SL811HS/SL811HST Application Notes

Cypress Semiconductor Corporation • **3901 North First Street** • **San Jose** • **CA 95134** • **408-943-2600**

CONVENTIONS

1,2,3,4	Numbers without annotations are decimals.
Dh, 1Fh, 39h	Hexadecimal numbers are followed by an “h”.
0101b, 010101b	Binary numbers are followed by a “b”.
bRequest, n	Words in *italics* indicate terms defined by USB Specification or by this Specification.
SL811HS	refers to SL811HS in either 28PLCC or SL811HST - 48LPQFP package.

DEFINITIONS

USB	**U**niversal **S**erial **B**us

REFERENCES:

[Ref 1] SL811HS Specification
[Ref 2] USB Specification 1.1
[Ref 3] Human Interface Device support for SL811HS (sl811hs_hid_hub.pdf)
[Ref 4] Host Debugger Document (host_debugger.pdf)

Please direct any questions about this document to usbapps@cypress.com

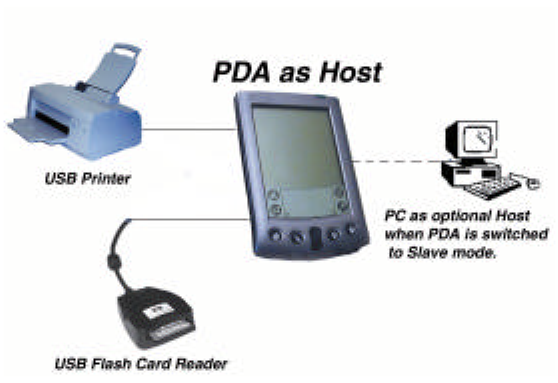
REVISION HISTORY

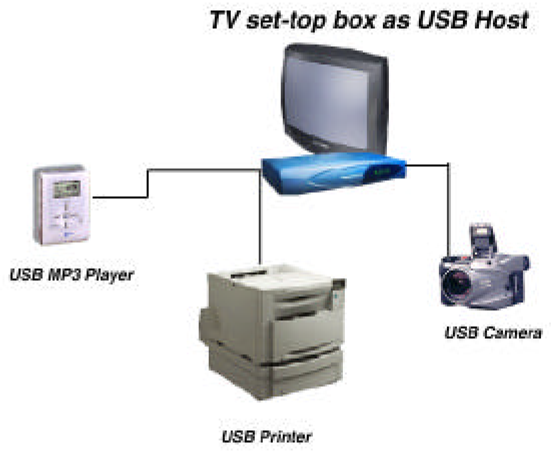
Name and Version	Date Issue	Description	Author/Revise
First Draft	Sept 26, 2000	Created	TBN
Second Draft	10/24/2000		Bernie
Preliminary	02/02/01	Added USB software Tech. support	TBN
Version 1.13	02/06/2001	Version 1.13	KPS
Version 1.16	02/15/2001	Preliminary release	Bernie+TBN+KPS
Version 1.17-1.19	05/17/2001	Add handle USB hub & low speed	TBN
Version 1.21	12/12/2001	Changed name & logos to Cypress	SIK

1. INTRODUCTION

1.1. Overview

The SL811HS USB Host Controller is a single chip USB embedded host device that can communicate with either full or low speed USB peripherals. The SL811HS can interface to devices such as microprocessors, microcontrollers, DSP's, or to a variety of buses such as ISA, PCMCIA and others. The SL811HS is suitable for use in a variety of embedded systems that require configurable USB host or device features. Typical applications include: Personal Digital Assistants (PDA), Set-top boxes, POS, Medical Test Instruments, Digital Video or Still Cameras, Scanners, Printers, Mobile/Cellular Phones, Game devices, etc.





Host and Slave Application Example

Host Application Example

2. SL811HS HOST OR SLAVE MODE

2.1. Overview

The SL811HS supports both USB Host and USB Slave modes. These modes are selectable by a single pin that can be permanently connected to ground for Host mode, or left unconnected for Slave mode. Optionally, the Mode Select pin can be driven by a GPIO pin of a host CPU thus allowing software selection of the USB Host/Slave modes. In certain applications, the user may need to support both Host and Slave modes with a single SL811HS controller. The following will describe an application whereby the SL811HS can be utilized to support both modes. A schematic diagram is provided (Figure 1: SL811HS circuit diagram) to illustrate a possible implementation.

2.2. Hardware requirements

There is a minimum set of requirements to implement the host/slave functions with a single SL811HS. These are described below in Table 1 below.

2.1.1 External Controller and USB connectors

The application requires an external microcontroller that can provide at least two 3.3V GPIO lines to control the Host/Slave mode pin, and sufficient power to supply to the USB connector when Host mode is enabled.

The USB 1.1 specification requires the use of different USB connectors for Host and Slave ports, and specific termination resistors for each port. The schematic diagram (Figure 2: SL811HS circuit diagram/Figure 2: SL811HS host/slave diagram) illustrates the requirements for each mode.

2.3. Power On and Reset Mode

The SL811HS needs to be reset and reconfigured when switched to different modes. The SL811HS has a hardware RESET input that can be used rather than recycling power on/off whenever the mode switch is initiated. For instance, when switching to the Host mode, after the hardware reset, the user sets bit 7 = ‘1’ of Control2 Register (cSOFcnt =0x0F), SL811HS will be in host mode. Please note when both Host and Slave USB connectors are present as shown in the diagram, only one device can be attached at any one time. The following summary of actions has to be accomplished in switching between modes:

Host Mode:

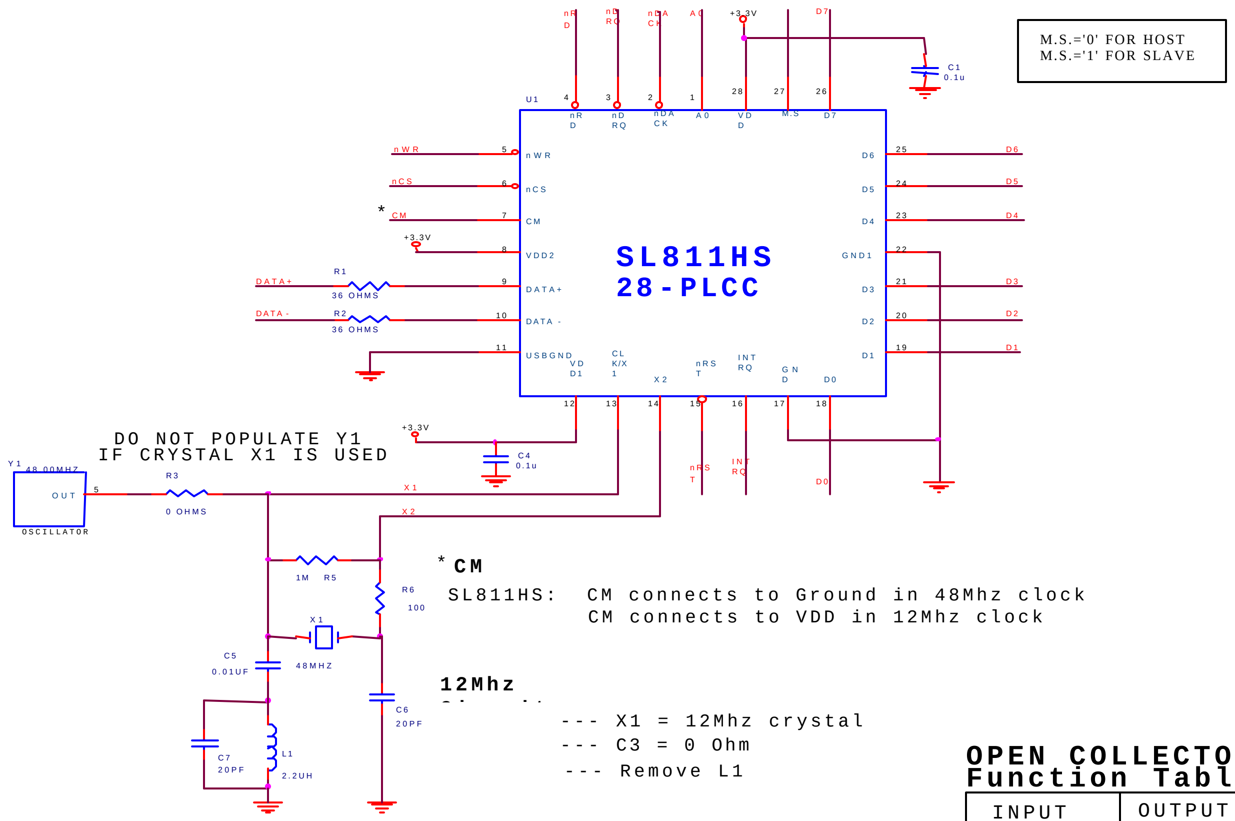
- Pin M/S must set low
- Issue HW reset.
- Set bit-7=1 of cSOFcnt register
- Data+ - switch pull-up 1.5K ohms resistor to disconnect
- Data+ and Data- switch pull-down 15k ohm resistors to ground
- Provide power to pin-1 of USB connector

Slave Mode:

- Pin M/S must set high.
- Issue HW reset.
- Data+ - switch pull-up 1.5K ohm resistor to Vdd.
- Data+ and Data- switch pull-down 15K ohm resistors to disconnect.
- Disconnect power from pin-1 of USB connector.

2.4. Switching Logic

The schematic diagrams illustrate a possible implementation using open collector logic devices to switch the pull-up and pull-down resistors, an inverter and a power switch.



