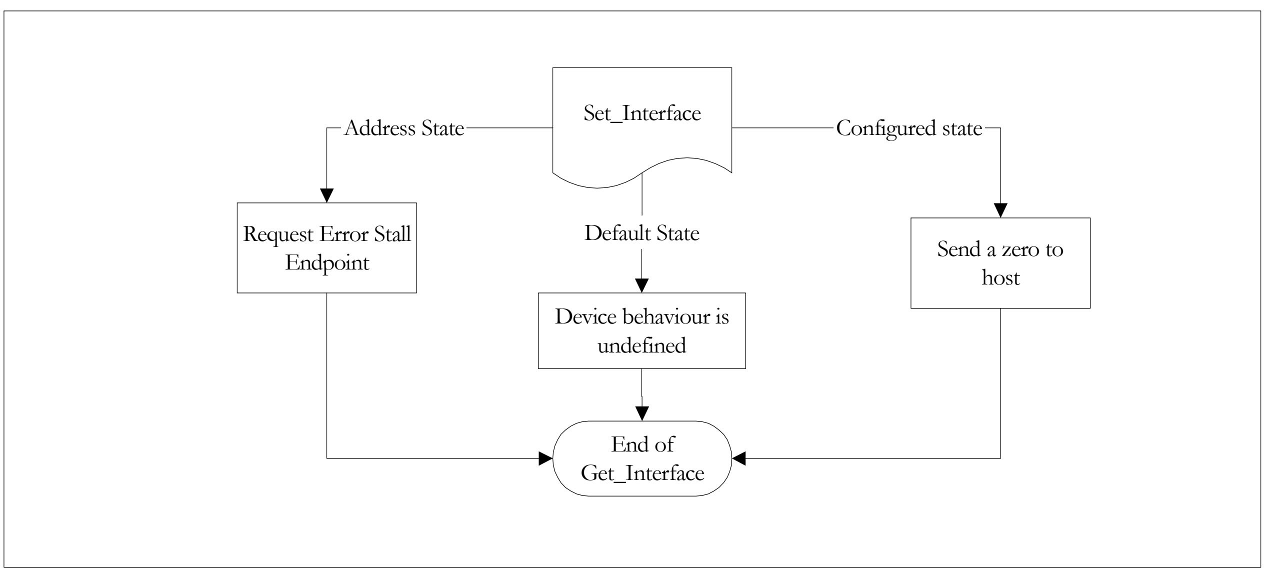

Figure 12-45: Flowchart of Set Interface

12.7.8. Set Feature Request

The Set Feature request is just the opposite of the Clear Feature request. Figure 12-46 contains the flowchart of Set Feature. If the recipient is a device, the microprocessor must set the feature of the device according to the feature selector in the Setup packet. Again, there is no support for the Interface recipient. For example, if the feature selector is 0 (which means enabling endpoint), the Device Controller of the ISP1362 specific endpoint must be stalled through the Write Endpoint Status command.

