MICROCHIP

AN1176

USB Device Stack for PIC32 Programmer's Guide

Author: *Bud Caldwell*
Microchip Technology Inc.

INTRODUCTION

The Universal Serial Bus (USB) has revolutionized the way the world connects peripherals to personal computers (PCs). It provides a simple common interface for nearly any type of peripheral device imaginable. The user simply plugs the peripheral into one of the computer's USB ports or into a hub connected to the computer using a common connector type, installs driver software (if the OS doesn't already support it), and the device is ready to use.

The flexibility and power of the USB requires managing protocols for device identification, configuration, control and data transfer. The Microchip PIC32 USB device firmware stack provides an easy-to-use framework to simplify the development of USB 2.0 compliant peripherals when using supported Microchip microcontroller families.

This application note describes the Microchip PIC32 USB peripheral firmware stack and acts as a programmer's reference manual for developers who need to design firmware for any type of USB peripheral device for which no Microchip sample implementation is available. It describes how to implement a function-specific driver that will interface with the Microchip USB peripheral firmware stack and shows how this simplifies the overall application development.

ASSUMPTIONS

1. Working knowledge of C programming language
2. Familiarity with the USB 2.0 protocol
3. Familiarity with Microchip MPLAB® IDE

FEATURES

- Supports USB peripheral device applications
- Handles standard USB device requests, as stated in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* (available on the Internet at the following URL:
  http://www.usb.org/developers/docs/
- Supports an multiple number of configurations and interfaces
- Simplifies definition of USB descriptors and configuration information
- Optional support for alternate interface settings
- Support for multi-function devices
- Event-driven system (interrupt or polled)
- Provides a simple Application Program Interface (API)
- Provides a simple Function Driver Interface (FDI)

LIMITATIONS

- Supports 32 USB endpoints, the maximum allowed (16 `IN` and 16 `OUT`)
- Supports up to 32 device functions
- Possible configurations limited only by available memory

SYSTEM HARDWARE

The USB firmware stack was developed for the following hardware:

- PIC32 microcontrollers supporting USB

PIC® MCU MEMORY RESOURCE REQUIREMENTS

For complete program and data memory requirements, refer to the release notes located in the installation directory.

PIC® MCU HARDWARE RESOURCE REQUIREMENTS

The Microchip USB device stack firmware uses the following I/O pins:

TABLE 1: PIC® MCU I/O PIN USAGE

I/O Pin	Usage
D+ (IO)	USB D+ differential data signal
D- (IO)	USB D- differential data signal
VBUS (Input)	Senses USB power (does not operate bus powered)
VUSB (Input)	Power input for the USB D+/D- transceivers

INSTALLING SOURCE FILES

The complete device stack source is available for download from the Microchip web site (see **Appendix D: "Source Code for the USB Device Stack Programmer's Guide"**). The source code is distributed in a single Windows® installation file.

Perform the following steps to complete the installation:

1. Execute the installation file. A Windows installation wizard will guide you through the installation process.
2. Before continuing with the installation, you must accept the software license agreement by clicking **I Accept**.
3. After completion of the installation process, you should see a new entry in the "PIC32 Solutions" program group for the PIC32 USB Device Stack. The complete source code will be copied in the selected directory.
4. Refer to the release notes for the latest version-specific features and limitations.

SOURCE FILE ORGANIZATION

The Microchip USB device stack contains the following source and header files:

TABLE 2: SOURCE FILES

File	Directory*	Description
`usb_device.c`	`Microchip\USB`	USB device layer (device abstraction and Ch 9 protocol handling)
`usb_hal.c`	`Microchip\USB`	USB Hardware Abstraction Layer (HAL) interface support
`usb_hal_core.c`	`Microchip\USB`	USB controller functions, used by HAL interface support
`usb_device_local.h`	`Microchip\USB`	Private definitions for USB device layer
`usb_hal_core.h`	`Microchip\USB`	Private definitions for HAL controller core
`usb_hal_local.h`	`Microchip\USB`	Private definitions for HAL
`usb.h`	`Microchip\Include\USB`	Overall USB header (includes all other USB-headers)
`usb_ch9.h`	`Microchip\Include\USB`	USB device framework (Chapter 9 of the *“Universal Serial Bus Specification, Revision 2.0”*) definitions
`usb_common.h`	`Microchip\Include\USB`	Common USB stack definitions
`usb_device.h`	`Microchip\Include\USB`	USB device layer interface definition
`usb_hal.h`	`Microchip\Include\USB`	USB HAL interface definition
`usb_config.h`	`<defined by application>`	Application-specific configuration options (see **Appendix A: “USB Firmware Stack Configuration”**)

*By default, the root of the installation will be `C:\PIC32 Solutions`, unless another location was chosen.

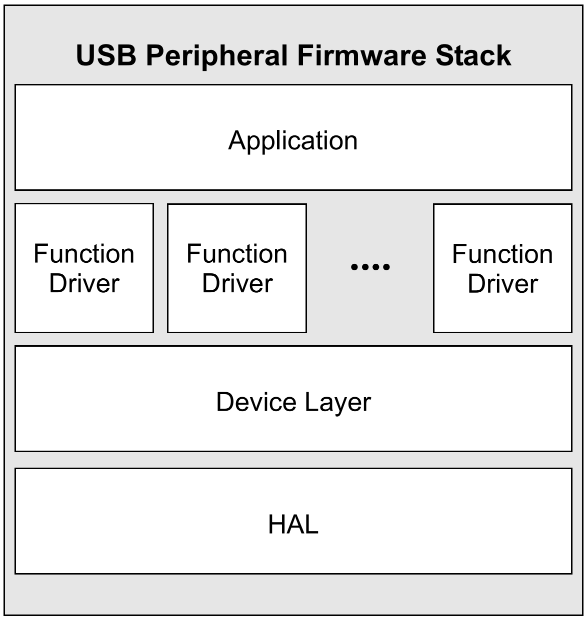
DEMO APPLICATION

The document does not refer to a demo application. It is intended to describe how to use the USB peripheral device stack in the cases where there are no sample applications available that fit the desired usage. See the **"References"** section for sample applications that microchip has provided.