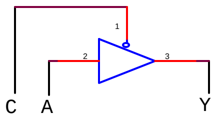
*CM
SL811HS: CM connects to Ground in 48Mhz clock
CM connects to VDD in 12Mhz clock

12Mhz
--- X1 = 12Mhz crystal
--- C3 = 0 Ohm
--- Remove L1

Circuit

OPEN COLLECTOR Function Table

INPUT	OUTPUT
C	Y
L	L
L	H
H	HI-Z



H = HIGH Logic Level
L = LOW Logic Level
X = Either LOW or HIGH Logic Level
HI-Z = 3-STATE (outputs are disabled)

Figure 1: SL811HS circuit diagram

NOTE:

CONNECT EXTERNAL DEVICE TO EITHER JP1 OR JP2.

NEVER CONNECT TO BOTH SIMULTANEOUSELY.

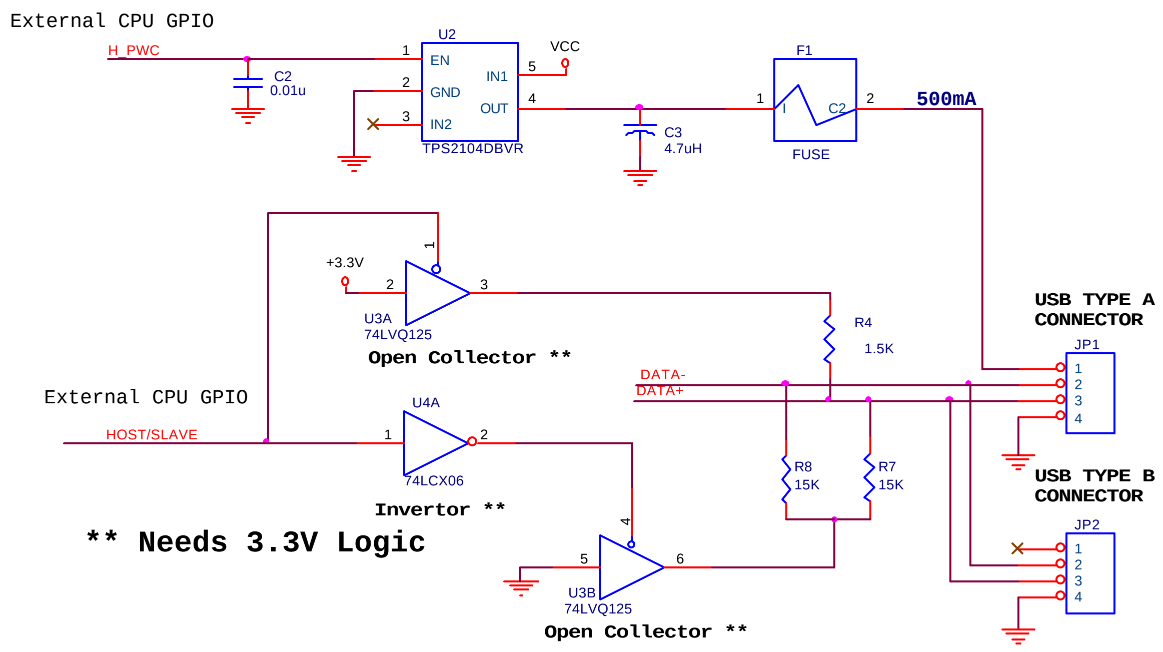


Figure 2: SL811HS Host/Slave diagram

2.5. Software Initialization

Cypress provides Host and Slave code in development Kit. To switch Host/Slave mode, you need to write an appropriate value to the register 0FH of SL811HS. After writing value to register 0FH, you need to do Initialization.

2.5.1 Initialization from Host to Slave

This software initialization is on SL811HS slave mode. This assumes hardware is already configured slave mode.

- Power up
- Config slave mode
  - Write value “0x00” to register 0FH
- Memory Test
- Slave Initialization:
  - Reset chip
  - Disable Interrupt Enable
    - Write value “0x00” to register 06H
  - Setup USB Address
    - Write value “0x00” to register 07H
  - Setup endpoint 0 address starts at 0x40
    - Write value “0x40’ to register 01H
  - Setup transfer length of endpoint 0
    - Write value “0x40” to register 12H
  - Set Arm and direction bit
    - Write value “0x03” to register 00H

  Endpoint 1 Set A:
  - Setup address Endpoint 1 starts at 0x60
    - Write value “0x60” to register 11H
  - Setup endpoint 1 transfer length
    - Write value “0x40” to register 12H
  - Set Arm and direction bit
    - Write value “0x03” to register 10H

  Endpoint 2 Set A and B:
  - Setup Endpoint2A address starts at 0x80
    - Write value “0x80” to register 21H
  - Setup Endpoint2B address starts at 0xC0
    - Write value “0xC0” to register 29H
  - Setup Endpoint2A transfer length
    - Write value “0x40” to register 22H
  - Set Arm and direction bit
    - Write value “0x03” to register 10H

  - Interrupt Enable for Endpoint 0, 1, 2, SOF and USB reset
    - Write value “0x67” to register 06H
  - Enable USB Transfer
    - Write value “0x01” to register 05H
  - Clear Interrupt Status
    - Write value “0xFF” to register 0DH

2.5.2 Initialization from Slave to Host

This software initialization is on SL811HS host mode. This assumes hardware is already configured host mode.

- Power up
- Config host mode
  - o Write value “0xAE” to register 0FH
- Enable power to USB device
- Delay for hardware stable before detect USB device
  - o Delayms (25,0); // Delay 25 mili-second
- Memory Test
- Host Initialization:
  - o Hardware reset chip
  - o Interrupt Enable on set USB_AB, Insert/Remove, and USB Reset
  - o USB Reset, set full speed, and clock 12/48 Mhz mode
    - ▪ Write a value “0x48” to register 05H
  - o Disable USB Transfer and SOF generation
    - ▪ Write a value “0x00” to register 05H

- Full/Low Speed Detection

  - o If bit-7 of Interrupt Status (0DH) equal 0, it is Low speed device.
    - ▪ Write value “0xFF” to register 0EH
    - ▪ Write value “0x0E” to register 0EH
    - ▪ Write value “0x21” to register 05H

  - o If bit- 7 of Interrupt Status (0DH) equal 1, then it is Full speed device.
    - ▪ Write value “0xAE” to register 0EH
    - ▪ Write value “0x0E” to register 0EH
    - ▪ Write value “0x05” to register 05H
  - o Otherwise if there is no power or no device attach clear Interrupt Status
    - ▪ Write value “0xFF” to register 0DH

- Generate SOF to USB device
  - o Write a value “0x50” to register 00H
  - o Write a value “0x00” to register 04H
  - o Write a value “0x01” to register 00H

- Delay for Hardware stable before enumeration
  - o Delayms (10,0); // Delay 10 mili-second

2.6. M/S pin (pin27/pin40)

It is recommended that M/S pin is "NC". If you want to set the "Host" or "Slave" mode as power-on-default, you need to make M/S pin be tied to

Mode	M/S pin
Host	GND
Slave	VCC3.3

2.7. Pull-up/down registers on USB lines

A 1.5K pull-up resister is required for "Slave" mode, a 15K pull-down resisters is required for "Host" mode. You will need to enable/disable each registers.

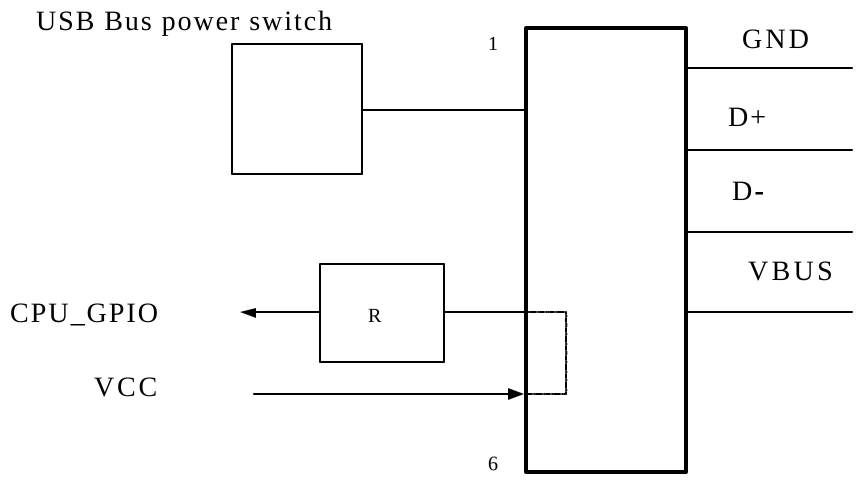
NOTE: See the "SL811HS and SL811HST Application Notes page.8" for more details.

2.8. Host/Slave detector

Since the SL811HS does not have a mechanism to know if the inserted device is operating as "Host" or as "Slave", you need to add a Host/Slave detector. Following are examples of this.

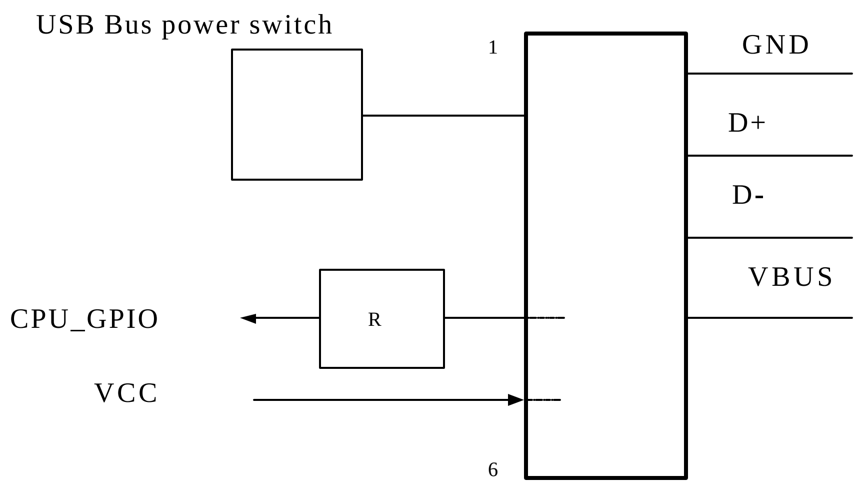
(Example.1) Using 2 different cable with special connectors either "Host" or "Slave"

Connector for Host (pin5 and 6 are inter-connected.)