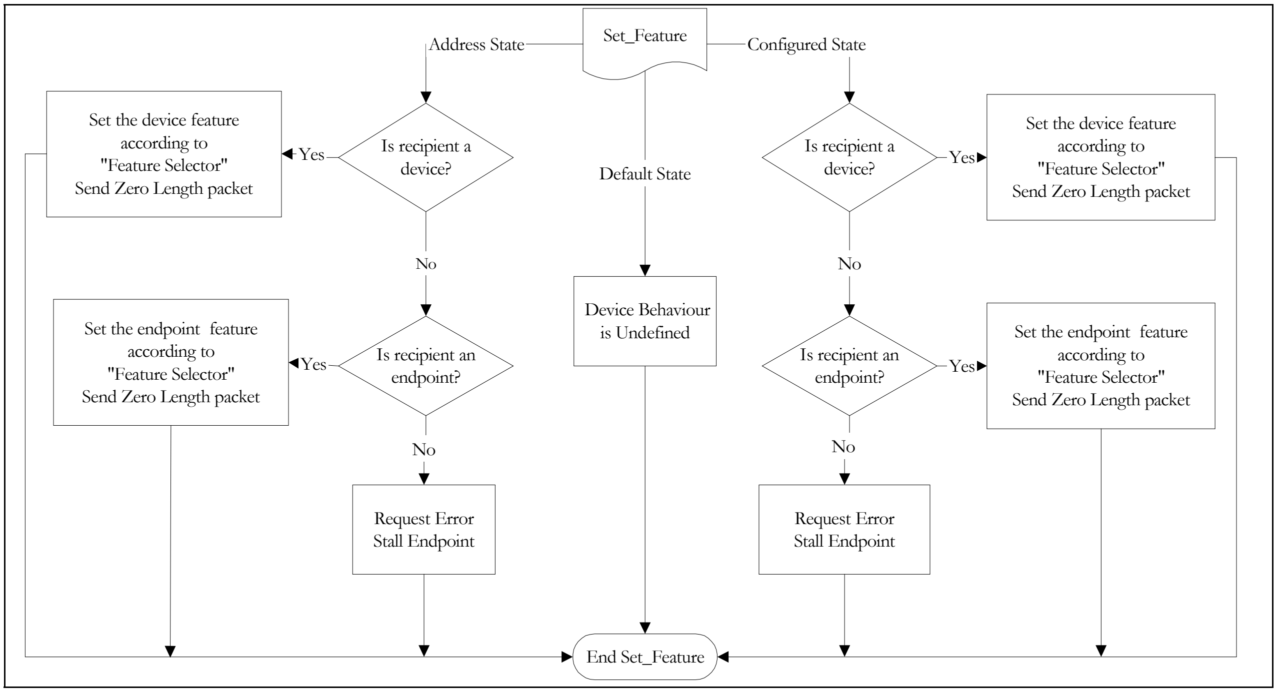

Figure 12-46: Flowchart of Set Feature

12.7.9. Class Request

Support for class requests is not included in the Device Controller of the ISP1362 sample firmware.

12.8. Vendor Request

In the ISP1362 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.

12.8.1. Vendor Request for the Bulk Transfer

The device request is defined in Table 12-13.

Table 12-13: Device Request

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

The details requested by the bulk transfer operation are sent in the Data phase after the Setup Token phase of the device request. The sample firmware and applet use a proprietary definition, which is given in Table 12-14.