STACK ARCHITECTURE

The USB peripheral firmware stack can be thought of as consisting of 4 layers, as shown in Figure 1.

FIGURE 1: CONCEPTUAL STACK



Application

The application consists of the firmware necessary to implement the device's desired behavior. This is customer designed and implemented code, although it may be based on Microchip supplied sample code. This code may communicate with the USB FW stack or with any other software in the system as necessary.

Note:	The USB descriptors and other configuration options are application-specific. Consequently, they need to be defined by the application. The application must provide a function that the USB firmware stack calls to retrieve the definitions. This function must be identified in the `usb_config.h` header file (see `USB_DEV_GET_DESCRIPTOR_FUNC`).

USB Function Driver

Every USB peripheral device implements a particular function (printer, mouse, mass storage, etc.). Some devices may have multiple functions. Function drivers implement the desired function behavior and provide function-specific control interfaces to the application. To access the USB and transfer data and control information, function drivers interact with the USB device layer.

Microchip supplies function drivers for several of the most commonly requested USB device functions. However, custom development of a specific USB function driver (which is the topic of this document) may be necessary.

USB Device

The device layer abstracts a USB device. It does not make any assumptions about what function that device may implement. Its primary job is to handle the USB protocol elements specified in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"*. It also provides all the access to the USB that a function of any type may need. It does this by implementing a well-defined interface (described within this document) used by all function drivers to access the USB. The device layer then communicates with the HAL as necessary to support this interface.

HAL

The HAL (Hardware Abstraction Layer) abstracts the USB controller hardware. It provides access to all of the features that the controller implements to support a USB peripheral device.

CREATING A USB APPLICATION

This section describes the steps necessary to design and implement a USB peripheral device application, how to implement a function driver, and how to integrate it with the Microchip PIC32 USB device firmware stack.

Overview:

1. Implement the main application.
2. Implement the USB function driver.
3. Implement the application-specific USB support.
4. Configure USB stack options.

Implementing the Main Application

Using MPLAB IDE, create a new application for the supported microcontroller. (Refer to the MPLAB IDE online help for instructions on how to create a project.) Implement and test any non-USB application-specific support desired.

To support the USB FW stack, the application's main function must call `USBInitialize`, once before any other USB activity takes place. After `USBInitialize` has been called, the application must call `USBTasks` in a "polling" loop (as shown in Example 1) or it must directly link `USBTasks` to the processor's USB Interrupt Service Routine (ISR).

EXAMPLE 1: MAIN APPLICATION LOGIC

```
// Initialize the USB stack.
USBInitialize(0);

// Main Processing Loop
while(1)
{
    // Check USB for events and
    // handle them appropriately.
     USBHandleEvents();

     // Perform any additional
    // IO processing needed.
}
```

The interface between the application and the function driver is completely up to the designer of the function driver, thus it is beyond the scope of this document. However, it is recommended that the application implement an event routine similar to the `Func-Event Handling Routine` defined by the Function Driver Interface (FDI) to receive events from the USB stack. If this is done, the function driver can be designed to call the application and pass events to it similar to the way the driver receives events from the USB stack. (See **"Implementing the USB Function Driver"**.)

Notes:	Code executing within the polling loop must not block or wait on anything taking more then a few microseconds. If blocking behavior is required, the USB FW stack must be used in an interrupt-based environment. If executed in an interrupt-based environment, the function driver's event handling routine (and the application's, if one is implemented) will be called in an interrupt context.

Implementing the USB Function Driver

The purpose of the USB function driver is to implement the features of the class-or-vendor-specific USB function. The function driver must interface with the FDI routines to transfer data on the USB and to receive notification of events that occur on the bus.

Initialization

The function driver is initialized by the USB stack when the device configuration has been selected by the host. In order for this to happen, the driver must implement an initialization routine with a specific C-language-function signature. This routine is called via a pointer in the "Function Driver Table".

The purpose of the initialization routine (see Example 2) is to reset and initialize the state of the function driver and prepare for any function-specific activity on the bus.

EXAMPLE 2: INITIALIZATION ROUTINE

```
BOOL USBGenInitialize ( unsigned long flags
{
    // Initialize the driver state
    memset(&gGenFunc, sizeof(gGenFunc), 0);

    // Set initialized flag!
    gGenFunc.flags =
      GEN_FUNC_FLAG_INITIALIZED;

    return TRUE;
}
```

Notice that all the example routine does is initialize the data structure that maintains the function's state (`gGenFunc`). Actual initialization of the endpoint hardware is handled by the USB FW stack, based on an application-specific configuration table.

Event Handling

The other thing that a function driver must do is handle device-or-function-specific events that occur on the USB. To do this, it must implement an event-handling routine with a specific C language function signature. This routine is called by a lower level of the USB stack using a pointer in the Function Driver Table.

The purpose of the event-handling routine is to respond to the appropriate events and provide the required behavior. Events are defined by the `USB_EVENT` enumerated data type found in the in the `usb_common.h` header file. The function driver's event-handling routine must perform the correct action to support the desired function behavior. Exactly what action must be taken for each event is function-specific and beyond the scope of this document. The function-driver designer must have detailed knowledge of the required behavior for the desired USB peripheral function in order to implement a driver for it.

Example 3 shows when the function driver should call the `USBDEVGetLastError` FDI routine. This routine must be called when an `EVENT_BUS_ERROR` event is "thrown" to the function driver by the stack to clear error indication bits. Most of these events will be handled directly by the stack itself. However, if it is not handled by the stack, the function driver may need to take error-specific and function-specific action.

The other thing that most function drivers will need to do is transfer data across the USB. This is done by calling the `USBDEVTransferData` FDI routine. The usual reason to call this routine is when a function-specific request has been received, indicated by an `EVENT_TRANSFER` event, and it has been decoded as a function-specific request to transfer data. The event-handling logic of the driver may then need to call the `USBDEVTransferData` routine to satisfy the request. For an example of how to use these FDI functions, see their descriptions in **Appendix C: "USB Function Driver Interface"**.