1. CPU_GPIO detects that a cable is for “Host”.
2. Start supplying USB Bus power.

Connector for Slave



1. CPU_GPIO detects that a cable is for “Slave”.
2. Never supply USB Bus power.

(Example.2) Using two connectors on host side.
See “SL811HS and SL811HST: Application Notes page.8” for the layout.

Supplier must explain to the customer, “ Host and Slave must not be used at the same time” because it will break the system.

3. PROGRAMMING INFORMATION

3.1. Host and Device Programming

The SL811HS supports auto increment mode for Read/Write Cycles in the A0 mode. In A0 mode, the Micro Controller sets up the address only once. On any subsequent DATA Read/Write access, the internal address pointer will advance to the next DATA location.

In A0 mode (shown in the diagram above), the internal register address is input to the device in a normal I/O or memory mapped I/O write operation with the A0 address select input driven low ('0'). This operation results in the address being latched internally so that the subsequent Read or Write operation with A0 driven high ('1') will result in a data transfer.

1.

D7	D6	D5	D4	D3	D2	D1	D0	A0
REG/MEMORY ADDRESS								'0'

WRITE ADDRESS

2.

D7	D6	D5	D4	D3	D2	D1	D0	A0
REG/MEMORY DATA								'1'

READ/WRITE REGISTER OR MEMORY

SL811H Register/Memory I/O Operations.
1. Write Address to I/O location with A0 = '0'.
2. Read/Write data from/to location. A0 = '1'.

3.2. SL811HS USB Control Registers

The SL811HS can communicate to any USB Device with any type of configuration or function including any specific endpoints via the USB control registers. The SL811HS can address up to 127 devices and 16 endpoints.

Each endpoint has an associated set of registers (this is further described in the SL811HS specification). Each must be programmed in order to initiate or respond to transactions on the USB. The host initiates USB Control Register's transactions during setup and configuration. Typically, the host will request information from the device during setup to determine the device's characteristics, and assign a USB ID to the device. The complete definition of control messages and transactions are defined in Chapter 9 of the USB Specification 1.1.

3.3. SL811HS MEMORY MAP

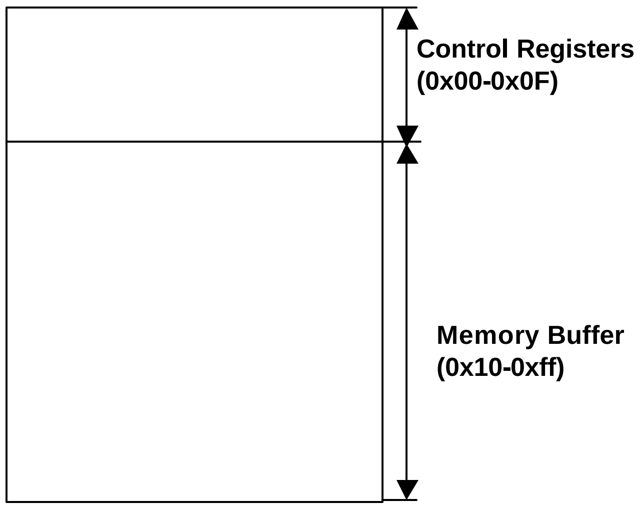
The SL811HS contains 256 bytes of internal memory buffer. The first 16 bytes of memory represent control and status registers for programmed I/O operations. The remaining memory locations are used for data buffering (max 240 Bytes).

The SL811HS can be mapped into the users processor I/O or memory space. In the following example, which is used in the SL811HS DVK, The SL811HS is mapped at an ISA bus I/O location.

Table 1: SL811HS and PC address

Register Name	Address	Register Function
	Host Mode	
SL11_ADDR	0x290	SL811HS I/O Address

Figure 3: SL811HS Internal Memory Map



3.3.1 Control Registers

In Host mode, SL811HS has two sets of USB Endpoint Control Registers, which allow for overlapped operation. The registers allow new transactions to be set up while a current transaction is underway. The following Tables show various Register addresses and values used in the DVK software routines

Table 2: USB Control Register Memory

USBControl0		**Description**
EP0Control	0x00	Control Register0
EP0Address	0x01	Address Register0
EP0XferLen	0x02	Transfer length Register0
EP0Status	0x03	Status Register0
EP0Counter	0x04	Counter Register0
USBControl1		**Description**
EP0Control	0x08	Control Register1
EP0Address	0x09	Address Register1
EP0XferLen	0x0a	Transfer length Register1
EP0Status	0x0b	Status Register1
EP0Counter	0x0c	Counter Register1

Table 3: SL811HS Control Register Memory Map

Register Name	Address	Register Function
CtrlReg	0x05	Control Register
IntEna	0x06	Interrupt Enable
IntStatus	0x0d	Interrupt Status
DATASet	0x0e	DATA Set
CSOFcnt	0x0f	SOF Counter High and Control Register

3.3.2 Memory Buffer

The SL81HS Memory buffer is located between address 0x10 and 0xFF inclusive, and is used for data buffering. The following Tables show Register labels and variables mapped to specific memory and register locations that are used in the example code.

Table 4: DATA0/DATA1 Mapping

Register Name	Address	Register Function
cMemStart	0x10	Host Memory Start
ubufA	0x80	Buffer A address for DATA0
ubufB	0xc0	Buffer B address for DATA1
uxferLen	0x40	Xfer Length
sMemSize	0xc0	Total SL811HS memory size
cMemEnd	256	Upper limit of memory

Table 5: Buffers use for configuration and Vendor Specific command

Register Name	Address/ Value	Register Function
EP0Buf	0x40	Endpoint 0 Buffer where SL811HS memory starts
EP0Len	0x40	Length of config buffer EP0Buf

Table 6: SL811HS Code Bit Definitions

Name	Value	Description
DATA0_WR	0x07	Arm+Enable+tranmist to Host+DATA0
DATA1_WR	0x05	Arm+Enable+tranmist to Host on DATA1
ZDATA0_WR	0x47	Arm+Transaction Ignored+tranmist to Host+DATA0
ZDATA1_WR	0x45	Arm+Transaction Ignored+tranmist to Host+DATA1
DATA0_RD	0x03	Arm+Enable+received from Host+DATA0
DATA1_RD	0x43	Arm+Enable+received from Host+DATA1
PID_SOF	0xa5	SOF Token
PID_SETUP	0x2d	SETUP Token
PID_IN	0x69	IN Token
PID_OUT	0xe1	OUT Token
PID_PRE	0x3c	PRE Token
PID_NAK	0x5a	NAK Token
PID_STALL	0x1e	STALL Token
PID_DATA0	0xc3	DATA0 Token
PID_DATA1	0X4b	DATA1 Token
MAX_RETRY	0xffff	Number of Re-try
TIMEOUT	6000L	Time out

3.4. Interrupt Status Register

The ISR is a Read/Write register providing interrupt status. Writing to this register can clear interrupts. To clear a specific interrupt, the register is written with corresponding bit set to “1”.

Bit Position	Bit Name	Function
0	USB-A	USB-A done Interrupt.
1	USB-B	USB-B done Interrupt.
2	Babble Detection	1=enable interrupt on babble detection
3	Reserved	
4	SOF timer	1=enable interrupt on 1ms SOF timer
5	Insert/Remove	Slave Insert/Remove detection (Note 2)
6	USB Reset/Resume	Enable USB Reset/Resume Interrupt. (note 1)
7	D+	Value of the Data+ pin

NOTE1:
This bit is shared between USB_RESET and Resume interrupt detection. When bit-6 of register 05H is set to one, this bit will be the Resume detection Interrupt bit. Otherwise, this bit is used to indicate detection of USB RESET.

NOTE2:
Bit-5 is provided to support USB cable Insertion/Removal detection for the SL811HS in Host Mode. This bit is set whenever a cable is inserted or removed from the USB port. It should be cleared after a removal or insertion is detected so that the next event can be detected.

Bit 6 is updated whenever a USB reset is detected, or when in suspend state the detection of a ‘K’ or SE0 state

- Bit 7 provides continuous USB Data+ line status. Bit 7 in conjunction with bit 6 and 5 can be used to detect if a Low or Full speed device is connected after USB reset is issued. The bit should be read immediately after a USB reset and before USB activity begins for a valid state.
- On Power up, before generating a USB Reset, the state of the USB Reset bit will be = ‘1’ if no device is connected, i.e. a ‘SE0’ on the USB. If the bit is ‘0’ then it indicates a device is attached. If the D+ (bit 7) is a ‘1’ then the attached device is a Full Speed device, if D+ is a ‘0’ then it is a Low Speed Device.
- After a USB Reset has been generated and the USB is idle, the same procedure can be used to determine if a device is attached. The SL811HS is continually monitoring the USB and if a ‘SE0” state continuously exists on the USB the USB reset bit will be ‘1’ indicating no device attached. If ‘0’ then device attached and speed can be determined by viewing the state of bit 7.

4. SL811HS HOST SOFTWARE

These sample IO subroutines are based on the SL811HS DVK. The Address location is mapped for the ISA bus space.

The User should redefine the SL11_ADDR variable to match user's mapping.

