TMS320C55x CSL USB Programmer's Reference Guide

SPRU511
October 2001

PRINTED WITH SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

The TMS320C55x™ CSL USB Programmers Reference Guide provides C-program functions to configure and control on-chip Universial Serial Bus (USB) peripherals. It is intended to make it easier to get algorithms running in a real system. The information found in this document is targeted to the USB spec 1.1 compliant USB module.

This document provides reference information for the USB and is organized as follows:

- Overview – high level overview of the USB
- DSP Resource Requirements
- Module Initialization
- Data Structures
- API Routines
- Module Drivers
- Symbolic Constants
- Enumerated Data Types
- USB Data Structures
- USB Functions
- Configuration of the USB module using the CSL graphical user interface (GUI).
- Appendix that details USB Terminology

How to Use This Manual

The information in this document describes the contents of the TMS320C55x™ DSP USB Reference Guide as follows:

- ❑ Chapter 1 provides an overview of the USB, includes figures, tables, and examples showing USB module support for VC5509 devices.
- ❑ Chapter 2 provides essential USB functions with examples and descriptions of their use.
- ❑ Chapter 3 provides a USB Demo Application sample and instructions.
- ❑ Appendix A provides an overview of USB terminology.

Notational Conventions

This document uses the following conventions:

- ❑ Program listings, program examples, and interactive displays are shown in a `special typeface`.
- ❑ In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- ❑ Macro names are written in uppercase text; function names are written in lowercase.
- ❑ TMS320C55x™ DSP devices are referred to throughout this reference guide as C5501, C5502, etc.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x™ DSP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number. Many of these documents are located on the internet at http://www.ti.com.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, Code Composer, DSP/BIOS, and TMS320C5000.

Microsoft® and Windows® 2000 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

Figures

Tables

Examples

Chapter 1

USB Overview

This chapter is an overview of the components, features and benefits, routines, drivers, and configuration settings found in the USB module.

1.1 CSL USB Module Overview

Features and benefits:

- Complete hardware abstraction
- Single call device configuration
- Single API for posting all four types of transfers:
  - Control
  - Bulk
  - Interrupt
  - Isochronous
- Supported data buffers:
  - Single data buffer
  - Multiple data buffers in linked list form. All the members of the linked list can be concatenated into a single USB transfer or each member of the linked list can be posted as an individual transfer.
- USB bus events and endpoint events are broadcast to the user selected event-handler routines.

Note: One endpoint event handler routine is allowed per active endpoint. Event handler routines are bound to their respective endpoint objects during the endpoint object initialization.

1.1.1 Components of the CSL USB Module

Detailed descriptions for these components begin on page 1-4

- Data Structures:
  - Endpoint Object
  - Endpoint Data Buffer
- API Routines:
  - Software initialization
  - Module initialization
  - Module Control
  - Data Transfer
  - Status Query
- Module Drivers:
  - Data Buffer Handler
  - Event Dispatcher

1.2 DSP Memory Resource Requirements

- A typical USB application built using the CSL USB module requires 512 bytes of software stack and 512 bytes of system stack.
- The upper 256 bytes shared RAM of the USB module are reserved for use by the CSL USB components for internal variables. These 256 bytes of USB shared RAM can not be used as an endpoint data buffer.
- Each Endpoint object requires twenty words of DSP data memory.

1.3 USB Components Overview

1.3.1 Data Structures

Endpoint Object

Endpoint objects are the starting point of building a USB application using the USB Module Support Library.

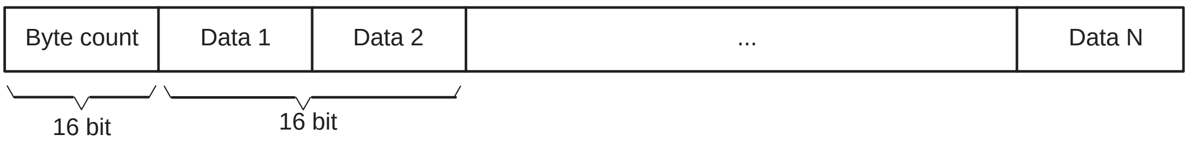
In an application, every active USB endpoint must be represented by initialized endpoint objects. An initialized endpoint object holds the Runtime characteristics of a physical endpoint. The total number of endpoint objects in an application depends on the number of physical endpoints the application intends to use.

The endpoint objects can be initialized either by the CSL USB GUI, or by the endpoint initialization API. The initialized endpoint objects are bound to the respective endpoints by the USB module initialization API. Once the endpoint objects are successfully bound, the application can use a handle (pointer) to the endpoint object to communicate with the endpoint. A collection of initialized endpoint objects represents a USB configuration.

Endpoint Data Buffer

The USB driver supports both single and multiple data buffers. Multiple data buffers are supported as a linked list. Figure 1–1 illustrates the format of the endpoint data buffer supported by the USB driver.

Figure 1–1. Endpoint Data Buffer Format



Each 16-bit word of DSP data memory holds two bytes of USB data. The USB driver uses the very first word of the data buffer to store the number of bytes being transmitted or received. The format of the data buffers used for IN and OUT endpoints is the same. The following equation can be used to determine the effective length of the USB data buffer:

Buffer Length = 1 + int[(n +1)/2] words; Where n = number of bytes

The USB driver also supports a NULL buffer pointer to send and receive 0-byte handshake packets to indicate the closure of a setup packet. The NULL buffer pointer is a special case and can only be used with the control endpoints (endpoint0).

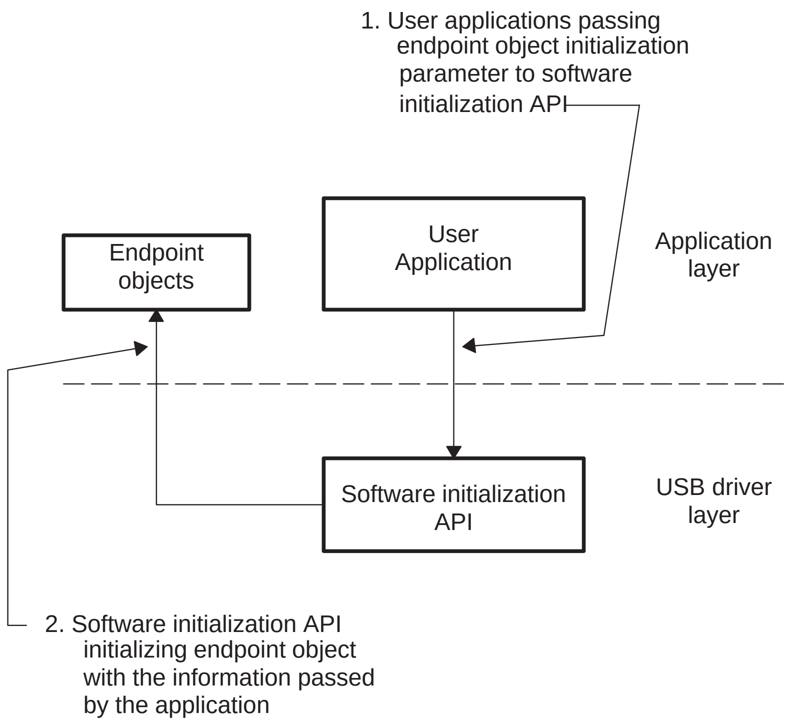
1.3.2 API Routines

Software initialization

The software initialization API initializes the endpoint objects. The application calls the software initialization API with the parameters such as handle to an endpoint object, endpoint number, endpoint type (control, bulk, interrupt, isochronous, and host port), endpoint size, and pointer to the endpoint event handler routine. The software initialization API updates the endpoint object based on the parameters passed.

The CSL USB GUI can also be used to initialize endpoint objects. The CSL USB GUI initialize endpoint objects statically, whereas the software initialization API initializes an endpoint object dynamically. Figure 1–2 illustrates the relationship between the USB software initialization API and a USB application.

Figure 1–2. Software initialization



Module initialization

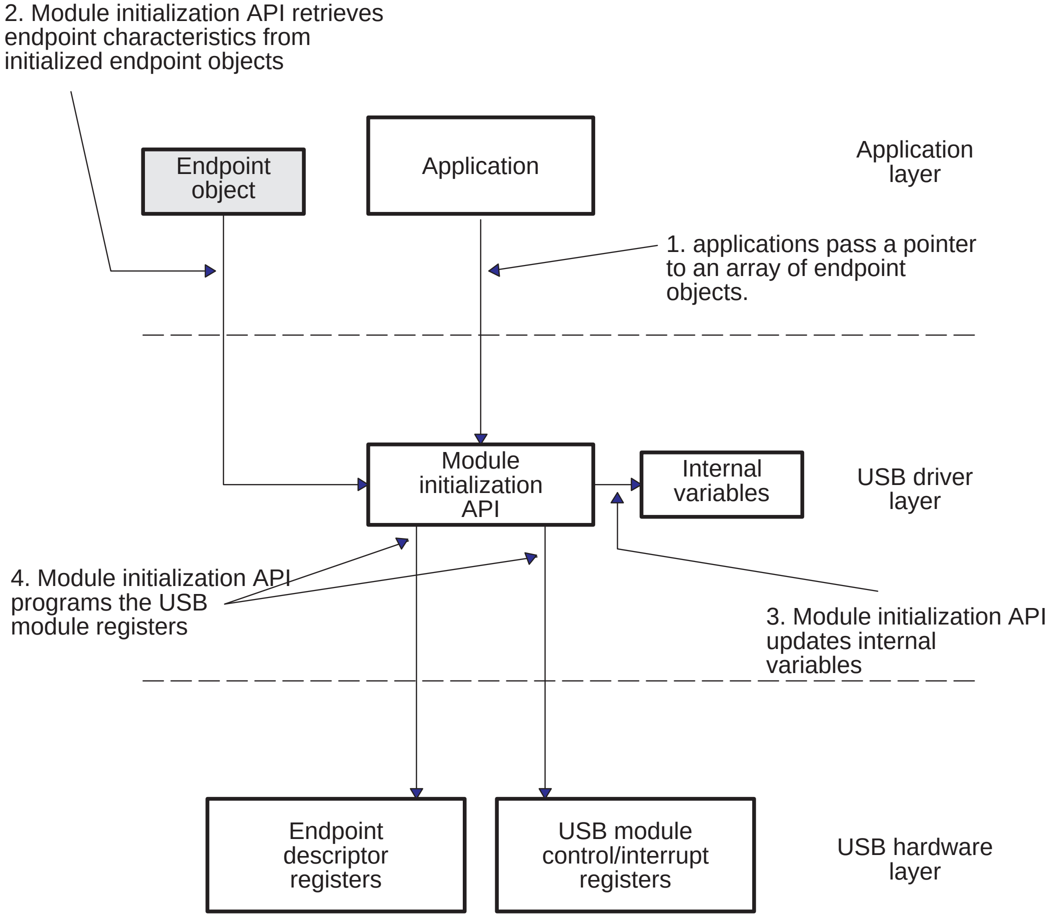
The module initialization API, based on the endpoint characteristics defined by the initialized endpoint objects, programs the endpoint descriptor (registers), and the USB module control and interrupt enable registers. The initialized endpoint objects are passed to the module initialization API as a pointer to a NULL terminated array of handles (pointers to the endpoint objects).

The module initialization API stores a copy of the endpoint handles internally for runtime use by the USB driver. Upon successful execution of the module initialization API, the USB module is initialized to the configuration defined by the endpoints passed as the array of endpoint handles.

The USB Module initialization API configures the USB Module, but does not connect the module to the bus. The USB module is connected to the bus by calling the appropriate module control API.

Figure 1–3 illustrates the USB module initialization.

Figure 1–3. Module initialization



Note: Gray backgrounds represent components that are indirectly involved in the process.