EXAMPLE 3: EVENT-HANDLING ROUTINE

```
BOOL USBGenEventHandler ( USB_EVENT event, void *data, unsigned int size )
{
    unsigned long error;

    // Abort if not initialized.
    if ( !(gGenFunc.flags & GEN_FUNC_FLAG_INITIALIZED) ) {
        return FALSE;
    }

    // Handle specific events.
    switch (event)
    {
    case EVENT_TRANSFER:      // A USB transfer has completed.
        return HandleTransferDone((USB_XFER_EVT_DATA *)data);

    case EVENT_DETACH:    // USB cable has been detached
        // De-initialize the general function driver.
        gGenFunc.flags   = 0;
        gGenFunc.rx_size = 0;
        return TRUE;

    case EVENT_BUS_ERROR:   // Error on the bus
        error = USBDEVGetLastError();
        // Should capture the error and do something about it.
        return TRUE;

    // Handle any other events required by the application or function.

    default:            // Unknown event
        return FALSE;
    }
}
```

Notes: Code executed within the context of the event-handling routine must not block.

Implementing the Application-Specific USB Support

In order to integrate one or more function drivers with the USB FW stack and to configure the stack for the application, the user must define three tables and implement functions (or macros) to provide the USB device stack with access to them.

Application-Specific USB Tables:

1. USB Descriptor Table
2. Endpoint Configuration Table
3. Supported-Function-Drivers Table

All three tables are interrelated. Together, they identify the features, endpoint configurations, and functions that are supported by the USB stack and the device itself.

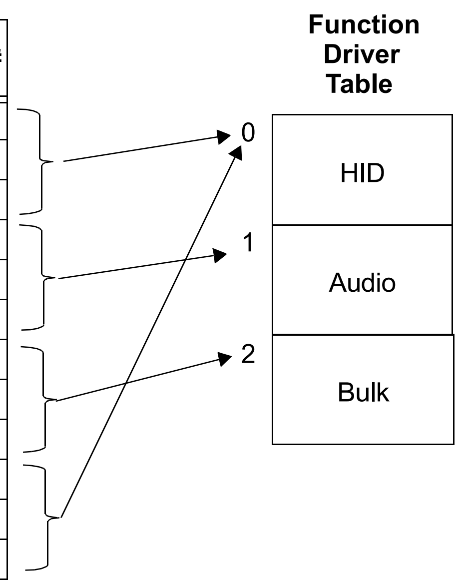
The requirements for the USB descriptors are defined in the *"Universal Serial Bus Specification 2.0"* and in the class-specific supplements for practically any class of device that the user might want to design. Section **"Implementing the USB Descriptor Table"** describes a method for implementing these descriptors and providing the USB FW stack with access to them.

The endpoint configuration and function driver tables allow the application to support a device with any number of endpoint configurations and practically any number of USB peripheral device functions. The endpoint configuration table identifies which function driver should receive events for each endpoint (for each configuration, interface, and alternate interface setting). A graphical depiction of the function driver table is shown in Figure 2. The arrows show the relation ship between the entries in the endpoint configuration table and the function driver table.

The following sub-sections describe these tables in detail and show how to implement them, and their access routines.

FIGURE 2: ENDPOINT CONFIGURATION AND FUNCTION DRIVER TABLES

Config. #	Intf #	Alt Intf #	EP #	EP Config. Data	Func. #
1	0	0	1	Tx, HndShk	0
1	0	0	2	Rx, HndShk	0
1	0	0	3	Tx, HndShk	0
1	1	0	5	Tx, HndShk	1
1	1	0	7	Rx, HndShk	1
1	1	0	9	Rx, HndShk	1
1	1	1	5	Tx	2
1	1	1	7	Rx	2
1	1	1	9	Rx	2
2	0	0	1	Tx, HndShk	0
2	0	0	2	Rx, HndShk	0
2	0	0	3	Tx, HndShk	0



IMPLEMENTING THE USB DESCRIPTOR TABLE

Every USB device must be able to provide a set of descriptors (data structures), describing the device and providing details to the host about how to use it. Exactly how these descriptors must be provided and what information they must contain is defined in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* and its class-specific supplements. Please refer to these documents for full details. In the Microchip USB stack, these descriptors are created using data types defined in the `usb_ch9.h` header file.

The descriptors for a USB device can be thought of as belonging to one of three different groups: those describing the overall device, those describing possible device configurations, and those providing user-readable information. Each USB device has one and only one device descriptor, to uniquely identify the device and give the number of possible configurations. Each configuration has its own set of descriptors describing the details of that configuration. There may also be any number of user-readable "string" descriptors.

Example 4 shows an example of how a USB descriptor table might be defined using the provided type definitions for the USB descriptors.

EXAMPLE 4: DESCRIPTOR TABLE DEFINITION

```
#define NUM_LANGS   1
#define LANG_1_ID   0x0409                                     // English
#define STR_1_LEN   25
#define STR_2_LEN   27
#define STR_3_LEN   10

typedef struct _config1_descriptors
{
    USB_CONFIGURATION_DESCRIPTOR     cfg_desc;                   // Configuration 1
    USB_INTERFACE_DESCRIPTOR         intf0_desc;                // Config 1, Interface 0
    USB_ENDPOINT_DESCRIPTOR          intf0_ep1_in_desc;         // Endpoint 0 in (Tx)
    USB_ENDPOINT_DESCRIPTOR          intf0_ep1_out_desc;        // Endpoint 0 out (Rx)
} CONFIG1_DESC, *PCONFIG1_DESC;

typedef struct _string0_descriptor
{
    USB_STRING_DESCRIPTOR            string;                    // String0 Descriptor
    WORD                             langid[NUM_LANGS];
} STR0_DESC, *PSTR_DESC;

typedef struct _string1_descriptor
{
    USB_STRING_DESCRIPTOR            string;                    // String1 Descriptor
    WORD                             string_data[STR_1_LEN];
} STR1_DESC, *PSTR1_DESC;

typedef struct _string2_descriptor
{
    USB_STRING_DESCRIPTOR            string;                    // String2 Descriptor
    WORD                             string_data[STR_2_LEN];
} STR2_DESC, *PSTR2_DESC;

typedef struct _string3_descriptor
{
    USB_STRING_DESCRIPTOR            string;                    // String3 Descriptor
    WORD                             string_data[STR_3_LEN];

} STR3_DESC, *PSTR3_DESC;
```

Note:	There is no endpoint descriptor for Endpoint zero (`0`). The *"Universal Serial Bus Specification, Revision 2.0"* explicitly defines the behavior of Endpoint zero. Only the packet size is configurable.