The next section shows how a Microcontroller can communicate with SL811HS. In successive sections, examples will be shown for two possible operations to be performed with the SL811HS:

- **Transmit Data**
- **Receive Data**

4.1. SOFTWARE APPLICATION

The SL811HS is intended for embedded systems. This document describes general interface routines that are required by the SL811HS host USB interface. For integration with a specific external Microcontroller, you will need to understand the following sections to know how to communicate with the SL811HS. The interface to the SL811HS is described using an ISA bus, but these functions can be translated for use with any external CPU or other type of interface bus.

4.1.1 Single Write Operation

Writing a byte to the SL811HS involves two write cycles. On the first write, the application must write a register address into the SL811HS's Address Pointer Register. On the second write, the actual data is written to the chip.

In an I/O mapped application, the function to write a byte to an SL811HS register is shown below:

Program Sample 1: Single Write Function

```
void SL11Write(BYTE a, BYTE d)
{
  outportb(SL11_ADDR,a);
  outportb(SL11_ADDR+1,d);
}
```

The function to write a block of data of a specific length into the SL811HS from the buffer, which is defined by the pointer "addr".

Program Sample 2: Write Buffer Function

```
void SL11BufWrite(short addr, BYTE *s, short c)
{
   if(c<=0) return;
   outportb(SL11_ADDR,addr);
   while (c--) outportb(SL11_ADDR+1,*s++);
}
```

4.1.2 Single Read Operation

Reading a byte from the SL811HS involves an address write cycle, followed by a read cycle. First, as in a data write, the application writes a register address into the SL811HS's Address Pointer Register. The data is then read from the chip in a read cycle.

The function to read an SL811HS register is shown below:

Program Sample 3: Single Read Function

```
BYTE SL11Read(BYTE a)
{
  outportb(SL11_ADDR,a);
  return (inportb(SL11_ADDR+1));
}
```

The Read Buffer function reads a block of data of a specific length from the buffer, which is located by the pointer “addr.”

Program Sample 4: Read Buffer Function

```
void SL11BufRead(short addr, BYTE *s, short c)
{
   if( c <= 0) return;
   outportb(SL11_ADDR, addr);
   while (c--) *s++ = (BYTE)inportb(SL11_ADDR+1);
}
```

4.1.3 Memory Test

The Memory test verifies read/write operation of the SL811HS internal memory.

Program Sample 5: Memory Test

```
int SL11HMemTest()
{
   int errors = 0, i;
   for (i = EP0Buf; i < cMemEnd; i++)    // addr = data
      SL11Write((BYTE)i, (BYTE)i);
```

```
        for (i = EP0Buf; i < cMemEnd; i++)    // verify data
        {
           if ((BYTE)i != SL11Read((BYTE)i))
              errors++;
           SL11Write ((BYTE)i, (BYTE)~i);
           if ((BYTE)~i != SL11Read((BYTE)i))
              errors++;
        }

        // auto increment: addr = data
        for (i = EP0Buf, outportb(SL11_ADDR,EP0Buf); i < cMemEnd; i++)
            outportb(SL11_ADDR+1, i);

        // auto: addr = data
        for (i = EP0Buf, outportb(SL11_ADDR, EP0Buf); i < cMemEnd; i++)
        {
           if ((BYTE)i != (BYTE)inportb(SL11_ADDR + 1))
              errors++;
        }
        // clear all SL811H/SL11H Memory
        for (i = EP0Buf, outportb(SL11_ADDR, EP0Buf); i<cMemEnd; i++)
            outportb(SL11_ADDR + 1, 0);
        printf("Memory test done %x\n",errors);
        return errors;         // Return number of error
    }
```

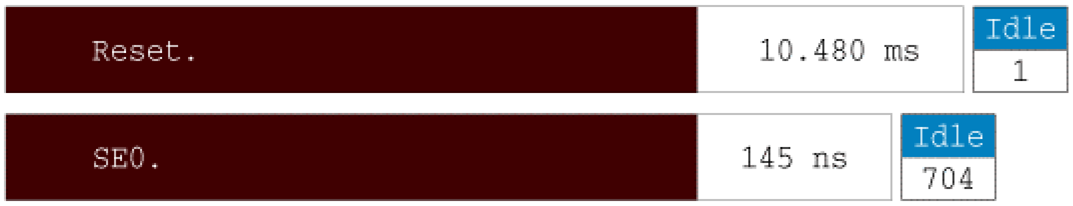
4.1.4 USB Reset

Before accessing a USB device, the SL811HS must generate a USB_RESET, which forces the slave device to its default address of zero. The minimum time required to hold the USB bus in Reset is ≥ 10 milliseconds. Every USB device after detecting Reset responds to USB address zero. After Reset, configuration software can read every device’s descriptor at the same default address, one device at a time.

Program Sample 6: USB Reset Function

```
    //--------------------------------------------------------------------------
    // UsbReset:
    //--------------------------------------------------------------------------
    void USBReset()
    {
        BYTE tmp;
        tmp =  SL11Read(CtrlReg);          // Read Control Status
        SL11Write(CtrlReg,tmp | 8);        // Setup USB Reset
        Delayms(250,0);
        SL11Write(CtrlReg,tmp | 0x18);     // suspend/resume, reset
        Delayms(150,0);
        SL11Write(CtrlReg,tmp | 8);        // Setup USB Reset
        Delayms(10,0);                     // Delay 10 ms
        SL11Write(CtrlReg,tmp);            // enable USB
    }
```

USB Trace 1: USB Reset



Reset.	10.480 ms	Idle	1
SE0.	145 ns	Idle	704

4.1.5 CRC5/16 Generation

The SL811HS automatically computes and verifies all CRC5 and CRC16 by hardware. No CRC is required to be generated or verified by external firmware.

4.1.6 Zero Length Packet

The Zero Length Packet is a NULL packet during the IN or OUT transaction phase. For example, SL811HS will send the Zero Length Packet after a successful IN, OUT transaction to verify that the device endpoint has successfully returned the contents of the descriptor.

Figure 4: Zero Length Data Packet

SYNC	IN/OUT Token	DATA1 [zero length]	ACK

4.1.7 Double Buffer Operation

SL811HS memory structure is a ping-pong (double buffer) scheme. The software sequentially sets up Data PID, Endpoint, Device Address, and Payload for each USB packet transfer. The Data PID includes DATA0 PID and DATA1 PID. The DATA0 starts from a memory location at 0x10 and DATA1 PID starts at (DATA0 + Payload). The SL811HS provides two control registers to setup Endpoint, and Address to USB Device. The Payload is available from 8,16, 32, 64 for Control/Interrupt/Bulk USB Transactions and up to 118-bytes for ISO transactions.

DATA0/DATA1
Endpoint and Address
Payload

Program Sample 7: Data, Endpoint, and Address

SL811HS provides two EP0Status=0x03 and EP0Counter =0x04 registers for setting USB device PID, Endpoint, and Address. SL811HS host mode allows the user to setup any of the 16 endpoints and any of the 127 addresses.

```
ep    = (crc>>7) & 0xf;
data0 = 0x10;   // Memory start

SL11Write(EP0Address, (BYTE)data0);         // DATA0
...
// Setup PID, Endpoint, and device Address

SL11Write(EP0Status,  (BYTE)( (pid << 4) + ep ) );
SL11Write(EP0Counter, (BYTE)(crc & 0x7f));

     // setup 3-byte header + payload

SL11Write(EP0XferLen,(BYTE)(payload));

...
// setup next ping-pong buffer
data1 = data0 + payload;   // next buffer
...
```

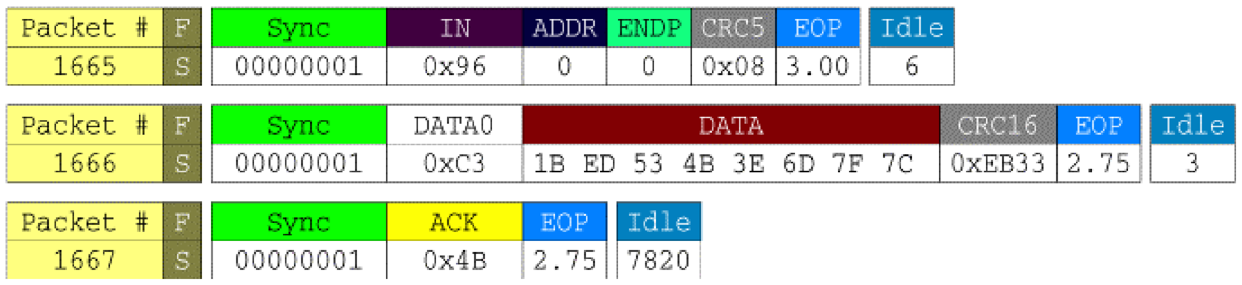
4.1.8 Single IN Packet

Example of a single IN Packet transaction: The IN Packet includes transmission of IN PID with device address and endpoint. The device responds with either DATA0 or DATA1 token and data payload. The program sample shows PID IN packet followed by a Data0 packet. For continuous IN packets, the device is required to toggle Data tokens i.e., Data0, Data1, and so on.