Table 12-14: Proprietary Definition of the Sample Firmware and Applet

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

12.8.2. CATC Capture of a PIO OUT Transfer

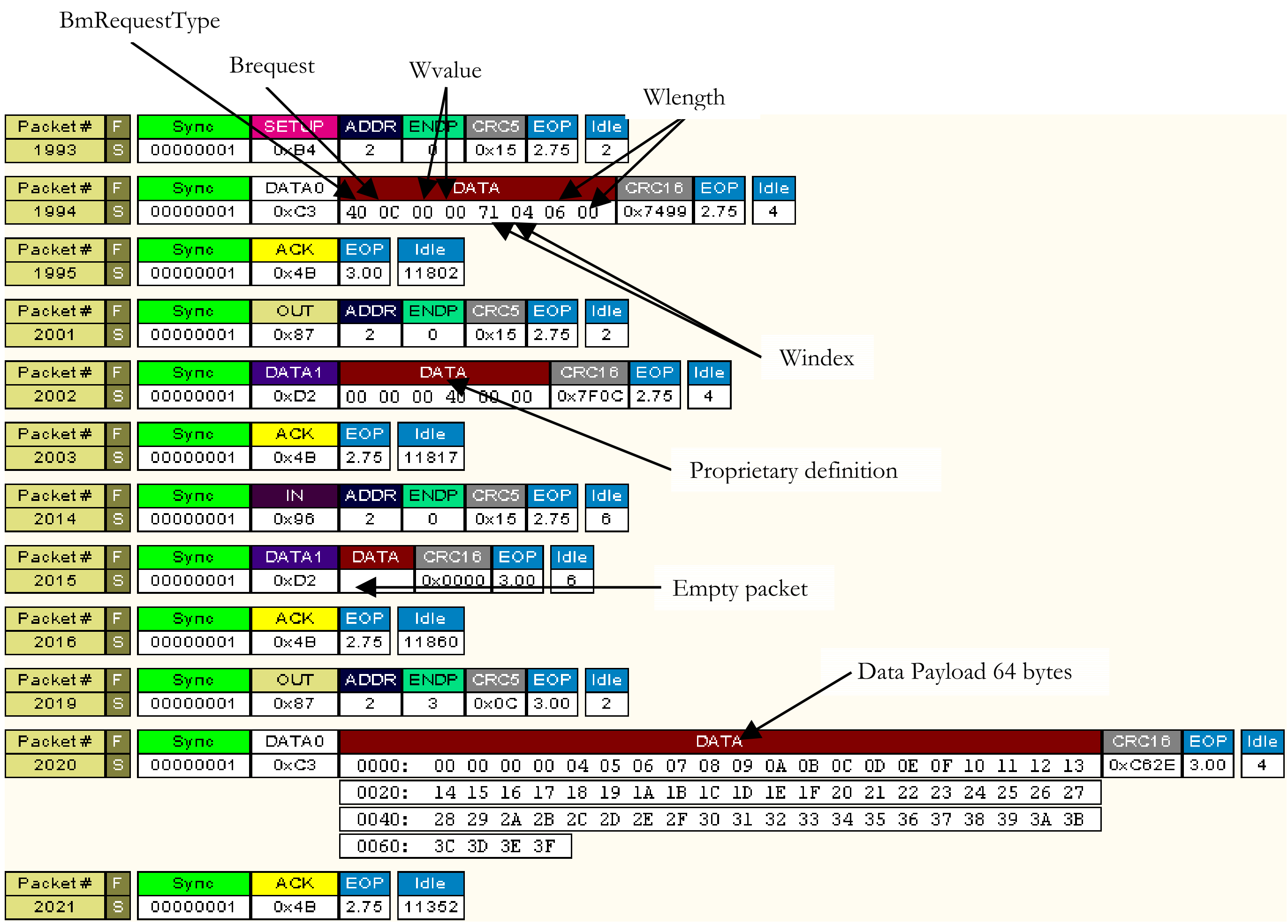

Figure 12-47: CATC Capture of a PIO OUT Transfer

12.8.3. CATC Capture of a PIO IN Transfer

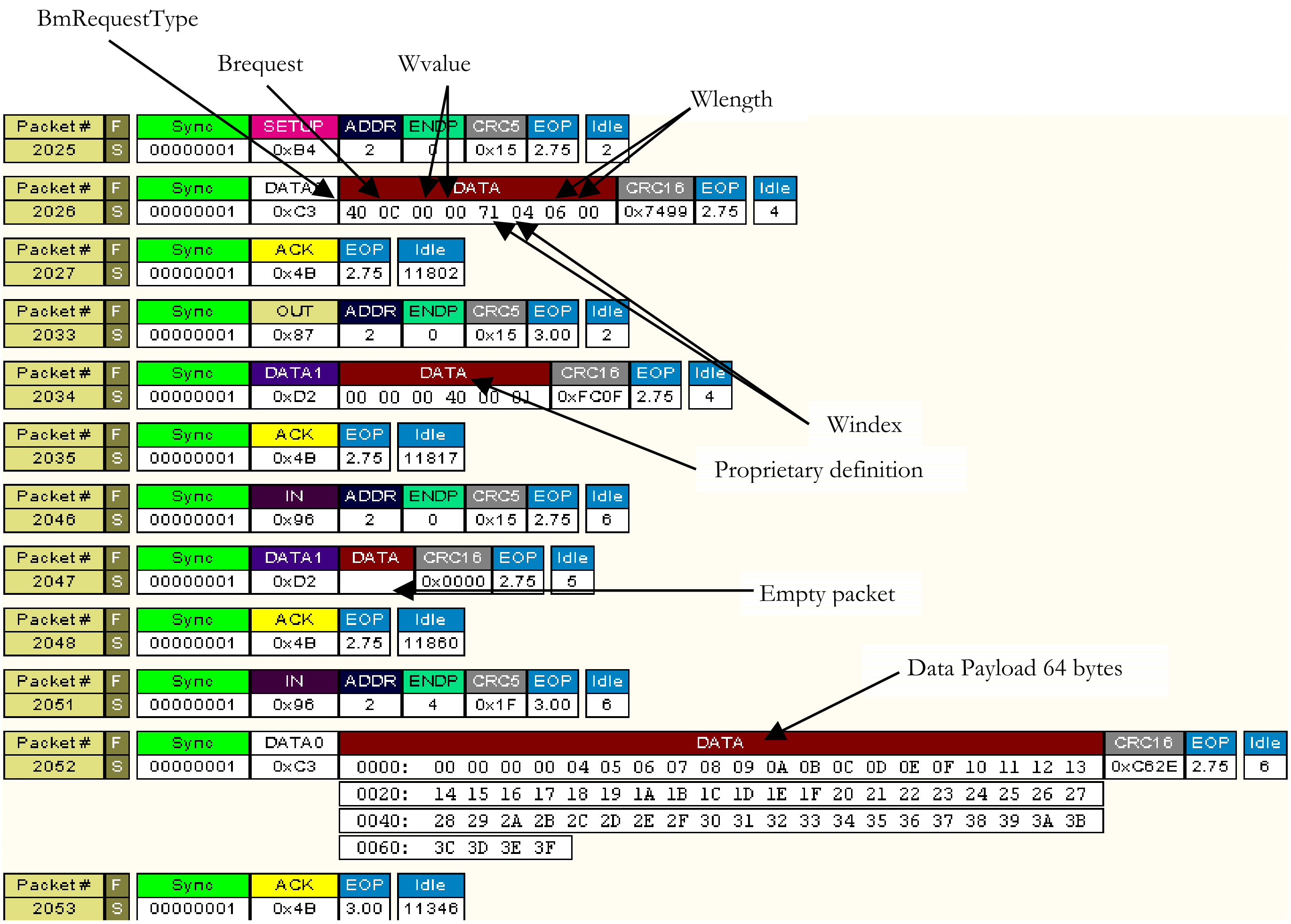

Figure 12-48: CATC Capture of a PIO IN Transfer

ISP1362 Embedded Programming Guide Rev. 0.9

12.8.4. Vendor Request for the ISO Transfer

The device request is defined in Table 12-15.

Table 12-15: Device Request

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

For the ISO transfer, the applet and the firmware must prearrange 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.