The “device” descriptor (which uses the standard definition provided in `usb_ch9.h`) can be provided alone. The Configuration descriptors (`cfg_desc`, `intf0_desc`, `intf0_ep1_in_desc`, and `intf0_ep1_out_desc`) must be provided together as a contiguous set. The host will request the specific amount of data it wishes to receive along with a zero-based index indicating which configuration-descriptor set it wishes to receive. String descriptors will also be requested by index (string 0, string 1, etc.) as well as by language ID.

> **Note:** The string descriptor consists of both the `USB_STRING_DESCRIPTOR` structure and the WORD array of string data, stored contiguously in memory.
>
> Descriptor data, which must be contiguous, should be placed in packed structures.

Using the descriptor table definition shown in Example 4, an example of how to initialize it is shown in Example 5.

EXAMPLE 5: DESCRIPTOR TABLE INITIALIZATION

```
USB_DEVICE_DESCRIPTOR  dev_desc =
{
    sizeof(USB_DEVICE_DESCRIPTOR),        // Size of this descriptor in bytes
    USB_DESCRIPTOR_DEVICE,                // DEVICE descriptor type
    0x0200,                               // USB Spec Release Number in BCD format
    0x00,                                 // Class Code
    0x00,                                 // Subclass code
    0x00,                                 // Protocol code
    USB_DEV_EP0_MAX_PACKET_SIZE,          // Max packet size for EP0, see usbcfg.h
    0x04D8,                               // Vendor ID
    0x000C,                               // Product ID: PICDEM FS USB (DEMO Mode)
    0x0000,                               // Device release number in BCD format
    0x01,                                 // Manufacturer string index
    0x02,                                 // Product string index
    0x00,                                 // Device serial number string index
    0x01                                  // Number of possible configurations
};

CONFIG1_DESC           config1 =
{
    {   /* Configuration Descriptor */
        sizeof(USB_CONFIGURATION_DESCRIPTOR),// Size of this descriptor in bytes
        USB_DESCRIPTOR_CONFIGURATION,        // CONFIGURATION descriptor type
        sizeof(CONFIG1_DESC),                // Total length of data for this cfg
        1,                                   // Number of interfaces in this cfg
        USBGEN_CONFIG_NUM,                   // Index value of this configuration
        0,                                   // Configuration string index
        0x01<<7,                             // Attributes, see usbdefs_std_dsc.h
        50                                   // Max power consumption (2X mA)
    },
    {   /* Interface Descriptor */
        sizeof(USB_INTERFACE_DESCRIPTOR),    // Size of this descriptor in bytes
       USB_DESCRIPTOR_INTERFACE,             // INTERFACE descriptor type
        USBGEN_INTF_NUM,                     // Interface Number
        0,                                   // Alternate Setting Number
        2,                                   // Number of endpoints in this intf
        0x00,                                // Class code
        0x00,                                // Subclass code
        0x00,                                // Protocol code
        0                                    // Interface string index
    },
    /* Endpoint Descriptors */
    {   /* EP 1 - Out */
        sizeof(USB_ENDPOINT_DESCRIPTOR),
        USB_DESCRIPTOR_ENDPOINT,
        {EP_DIR_OUT|USBGEN_EP_NUM},
        {EP_ATTR_INTR},
        EP_MAX_PKT_INTR_FS,
        32
    },
```

```
    {   /* EP 1 - In */
        sizeof(USB_ENDPOINT_DESCRIPTOR),
        USB_DESCRIPTOR_ENDPOINT,
        {EP_DIR_IN|USBGEN_EP_NUM},
        {EP_ATTR_INTR},
        EP_MAX_PKT_INTR_FS,
        32
    }
};

STR0_DESC string0 =
{
    {   // Language ID: English
        sizeof(STR0_DESC),
        USB_DESCRIPTOR_STRING
    },
    {LANG_1_ID}
};

STR1_DESC string1 =
{
    {   // Vendor Description
        sizeof(STR1_DESC),
        USB_DESCRIPTOR_STRING
    },
    {'M','i','c','r','o','c','h','i','p',' ',
    'T','e','c','h','n','o','l','o','g','y',' ',
    'I','n','c','.'}
};

STR2_DESC string2 =
{
    {   // Device Description
        sizeof(STR2_DESC),
        USB_DESCRIPTOR_STRING
    },
    {'P','I','C','3','2',' ','P','I','C','D','E','M',' ',
     'D','e','m','o',' ','E','m','u','l','a','t','i','o','n'}
};

STR3_DESC string3 =
{
    {   // Serial Number
        sizeof(STR3_DESC),
        USB_DESCRIPTOR_STRING
    },
    {'0','0','0','0','0','0','0','0','0','0'}
};
```

Along with the necessary set of descriptors, the application must also provide a routine to access them. Example 6 shows an implementation of this routine. It must have the C-language-function signature described by the `USB API – USB_DEV_GET_DESCRIPTOR_FUNC` definition of **“Application Programming Interface”**.

EXAMPLE 6: GET DESCRIPTOR ROUTINE AND SUPPORT CODE

```
static inline const void *GetConfigurationDescriptor( BYTE config, unsigned int *length )
{
    switch (config)
    {
    case 0: // Configuration 1 (default)
        *length = sizeof(config1);
        return &config1;

    default:
        return NULL;
    }

} // GetConfigurationDescriptor

static inline const void *GetStringDescriptor( PDESC_ID desc, unsigned int *length )
{
    // Check language ID
    if (desc->lang_id != LANG_1_ID) {
        return NULL;
    }

    // Get requested string
    switch(desc->index)
    {
    case 0: // String 0
        *length = sizeof(string0);
        return &string0;

    case 1: // String 1
        *length = sizeof(string1);
        return &string1;

    case 2: // String 2
        *length = sizeof(string2);
        return &string2;

    case 3: // String 3
        *length = sizeof(string3);
        return &string3;

    default:
        return NULL;
    }

} // GetStringDescriptor

const void *USBDEVGetDescriptor ( PDESC_ID desc, unsigned int *length )
{
    switch (desc->type)
    {
    case USB_DESCRIPTOR_DEVICE:       // Device Descriptor
        *length = sizeof(dev_desc);
        return &dev_desc;

    case USB_DESCRIPTOR_CONFIGURATION:// Configuration Descriptor
        return GetConfigurationDescriptor(desc->index, length);

    case USB_DESCRIPTOR_STRING:       // String Descriptor
        return GetStringDescriptor(desc, length);
```

```
    // Fail all un-supported descriptor requests:

    default:
        return NULL;
    }

} // USBDEVGetDescriptor
```

Note: In Example 6, the `USBDEVGetDescriptor` routine is implemented using the inline helper functions `GetStringDescriptor` and `GetConfigurationDescriptor` to make the code more readable without incurring the overhead of a function call.