Program Sample 8: Single IN Packet

```
short addr = 3;           // address 3
...
SL11Write(EP0Address, (BYTE)cMemStart);            // DATA0

// PID_IN and endpoint 1
SL11Write(EP0Status,  ((PID_IN & 0xff)<<4) | 1); // PID + ep
SL11Write(EP0Counter, addr);
SL11Write(EP0XferLen,payload);  // payload
SL11Write(EP0Control,DATA0_RD);
```

USB Trace 2: IN Packet



Packet #	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
1665	S	00000001	0x96	0	0	0x08	3.00	6

Packet #	F	Sync	DATA0	DATA	CRC16	EOP	Idle
1666	S	00000001	0xC3	1B ED 53 4B 3E 6D 7F 7C	0xEB33	2.75	3

Packet #	F	Sync	ACK	EOP	Idle
1667	S	00000001	0x4B	2.75	7820

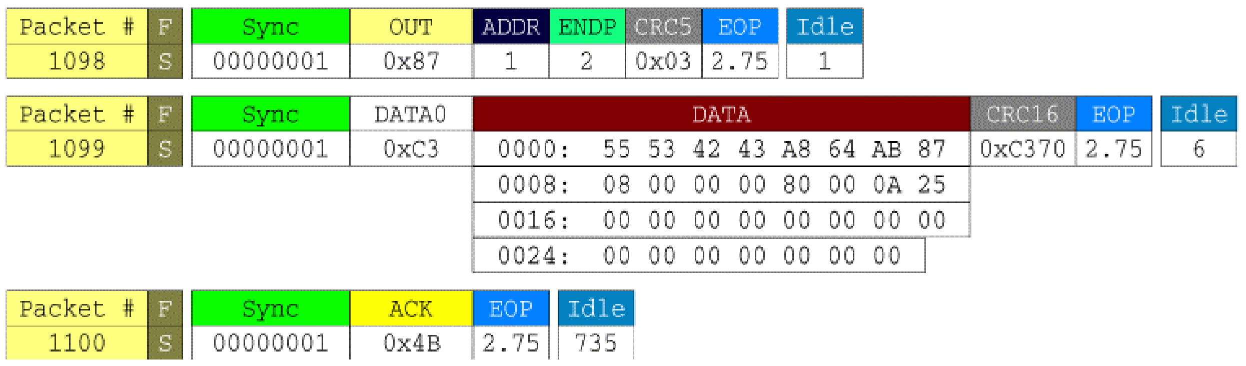
4.1.9 Single OUT Packet

Example of a single OUT Packet: The OUT Packet transaction consists of an OUT PID containing device USB address and Endpoint. It is followed by a DATA token with data payload. The program sample shows a PID OUT token followed by a Data0 packet. For continuous OUT packets, Data tokens are required to toggle between Data0 and Data1, and so on.

Program Sample 9: Single OUT Packet

```
short addr = 3;             // address 3
...
SL11Write(EP0Address, (BYTE)cMemStart);            // DATA0

// PID_IN and endpoint 2
SL11Write(EP0Status,  ((PID_OUT & 0xff)<<4) | 2); // PID + ep
SL11Write(EP0Counter, addr);
SL11Write(EP0XferLen,payload);  // payload
SL11Write(EP0Control,DATA0_WR);
```

USB Trace 3: OUT Packet



Packet #	F	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle
1098	S	00000001	0x87	1	2	0x03	2.75	1

Packet #	F	Sync	DATA0	DATA	CRC16	EOP	Idle
1099	S	00000001	0xC3	0000: 55 53 42 43 A8 64 AB 87	0xC370	2.75	6
				0008: 08 00 00 00 80 00 0A 25			
				0016: 00 00 00 00 00 00 00 00			
				0024: 00 00 00 00 00 00 00			

Packet #	F	Sync	ACK	EOP	Idle
1100	S	00000001	0x4B	2.75	735

4.1.10 Short Data Packet

The short Data Packet occurs when the host does not know the transfer length of the USB device during the IN transaction. This can cause a Bus Time-out or a Babble condition.

Program Sample 10: Short Data Packet

```
...    // check PID IN
if (pid == PID_IN)
   rem = (BYTE)SL11Read(EP0Counter);
   len -= rLen;     // rLen = actual read/write
   if (bLen  &&  len > 0)
   {
        SL11Write(EP0XferLen, (BYTE)(bLen));
        SL11Write(EP0Address, addr);     // data address
        // Clear the Interrupt Status
        SL11Write(IntStatus, 0xff);
        if (pid == PID_IN && rem == 0)  // Check to arm
             SL11Write(EP0Control, (DATA0_RD));
    }
    if (pid == PID_IN)
    {
      SL11BufRead((short)((Cmd & 0x40) ? data0: data1), buf, rLen);
         //Short packet detection
         if (rem > 0)        // Remainder Packet
         {
             printf("\nShort packet detection %x\n", rem);
         return len;   // return Length
         }
      }
  ...
```

4.1.11 USB Device Detection

USB device attachment and the speed of the attached device can be determined by monitoring the SL811HS Interrupt status register bits -5 and bit-7 (IntStatus =0x0d). When a full-speed device is attached, both bits -5 and bit-7 will be set ='1'. If a low speed device is attached, then bit- 5 = '1' and bit- 7=0.

SL811HS detects HUB as a full-speed USB device. After Enumeration process, SL811HS software is able to determine the devices attached downstream of the HUB.

4.1.12 Full/Low Speed Detect

The SL811HS is able to detect attachment of full or low speed devices by monitoring bit-5, bit-6, and bit-7 of register 0x0d. The speed_detect () function detects full/low speed device attachment to the USB port. Also, this function generates the SOF/EOP for low/full speed within 1ms.

When the SL811H is connected to a hub, the software must perform the following to avoid a babble problem. This software implementation is a requirement for our chip. The developer must follow this code so they will not have a missing packet problem. Initialization is required for the interrupt.

Program Sample 11: Speed Detect Function

```
int speed_detect()
{
    full_speed = 0xffff;
    int i =0;

    SL11HMemTest();                    // Memory test
    SL11Write(IntEna, 0x63);          // USBA/B, Insert/Remove,USBRest/Resume.
    SL11Write(cSOFcnt, 0xae);          // Set SOF high counter, no change D+/D-
    SL11Write(CtrlReg, 0x48);          // Setup Normal Operation
    SL11Write(CtrlReg, i);            // Disable USB transfer operation and SOF

#ifdef SL811H
    SL11Write(cSOFcnt, 0xae);          // Set SOF high counter, no change D+/D-
    SL11Write(CtrlReg, 0x48);       // Clear SL811H mode and setup normal operation
    Delayms(10,0);                              // Delay for HW stablize
    SL11Write(CtrlReg, 0);            // Disable USB transfer operation and SOF
    Delayms(10,0);
    i =  SL11Read(IntStatus);   // Read Interrupt Status

   if(i & 0x40)
   {
       SL11Write(IntStatus,0x40);
       printf("No device or No Power\n");
   }
   if(!(SL11Read(IntStatus)&0x40))
   {
        if ((i & 0x80) == 0)           // Checking full or low speed
        {
            printf("Low Speed is detected %x\n", i);
            // Set up Master and low Speed direct and SOF cnt high=0x2e
            SL11Write(cSOFcnt,0xee);
            // SOF Counter Low = 0xe0; 1ms interval
            SL11Write(cDATASet,0xe0);
            // Setup 6MHz and EOP enable
            SL11Write(CtrlReg,0x21);
            full_speed = 0;
         }
         else
         {
            printf("Full Speed is detected %x\n", i);
            // Set up Master and low Speed direct and SOF cnt high=0x2e
            SL11Write(cSOFcnt,0xae);
            // SOF Counter Low = 0xe0; 1ms interval
            SL11Write(cDATASet,0xe0);
            // Setup 48MHz and SOF enable
            SL11Write(CtrlReg,0x05);
         }
     }
    else
    {
        SL11Write(IntStatus,0xff);
     }
     SL11Write(EP0Status,  0x50);   // Setup SOF Token
     SL11Write(EP0Counter, 0);
     SL11Write(EP0Control, 0x01);   // start generate SOF or EOP
     Delayms(25,0);                 // Hub required approx. 24.1mS
#endif
    return 0;
}
```

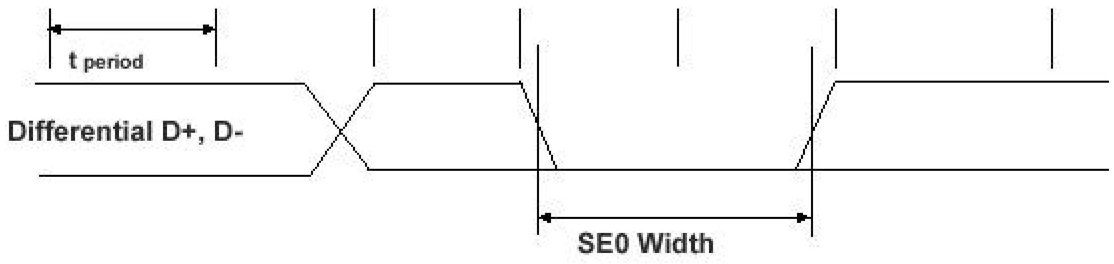
Example for doing PID_IN or PID_OUT or PID_SETUP with payload = 8;
then the Software requirement is:

```
static short low_speed_via_hub = 0;

// the low_speed_via_hub will be set high only when the HUB detect low
// speed device attach to the HUB.

    short int_cnt=0;
    short pl, Cmd=DATA0_RD, loop=30000;

    if (low_speed_via_hub)
    {
        pl = payload + 16;  // low speed
        Cmd |= 0x80;        //
    }
    else  pl = (payload>>3) + 3;

    SL11Write(EP0Status,  (BYTE)( (PID_IN << 4) + 0 ) );
    SL11Write(EP0Counter, 0);
    SL11Write(EP0XferLen, payload);  // setup 3-byte header + payload
    SL11Write(EP0Address, 0x10);     // DATA0

    while( loop-- )
    {
         int_cnt++;
         SL11Write(IntStatus,0xff);
         if (SL11Read(0xf) > (BYTE)pl)
            SL11Write(EP0Control,Cmd);        // Enable ARM
         else
            SL11Write(EP0Control,Cmd|0x20);   // Enable ARM and Wait after SOF

         // just for example only.  If CPU uses the Interrupt
         Delay1ms(2,0);



        // should check the interrupt variable "int_cnt"
         if ((SL11Read(IntStatus)&1)==0)
            printf("Missing Interrupt\n");
         else int_cnt--;
       }
       if (int_cnt)
          printf("Missing interrupt\n");
```

4.1.13 SOF or EOP Generation

The USB Specification v1.1 requires a host controller to generate an End of Packet (EOP) at regular timed intervals - nominally every 1 millisecond - to keep low speed devices alive. An EOP is defined as two single-ended zeros (SE0) followed by a J State. See figure below for EOP pulse width.