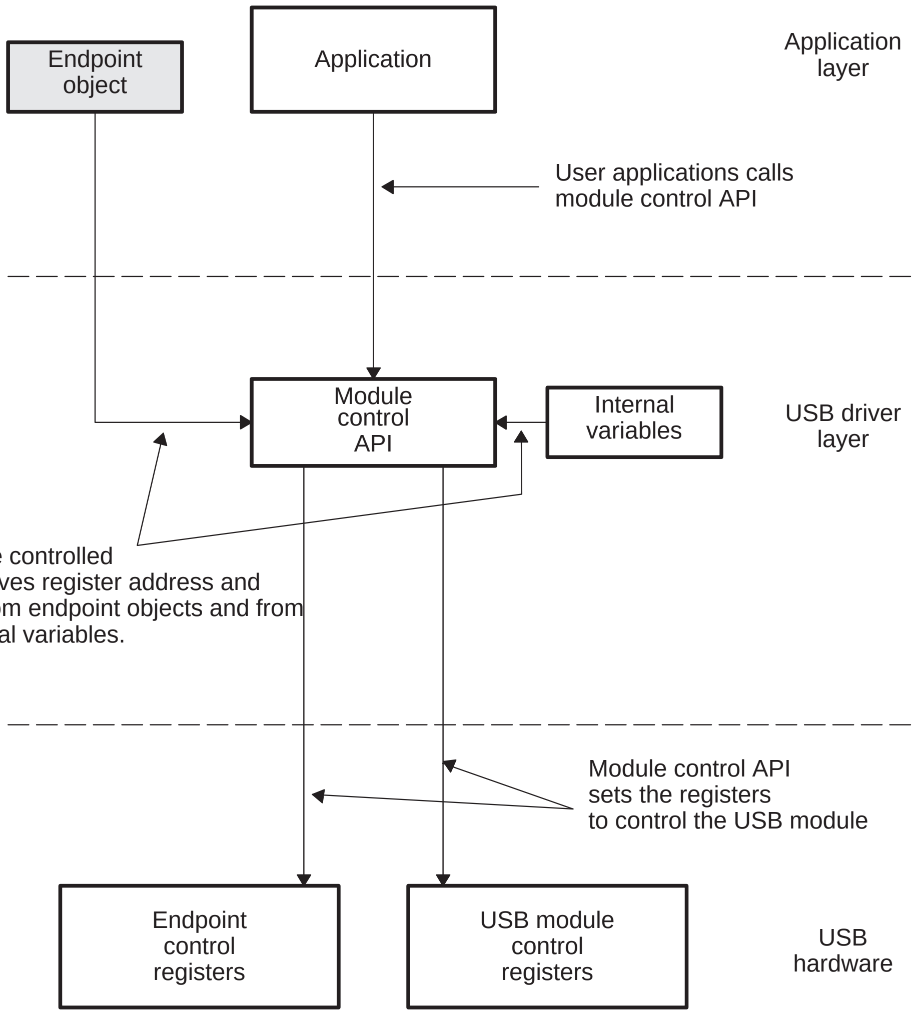
Module Control

The module control APIs allow the user application to control the hardware features of the USB module. Some of the features that can be controlled by the module control APIs are:

- Connect/disconnect from the bus
- Setting of the device address
- Stalling of an endpoint

The module control APIs access the internal variables and the endpoint objects associated with the current USB device configuration to determine the values and addresses of the registers to be modified. Figure 1–4 illustrates USB Module Control API.

Figure 1–4. Module Control



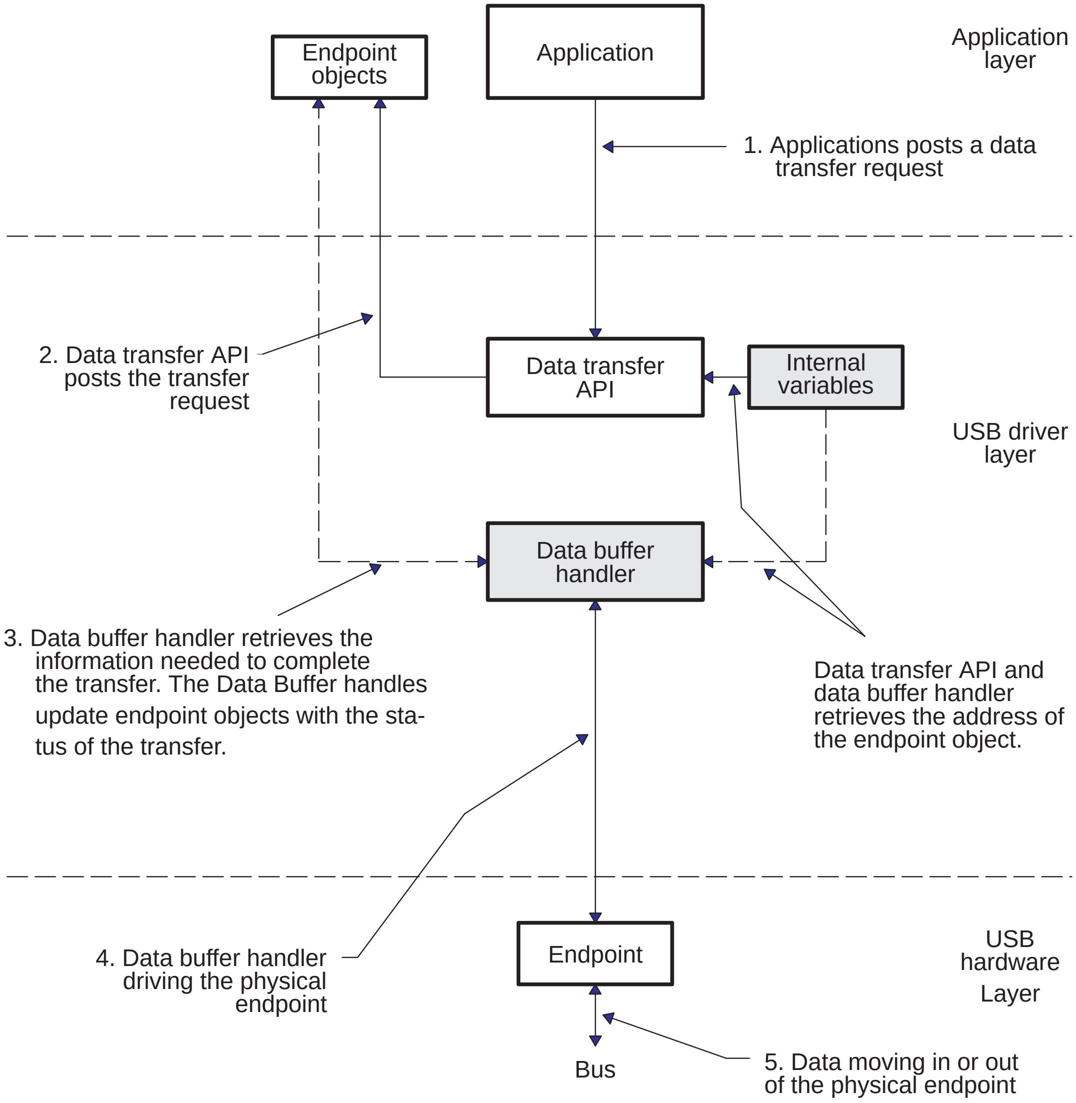
Note: Gray backgrounds represent components that are indirectly involved in the process.

Data Transfers

The data transfer API allows an application to send and receive USB data (single buffer or linked list of multiple buffers) through any active endpoint defined in the current configuration. If there is no transfer in progress, the data transfer API posts the request to the data buffer handler module of the USB driver.

Upon completion of the transfer, the data buffer handler sets the End-of-Transfer event flag. in the associated endpoint object. The application can query the status of a posted data transfer by calling the appropriate status query API.

Figure 1–5. Data Transfer



Note: Gray backgrounds represent components that are indirectly involved in a process.

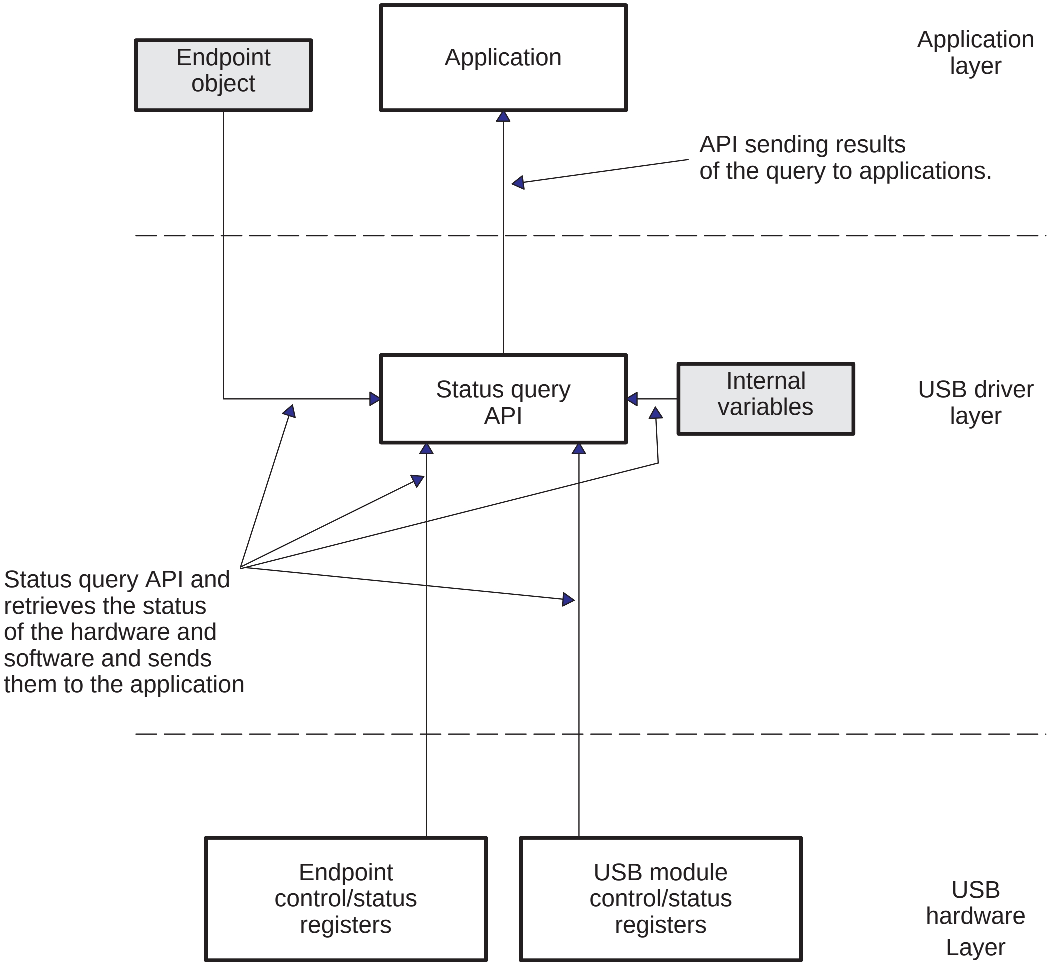
Status Query

The status query API posts the application with runtime software and hardware status of the USB module.

Remote wakeup, current USB frame number, and endpoint stall are examples of the types of status that an application may query during runtime.

Depending on the type of query, the status query APIs access the appropriate USB resource (hardware or software) and returns the requested status to the application.

Figure 1–6. Status Query



Note: Gray backgrounds represent components that are indirectly involved in a process.