IMPLEMENTING THE ENDPOINT CONFIGURATION TABLE

The endpoint configuration table identifies direction and protocol features for every endpoint required by each interface or alternate setting for every supported configuration of the USB device. The table also identifies which function driver will be used to service events that occur related to each endpoint. The only exception is that Endpoint zero (`0`) is configured automatically by the stack and is not included in the endpoint configuration table.

The `EP_CONFIG` structure and flags are defined in the `usb_device.h` header file.

Each entry in the table is made up of the following data structure:

FIGURE 3: ENDPOINT CONFIGURATION TABLE STRUCTURE

```
typedef struct
{
    UINT16  max_pkt_size;
    UINT16  flags;
    BYTE    config;
    BYTE    ep_num;
    BYTE    intf;
    BYTE    alt_intf;
    BYTE    function;

} EP_CONFIG, *PEP_CONFIG;
```

The `max_pkt_size` field defines how many bytes this endpoint can transfer in a single packet. The `ep_num` field identifies which endpoint the structure describes. The `config`, `intf`, and `alt_intf` fields identify which device configuration, interface and alternate interface setting that this structure describes. The `function` field identifies which function driver uses the endpoint identified by `ep_num`. It does this by providing the index into the "Supported-Function-Drivers Table", as illustrated by the arrows in Figure 2. The `flags` field provides the information used to configure the behavior of the endpoint. The flags are described in Table 3.

TABLE 3: ENDPOINT CONFIGURATION FLAGS

Flag	Description
`USB_EP_TRANSMIT`	Enable endpoint for transmitting data
`USB_EP_RECEIVE`	Enable endpoint for receiving data
`USB_EP_HANDSHAKE`	Enable generation of handshaking (ACK/NAK) packets (non-isochronous endpoints only)
`USB_EP_NO_INC`	Used only for direct DMA to another device's FIFO

TERMINOLOGY

The terminology can be confusing when discussing the direction of data flow on the bus.

The USB specification uses the term `OUT` to refer to data flow from the host (PC) to the device (peripheral) and the term `IN` to refer to data flow from the device to the host.

Since the USB interface on the microcontroller may also support USB host functionality, the Microchip PIC32 USB stack uses the term `TRANSMIT` to refer to data flowing out of the microcontroller (onto the bus) and the term `RECEIVE` to refer to data flowing from the USB into the microcontroller. To help clarify, the following table summarizes the relationship between these terms.

TABLE 4: DATA FLOW DIRECTION SUMMARY FOR A PERIPHERAL DEVICE

USB Term	FW Stack Term	Description
`IN`	`TRANSMIT`	Data flows from the device to the host.
`OUT`	`RECEIVE`	Data flows from the host to the device.

Simple Example

The following code snippet provides an example of how to initialize the endpoint configuration table in a way that is consistent with the descriptor table shown in Example 4 (see **"Implementing the USB Descriptor Table"**). It is vital that the information reported to the host in the descriptor table match with the information provided in the endpoint configuration table that is used to configure the hardware.

EXAMPLE 7: SIMPLE ENDPOINT CONFIGURATION TABLE

```
const EP_CONFIG gEpConfigTable[] =
{
    {   // EP1 - In & Out
        EP_MAX_PKT_INTR_FS,     // Maximum packet size for this endpoint
        USB_EP_TRANSMIT |       // Configuration flags for this endpoint
        USB_EP_RECEIV

        USB_EP_HANDSHAKE,
        USBGEN_EP_NUM,          // Endpoint number.
        USBGEN_CONFIG_NUM,      // Configuration number
        USBGEN_INTF_NUM,        // Interface number
        0,                      // Alternate interface setting
        0                       // Index in device function table (see below)
    }
```

Complex Example

A more complex device might have multiple configurations, or multiple interfaces, within a configuration. Example 8 presents an endpoint configuration table that could be for a device that has two configurations. Configuration 1 has two interfaces (`0` and `1`). Each interface has two endpoints, one for transmitting data, and one for receiving it. Configuration 2 has one interface with two endpoints; again, one for transmitting data and one for receiving it.

EXAMPLE 8: COMPLEX ENDPOINT CONFIGURATION TABLE

```
const EP_CONFIG gEpConfigTable[] =
{
    // Device Configuration 1 Endpoint Configurations.
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_TRANSMIT|        // Configuration flags for this endpoint
        USB_EP_HANDSHAKE,
        1,                      // Endpoint number.
        1,                      // Configuration number (starts at 1)
        0,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    },
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_RECEIVE|         // Configuration flags for this endpoint
        USB_EP_HANDSHAKE,
        2,                      // Endpoint number.
        1,                      // Configuration number (starts at 1)
        0,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    },
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_TRANSMIT|        // Configuration flags for this endpoint
        USB_EP_HANDSHAKE,
        3,                      // Endpoint number.
        1,                      // Configuration number (starts at 1)
        1,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    },
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_RECEIVE|         // Configuration flags for this endpoint (see below)
        USB_EP_HANDSHAKE,
        4,                      // Endpoint number.
        1,                      // Configuration number (starts at 1)
        1,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    },
    // Device Configuration 2 Endpoint Configurations.
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_TRANSMIT|        // Configuration flags for this endpoint
        USB_EP_HANDSHAKE,
        1,                      // Endpoint number.
        2,                      // Configuration number (starts at 1)
        0,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    },
    {
        64,                     // Maximum packet size for this endpoint
        USB_EP_RECEIVE|         // Configuration flags for this endpoint
        USB_EP_HANDSHAKE,
        2,                      // Endpoint number.
        2,                      // Configuration number (starts at 1)
        0,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table
    }
};
```

IMPLEMENTING THE SUPPORTED FUNCTION DRIVERS TABLE

Since a device may implement more then one class-or-vendor specific USB function, the Microchip PIC32 USB FW stack uses a table to manage access to supported function drivers. Each entry in the table contains the information necessary to manage a single function driver. If a device only implements one USB function, the table will only contain one entry. The following data structure defines an entry in the function-driver table.