Figure 5: EOP Pulse Width



The width of the SE0 in the EOP is approximately 2 * TPERIOD. The SE0 width is measured with the same load used for maximum rise and fall times and is measured at the same level as the differential signal crossover points of the data lines.

Program Sample 12: Hardware EOP Generation

```
int hw_sof()
{
    ...                           // Need USB Reset
    SL11Write(cDATASet,0xe0);     // Set up SOF or EOP for 1ms interval
    SL11Write(cSOFcnt,0xae);      // Set up frame counter
    SL11Write(CtrlReg,05);        // Enable SOF/EOP
    SL11Write(EP0Control, 0x01); // Start SOF/EOP
}
```

An EOP is generated by within the SL811HS hardware; therefore, there is no need to use an external CPU to generate an EOP when using the SL811HS. The following is a sequence showing the SL811HS generating the EOP during Token, Data, and Handshaking phases.

Figure 6: Token Phase

SYNC	IN/OUT Token	EOP

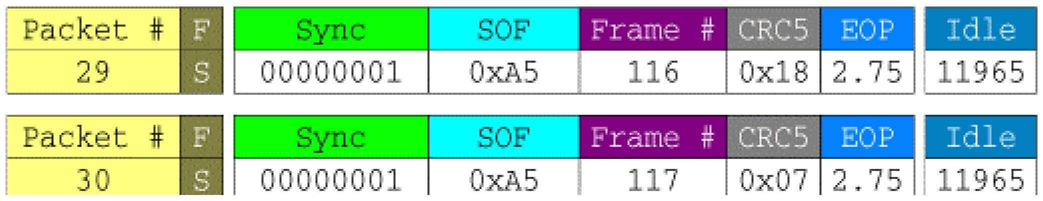
Figure 7: Data Phase

SYNC	Data Packet	EOP

Figure 8: Handshaking phase

SYNC	ACK/NAK/STALL	EOP

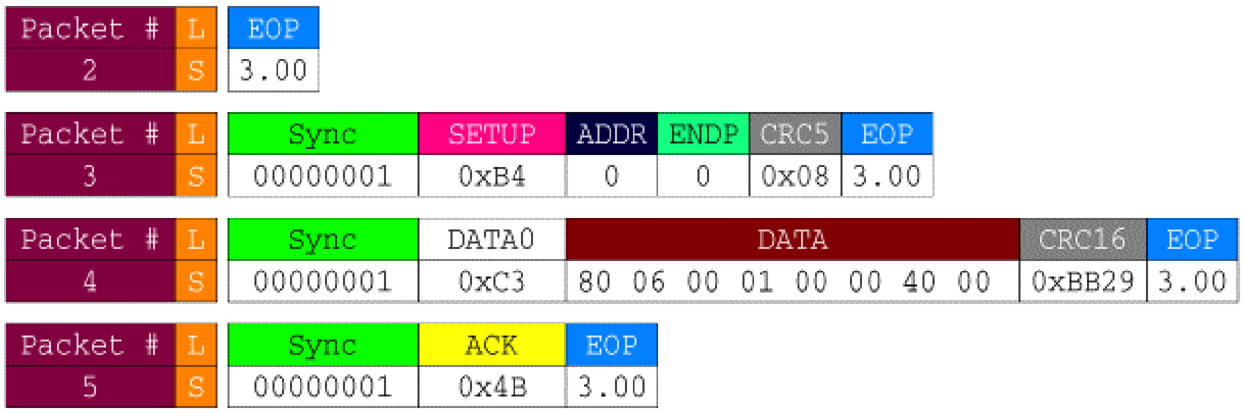
USB Trace 4: SOF Packet



Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
29	S	00000001	0xA5	116	0x18	2.75	11965

Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
30	S	00000001	0xA5	117	0x07	2.75	11965

USB Trace 5: EOP Packet



Packet #	L	EOP
2	S	3.00

Packet #	L	Sync	SETUP	ADDR	ENDP	CRC5	EOP
3	S	00000001	0xB4	0	0	0x08	3.00

Packet #	L	Sync	DATA0	DATA	CRC16	EOP
4	S	00000001	0xC3	80 06 00 01 00 00 40 00	0xBB29	3.00

Packet #	L	Sync	ACK	EOP
5	S	00000001	0x4B	3.00

4.1.14 Suspend State

When SL811HS is placed in suspend state, it will stop all activity on the USB Bus. Since no activity is seen downstream, slave devices will also suspend activity.

Program Sample 13: Suspend State

```
...
SL11Write(CtrlReg, 0);        // Stop sending SOF
SL11Write(CtrlReg,0x40);      // Suspend enable
...
```

4.1.15 Resume Wakeup

Devices that are in suspend state because of lack of USB activity, will be forced to resume operation when the SL811HS resumes SOF transmission. The SL811HS will do the following:

- Enable SOF generation
- Disable Suspend states
- Starts normal operation

Program Sample 14: Resume Wakeup

```
...
USBReset(10);               // Do USB_RESET
```

```
       // Generate SOF, setup 48Mhz, suspend disable
       SL11Write(CtrlReg, 0x01);
       ...
```

4.1.16 Specific Command Transfer

This subroutine is similar to the DeviceIoControl in the Window Driver Model. See the USB specification v1.1, chapter 9 for more details.

Program Sample 15: Vendor Command

The VendorCmd () setup specific request parameters to the USB devices.

Program Sample 16: Vendor Command

```
//-------------------------------------------------------------------
// Control endpoint
// return:
//      -1 on failure, 0 on success > 0 if remainder
//--------------------------------------------------------------------
int VendorCmd(BYTE bReq, BYTE bCmd, WORD wValue, WORD wIndex,
              WORD wLen,BYTE* pData)
{
    SetupPKG setup;

    setup.bmRequest = bReq;
    setup.bRequest  = bCmd;
    setup.wValue    = wValue;
    setup.wIndex    = wIndex;
    setup.wLength   = wLen;
    return ep0Xfer(uDev.wEPxCRC[0], uDev.wPayLoad[0], &setup, pData);
}
```

4.1.17 USB Transfer (Bulk/ISO/Control/Interrupt)

The following subroutines simulate the Micro USBD stack function on the top level.

Program Sample 17: USB Control Transfer

```
//--------------------------------------------------------------------
//   ep0Xfer
//   Dev contains the CRC of EP | Address
//   Return:
//           - Fail   -1
//           - Sucess  0
//           - non-zero   remainder (0 < wLen < setup->wLength)
//
//--------------------------------------------------------------------
int ep0Xfer(WORD Dev, WORD payload, PSetupPKG setup, BYTE *pData)
{
   BYTE pid=PID_IN;
   WORD wLen = setup->wLength;
```

```
    if   (usbXfer(Dev,   PID_SETUP,   0,   payload,   8,   (BYTE*)setup)==-1)
            return FALSE;
    if (wLen)
    {
        if (setup->bmRequest & 0x80)
        {
            pid  = PID_OUT;
            wLen = (WORD)usbXfer(Dev, PID_IN, 0, payload, wLen, pData);
            payload =0;
        }
        else
            wLen = (WORD)usbXfer(Dev, PID_OUT, 0, payload, wLen, pData);
    }
    usbXfer(Dev, pid, 0, payload, 0, NULL);
    return wLen >= 0;                    // return remainder
}
```

4.1.18 Device Enumeration

The function FindUsbDev () is used to enumerate full speed devices or Hub. The process of Enumeration includes Get_Descriptor, Set_Address, Get_Configuration, and Set_Configuration. This is standard USB Device enumeration for all full/low devices or Hubs.

Program Sample 18: Device Enumeration

The complete code is in the SL811HR.C Does this software exist and does it come standard with the SL811HS DVK? Otherwise, it should either not be here at all or this software should be made available for the SL811HS DVK.