12.8.5. CATC Capture of an ISO OUT Transfer

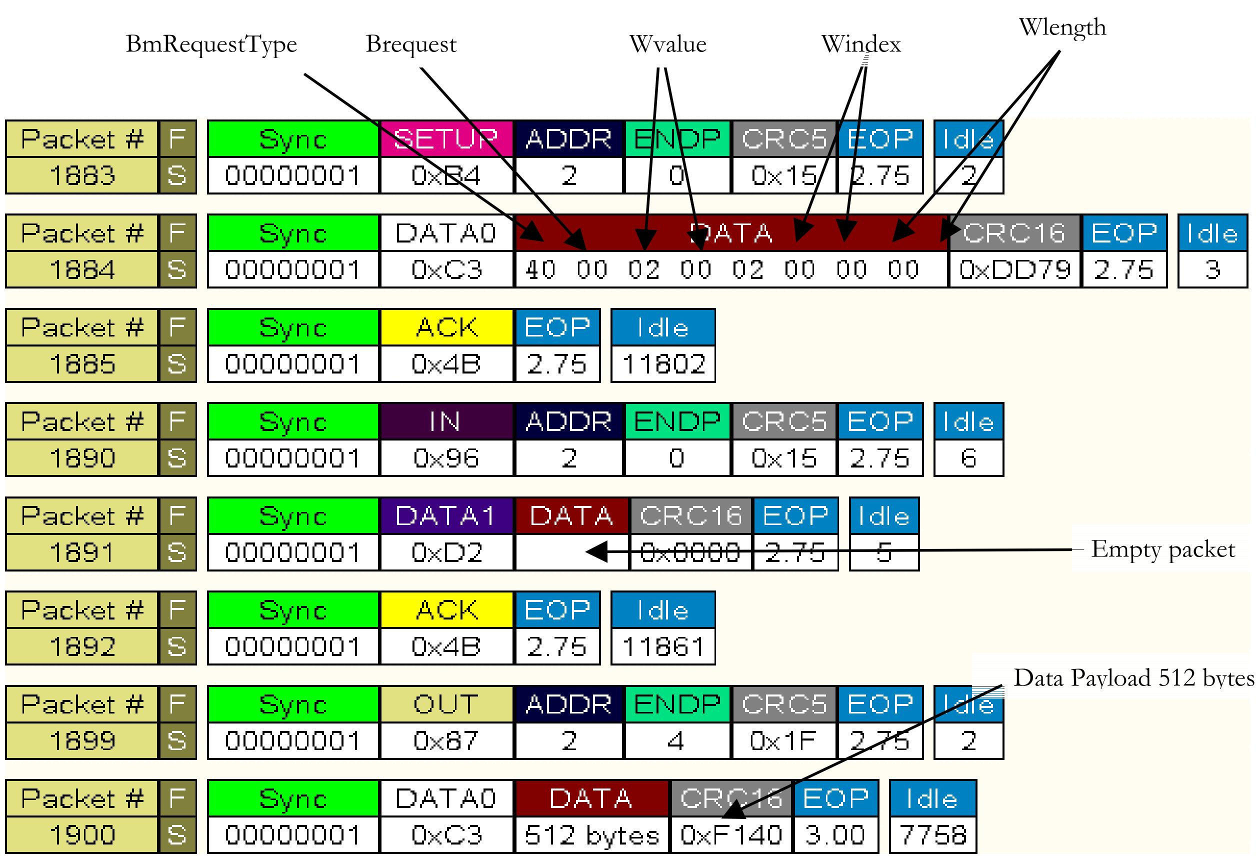

Figure 12-49: CATC Capture of an ISO OUT Transfer

12.8.6. CATC Capture of an ISO IN Transfer

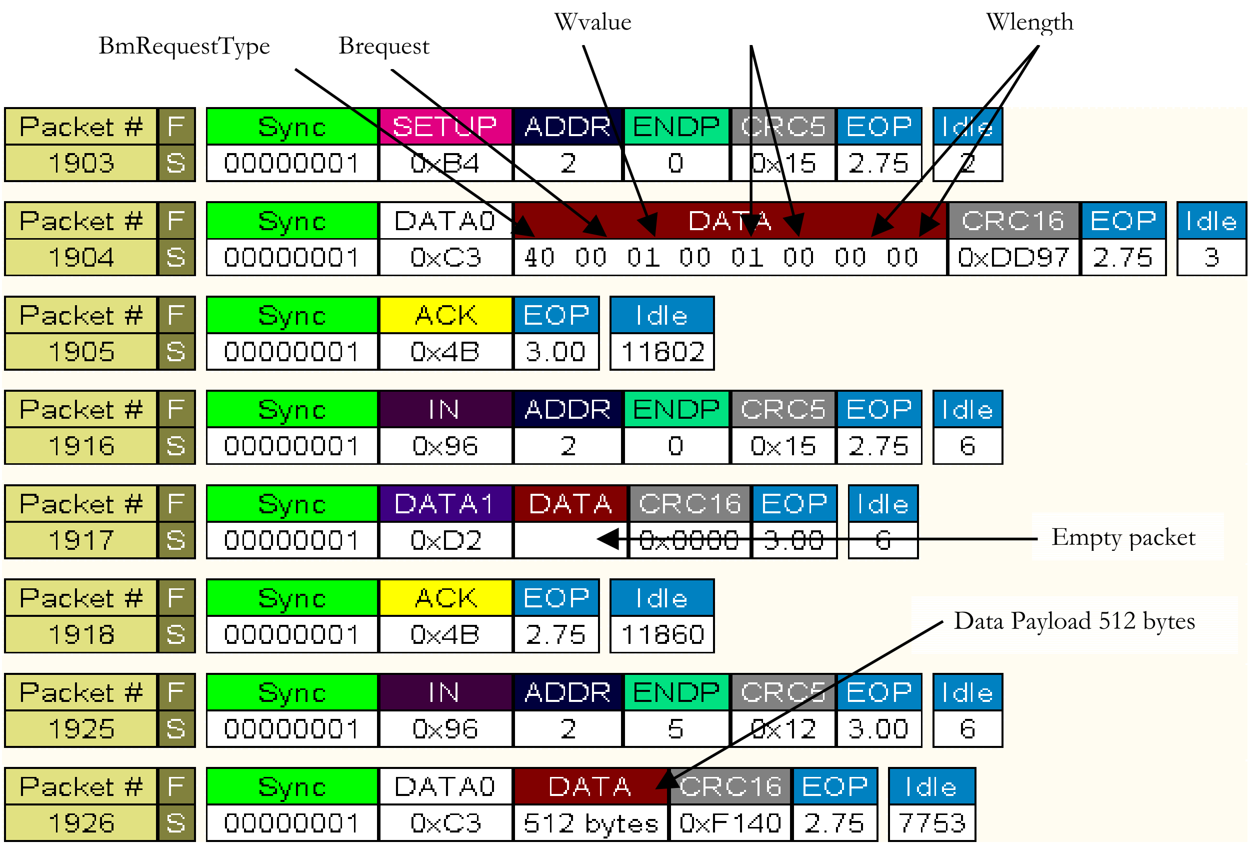

Figure 12-50: CATC Capture of an ISO IN Transfer

13. References

- *ISP1362 Single-chip USB OTG controller datasheet*
- *Universal Serial Bus Specification Rev. 2.0 (full-speed section)*
- *Open Host Controller Interface Specification for USB, Release: 1.0a*
- *On-The-Go Supplement to the USB 2.0 Specification Rev. 1.0.*