FIGURE 4: FUNCTION DRIVER TABLE ENTRY

```
struct _function_driver_table_entry
{
    USBDEV_INIT_FUNCTION_DRIVER Initialize;      // Init routine
    USB_EVENT_HANDLER           EventHandler;    // Event routine
    BYTE                        flags;           // Init flags
};
```

The `Initialize` field holds a pointer to the function driver's initialization routine. The `EventHandler` field holds a pointer to the function driver's routine for handling vender-or-class-specific USB events. The flags field contains any driver-specific flags that will be passed into the initialization routine. Refer to **"Implementing the USB Function Driver"** for details on what these routines do and how to implement them.

The data in these three fields is all that is required to for the USB stack to manage the function driver. The table entry allows the USB stack to dynamically choose which function driver is called once the host has configured the device. The function driver itself can directly link to the `USBDEVTransferData` and `USBDEVGetLastError` routines, using the normal method since there is only one implementation for each of these routines in the USB stack. Example 9 shows a sample implementation.

EXAMPLE 9: FUNCTION DRIVER TABLE

```
const FUNC_DRV gDevFuncTable[2] =
{
    {   // General Function Driver
        USBGenInitialize,           // Init routine
        USBGenEventHandler,         // Event routine
        2                           // Endpoint Number (bottom 4 bits)
    },
    {   // HID function Driver
        USBHIDInitialize,           // Init routine
        USBHIDEventHandler,         // Event routine
        0                           // No flags supported
    }
};
```

In addition to the table, the application must implement a routine or macro to provide the base address of the table. Example 10 shows how this might be implemented.

The routine only needs to provide a pointer to the base of the table. No information is needed regarding the number of entries in the table since the endpoint configuration table provides the indices of every possible entry in the function driver table (see **"Implementing the Endpoint Configuration Table"**).

EXAMPLE 10: GET FUNCTION DRIVER TABLE ROUTINE

```
const FUNC_DRV *USBDEVGetFunctionDriverTable ( void )
{
    return gDevFuncTable;
}
```

Configuring the USB Stack Options

This section highlights several key configuration options necessary to ensure proper operation of the USB peripheral device stack.

First, to ensure that the USB stack is built for Peripheral-Device-Only mode, be sure to define the `USB_SUPPORT_DEVICE` macro. Otherwise, the behavior of the USB stack will not be appropriate for a USB peripheral device application. Second, to ensure that the USB stack does not allocate any more RAM then is required, be sure to define the following macros correctly:

Macros Directly Effecting RAM Usage:

- `USB_DEV_HIGHEST_EP_NUMBER`
- `USB_MAX_NUM_PIPES`
- `USB_DEV_EP0_MAX_PACKET_SIZE`
- `USB_DEV_SUPPORTS_ALT_INTERFACES`

The first three of these macros must be defined as an appropriate integer, as required for the device's application. RAM is allocated to track state information for each endpoint used, from Endpoint zero (`0`) up to the highest endpoint number used. So, one way to conserve RAM is to allocate the endpoints required from the lowest numbers available. To indicate this to the USB stack, define the `USB_DEV_HIGHEST_EP_NUMBER` macro to be equal to this number.

Endpoint zero (`0`) can support buffer sizes of 8, 16, 32, or 64 bytes. The RAM for the this buffer is allocated based upon how the `USB_DEV_EP0_MAX_PACKET_SIZE` macro is defined. It must be defined to equal one of these values.

If support for alternate interfaces is required, the macro `USB_DEV_SUPPORTS_ALT_INTERFACES` must be defined. Otherwise, the USB stack will not support the use of USB interfaces with alternate settings. This support is not always required, and including it will use additional RAM and Flash (see **"PIC® MCU Memory Resource Requirements"**).

Third, to ensure that the USB stack can call the three user-defined routines described in **"Implementing the Application-Specific USB Support"**, the `USB_DEV_GET_DESCRIPTOR_FUNC`, `USB_DEV_GET_EP_CONFIG_TABLE_FUNC`, and `USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC` macros must be defined to equal the names of their associated routines. The following example shows how these macros would be defined using the example routines shown in **"Implementing the Application-Specific USB Support"**.

EXAMPLE 11: FUNCTION IDENTIFICATION MACRO DEFINITIONS

```
#define USB_DEV_GET_DESCRIPTOR_FUNC           USBDEVGetDescriptor
#define USB_DEV_GET_EP_CONFIG_TABLE_FUNC      USBDEVGetEpConfigurationTable
#define USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC USBDEVGetFunctionDriverTable
```

Note:	See **Appendix A: "USB Firmware Stack Configuration"** for additional details on configuration options.

CONCLUSION

The Microchip PIC32 USB peripheral firmware stack makes it easy for a developer to manage USB device identification, configuration, control and data transfer. The stack simplifies support for practically any number of configurations or interfaces. Most importantly, it provides a simple function-driver interface that makes it easy to design a single or multi-function device.

REFERENCES

- "*Universal Serial Bus Specification, Revision 2.0"* http://www.usb.org/developers/docs
- "*OTG Supplement, Revision 1.3*" http://www.usb.org/developers/onthego
- Microchip MPLAB® IDE
  In-circuit development environment, available free of charge, by license, from www.microchip.com/mplabide
- Microchip Application Note AN1163*, "USB HID Class on an Embedded Device"*
- Microchip Application Note AN1169*, "USB Mass Storage Class on an Embedded Device"*
- Microchip Application Note AN1164*, "USB CDC Class on an Embedded Device"*
- Microchip Application Note AN1166*, "USB Generic Function on an Embedded Device"*

APPENDIX A: USB FIRMWARE STACK CONFIGURATION

The peripheral stack provides several configuration options to customize it for your application. The configuration options must be defined in the file `usb_config.h` that must be implemented as part of any USB application. Once any option is changed, the stack must be built "clean" to rebuild all related binary files.