Program Sample 19: Device Enumeration

```
//----------------------------------------------------------------------
// Find Full Speed USB Device
//----------------------------------------------------------------------

int FindUsbDev(short DevNum)
{
    pDevDesc   pDev;
    pConfDesc  pConf;
    pIntfDesc  pIntf;
    int        i,j;
    BYTE       *p, *ep;
    sEPDesc    sEP;

    if (DevNum==1) USBReset();    // root hub

    uDev.wEPxCRC[0]  = crc5(0);   // address = 0, endpoint = 0
    uDev.wPayLoad[0] = 8;

    printf("Get Device Descriptor: .. \n");
    pDev = (pDevDesc)SCMD;
    if (!GetDesc(DEVICE<<8,0x40,(BYTE*)SCMD)) return FALSE;

    printf("Set Address: %x ", DevNum);
    if (!SetAddress(DevNum)) return FALSE;

    printf("len=%02x ep0Len=%d\n", pDev->bLength, pDev->bMaxPacketSize0);
    uDev.wPayLoad[0] = (pDev->bMaxPacketSize0&0xff)?pDev->bMaxPacketSize0:8;
    printf("Get Device Descriptor: ..");
    uDev.wEPxCRC[0] = crc5((WORD)(DevNum+(0<<7)));// Endpoint 0 |assign new addr

    if (!GetDesc(DEVICE<<8,0x09, (BYTE*)SCMD)) return FALSE;
    printf("Vendor: %04x Product: %04x\n",pDev->idVendor,pDev->idProduct);
    uDev.wVID = pDev->idVendor;
    uDev.wPID = pDev->idProduct;

    printf("Get Configuration\n");
    if (!GetDesc(CONFIGURATION<<8,0xff, (BYTE*)SCMD)) return FALSE;
    if (!GetDesc(CONFIGURATION<<8,0x12, (BYTE*)SCMD)) return FALSE;
    if (!GetDesc(CONFIGURATION<<8,0x09, (BYTE*)SCMD)) return FALSE;
    if (!GetDesc(CONFIGURATION<<8,0x20, (BYTE*)SCMD)) return FALSE;

    pConf = (pConfDesc)SCMD;
    //printf("Len=%d, type: %02x\n", pConf->wLength, pConf->bType);
    //printf("Set Configuration\n");
    if (!Set_Configuration(DEVICE)) return FALSE;
    p = (BYTE*)SCMD;
    i = *p;
    pIntf = (pIntfDesc)&p[i];              // point to Interface Desc

    p = (BYTE*)&p[i+pIntf->bLength];      // point to EP Desc
    printf("Number of Endpoint = %d\n",pIntf->bEndPoints);
    uDev.bNumOfEPs = pIntf->bEndPoints;
    if (uDev.bNumOfEPs > cMaxEP) return FALSE;

    ep = (BYTE*)&sEP;
    for (i=1; i <= pIntf->bEndPoints; i++)
    {
        uDev.wEPxCRC[i] = crc5((WORD)(DevNum+(i<<7))); // Endpoint CRC
        for (j=0; j<7; j++) ep[j] = *p++;
        if (sEP.bEPAdd&0x80) Rd_EP = sEP.bEPAdd&0xf;   // index of Read EP
        else Wr_EP = sEP.bEPAdd;

        uDev.bEPAddr[i]  = sEP.bAttr;
        uDev.wPayLoad[i] = sEP.wPayLoad;

        printf("Length %x Type %x EndPoint Addr %02x Attr %02x wLength=%04x\n",
                sEP.bLength, sEP.bType, sEP.bEPAdd, sEP.bAttr, sEP.wPayLoad);
        DToggle[i] = 0;
    }
    printf("Write Endpoint %d, Read Endpoint %d\n", Wr_EP, Rd_EP);
    return TRUE;
}
```

5. SL811HS USB TRANSFER

The UsbXfer() subroutine handles USB packet transactions. It supports ping-pong operation for SETUP, PID_IN and PID_OUT Tokens. This code is currently designed to support Control pipe, Bulk Data Pipe. This code can be expanded to support ISO and Interrupt endpoints.

The usbXfer() handles USB Transactions. It will report USBStatus register contents:

```
//------------------------------------------------------------------------
// usbXfer:
// NOTE: The CRC contains Device 4-bit of EndPoint Number | 7-bit of Device Addr
// iso = 1 transfer with ISO mode, else Bulk/Interrupt/Control
// return 0 on Success
//
// USB Status
//    0x01   ACK
//   0x02   Device Error Bit
//   0x04   Device Time out
//   0x08   Toggle bit
//   0x10   SET_UP packet bit  (Not used in SL11H)
//   0x20   Overflow bit
//   0x40   Device return NAKs, forever
//   0x80   Device return STALL
//   need to keep track DATA0 and DATA1 for write endpoint
//--------------------------------------------------------------------------
int usbXfer(WORD crc, BYTE pid, BYTE iso, WORD payload, int len, BYTE *buf)
{
   short result,  bLen, rLen, time_out, data0, data1, D0, ep;
   WORD  retry;
   BYTE  Cmd=DATA0_RD, addr, rem, pl;
   extern short dbug;

   if (dbug) pl = payload + 24;  // low speed debug
   else  pl = (payload>>3) + 3;

   ep    = (crc>>7) & 0xf;
   data0 = cMemStart;

#ifdef SL811H
   SL11Write(EP0Status,  (BYTE)( (pid << 4) + ep ) );
   SL11Write(EP0Counter, (BYTE)(crc & 0x7f));

#ifdef REV13
   if (dbug && pid==PID_IN)          // handle low speed via Hub
   {
      SL11Write(EP1Status, 0xc0);
      SL11Write(EP1XferLen, 0);
   }
#endif

#else
   SL11Write(data0,pid);                                 // PID
   SL11BufWrite((short)(data0+1),(BYTE*)&crc, 2);  // 2 bytes CRC
#endif

   if (len>=(int)payload)
   {  // setup ping pong buffer
      data1 = data0 + payload + cOFFSET;             // next buffer
#ifndef SL811H
      SL11Write(data1,pid);                           // PID
      SL11BufWrite((short)(data1+1),(BYTE*)&crc,2);
#endif
      rLen = payload;
   }
   else
      rLen = (short)len;

   SL11Write(EP0XferLen,(BYTE)(cOFFSET+rLen));  // setup 3-byte header + payload
   SL11Write(EP0Address, (BYTE)data0);          // DATA0

#ifndef SL811H
   if (pid == PID_SOF)                          // Send Software SOF
   {
       SL11Write(EP0Control, 1);           // arm and return
       return len;
   }
#endif
   if (iso) Cmd |= ISO_BIT;                     // if iso setup ISO mode

   if (pid != PID_IN)
   {  SL11BufWrite((short)(data0+cOFFSET),buf, rLen);    // DATA to SL11 Memory
      Cmd = DATA0_WR;
   }
```

```
      if (ep==0 && pid != PID_SETUP) Cmd |= 0x40; else Cmd |= DToggle[ep];
      retry = 1;  rem = time_out = result = D0 = 0;
      Cmd |= dbug;                                 // set PRE for low speed
      while (1)
      {
         if (retry>0)
         {
            SL11Write(IntStatus,0xff);
            if (full_speed)
            {
               if (SL11Read(0xf) > (BYTE)pl)
                  SL11Write(EP0Control,Cmd);           // Enable ARM
               else
                  SL11Write(EP0Control,Cmd|0x20);      // Enable ARM
            }
            else
               SL11Write(EP0Control,Cmd);              // Enable ARM
   #ifdef REV13
            if (dbug && pid==PID_IN)                  // handle low speed via Hub
               SL11Write(EP1Control,1);
   #endif
         }
        retry++;
         bLen = ((int)(len - rLen) >= (int)payload) ? payload : (BYTE)(len - rLen);
         addr = (D0 & 1) ? data0 : data1;    // next ping pong buffer

         // Write ahead for ping pong buffer
         if (pid == PID_OUT && retry <= 2 && bLen)
            SL11BufWrite((short)(addr + cOFFSET), buf + rLen, bLen);

         if (Delayms(TIMEOUT, 1))
         {
            //printf("TIMEOUT: pid %x result %x\n", pid, SL11Read(EP0Status));
            result = 4;  // indicate it is time-out
         }
         else
         {
            result=SL11Read(EP0Status);
   #ifdef REV12
            if (pid==PID_IN && dbug && !(result&0x40)) // handle low speed via hub
            {
               SL11Write(IntStatus,0xff);
               SL11Write(EP0Control,9);
               Delayms(1,1);
            }
   #endif
         }
         if (result & 1)
         {
             D0++;
             retry = 0;
             Cmd ^= 0x40;                        // toggle DATA0/DATA1
            //Add for Shortpacket detection
                if (pid==PID_IN)
                 rem = (BYTE)SL11Read(EP0Counter);
             len -= rLen;    // rLen = actual read/write
             if (bLen && len && rem==0)
             {
                 SL11Write(EP0XferLen, (BYTE)(cOFFSET+bLen));
                 SL11Write(EP0Address, addr);            // data addr
                 SL11Write(IntStatus, 0xff);
                  if (full_speed)
                  {
                     if (SL11Read(0xf) > (BYTE)pl)
                        SL11Write(EP0Control,Cmd);      // Enable ARM
                     else
```

```
                        SL11Write(EP0Control,Cmd|0x20); // Enable ARM
                    }
                    else
                        SL11Write(EP0Control,Cmd);          // Enable ARM
#ifdef VER13
                   if (dbug && pid==PID_IN)              // handle low speed via hub
                        SL11Write(EP1Control,1);
#endif
                }
                if (pid==PID_IN)
                {
                   SL11BufRead((short)((D0&1)?data0+cOFFSET:data1+cOFFSET), buf, rLen);
                   if (rem) // Short Packet detection
                   {
                       //printf("Short packet detection %x\n", rem);
                     break;
                   }
                }
                 if (len == 0) break;
                rLen = bLen;
                buf += rLen;
            }
            if (result & 0x04)
            {
                 if (time_out > 100)
                 {
                     printf("WARNNING: time-out, pid=%x, status %x len %x\n", pid,
                                                                     result, len);
                     break;
                 }
                // should be 100 time retry on time_out, check the babble detection
                 time_out++;
            }
            else
                 time_out = 0;
            if (result & 0x80)    // STALL
            {
               printf("WARNNING: Stall is detected, status %x, len %x\n", result, len);
                break;
            }
            if (retry >= MAX_RETRY) break;
        }
        DToggle[ep] = Cmd & 0x40;
        if (result & 1) return len;
        printf("Exit: PID=%x Result = %x len=%d, retry %d\n",pid, result, len, retry);
        return -1;
}
```

5.1. Control Transfer

The Control pipe includes the Token (SETUP, IN, OUT), Data (DATA0/DATA1), and Handshake (ACK, NAK, STALL) phases. During the SETUP transaction, data is always sent to the USB peripheral with a specific request. Following the SETUP transaction, the SL811HS initiates one or more IN transactions, which prompts the target to return the requested data. SL811HS completes the Control Transfer by using OUT transactions to request verification that the device endpoint has successfully returned the contents of the descriptor. The Control Pipe is used for USB device enumeration.