1.3.3 Module Drivers

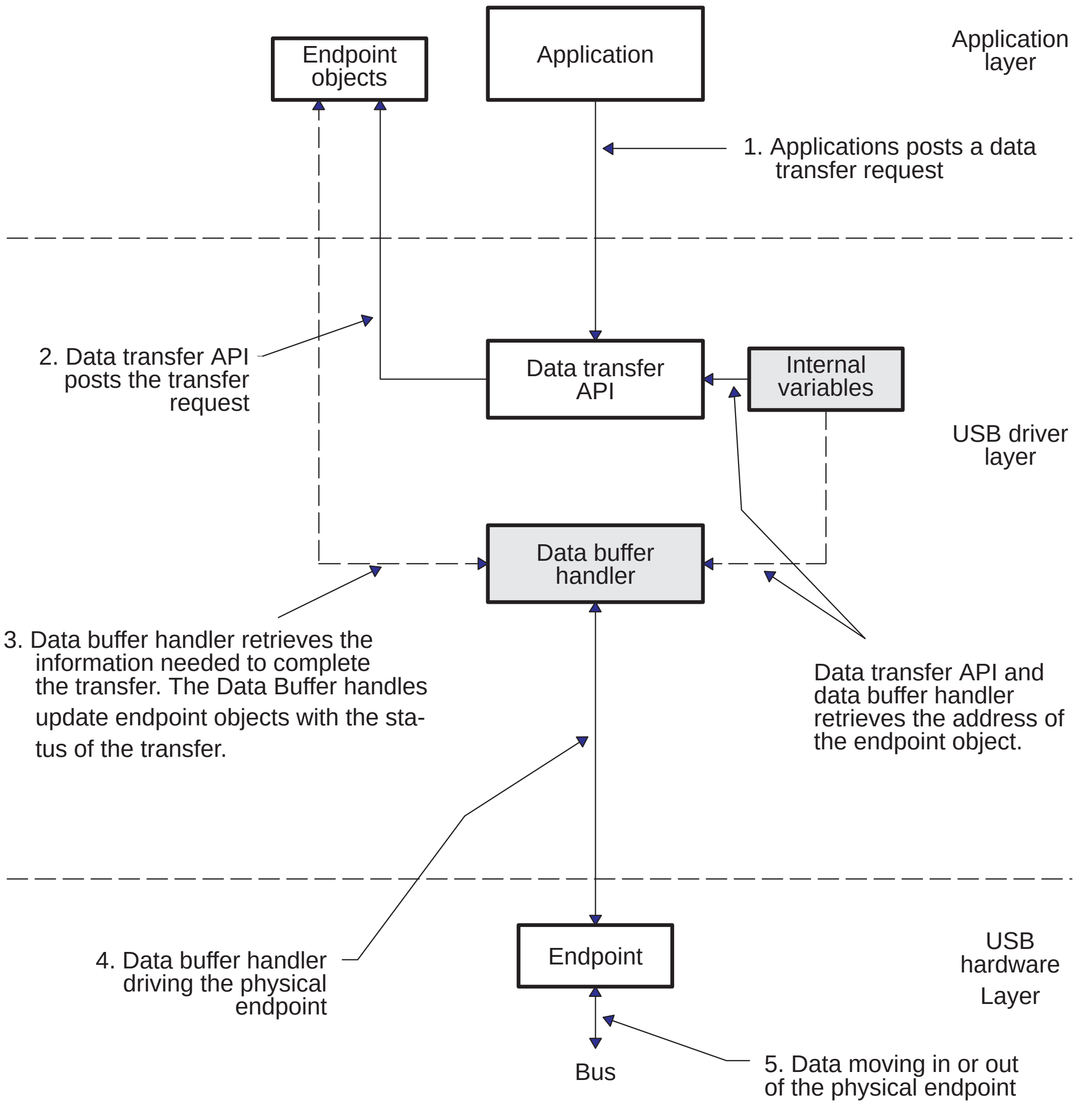
Data Buffer Handler

The data buffer handler sends and receives data that has been posted by the data transfer API.

Data buffers are handled on an endpoint basis. The endpoint characteristics defined during the endpoint object initialization and the data transfer qualification flags set by the application determine the way each transfer should be handled. If necessary, the data buffer handler breaks down each transfer into multiple packets based on the maximum packet size supported by the endpoint.

Upon completion of a transfer, the data buffer handler sets the End-Of-Transfer flag in the associated endpoint object. The application can query the status of the posted data transfer by using the status query API. The data buffer handler uses DMA channels to move data in and out of the general-purpose endpoints (endpoints 1-7). The control (endpoint0) transfers are handled by dedicated data handler routines; hence they require more CPU overhead than those of general-purpose endpoints. Figure 1–7 illustrates the data buffer handler.

Figure 1–7. Data Buffer Handler



Note: Gray backgrounds represent components that are indirectly involved in a process.

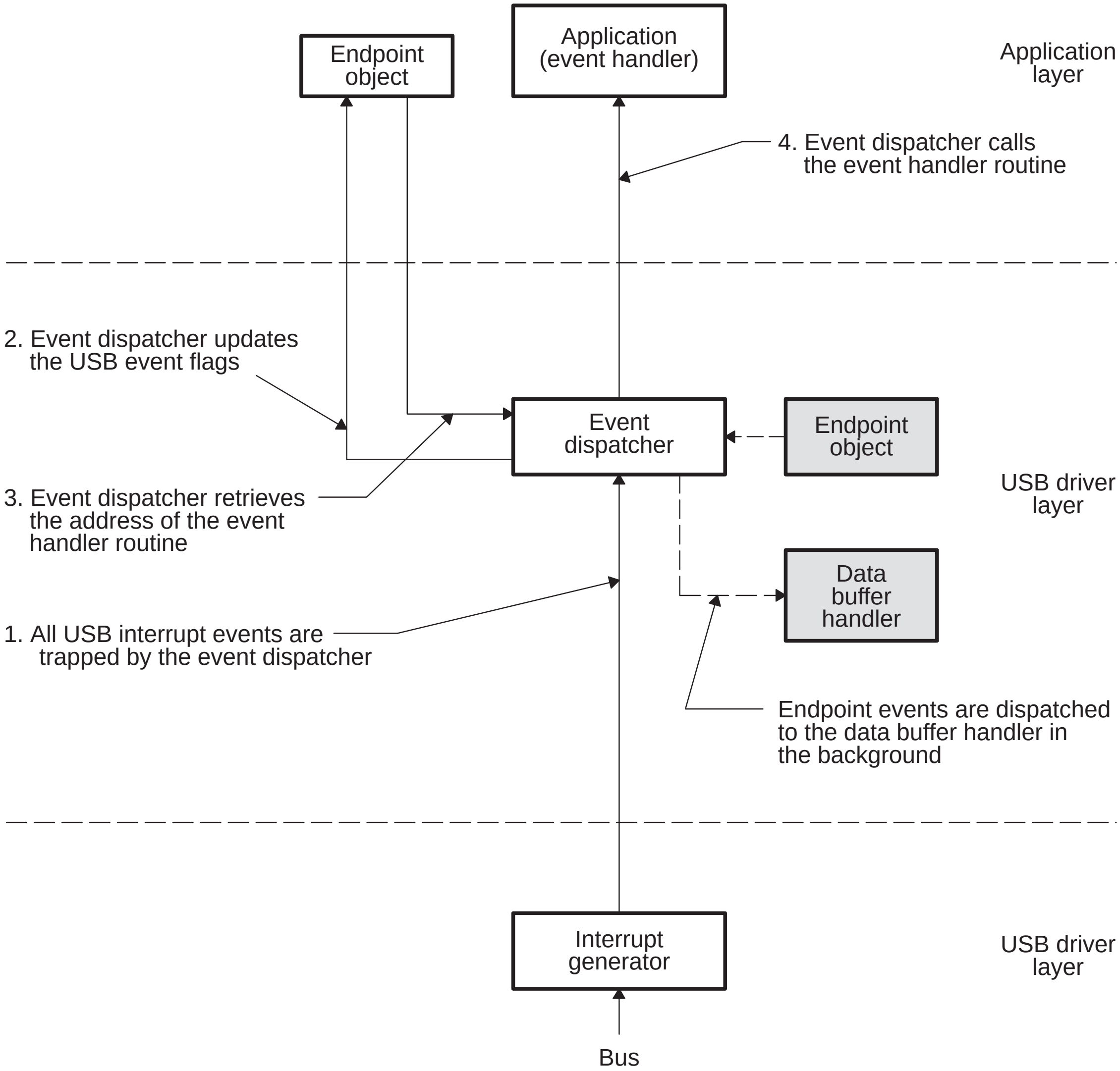
Event Dispatcher

The event dispatcher traps the USB bus events (bus reset, setup packet received, suspend request, etc.) and the endpoint event (end of data transfer) and broadcasts them to the user-selected event handler routines. Event handler routines are bound with the endpoint objects during the endpoint object initialization. Only one event handler routine is allowed per active endpoint.

In the background, the event dispatcher also broadcasts the endpoint events to the data buffer handler. The Data Buffer Handler depends on the End-of-Transfer events to drive data in and out of the active USB endpoints. Figure 1–8 illustrates the event dispatcher into action.

Figure 1–8. Event Dispatcher

Note: Gray backgrounds represent components that are indirectly involved in a process.



1.4 USB Configuration and Interfaces

The USB driver supports single configurations with single interface. At any time, a configuration is represented by a collection of initialized endpoint objects.

An application can define multiple configurations by creating multiple arrays of initialized endpoint objects. You can easily switch among configurations by:

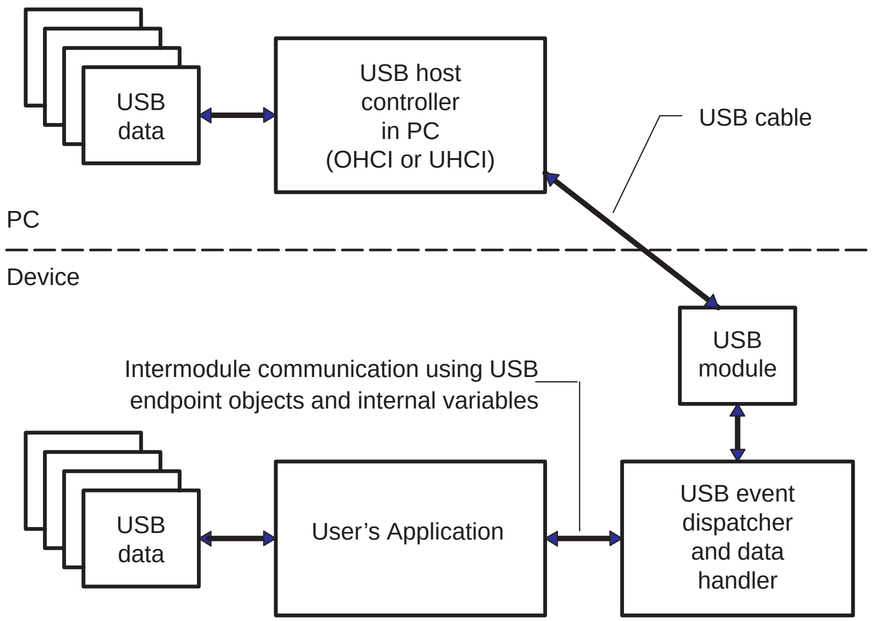
Step 1: Resetting the USB module

Step 2: Calling the hardware initialization API with the array of endpoint objects representing the desired configuration.

1.5 Typical Application Using The CSL USB Module Support Library

Figure 1–9 illustrates a generic representation of a Universal Serial Bus (USB) application using the various components available to you via the USB Module Support Library Components.

Figure 1–9. Generic USB Application using CSL USB Module Support Library Components



Chapter 2

CSL USB Module Components

This chapter contains descriptions and examples for the symbolic constants, enumerated data types, data structures and the API routines for the CSL USB module.

Topic **Page**

2.1 Symbolic Constants

2.1.1 USB Data Transfer Flags

These qualifying flags are used with the USB Data Transfer API.

Table 2–1. USB Data Transfer Flags

Constant	Description
USB_IOFLAG_NONE	Default value
USB_IOFLAG_NOSHORT	Does not expect or insert a 0-byte packet after a full size packet.
USB_IOFLAG_SWAP	Swaps hi/lo bytes before data is transmited or after data is received
USB_IOFLAG_LNK	Data buffer (transmit or receive) passed is a linked list
USB_IOFLAG_CAT	Concatenates multiple data buffers (linked list) into a single transfer.
USB_IOFLAG_EOLL	Ignores the argument ByteCnt, transfers ends when the end of the linked list is reached

2.1.2 USB Interrupt Events

These Symbolic Constants are used by the CSL graphical user interface (GUI) and by the CSL USB Module functions to initialize the Endpoint objects.

Table 2–2. USB Interrupt Events

Constant	Description
USB_EVENT_NONE	No interrupt is received
USB_EVENT_RESET	Bus Reset
USB_EVENT_SOF	Start of Frame
USB_EVENT_SUSPEND	Bus Suspend
USB_EVENT_RESUME	Bus Resume
USB_EVENT_SETUP	Setup Packet Received
USB_EVENT_EOT	End of posted transfer

Table 2–2. USB Interrupt Events(Continued)

Constant	Description
USB_EVENT_STPOW	Setup Packet Overwrite
USB_EVENT_PSOF	Pre-Start of Frame
USB_EVENT_HINT	Host Interrupt
USB_EVENT_HERR	Host Error

NOTE: USB_EVENT_EOT is not an actual hardware interrupt. This flag is set by the USB event dispatcher to indicate the completion of the latest posted transaction for an endpoint.

2.2 Enumerated Data Types

2.2.1 Endpoint Numbers

Endpoint Numbers are used by either the CSL USB GUI or the CSL USB Module functions to initialize the USB Endpoint Objects.