The following is a list of peripheral stack configuration options:

- `USB_SUPPORT_DEVICE`
- `USB_DEV_EVENT_HANDLER`
- `USB_DEV_HIGHEST_EP_NUMBER`
- `USB_DEV_EP0_MAX_PACKET_SIZE`
- `USB_DEV_SUPPORTS_ALT_INTERFACES`
- `USB_DEV_GET_DESCRIPTOR_FUNC`
- `USB_DEV_GET_EP_CONFIG_TABLE_FUNC`
- `USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC`
- `USB_DEV_SELF_POWERED`
- `USB_DEV_SUPPORT_REMOTE_WAKEUP`
- `USB_SAFE_MODE`

USB_SUPPORT_DEVICE

Purpose This macro determines that the application being implemented supports peripheral device operation.

Precondition None

Valid Values This macro does not need to have a value assigned to it. Defining it is sufficient to select the USB role of the application.

Default: Not defined

Example `#define USB_SUPPORT_DEVICE`

USB_DEV_EVENT_HANDLER

Purpose This macro identifies the name of the bus-event-handling routine for the device support layer of the USB stack. The device support layer handles all standard (Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"*) requests. The macro should always be defined as shown in the example unless the user wishes to handle standard device requests directly.

Precondition None

Valid Values This macro needs to be equal to the name of a routine capable of handling all USB device requests.

Default: `USBDEVHandleBusEvent`

Example `#define USB_DEV_EVENT_HANDLER   USBDEVHandleBusEvent`

USB_DEV_HIGHEST_EP_NUMBER

Purpose This macro determines the highest endpoint number to be used by the application.

> **Note:** The USB peripheral SW stack will use additional RAM on a per-endpoint basis to manage data transfer (see **"PIC® MCU Memory Resource Requirements"**).

Precondition None

Valid Values Valid values are any integer between 1 and 15.

Default: None – must be defined by application

Example `#define USB_DEV_HIGHEST_EP_NUMBER15`

USB_DEV_EP0_MAX_PACKET_SIZE

Purpose This macro defines the maximum packet size allowed for Endpoint `0`.

> **Note:** The USB peripheral SW stack will use additional bytes of RAM equal to the definition of this macro (see **"PIC® MCU Memory Resource Requirements"**).

Precondition None

Valid Values This macro must be defined as 8, 16, 32, or 64 bytes

Default: 8

Example `#define USB_DEV_EP0_MAX_PACKET_SIZE8`

USB_DEV_SUPPORTS_ALT_INTERFACES

Purpose When this macro is defined, the USB device FW stack includes support for alternate interfaces within a single configuration.

> **Note:** The USB device FW stack will use additional Flash and RAM to manage alternate interfaces when this macro is defined (see **"PIC® MCU Memory Resource Requirements"**).

Precondition None

Valid Values This macro does not need to have a value assigned to it. Defining it is sufficient to enable support for alternate interfaces

Default: Not defined

Example

```
#define USB_DEV_SUPPORTS_ALT_INTERFACES
```

USB_DEV_GET_DESCRIPTOR_FUNC

Purpose This macro defines the name of the routine that provides the descriptors to the USB FW stack. This routine must be implemented by the application. The signature of the function must match that defined in the `usb_device.h` header.

Precondition None

Valid Values This macro must be defined to equal the name of the application's "get descriptor" routine to support USB peripheral device operation.

Default: None – must be defined by application

Example

```
#define USB_DEV_GET_DESCRIPTOR_FUNC     USBDEVGetDescriptor
```

USB_DEV_GET_EP_CONFIG_TABLE_FUNC

Purpose This macro defines the name of the routine that provides a pointer to the endpoint configuration table used to configure endpoints as desired. The signature of the function must match the one defined in `usb_device.h`.

Precondition None

Valid Values This macro must be defined to equal the name of the application's "get endpoint configuration table" routine to support USB peripheral device operation.

Default: None – must be defined by application

Example

```
#define USB_DEV_GET_EP_CONFIG_TABLE_FUNC USBDEVGetEpConfigurationTable
```

USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC

Purpose This macro defines the name of the routine that provides the pointer to the function driver table. The signature of the function must match the one defined in `usb_device.h`.

Precondition None

Valid Values This macro must be defined to equal the name of the application's "get function driver table" routine to support USB peripheral device operation.

Default: None – must be defined by application

Example

```
#define USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC \
        USBDEVGetFunctionDriverTable
```

USB_DEV_SELF_POWERED

Purpose This should be defined if the system acts as a self powered USB peripheral device.

> **Note:** Must match the information provided in the descriptors.

Precondition None

Valid Values This macro does not need to have a value assigned to it. Defining it is sufficient to enable support for self powered devices in the USB peripheral SW stack.

Default: Not defined

Example `#define USB_DEV_SELF_POWERED`

USB_DEV_SUPPORT_REMOTE_WAKEUP

Purpose This should be defined if the system is to support remotely waking up a host.

Precondition None

Valid Values This macro does not need to have a value assigned to it. Defining it is sufficient to enable support for remote wake-up.

Default: Not defined

Example `#define USB_DEV_SUPPORT_REMOTE_WAKEUP`

USB_SAFE_MODE

Purpose Define this macro to enable parameter and bounds checking in various places throughout the USB SW stack.

> **Note:** This feature can be removed for efficiency by not defining this label once careful testing and debugging have been done.

Precondition None

Valid Values This macro does not need to have a value assigned to it. Defining it is sufficient to enable safe mode.

Default: Not defined

Example `#define USB_SAFE_MODE`

APPENDIX B: APPLICATION PROGRAMMING INTERFACE

This section describes the Application Programming Interface (API) to the USB device firmware stack. This API is used by the application to initialize and maintain the USB firmware stack. It also provides application-specific configuration information such as device descriptors, endpoint configurations, and access to function drivers.