The USB defines the allowable maximum control data payload sizes for full-speed devices to be 8, 16, 32 or 64 bytes. However, low speed devices are limited to an 8-byte maximum data payload size.

5.1.1 SETUP Transaction

The SETUP transaction is a USB Host specific command that requests USB device information whenever a USB device is attached.

USB Trace 6: SETUP Transaction

SYNC	SETUP	ADDR	ENDP	CRC5	EOP	IDLE

SYNC	DATA0	DATA PAYLOAD	CRC16	EOP	IDLE

SYNC	ACK	EOP	IDLE

Note: Data payload for SETUP Token is 8 bytes.
Depending on the wLength of the transaction descriptor, the SETUP transaction is followed by either one or more IN transactions that the host requests from the device. See example below showing an instance of more than one IN transaction.

5.1.2 Control Read

The SL811HS read IN Token from USB device, which is always started with DATA1 during the Data phase. SL811HS toggles to DATA0 then DATA1 and so on.

USB Trace 7: IN Transaction from Device to Host

To transfer 18-byte from device to host, it takes 3 IN transactions with 8-byte data payload.

PACKET #	SYNC	IN	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	8-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

PACKET #	SYNC	IN	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA0	8-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

PACKET #	SYNC	IN	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	2-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

SL811HS completes the Control Transfer by using OUT transaction with either handshaking or zero length data to request verification that the device endpoint has successfully returned the contents of the descriptor.

USB Trace 8: OUT Transaction from Host to Device

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	ZERO LENGTH PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

5.1.3 Control Write

The SL811HS writes an OUT Token to the USB device, which always starts with DATA1 during the Data phase. SL811HS toggles to DATA0 then DATA1 and so on.

USB Trace 9: OUT Transaction from Host to Device

To transfer 18-bytes from the host to device by Control transfer, it takes 3 OUT transactions with 8-byte data payload.

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	8-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA0	8-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	2-BYTE DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

The SL811HS send an IN Token during Control Write to initiate the Status Stage. The USB device responds either by a handshake or a zero length data to indicate its current status.

5.2. Bulk Transfer

The Bulk pipe includes the Token (IN, OUT), Data (DATA0/DATA1), and Handshake (ACK, NAK, STALL) phases. The USB defines the allowable maximum control data payload sizes for full-speed devices to be 8, 16, 32 or 64 bytes. However, low speed devices are limited to an 8-byte maximum data payload size.

5.2.1 BULK Write

The BULK Write endpoint's toggle sequence is initialized to DATA0/DATA1 and so on.

USB Trace 10: Bulk Write

To transfer 512-bytes from host to device by BULK transfer, it takes 8 OUT transactions with a 64-byte data payload.

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA0	DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA1	DATA PAYLOAD	CRC16	EOP	IDLE

PACKET #	SYNC	ACK	EOP	IDLE

...

PACKET #	SYNC	OUT	ADDR	ENDP	CRC5	EOP	IDLE

PACKET #	SYNC	DATA0	DATA PAYLOAD	CRC16	EOP	IDLE

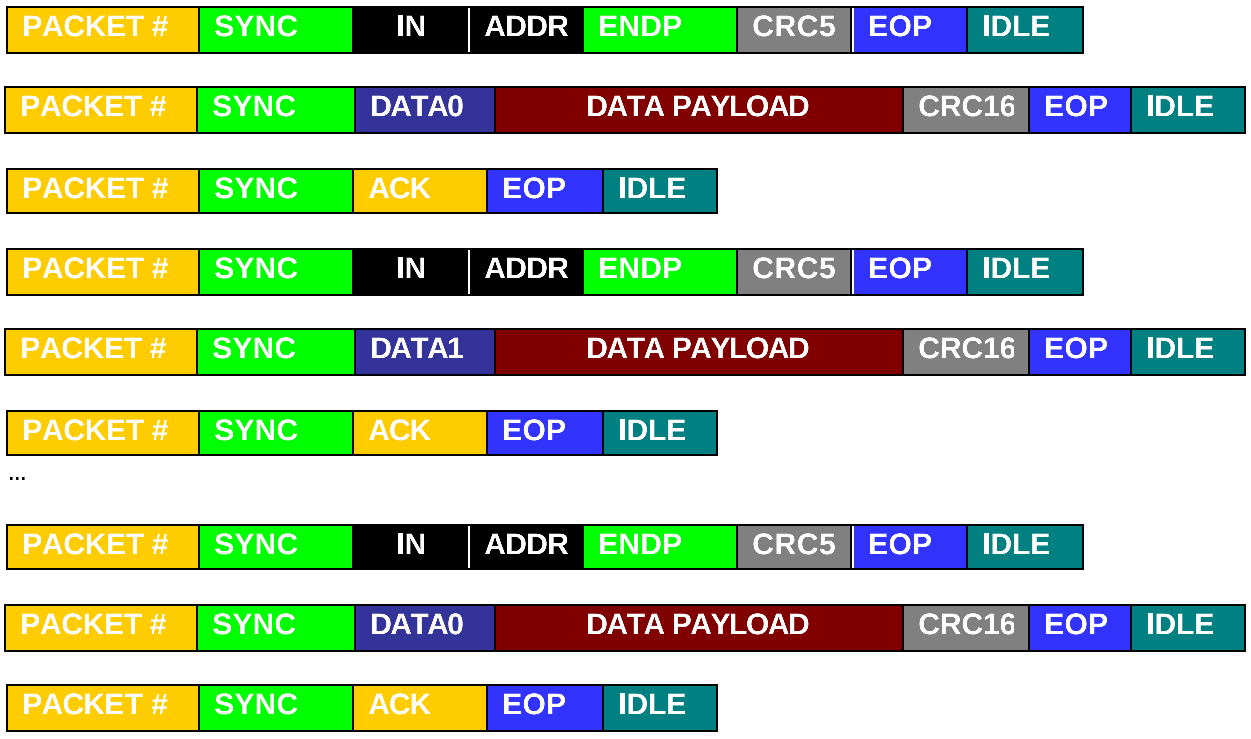
PACKET #	SYNC	ACK	EOP	IDLE

5.2.2 BULK Read

The BULK Read endpoint's toggle sequence is initialized to DATA0/DATA1 and so on.

USB Trace 11: Bulk Read

To transfer 512-bytes from device to host, it takes 8 IN transactions with a 64-byte data payload.


PACKET # | SYNC | IN | ADDR | ENDP | CRC5 | EOP | IDLE
PACKET # | SYNC | DATA0 | DATA PAYLOAD | CRC16 | EOP | IDLE
PACKET # | SYNC | ACK | EOP | IDLE
PACKET # | SYNC | IN | ADDR | ENDP | CRC5 | EOP | IDLE
PACKET # | SYNC | DATA1 | DATA PAYLOAD | CRC16 | EOP | IDLE
PACKET # | SYNC | ACK | EOP | IDLE
...
PACKET # | SYNC | IN | ADDR | ENDP | CRC5 | EOP | IDLE
PACKET # | SYNC | DATA0 | DATA PAYLOAD | CRC16 | EOP | IDLE
PACKET # | SYNC | ACK | EOP | IDLE

6. ERROR RECOVERY MANAGEMENT

Error recovery is the responsibility of the host controller. The SL811HS host controller provides error detection and status for USB transactions. Host software should be written to recover from reported error conditions.

6.1. SL811HS Error Conditions

Error conditions are reported in the Status register. An interrupt can be generated on an error condition by enabling the Interrupt bit in the Interrupt control register. Polling the USBStatus register can monitor the error conditions.

USB Status:

0x01	ACK
0x02	Device Error Bit
0x04	Device Time out
0x08	Toggle bit
0x10	SET_UP packet bit (Not used in SL11H)
0x20	Overflow bit
0x40	Device returned NAK to last transaction
0x80	Device returned STALL

Table 7: Error Conditions

Error Conditions	SL811HS
Packet ID Check	√
Bit Stuff Error	√
Bus Time-out	√
False EOP	√
CRC Check	√
Babble	√
Loss of Activity	√

6.2. Data Toggle Error

Data transfers on the USB use Data0 and Data1 packets so that the receiver can remain synchronized with the transmitter. If synchronization is lost, a Data Toggle Error condition occurs. Data Toggle Error reporting is supported for Interrupt, Bulk, and Control transfers only.

Software is required to recover from a Data Toggle Error whenever it occurs. SL811HS provides bit-3 of USBStatus Register to report whenever a Data0 and Data1 mismatch occurs.

6.3. Bus Timeout Error

A corrupted Token, Data, or handshake results in a Bus Timeout error, which is reported in the status register. This can result in a retry of the same Data toggle setting.

6.4. Babble and Loss of Activity (LOA)

On USB Bus, the Babble condition is reported in the Status register. When bit 2 is set in the interrupt enable register, the babble condition causes an interrupt. This interrupt only occurs during the period between the EOF2 and the next SOF token generation. See chapter 5 in USB Specification v1.1 for more details.

The SL811HS is able to detect a babble condition on bit-2 of the Interrupt Status (IntStatus =0x0d). When the babble condition is detected, SL811HS will be prevented from sending any packets during EOF2 time frame. SL811HS must discontinue the transactions prior to the end of packet to ensure that no bus activity occurs at the end of frame.

Loss of Activity (LOA) is similar to Babble. A device starting a packet transfer followed by a constant J or K state on the bus and no EOP characterizes LOA. LOA, like Babble, has the potential to deadlock the bus and must be prevented.