USB_EpNum

Data Type USB_EpNum;

Members

OUT Endpoints

USB_OUT_EP0 = 0x00	/* Out Endpoint 0 – Control Out Endpoint */
USB_OUT_EP1 = 0x01	/* Out Endpoint 1 */
USB_OUT_EP2 = 0x02	/* Out Endpoint 2 */
USB_OUT_EP3 = 0x03	/* Out Endpoint 3 */
USB_OUT_EP4 = 0x04	/* Out Endpoint 4 */
USB_OUT_EP5 = 0x05	/* Out Endpoint 5 */
USB_OUT_EP6 = 0x06	/* Out Endpoint 6 */
USB_OUT_EP7 = 0x07	/* Out Endpoint 7 */

IN Endpoints

USB_IN_EP1 = 0x09	/* IN Endpoint 1
USB_IN_EP0 = 0x08	/* IN Endpoint 0 – Control IN Endpoint */
USB_IN_EP2 = 0x0A	/* IN Endpoint 2 */
USB_IN_EP3 = 0x0B	/* IN Endpoint 3 */
USB_IN_EP4 = 0x0C	/* IN Endpoint 4 */
USB_IN_EP5 = 0x0D	/* IN Endpoint 5 */
USB_IN_EP6 = 0x0E	/* IN Endpoint 6 */
USB_IN_EP7 = 0x0F	/* IN Endpoint 7 */

2.2.2 USB Transfer Types

The USB Transfer types are used by either the CSL USB GUI or CSL USB functions to initialize the USB Endpoint Objects.

USB_XferType

Data Type

USB_XferType;

Members

Transfer Type

```
USB_CTRL   = 0x00   /* Endpoint functions as a control endpoint           */
USB_BULK   = 0x01   /* Endpoint functions as a bulk endpoint.             */
USB_INTR   = 0x02   /* Endpoint functions as an interrupt endpoint.       */
USB_ISO    = 0x03   /* Endpoint functions as an isochronous endpoint.     */
USB_HPORT  = 0x04   /* Endpoint functions as  a Host Port                 */
```

NOTE: USB_HPORT is a special feature and is not a part of USB specifications. For details on host port mode, please refer to the *TMS320C55x DSP Peripheral Reference Guide (SPRU317B)*.

2.2.3 USB Device Number

Device numbers are used by Device Control, Status Query and Data Transfer functions.

USB_DevNum

Data Type USB_DevNum;

Members

Device Number

USB0 = 0x00	/* 1st USB module */
USB1 = 0x01	/* 2nd USB module – Use only if the DSP supports two USB Modules. */
USB2 = 0x02	/* 3rd USB module – Use only if the DSP supports three USB Modules. */

NOTE: At this time, USB0 is the only supported device.

Comments Device Numbers are implemented to support multiple USB modules in a single DSP. Currently, only USB0 is supported.

Example None

2.2.4 USB Boolean

USB_Boolean

Data Type USB_boolean

Members

USB_FALSE = 0,
USB_TRUE = 1

Comments None

Example None

2.3 USB Data Structures

Every active USB endpoint is associated with an endpoint object that keeps track of the endpoint related initialization and runtime information.

2.3.1 USB Setup Packet

USB_SetupStruct *Data Structure to hold USB setup packet*

Structure USB_SetupStruct

Members int New /* New = 1, Structure holds new setup packet */

```
Uint16    bmRequest Type    /* Byte 0 of setup packet          */
Uint16    bRequest          /* Byte 1 of setup packet          */
Uint16    wValue            /* Byte 2 and 3 of setup packet */
Uint16    wIndex            /* Byte 4 and 5 of setup packet */
Uint16    wLength           /* Byte 6 and 7 of setup packet */
```

Comments Data structure to hold the USB setup packet. A function call to USB_getSetupPacket (USB_DevNum DevNum, USB_SetupStruct *USB_Setup) returns the most recent setup packet.

Example None

2.3.2 Data Structure

USB_DataStruct *Data Structure to send and receive USB data as a linked list*

Structure USB_DataStruct

Members

```
Uint16                      Bytes;         /* Total number of bytes in the buffer */
Uint16                      pBuffer;       /* pointer to the start of buffer          */
struct USB_DataStructDef    *pNextBuffer;  /* pointer to the next structure           */
```

Comments USB_DataStruct is used by the USB Data Transfer API to send and receive USB data in linked list form.

2.3.3 USB Endpoint Object

The Endpoint Objects hold USB endpoint related initialization and runtime information.

USB_EpObj — *Data Structure for USB Endpoint Objects*

Structure USB_EpObj, *USB_EpHandle;

Members

```
USB_EpNum        EpNum;         /* USB endpoint number                      */
USB_XferType     XferType;      /* USB xfer type supported by the endpoint  */
Uint16           MaxPktSiz;     /* Max pkt size supported by the endpoint   */
Uint16           EventMask;     /* OR'ed value of USB_EVENTS. The USB       */
                                /* event dispatcher will call the           */
                                /* ISR if the  event  matches the EventMask */
USB_EVENT_ISR    Fxn;           /* Pointer to USB event ISR                 */
Uint16           DataFlags;     /* OR'ed combination of USB_DATA_IN         */
                                /* OUT_FLAGS                                */
Uint16           Status;        /* Reserved for future use                  */
Uint16           EDReg_SAddr;   /* Endpoint desc reg block start addr. 2 regs */
                                /* for EP0 and 6 regs for all others.       */
Uint16           DMA_SAddr;     /* DMA reg block start addr. Used ONLY for  */
                                /* EP1 - EP7                                */
Uint16           TotBytCnt;     /* Total number of bytes to xfer            */
Uint16           BytInThisSeg;  /* # of bytes in the active node of the linked list */
Uint16           *pBuffer;      /* Active data buffer pointer */
USB_DataStruct   *pNextBuffer;  /* Pointer to the next node of the linked list */
Uint16           EventFlag;     /* Flag to indicate the event that caused the */
                                /* USB interrupt                            */
```

Comments User's code should not modify Endpoint Objects directly.

2.4 USB Functions

Table 2–3. Summary of USB API functions

(a) USB Event Dispatcher

Function	Purpose	See page...
USB_evDispatch	Sets the USB event flags	2-11

(b) Software Initialization

Function	Purpose	See page...
USB_initEndptObj	Initializes an endpoint object	2-12
USB_setAPIVectorAddress	Initializes the USB API vector pointer	2-14

(c) Software Control

Function	Purpose	See page...
USB_setRemoteWakeup	Sets or clears the remote wakeup feature	2-15
USB_abortTransaction	Aborts a Data transfer in progress	2-16
USB_abortAllTransaction	Aborts all Data transfers in progress	2-16

(d) Module Initialization

Function	Purpose	See page ...
USB_init	Initializes the USB Module	2-18

(e) Data Transfer

Function	Purpose	See page ...
USB_getSetupPacket	Reads the setup packet from the setup data buffer	2-24
USB_postTransaction	Transmits or receives USB data through an endpoint	2-25

(f) Module Control

Macro	Purpose	See page ...
USB_initPLL	Initializes the USB PLL module	2-20
USB_connectDev	Connects the USB module to the upstream port	2-20
USB_disconnectDev	Disconnects the USB module from the upstream port	2-21
USB_issueRemoteWakeup	Issues a remote wakeup signal to the host	2-21
USB_resetDev	Reset the USB module	2-22
USB_setDevAddr	Sets the USB device address	2-22
USB_stallEndpt	Stalls an endpoint	2-23
USB_clearEndptStall	Clears an endpoint stall	2-23

(g) Status Query

Function	Purpose	See page ...
USB_getFrameNo	Returns the current Frame Number	2-30
USB_getEvents	Reads and clears all pending USB_EVENTS for an endpoint	2-30
USB_getRemoteWakeupStat	Gets the status of the remote wakeup feature	2-32
USB_peekEvents	Reads all the pending USB_EVENTS for an endpoint	2-32
USB_isTransactionDone	Returns the Status of a previously posted data transfer request	2-34
USB_bytesRemaining	Returns the number of bytes awaiting transfer	2-35
USB_getEndptStall	Determines if an endpoint is stalled	2-36

(h) Miscellaneous functions

Function	Purpose	See page ...
USB_epNumToHandle	Returns a handle to an endpoint object	2-37

2.4.1 USB Events Dispatcher

USB_evDispatch *Traps and broadcasts USB events to user selected event handler routines*

Function void USB_evDispatch(void);

Category Module Driver

Arguments None

Return Value None

Comments Any USB application build on CSL USB component must use this USB event dispatcher function to handle the USB interrupts. There are two ways to use this function. The first method is interrupt polling. This is where the user's code polls the USB interrupt flag bit periodically and calls the USB Event Dispatcher function every time the USB interrupt flag is set. The second method is to encapsulate the USB Event Dispatcher function in an ISR and set up the DSP interrupt vector table to service this ISR every time a USB event occurs.

Example 1 Polling Method:

```
if(IFR0 & IFR0_USBMSK)
  {
    IFR0 |= IFR0_USBMSK; /* Clear USB interrupt flag */
     USB_evDispatch();   /* Handle all USB events */
   }
```

Example 2 ISR Method:

```
interrupt void USB_ISR(void)
{
 USB_evDispatch();  /* call USB event dispather to */
                    /*  handle all USB events       */

}
```

2.4.2 Software Initialization

USB_initEndptObj *Initializes an endpoint object*

Function

```
USB_Boolean USB_initEndptObj (USB_DevNum      DevNum,
                              USB_EpHandle    hEp,
                              USB_EpNum       EpNum,
                              USB_XferType    XferType,
                              Uint16          MaxPktSiz,
                              Uint16          EvMsk,
                              USB_EVENT_ISR   Fxn);
```

Category Software Initialization

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.
hEp	Handle or a pointer to an initialized endpoint object
EpNum	USB endpoint number, enumerated data type of USB_EpNum
XferType	Type of data transfer to be supported by the endpoint, enumerated data type of USB_XferType.
MaxPktSiz	Max data packet size supported by the endpoint
EvMsk	ORed combination of USB Interrupt Events to be broadcasted to the associated Endpoint event handler
Fxn	Associated Endpoint event handler routine

Return Value USB_TRUE if the initialization was successful; otherwise, USB_FALSE is returned.

Description Initializes an endpoint object so that it may be used with other USB functions at a later time.

Comments The event handler must be in void Fxn (void) form. **Do not use Interrupt Pragma to define a USB Event handler routine**. Once the program control branches to Fxn, the code is free to call other functions or post a DSP/BIOS software interrupt.

This function can be replaced by using the C55x Chip Support Library GUI to create an initialized endpoint object.

Example The following example initializes an endpoint object endpoint 0 Out (control OUT endpoint):

```
/* create an instance of an endpoint object */
USB_EpObj EndptObjOut0;

/* USB driver will call the event handler routine */
/* associated with this endpoint if any of the    */
/* following events are detected                  */
Uint16 event_mask = USB_EVENT_RESET | USB_EVENT_SETUP |
                    USB_EVENT_SUSPEND | USB_EVENT_RESUME |
                    USB_EVENT_EOT;

extern void Endpt0EvHndler( );  // Endpoint0 event handler
                                   routine

USB_initEndptObj(USB0,          // endpoint is associated with
                                  USB0 module

&EndptObjOut0,                  // handle to endpoint object

USB_OUT_EP0,                    // endpoint associated with
                                   the endpoint object

USB_CTRL,                       // transfer type this endpoint
                                  will support

0x40,                           // max packet the endpoint can
                                  handle

event_mask,                     // call the event handler if
                                   these events are detected

Endpt0EvHndler);                // endpoint even handler
                                   routine
```

USB_setAPIVectorAddress

Initializes the API vector pointer

Category
Software Initialization

Function
void USB_setAPIVectorAddress()

Arguments
None

Return Value
None

Comments
USB_setAPIVectorAddress allows the user application to access the CSL USB API via a relocatable function call table.

USB buffer RAM locations 0x667E and 0x667F are reserved to point to the API Vector Table. These are 8-bit locations and hold the two buytes of a 24-bit address. The lower byte is assumed to be 0, thus forcing the table to be allocated on a 256-byte boundary.

Before you call a CSL USB function, you must initialize the API Vector pointer by calling USB_setAPIVectorAddress(). Failure to initialize the API Vector pointer will result in a malfunction within the application.

Example

```
void USB_setAPIVectorAddress()
```

2.4.3 Software Control

USB_setRemoteWakeup *Sets or clears the remote wakeup feature*

Function void USB_setRemoteWakeup(USB_DevNum DevNum, USB_Boolean RmtWkpStat);

Category Software Control

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.
RmtWkpStat:	If USB_TRUE, the driver sets the remote wakeup feature and a subsequent call to USB_issueRemoteWakeup( ) causes the driver to generate a remote signal on the bus. If USB_FALSE, the driver clears the remote wakeup feature and a subsequent call to USB_issueRemoteWakeup( ) does not generate a remote signal on the bus.

Return Value None

Comments The Host must set the remote wake up feature first. An application must verify if the remote wakeup feature is set before generating a remote wakeup signal.

Example The following example enables the remote wakeup feature for USB0:

```
USB_setRemoteWakeup(USB0, USB_TRUE);
```

The following example disables the remote wakeup feature for USB0:

```
USB_setRemoteWakeup(USB0, USB_FALSE);
```

USB_AbortAll-Transaction

Aborts all data transfers

Function USB_Boolean USB_abortAllTransaction(USB_DevNum DevNum);

Category Software Control

Arguments DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value USB_TRUE if all transfers have been successfully terminated; otherwise, USB_FALSE is returned.

Description USB_AbortAllTransaction terminates all data transfers in progress and makes endpoints free for new data transfer requests.

Comments None

Example The following example aborts all transfers in progress via the USB0 module and frees up the endpoints to post new data transfer requests:

```
USB_abortAllTransaction(USB0);
```

USB_Aborttran-saction

Aborts a data transfer in progress

Function USB_Boolean USB_abortTransaction(USB_EpHandle hEp);

Category Software Control

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value USB_TRUE if transfer has been successfully terminated; otherwise, USB_FALSE is returned.

Description Terminates a data transfer in progress, allowing a new data transfer request to be posted.

Comments The endpoint handle determines the endpoint associated with the data transfer in progress.

Example

The following example aborts a transfer in progress through endpoint 6 OUT and requests a new transfer to fill up usb_OutData2 buffer with 17 bytes of data from the host:

```
Uint16 usb_OutData2[33];    /* 2+64 byte buffer                 */
                      .     /* the first two bytes indicate the */
                      .     /* actual number of bytes received  */
if(!USB_isTransactionDone(&EndptObjOut6))
                                 /* if transfer in progress */
        USB_abortTransaction((&EndptObjIn6)
/* abort the transfer          */

USB_postTransaction(&EndptObjOut6, 17, &usb_OutData2,
USB_IOFLAG_NONE);
```

2.4.4 Module Initialization

USB_init *Initializes the USB_Module to operation mode*

Category Module Initialization

Function

```
USB_Boolean USB_init (USB_DevNum    DevNum,
                      USB_EpHandle  hEpObj[ ],
                      Uchar         PSofTmrCnt);
```

Arguments

DevNum — USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

hEpObj[ ] — Pointer to a NULL terminated array of handles (pointers) of initialized endpoint objects. Maximum number of handles in the array cannot be more than 16 (excluding the NULL handle).

PSofTmrCnt — 8-bit counter value for the pre SOF timer

Return Value USB_TRUE if the module initialization is successful, otherwise, USB_FALSE is returned.

Description Initializes the USB_Module to operation mode.

Comments Upon successful return from the function call, the USB module is ready for operation (all registers are configured and unmasked interrupts are enabled). Once the USB module is initialized, it is necessary that the application unmask the USB interrupt mask bit in the IER0 register and enable the DSP global interrupt. Finally, the application must call the function USB_connectDev(USB0) to connect the USB module to the USB bus.

If DSP/BIOS is used, the USB interrupt must be enabled through the BIOS hardware interrupt configuration too.

If the PSofTmrCat is not zero, then the pre-SOF interrupt will occur approximately (PSofTmrCnt) x 16cycles(12Mhz clock) prior to every SOF.

Example The following example initializes the USB0 module to support a USB device with one interrupt and two bulk endpoints in addition to two control endpoints. The function call also enables the pre-SOF interrupt. It is assumed that the endpoint objects are initialized prior to initializing the USB Module:

```
/* Control endpoint objects */
USB_EpObj    EndptObjOut0, EndptObjIn0;

/* Bulk endpoint objects */
USB_EpObj    EndptObjOut2, EndptObjIn2;

/* Interrupt endpoint object */
USB_EpObj    EndptObjIn3;

/* create a Null Terminated array of endpoint objects */
USB_EpHandle  hEpObjArray[] = {& EndptObjOut0, & EndptObjIn0,
                               & EndptObjOut2, &EndptObjIn2,
                               & EndptObjIn3, NULL};

{
/* Initialize endpoint objects here */
}

USB_Boolean USB_init(USB0, hEpObjArray, 0x40);
```

2.4.5 Module Control

USB_initPLL *Initializes the USB PLL*

Function void USB_initPLL(Uint16 inclk, Uint16 outclk, Uint16 plldiv);

Category Module Control

Arguments

inclk:	Input clock (supplied at CLKIN pin) frequency (in MHz)
outclk:	Desired clock frequency (in MHz) for the USB modules The outclk must be 48 MHz for the proper operation of the USB module.
plldiv:	Input clock (supplied at CLKIN pin) divide down value, used for USB PLL_enable as well as USB PLL bypass mode

Return Value None

Comments pllmult = (outclk * (plldiv+1)) / inclk

if pllmult > 1, outclk = (pllmult / (plldiv + 1)) * inclk
if pllmult < 1, outclk = (1 / (plldiv + 1)) * inclk

Example The following example initializes the USB PLL to generate a 48 MHz USB clock from a 12 MHz source clock:

```
USB_initPLL(12, 48, 0);
```

USB_connectDev *Connects the USB module to the upstream port*

Function void USB_connectDev(USB_DevNum DevNum);

Category Module Control

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value None

Comments Connects the USB module to the upstream port (D+ pull-up enabled).

Example The following example connects USB0 to the bus:

```
USB_connectDev(USB0);
```

USB_disconnectDev — *Disconnects the USB module from the upstream port*

Function void USB_disconnectDev(USB_DevNum DevNum);

Category Module Control

Arguments DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value None

Comments Disconnects the USB module from the upstream port (D+ pull-up disabled).

Example The following example disconnects USB0 from the bus:

```
USB_disconnectDev(USB0);
```

USB_issueRemoteWakeup — *Issues a remote wakeup signal to the host*

Function USB_Boolean USB_issueRemoteWakeup(USB_DevNum DevNum);

Category Module Control

Arguments DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value USB_TRUE if successful, else USB_FALSE (if remote wakeup feature is not set prior to calling this function).

Comments The USB driver generates a remote wakeup signal on the bus only if the remote wakeup is enabled. An application must enable the remote wakeup feature by calling the USB_setRemoteWakeup( ) routine when a Set Remote Wakeup request is received from the host.

Example The following example causes the USB0 to generate a remote wakeup signal on the bus if the host has already set the remote wakeup feature:

```
USB_Boolean   status;

status  = USB_issueRemoteWakeup (USB0)
```

USB_resetDev — *Resets the USB module*

Function void USB_resetDev(USB_DevNum DevNum);

Category Module Control

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value None

Comments Once the module has been reset, the control and status registers are returned to power-up reset values and the USB module is disconnected from the upstream port.

Example The following example resets all the status and control registers of USB0 to power-on reset value and disconnects the USB module from the bus:

```
USB_resetDev (USB0);
```

USB_setDevAddr — *Sets the USB device address*

Function void USB_setDevAddr(USB_DevNum DevNum, Uchar addr);

Category Module Control

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.
addr	7-bit USB device address

Return Value None

Comments None

Example The following example sets the address of USB0 to 03h. After execution of the function the USB module responds to this address:

```
USB_setDevAddr (USB0, 0x03);
```

USB_stallEndpt

Stalls an endpoint

Function void USB_stallEndpt(USB_EpHandle hEp);

Category Module Control

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value None

Description Stalls an endpoint.

Comments The endpoint handle determines the endpoint to stall.

Example The following example stalls endpoint 5 IN:

```
USB_stallEndpt (&EndptObjIn5);
```

USB_clearEndpt-Stall

Clears an endpoint stall

Function void USB_clearEndptStall(USB_EpHandle hEp);

Category Module Control

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value None

Description Clears an endpoint stall.

Comments The endpoint handle determines the endpoint to bring out of stall.

Example The following example clears the stall condition of endpoint 5 IN:

```
USB_clearEndptStall (&EndptObjIn5);
```

2.4.6 Data Transfer

USB_getSetup-Packet *Read the setup packet from the setup data buffer*

Category Data Transfer

Function USB_Boolean USB_getSetupPacket(USB_DevNum DevNum, USB_SetupStruct *USB_Setup);

Arguments

DevNum	USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.
*USB_Setup	Pointer to a structure of type USB_SetupStruct

Return Value USB_TRUE if successful. Otherwise, USB_FALSE. If successful, the USB_Setup structure holds the new setup packet.

Description None

Comments None

Example The following example returns the data from the most recent setup packet received by USB0:

```
USB_SetupStruct USB0_SetupPkt;
              .
              .


void USB_Endpt0EventHandler(void)
{
                      .
                      .
    if(USB_getEvents(EndptObjIn0)  & USB_EVENT_SETUP)
    {
       if(USB_getSetupPacket(USB0, USB0_SetupPkt) == USB_TRUE)
          {
/* Application code for handling setup packet. */
                      .
                      .
          }
     }
                       .
}
```

USB_postTransaction

Transmit or receive USB data through an endpoint.

Category

Data Transfer

Function

```
USB_Boolean USB_postTransaction(USB_EpHandle    hEp,
                                Uint16          ByteCnt,
                                void            *Data,
                                USB_FLAGS        Flags);
```

Arguments

hEp	Handle or a pointer to an initialized endpoint object
ByteCnt	Total number of bytes in the buffer pointed by *Data
*Data	A pointer to a data buffer or a linked list of type USB_DataStruct
Flags	ORed combination of USB Data Transfer Flags

Return Value

If the previously posted transfer is not completed, USB_FALSE is returned. USB_TRUE is returned if the data transfer request was posted successfully.

Description

Comments

The endpoint handle determines if the data moves in or out of the USB module. If the USB_IOFLAG_EOT event mask is set and an event handler routine is supplied during the endpoint object initialization, the USB event dispatcher calls the associated event handler routine at the end of the handler routine.

Known Limitations: This limitation applies to OUT Endpoints only if the data buffer is a linked list and USB_IOFLAG_CAT is set. If there is an instance where the host prematurely terminates the data transfer (with or without a short packet), the driver will attempt to fill the rest of the data buffers in the linked list.

As a result, the posted transaction appears to be in progress, and a call to USB_isTransactionDone (..) will return USB_FALSE.

If the current node is the very last node of the linked list, the driver treats this as a termination of transfer and a call to the routine USB_isTransactionDone (..) returns USB_TRUE.

Reasons for the limitations: In order to move data more efficiently, the driver, when possible, programs the DMA active/reload registers at the same time. It is beyond the scope of the driver to anticipate an early termination of data transfer and not program the DMA Reload registers.

Workaround: If there are concerns that the host may prematurely terminate the transfer, then avoid using the USB_IOFLAG_CAT with a linked list for OUT transfers.

Affected Endpoints: OUT [1..7]

NOTE: Since endpoint0 transfers are not done by the USB dedicated DMA, endpoint 0 IN/OUT transfers are not affected.

Example

Declare three standard and one linked list data buffer to be used by the data transfer examples (A C55x word holds two butes of USB data in BigEndian mode):

```
USB_EpObj   EndptObjIn2;
USB_EpObj   EndptObjOut3;
Uint16 usb_InData1[] = {0, 0x0100, 0x0302, 0x0004};
Uint16 usb_InData2[] = {0, 0x1110, 0x1312, 0x1514, 0x1716};
Uint16 usb_InData3[] = {0, 0x0100, 0x03002, .  .   .   .
0x7D7C, 0x7F7E};  // 128 bytes of data
uint16  USB_OutData[5];     //4-byte buffer for USB OUT data

USB_DataStruct usb_InLnk2 =
{
  8,                        // length of data buffer in bytes
  (Uint16 *)&usb_InData2,   // pointer to the data buffer
  NULL                      // pointer to next linked list
};

USB_DataStruct usb_InLnk1 =
{
  5,                       // length of data buffer in bytes
  (Uint16 *)&usb_InData1,  // pointer to the data buffer
  &usb_InLnk2              // pointer to next linked list
};
              .
              .
```

Sending Data to the Host

Case 1a:

Send 4 bytes from data buffer usb_InData1[] through Endpoint 2 IN. The data appears on the bus in the following order:
0x00, 0x01, 0x02, 0x03

```
USB_postTransaction(&EndptObjIn2, 4, &usb_InData1,
USB_IOFLAG_NONE);
```

.

Case 1b:

Send 4 bytes from data buffer usb_InData1[] through Endpoint 2 IN with high byte and low byte swapped. The data appears on the bus in the following order:
0x01, 0x00, 0x03, and 0x02

```
USB_postTransaction(&EndptObjIn2, 4, &usb_InData1,
USB_IOFLAG_SWAP);
```

Case 1c:

Send 96 bytes from data buffer usb_InData3[] through Endpoint 2 IN. Do not insert a 0-byte packet if the transfer ends with a full size packet. The data appears on the bus in the following order:
0x00 0x01 0x02, . . . 0x5E, and 0x5F

```
USB_postTransaction(&EndptObjIn2, 4, &usb_InData1,
USB_IOFLAG_NOSHORT);
```

Case 2a:

Send 10 bytes (all 5 bytes from usb_InData1[] and the rest from usb_InData2[]) of data through Endpoint 2 IN. Generate separate transfers for usb_InData1[] and usb_InData2[].

The data appears on the bus in the following order:
0x00, 0x01, 0x02, 0x03, and 0x04, for the first transfer
0x10, 0x11, 0x12, 0x13, and 0x14 for the second transfer

```
USB_postTransaction(&EndptObjIn2, 10, &usb_InLnk1,
USB_IOFLAG_LNK);
```

Case 2b:

Send 10 bytes (all 5 bytes from usb_InData1[] and the rest from usb_InData2[]) of data through Endpoint 2 IN. Concatenate usb_InData1[] and usb_InData2[] to send them as a single transfer.

The data appears on the bus in the following order:
0x00, 0x01, 0x02, 0x03, 0x04, 0x10, 0x11, 0x12, 0x13, and 0x14.

```
USB_postTransaction(&EndptObjIn2, 10, &usb_InLnk1,
USB_IOFLAG_LNK|USB_IOFLAG_CAT);
```

Case 2c:

Same as case 2a except that no 0-byte packets are inserted if any transfer ends with a full size data packet.

```
USB_postTransaction(&EndptObjIn2, 10, &usb_InLnk1,
USB_IOFLAG_LNK|USB_IOFLAG_NOSHORT);
```

Case 3a:

Send all the data bytes in the linked list usb_InLnk1 through Endpoint 2 IN. Generate separate transfers for each data buffer in the list. The transfer ends when the end of the linked list is reached. A NULL pointer to the next node indicates the end of the linked list.

The data appears on the bus in the following order:
0x00, 0x01, 0x02, 0x03, and 0x04 for the first transfer
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, and 0x17 for the second transfer.

```
USB_postTransaction(&EndptObjIn2, 0, &usb_InLnk1,
USB_IOFLAG_LNK|USB_IOFLAG_EOLL);
```

Case 3b:

Send all the data bytes in the linked list usb_InLnk1 through Endpoint 2 IN. Concatenate all the data buffers in the list to send them as a single transfer. The transfer ends when the end of the linked list is reached. A NULL pointer to the next node indicates the end of the linked list.

The data will appear on the bus in the following order:
0x00, 0x01, 0x02, 0x03, 0x04, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, and 0x17

```
USB_postTransaction(&EndptObjIn2, 0, &usb_InLnk1,
USB_IOFLAG_LNK|USB_IOFLAG_CAT|USB_IOFLAG_EOLL);
```

Case 3c:

Same as case 3b except that no 0-byte packets are inserted if the transfer ends with a full size data packet.

```
USB_postTransaction(&EndptObjIn2, 0, &usb_InLnk1,
USB_IOFLAG_LNK|USB_IOFLAG_CAT|USB_IOFLAG_EOLL|USB_IOFLAG_NOSHORT);

.

while(!USB_isTransactionDone(&EndptObjIn2)); /* wait until
last posted transfer is done */
USB_postTransaction(&EndptObjIn2, 10, &usb_InLnk1,
USB_IOFLAG_LNK);
```

Receiving Data from the host:

The function call examples shown above are also valid for receiving data from a USB host. To receive data, the pointer to the endpoint object passed, should be associated with a USB out Endpoint. For example, the following function call will receive 8-bytes of data in a USB_outData[] buffer.

```
USB_postTransaction (&EndptObjOut3, 8, &USB_outData,
                                   USB_IOFLAG_NONE);
```

Handling 0-byte Control Handshake Packets

Case 4a:

Send a 0-byte handshake packet to end a setup packet. Sending a NULL data buffer pointer is a special case supported only for control endpoints to send and receive a 0-byte handshake packet. Sending a NULL buffer pointer to the USB driver for data transfer through any other endpoint causes the driver to fail.

```
USB_postTransaction(&EndptObjIn0, 0, NULL, USB_IOFLAG_NONE);
```

Case 4b:

Receive a 0-byte handshake packet to end a setup packet. Sending a NULL data buffer pointer is a special case supported only for control endpoints to send and receive a 0-byte handshake packet. Sending a NULL buffer pointer to the USB driver for data transfer through any other endpoint causes the driver to fail.

```
USB_postTransaction(&EndptObjOut0, 0, NULL, USB_IOFLAG_NONE);
```

2.4.7 Status Query

USB_getFrameNo *Returns the current USB frame number*

Function Uint16 USB_getFrameNo(USB_DevNum DevNum);

Category Status/Query

Arguments DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value Current USB Frame Number

Comments None

Example The following example returns the current frame number for the USB0 module:

```
Uint16  CurFrmNo;
CurFrmNo  = USB_getFrameNo (USB0);
```

USB_getEvents *Reads and clears all pending USB_EVENTS*

Function USB_EVENT_MASK USB_getEvents(USB_EpHandle hEp);

Category Status/Query

Arguments hEp Handle or pointer to an initialized endpoint object

Return Value ORed combination of all the pending USB_EVENTS associated with a particular endpoint.

Description Get all the pending USB_EVENTS

Comments Calling this function also clears all the pending USB_EVENTS associated with a particular USB endpoint.

Example The following example returns all the events that occurred at Endpoint 0 OUT and **clears** the internal variable that holds the Endpoint 0 OUT events:

```
USB_EpObj   EndptObjOut0;
              .
              .

void USB_Endpt0EventHandler(void)
{
     USB_EVENT_MASK  mask;
               .
               .
    mask = USB_getEvents(EndptObjOut0);
    if(mask & USB_EVENT_RESET)
    {
/* Application code for handling the event. */
               .
               .
     }

    if(mask & USB_EVENT_SETUP)
    {
/* Application code for handling the event. */
               .
               .
     }
               .
               .
}
```

USB_getRemoteWakeupStat

Get the status of the remote wakeup feature

Function USB_Boolean USB_getRemoteWakeupStat(USB_DevNum DevNum);

Category Status Query

Arguments DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Return Value USB_TRUE if the remote wakeup feature is enabled in the software. USB_FALSE if the remote wakeup feature is disabled in the software.

Comments An application must verify if the remote wakeup feature is set before generating a remote wakeup signal.

Example The following example informs the firmware whether the remote wakeup feature for USB0 is set or not:

```
USB_Boolean   RmtWkpStat;

 RmtWkpStat = USB_getRemoteWakeupStat (USB0);
```

USB_peekEvents

Reads all pending USB_EVENTS

Function USB_EVENT_MASK USB_peekEvents(USB_EpHandle hEp);

Category Status/Query

Arguments hEp Handle or pointer to an initialized endpoint object

Return Value ORed combination of all the pending USB_EVENTS associated with a particular endpoint.

Description

Comments Calling this function does not clear the USB_EVENTS associated with a particular USB endpoint.

Example

The following example returns all the events that occurred at Endpoint 0 OUT, but **does not clear** the internal variable that holds the Endpoint 0 OUT events:

```
USB_EpObj   EndptObjOut0;
              .
              .

void USB_Endpt0EventHandler(void)
{
               .
               .
    if(USB_getEvents(EndptObjOut0) & USB_EVENT_RESET)
    {
/* Application code for handling the event. */
               .
               .
     }

    if(USB_getEvents(EndptObjOut0)  & USB_EVENT_SETUP)
    {
/* Application code for handling the event. */
               .
               .
     }
               .
               .
}
```

USB_isTransactionDone

Returns the status of a previously posted data transfer request

Function USB_Boolean USB_isTransactionDone(USB_EpHandle hEp);

Category Status/Query

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value USB_TRUE, if the previously posted transfer is completed, otherwise, USB_FALSE.

Description None

Example Send 5 bytes from data buffer usb_InData1[] through Endpoint 6 IN Wait for transfer to complete and then send 8 bytes from usb_InData2[].

```
Uint16 usb_InData1[] = {0, 0x0100, 0x0302, 0x0004};
Uint16 usb_InData2[] = {0, 0x1110, 0x1312, 0x1514, 0x1716};
                      .
                      .
USB_postTransaction(&EndptObjIn6, 5, &USB_InData1,
USB_IOFLAG_NONE);
                      .
while(!USB_isTransactionDone(&EndptObjIn6));  /* wait until
last posted transfer is done */
USB_postTransaction(&EndptObjIn6, 8, &USB_InData2,
USB_IOFLAG_NONE);
```

USB_bytesRemaining

Returns the number of bytes awaiting transfer

Function USB_BYTE_COUNT USB_bytesRemaining(USB_EpHandle hEp);

Category Status/Query

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value The number of bytes remaining to be transferred. Return value is 0xFFFF if USB_IOFLAG_NOBCNT flag is used while posting the transfer request.

Description Returns the number of bytes awaiting transfer from the previously posted data transfer request.

Comments The endpoint handle determines the endpoint the data moves through.

Example The following example returns the number of bytes to be received through endpoint-6 OUT to fill a buffer with 49 bytes of data from the host:

```
USB_BYTE_COUNT  bytes_to_recv
Uint16 usb_OutData1[65];     /* 128 byte buffer             */
                      .
                      .
USB_postTransaction(&EndptObjOut6, 49, &usb_OutData1,
                   USB_IOFLAG_NONE);
                      .
if(!USB_isTransactionDone(&EndptObjOut6))    /* if transfer is
                                            not compete */
        bytes_to_recv  = USB_bytesRemaining(&EndptObjIn6);
```

USB_getEndpt-Stall

Determines if an endpoint is stalled

Function USB_Boolean USB_getEndptStall(USB_EpHandle hEp);

Category Status/Query

Arguments hEp Handle or a pointer to an initialized endpoint object

Return Value USB_TRUE if the endpoint is stalled, otherwise, USB_FALSE is returned.

Description Determines if an endpoint is stalled.

Comments The endpoint handle selects the endpoint.

Example The following example returns the stall condition of endpoint 5 IN:

```
USB_Boolean  EndptStallStat;
                 .
                 .
EndptStallStat = USB_getEndptStall (&EndptObjIn5);
```

2.4.8 Miscellaneous

USB_epNumTo-Handle *Returns a handle or a pointer to an endpoint object*

Function USB_EpHandle USB_epNumToHandle(USB_DevNum DevNum, Uchar Endpt);

Category Misc

Arguments

DevNum USB device number, enumerated data type of USB_DevNum. Currently, the only active device number available is USB0.

Endpt 8-bit endpoint number per USB specifications:
0x00 -> Endpt 0 Out, 0x01 -> Endpt 1 Out ....
0x80 -> Endpt 0 In, 0x81 -> Endpt 1 In ....

Return Value A handle or a pointer to the endpoint object if a valid endpoint object exists, otherwise a NULL handle.

Description Delivers a handle or a pointer to an endpoint.

Comments Returns a handle to an endpoint object, which can be used to call other USB functions.

This routine is helpful when the application does not have any prior knowledge of the endpoint it is handling. For example, if the host request to stall an endpoint, the application can read the endpoint number from the setup packet, retrieve the handle to this endpoint by calling USB_epNumToHandle (..), and call USB_stallEndpt (..) with the handle to stall the endpoint.

Example The following example retrieves a handle to endpoint object for Endpoint 4 IN.

If an endpoint is not active, the function call returns a NULL pointer.

```
/* create an instance of a handle to an endpoint object */
USB_EpHandle    hEndptIn4;

/* retrieve the handle to the endpoint object */
hEndptIn4 = USB_epNumToHandle(USB0, 0x84),
if(hEndptIn4 != NULL)

/* stall endpoint */
      USB_clearEndptStall(hEndptIn4);
```

Chapter 3

Configuring The USB Module Using CSL GUI

This chapter contains instructions for the configuration of the USB module using the CSL graphical user interface (GUI).

Topic		Page

3.1 Overview

The CSL USB graphical user interface (GUI) facilitates the configuration of the Universal Serial Bus (USB) module. The USB GUI consists of an Endpoint Configuration Manager and a Resource Manager. The USB Endpoint Configuration Manager allows initialized endpoint objects to be used with the CSL USB API functions. The USB Resource Manager allows the user to group all initialized endpoint objects created by the USB Endpoint Configuration Manager into a user defined USB configuration array. The USB configuration array is used by the CSL USB API functions to initializae the USB module.

Figure 3–1 illustrates the USB sections menu on the CSL graphical user interface (GUI)

Figure 3–1. USB Sections Menu

The USB includes the following two sections:

❑ **USB Endpoint Configuration Manager**: Allows you to create initialized endpoint objects.

❑ **USB Resource Manager**: Allows you to group all initialized endpoint objects under a configuration array.

3.2 USB Endpoint Configuration Manager

The USB Endpoint Configuration Manager allows you to create initialized endpoint objects through the Properties page.

3.2.1 Creating/Inserting an Endpoint Object

There is no predefined endpoint object available.

To create a endpoint object, you must insert a new endpoint configuration object.

To insert a new endpoint configuration object, right-click on the USB Endpoint Configuration Manager and select insert USBCfg from the drop-down menu. The configuration objects can be renamed.

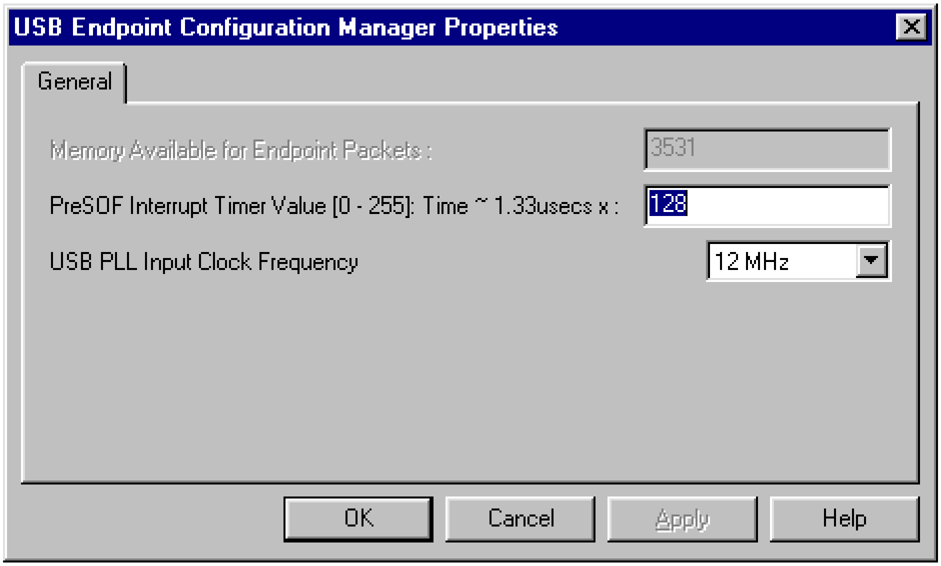
Note: Only one endpoing object is allowed per active endpoint in a USB application.

3.2.2 Deleting/Renaming an Endpoint Object

To delete or to rename an endpoint object, right-click on the endpoint object you want to delete or rename. Select Delete to delete an object. Select Rename to rename an object.

3.2.3 Configuring the Global Settings

Figure 3–2. Global Settings for the Configuration Manager



The following options are available for the Global Settings of the Configuration Manager:

- Memory Available for Endpoint Packets: Displays the memory (in byte) available in the USB Buffer RAM for endpoints yet to be added. Every time a new endpoint (object) is added to the configuration, a block of memory from the USB Buffer RAM is set aside for that that endpoint.

  The CSL USB API functions require all the active endpoints operating in double–buffer mode, hence the size of the memory reserved in the USB Buffer RAM for each endpoint is twice the size of the endpoint packet size.

  For more information regarding USB Buffer RAM, please refer to the USB chapter of the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C).

- PreSOF Interrupt Timer Value: Allows you to set the Pre-Start-of-Frame interrupt counter value. Please refer to the USB_init function, on page 2-18, for more information on setting the PreSOF Interrupt Timer Value.

- USB PLL Input Clock Frequency: Allows you to enter the Input clock frequency to the USB PLL. Please refer to the USB_initPLL function, on page 2-20, for more information on setting the USB PLL clock frequency.

3.2.4 Configuring the Endpoint Object Properties

The Properties pages allow you to set the characteristics of an endpoint associated with an endpoint object (see Figure 3–3, on page 3-5). To access the Properties dialog box, right-click on an endpoint object and select Properties. By default, the general page of the properties dialog box is displayed.

You can set the various configuration options through the following properties pages:

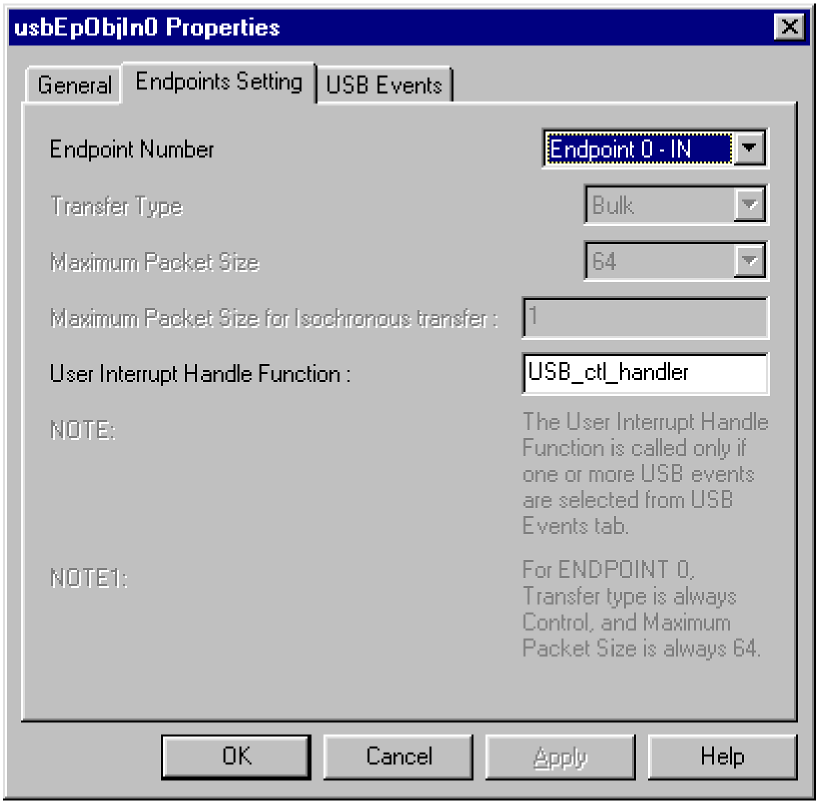
- General Settings: Allows general settings for the USB Module.

- Endpoint Settings: Allows you to configure a USB endpoint object.

- USB Events: Allows you to select the USB events used to trigger an endpoint event handler routine. For more information, please refer to the USB Event Dispatcher description found on page 1-12

Figure 3–3, on page 3-5, depicts the Properties Page.

Figure 3–3. USB Properties Page



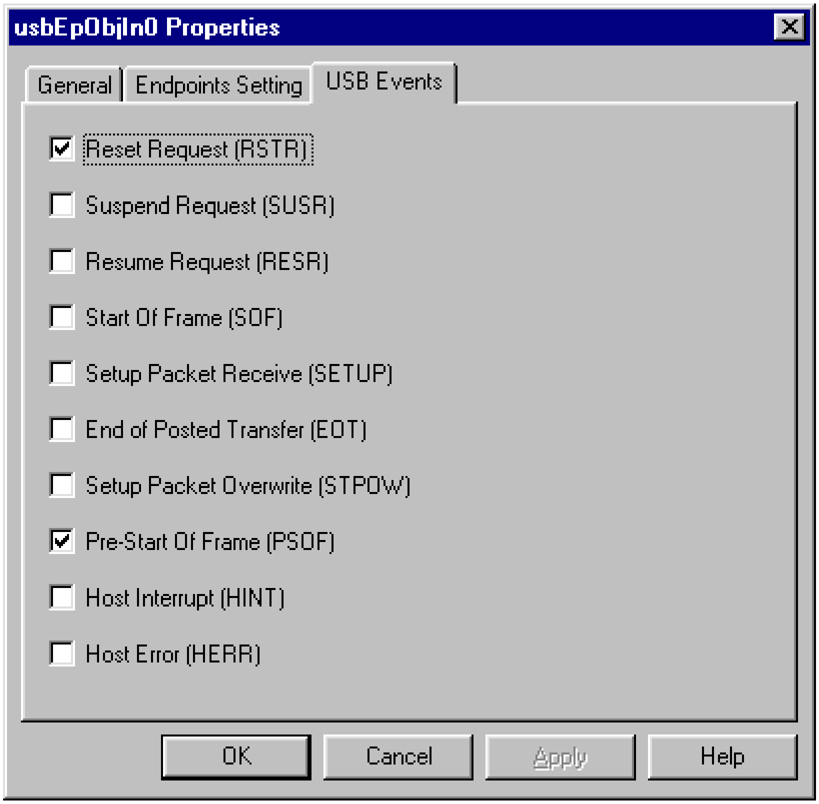
Each Tab page is composed of several options that are set to a default value (at device reset).

The USB Properties page allows you to configure endpoints with the following options:

- ❑ Endpoint Number: Allows you to select an endpoint to associate the the endpoint object.
- ❑ Transfer Type: Allows you to select the Transfer type to be supported by the endpoint.
- ❑ Maximum Packet Size: Allows you to determine the Maximum packet size supported by the endpoint.
- ❑ User Interrupt Handle Function: Allows you to set the User defined USB event handler routine to be called by the USB Event Dispatcher when any of the events selected from the USB Events tab occurs.

Figure 3–4 illustrates the USB Event tab.

Figure 3–4. USB Events Properties Page



3.3 USB Resource Manager

The USB Resource Manager allows you to generate the USB configuration arrays to be used by the USB_init() function.

Figure 3–5 illustrates the USB Resource Manager menu on the CSL Graphical User Interface (GUI).

Figure 3–5. USB Resource Manager Menu

3.3.1 Properties Page

You can generate the USB Initialization code through the Properties page.

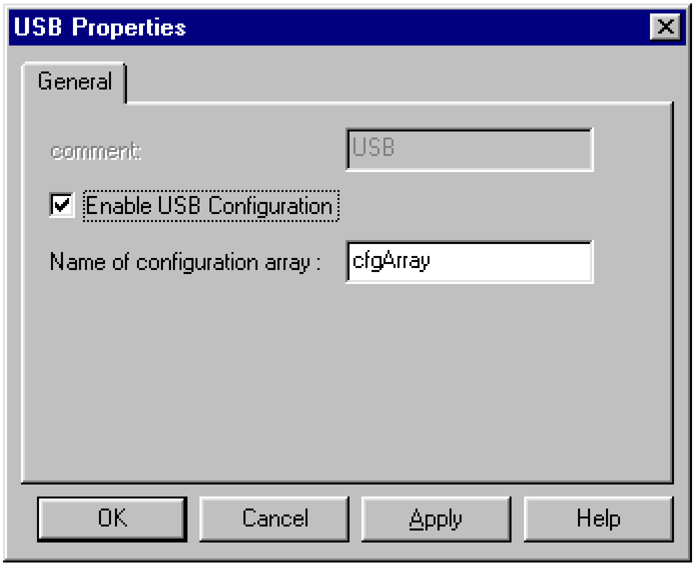
To access the Properties page, right-click on the predefined USB peripheral and select Properties from the drop-down menu (see Figure 3–6, on page 3-7).

To pre-initialize the USB peripheral, check the Enable USB Configuration box.

You can also change the name of the USB configuration array (default name is *cfgarray*), under which all initialized endpoint objects are grouped together.

In the example shown in Figure 3–6, the Enable USB configuration is selected, and the USB initialization code will be generated.

Figure 3–6. USB Resource Manager Properties Page



3.4 C Code Generation for the USB module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file

3.4.1 Header File

The header file includes all the CSL header files for the USB module and contains endpoint objects defined within the Endpoint Configuration Manager pages (see Example 3–1). The endpoint object structure is described in the USB Data Structures section (see section 2.3, on page 2-7).

Example 3–1. USB Header File

```
extern USB_EpObj usbEpObjOut0;
```

3.4.2 Source File

The source file includes the declaration (initialized structures) of endpoint objects (see Example 3–2). The endpoint object structure is described in the USB Data Structures section (see section 2.3, on page 2-7).

Example 3–2. USB Source File (Declaration Section)

```
/*  Config Structures */
USB_EpObj usbEpObjOut0 = {
      USB_OUT_EP0,   /* Endpoint Number                                    */
      USB_CTRL,      /* Transfer type value                                */
      0x0040,        /* Maximum Packet Size Supported by EP                */
      0x003d,        /* Event Mask                                         */
      USB_ctl_handler,/* Pointer to USB event ISR                          */
      0x0000,        /* Data Flags                                         */
      0x0000,        /* Status                                             */
      0x6782,        /* Endpoint descriptor reg block start addr           */
      0x6680,        /* DMA reg block start addr                           */
      0x0000,        /* Total byte count                                   */
      0x0000,        /* Number of bytes in the active node of the linked list */
      NULL           /* Pointer to store the number of bytes moved in (out)   */
      NULL           /* Active data buffer pointer                         */
      NULL           /* Pointer to NEXT Buffer                             */
      0x0000         /* Event Flag                                         */
};
```

The source file contains the USB module initialization code, using CSL USB API functions (see Example 3–3).

This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file. The USB API function calls are generated only if Enable USB configuration is checked under the USB Resource Manager Propteries page.

Example 3–3. USB Source File (Body Section)

```
void CSL_cfgInit()
{
       myUSBConfig[i++] = &usbEpObjOut0;
       myUSBConfig[i] = NULL;

       USB_setAPIVectorAddress();
       USB_initPLL(12, 48, 0);

       USB_init(USB0, myUSBConfig, 0x80);
}
```

3.5 Connecting the USB Module To a Host

Calling the CSL_cfgInit() from the main C file will program all USB module control and configuration registers. As discussed in chapter 1, all the CSL USB API functions work in conjunction with USB Event Dispatcher.

There are two options to use when enabling the USB Event Dispatcher:

- Interrupt Polling Method: The user's code polls the USB interrupt flag bit periodically and calls the USB Event Dispatcher functions every time the USB interrupt flag is set.
- Encapsulating the USB Event Dispatcher function: The user encapsulates the USB Event Dispatcher function in an ISR and sets the DSP to service this ISR every time a USB event occurs.

Once the CSL_cfgInit() is called and the USB Event Dispatcher function is enabled, the USB module can be connected to the bus by calling the USB_devConnect() function. At this point, the USB module can be connected to a USB traffic generator to transmit and receive raw USB data.

Connecting the USB module to a host requires additional code running on the DSP to support the USB protocol (for example, a PC running on the Windows® 2000 platform).

Without the USB protocol handling code, the DSP may not be able to process the data received, resulting in the possibility of the host PC locking up.

A chapter 9 compliant USB demo application using CSL USB API functions and the CSL USB GUI is available from the Texas Instruments' DSP Village web site.

Appendix A

USB Terminology

This appendix contains definitions, descriptions, and terminology associated with the USB.

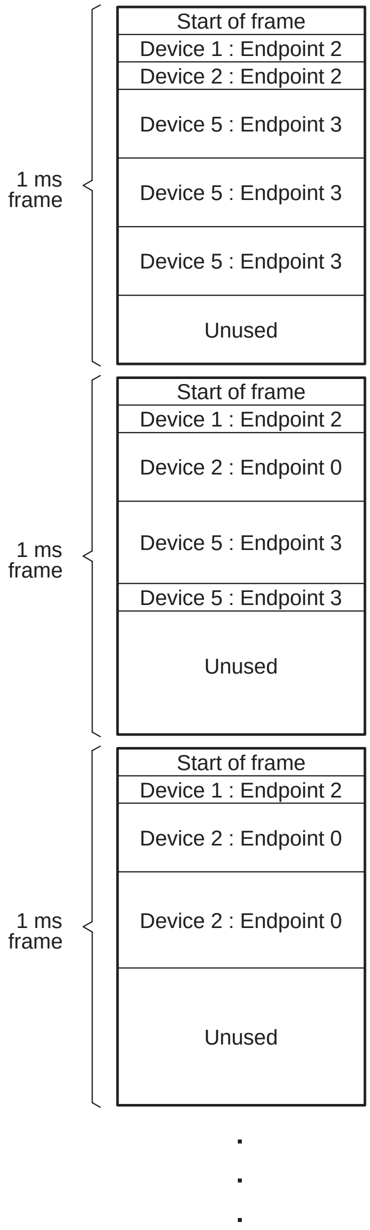
Topic		Page

A.1 USB Terminology

A.1.1 Frames

The USB is a single master (called host) and multiple slave (called device) interface. The host is in charge of initiating all transfers (control or data). In order to manage the bus traffic efficiently, the bus time is divided into 1-millisecond slots called frames. The host allocates a portion of the frame for each device successfully attached to the bus. The location of slots in a frame allocated for each device is not fixed. The host may allocate the slot anywhere within a frame. Figure A–1 illustrates the layout and components of a USB frame.

Figure A–1. USB Frame Layout



A.1.2 Transfers, Transactions, and Packets

A transfer is a collection of transactions. The USB transactions are made up of multiple packets. There are four types of data transfers:

- Control
- Bulk
- Interrupt
- Isochronous

A.1.2.1 Transfer Types

- **Control**

  Control transfers are performed with the device control endpoint , which is typically Endpoint0, and are given a standard protocol in the USB specification. The host allocates a small percentage (maximum 10%) of the bus bandwidth for control transfers. The host guarantees that some pending transactions belong to control transfers and are completed in every frame.

- **Bulk**

  Bulk transfers are used for moving large amounts of non-time critical data that requires reliable delivery. Bus bandwidth is not guaranteed for bulk transfers; all of the ”left-over” time on the bus is dedicated for bulk transfers only. On an idle bus, bulk transfers are fast.

- **Interrupt**

  An interrupt transfer is a limited latency delivery mechanism for moving moderate amounts of data from the device to the host. Interrupt transfers appear the same to the device as the bulk transfers, except the way they are scheduled by the host. Interrupt transfers, along with isochronous transfers, are guaranteed an allocated amount of bus bandwidth (maximum 90%). Full speed interrupt transfers can occur as often as every frame or as infrequently as every 255 frames.

- **Isochronous**

  Isochronous transfers are used for continuous real-time data delivery. Isochronous data is guaranteed a time of delivery, but not accuracy. In order to maintain the real-time delivery schedule, isochronous transfers do not include handshake packets. Unlike other transfers, isochronous transfers are not retried if an error occurs during the transfer.

A.1.2.2 Transactions

Transactions are the building blocks of a transfer. The USB uses four different transactions to address the four different transfers in section A.1.2.1, on page A-4. Except for Isochronous transfers, each transaction is made up of three packets:

- Token
- Data
- Handshake

The Isochronous transfer is made up of only two packets:

- Token
- Data

❑ **Control**

A basic control transaction consists of the following packets:

- Setup packet
- Data packet
- Handshake packet

The Data packet is optional. As a result, some control packets may only have a Setup packet and a Handshake packet.

❑ **Bulk**

A bulk transaction consists of three packets:

- IN/OUT token packet
- Data packet
- Handshake packet

To read data in, the host issues an IN token. The device responds with a Data packet and the host ends the transaction with a Handshake packet. To write data out, the host issues an OUT token packet, followed by a Data packet. The device completes the transaction by sending a Handshake packet.

❑ **Interrupt**

Being similar to bulk transactions, interrupt transactions are made up of three packets. Interrupt transactions always carry data from the device to the host. Therefore, an interrupt transaction starts with an IN token packet issued by the host, followed by a Data packet from the device, and ends with a Handshake packet from the host.

- **Isochronous**

  Isochronous transactions consist of a token packet and a Data (DATA0) Packet. The transaction starts with an IN/OUT token packet from the host, followed by a Data packet from either the host of the device, depending on the direction of the data flow. There are no handshake packets involved in an isochronous transaction; therefore, occasional errors are acceptable.

A.1.2.3 Packets

All USB transactions are made up of units referred to as “packets”. A USB packet always starts with a ”SYNC” field followed by a ”PID” and ends with an ”EOP”. The USB packets are grouped into four major categories:

- **Token**

  Token packets are used to identify a transaction. Four token packets are available as defined in the *USB 1.1 Specification*:

  - SOF

    The Start-Of-Frame packet is used to indicate the beginning of a 1-millisecond USB frame. An SOF packet has 11-bits of data and 5-bits of CRC error checking. The 11-bits of data in the packet is a monotonically increasing **Frame Number** that is typically used by the real-time devices to synchronize data transfers. The **Frame Number** rolls over every 2,048 milliseconds.

  - SETUP

    Setup packets are used to initiate a control transfer. A setup packet has a 7-bits of device address, 4-bits of endpoint address, and 5-bits of CRC error checking.

  - IN

    IN packets are used by the host to initiate a data transfer from a device to itself. An IN packet has 7-bits of device address, 4-bits of endpoint address, and 5-bits of CRC error checking. The IN packets are used in all four USB transfers.

  - OUT

    OUT packets are used by the host to initiate a data write from itself to the device. Like an IN packet, an OUT packet has 7-bits of device address, 4-bits of endpoint address, and 5-bits of CRC error checking. The OUT packets are used in control, bulk, and isochronous transfers.

- **DATA**

  Data packets are used to carry data payloads associated with a given transaction. Two types of data packets are available as defined in the *USB 1.1 Specification*:

  - DATA0 (even)

    A DATA0 packet is an even PID data packet. DATA0 packets are made up of 0-1023 bytes of data followed by a16-bit CRC value. DATA0 packets are used in all four transfers.

  - DATA1 (odd)

    A DATA1 packet is an odd PID data packet. DATA1 packets are made up of 0-64 bytes of data followed by a 16-bit CRC value. DATA1 packets are used in control, bulk, and interrupt transfers.

  DATA1 and DATA0 packets are used to keep the host and device synchronized during a transfer that requires multiple transactions. During a long transfer, the data packets are toggled between DATA0 and DATA1. This technique is known as a ***Data Toggle***. The sender and the receiver keep track of the data toggle bit to ensure that there are no lost transactions. ***Data Toggle*** is not implemented in isochronous transfers. Isochronous transfers use only DATA0 packets.

- **Handshake**

  Handshake packets are used by a receiver to indicate the reception status of a token and/or data packets. Three types of handshake packets are defined in the *USB 1.1 Specification*:

  - ACK

    ACK, or Acknowledgment packet, is used to indicate the error-free reception of a token or data packet.

  - NAK

    NAK, or Negative Acknowledgement packet, allows a device to report to the host that the receiver is not ready to handle a specific token or data packet at that particular time. A device is allowed to NAK any packet, the only exception being a setup packet. If a NAK packet is received, the host retries the packet at a later time. During interrupt transactions, a NAK means that no data is currently available to return to the host, meaning that no interrupt request is currently pending.

- STALL

  A device responds with a STALL handshake if any of the following cases are true:

  - Specific control requests (setup packet) are not supported
  - There is a failure to carry out the control request
  - The endpoint is halted (failed)

  In the case of a control endpoint, the stall condition is automatically cleared when the next setup packet arrives. A stalled bulk or interrupt endpoint requires the host to step in and clear the stall condition . The host never sends a STALL.

- **Special**

  The host uses special preamble packets (PRE) before initiating low speed packets. Upon receiving the PRE packet, the USB hubs enable low speed ports. Except for the hubs, all other devices ignore the PRE packet. The PRE packets do not end with an EOP.

Index

P

R

S

T

U

X