Some of these API routines are implemented by the USB firmware stack and are called directly by the application. Others are commonly referred to as "callouts" (or more correctly, "calls out"). These functions must be implemented by the application and are called "OUT" of the USB firmware stack into the application. In order for the USB firmware stack to know which routines to call, the function names of the "callout" routines must be identified during configuration of the stack. Refer to the `USB_DEV_GET_DESCRIPTOR_FUNC`, `USB_DEV_GET_EP_CONFIG_TABLE_FUNC`, and `_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC` macros in the **"USB Firmware Stack Configuration"** section to see how these routines are identified.

Table 5 summarizes the API and identifies which routines are call `IN` routines and which ones are call `OUT` routines.

TABLE 5: USB API SUMMARY

Operation	Call Type	Description
`USBInitialize`	`IN`	Initializes the USB firmware stack.
`USBHandleEvents`	`IN`	Identifies and handles bus events.
`USB_DEV_GET_DESCRIPTOR_FUNC`	`OUT`	This function must be implemented by the application to provide a pointer to the requested descriptor.
`USB_DEV_GET_EP_CONFIG_TABLE_FUNC`	`OUT`	This function must be implemented by the application to provide a pointer to the endpoint configuration table.
`USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC`	`OUT`	This function must be implemented by the application to provide a pointer to the function driver table.

Detailed descriptions of the API routines follow.

USB API – USBInitialize

This function performs initialization of the USB firmware stack, clears the USB state and attempts to connect to the bus.

Syntax

`BOOL USBInitialize (unsigned long flags)`

Parameters

`flags` – USB Initialization Flags (reserved, pass zero)

Return Values

`TRUE` if successful,
`FALSE` if not

Preconditions

None

Side Effects

The USB peripheral firmware stack has been initialized and the system is waiting for a connection on the bus.

Example

```
// Initialize the USB stack.
if (!USBInitialize(0))
    return FALSE;
```

Note:	This "function" may actually be implemented as a macro that calls more than one actual function to initialize multiple USB FW layers, depending on the current configuration of the USB stack.

USB API – `USBTasks`

This is the main USB state machine event “pump” routine. It checks for USB events that may have occurred and handles them appropriately. It may be called by the application in a polling loop or it may be called directly in response to the USB (or possibly a timer) interrupt.

Syntax

`void USBTasks (void)`

Parameters

None

Return Values

None

Preconditions

`USBInitialize` must have been called and returned a success indication.

Side Effects

Side effects will vary greatly, depending on the state of the USB peripheral firmware and activity on the bus. This routine will identify the bus event that has occurred and take appropriate action if it can. If the USB peripheral firmware cannot directly handle the event, then calling this routine will result in a “call out” to one or more function drivers to handle class and vendor specific events. The function driver may then call out to the application, depending on its design.

Example

```
// Main Processing Loop
while(1)
{
    // Check USB for events and
    // handle them appropriately.
    USBHandleEvents();

    // Handle other IO activity.
}
```

Note:	This “function” may actually be implemented as a macro that calls more than one actual function to maintain multiple USB FW layers, depending on the current configuration of the USB stack.

USB API – USB_DEV_GET_DESCRIPTOR_FUNC

This routine is a "call out" from the USB firmware that must be implemented by the application. The actual name is defined by the application. The device layer will call it using the USB_DEV_GET_DESCRIPTOR_FUNC macro (see **"USB Firmware Stack Configuration"**), which must be defined to equal the routine's actual name. The routine will be called in response to a GET_DESCRIPTOR request from the host. It must provide a pointer to and length of the indicated descriptor(s).

Syntax

const void * USB_DEV_GET_DESCRIPTOR_FUNC (BYTE type, BYTE index, unsigned int *length)

Where USB_DEV_GET_DESCRIPTOR_FUNC is an application-defined function name

Parameters

type – Identifies the type of descriptor requested.

index – Index of the desired descriptor.

length – Pointer to the variable that will receive the length of the requested descriptor.

Return Values

Returns a pointer to the requested descriptor(s)

Preconditions

USBInitialize must have been called and returned a success indication.

Side Effects

None

Example

```
// This is a sample implementation.  This routine is not called by the application.
const void *USBDEVGetDescriptor (BYTE type, BYTE index, unsigned int *length)
{
    switch (type)
    {
    case USB_DESCRIPTOR_DEVICE:       // Device Descriptor
        *length = sizeof(dev_desc);
        return &dev_desc;

    case USB_DESCRIPTOR_CONFIGURATION:// Configuration Descriptor
        *length = sizeof(conf_desc);
        return &conf_desc;

    // Handle other descriptors as needed.
    }
    return NULL;
}
```

USB API – USB_DEV_GET_EP_CONFIG_TABLE_FUNC

This routine is a "call out" from the device layer that must be implemented by the application. Since the routine's name is defined by the application, the USB firmware will call it using the USB_DEV_GET_EP_CONFIG_TABLE_FUNC macro (see **"USB Firmware Stack Configuration"**), which must be defined to equal the actual name. The function will be called to look up the appropriate endpoint configuration during device enumeration. It must provide a pointer to the endpoint configuration table and the number of entries in the table.

Syntax

```
const EP_CONFIG * USB_DEV_GET_EP_CONFIG_TABLE_FUNC (int *length)
```

Where USB_DEV_GET_EP_CONFIG_TABLE_FUNC is an application-defined function name.

Parameters

length – A pointer to the integer variable to receive the number of entries in the endpoint configuration table.

Return Values

This routine must return a pointer to the first element in the endpoint configuration table.

Preconditions

USBInitialize must have been called and returned a success indication.

Side Effects

None

Example

```
// This is a sample implementation. This routine is not called by the appplication.
const EP_CONFIG *USBDEVGetEpConfigurationTable (int *num_entries)
{
    // Provide the number of entries
    *num_entries = sizeof(gEpConfigTable)/sizeof(EP_CONFIG);

    // Provide the table pointer.
    return gEpConfigTable;

}
```

USB API – USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC

This routine is a "call out" from the USB firmware stack that must be implemented by the application. The actual name is defined by the application. The USB firmware will call it using the USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC macro (see **"USB Firmware Stack Configuration"**), which must be defined to equal the actual name. The function will be called to access the function-driver table when the device is configured by the host.

Syntax

```
const FUNC_DRV * USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC (void)
```

Where `USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC` is an application-defined function name

Parameters

None

Return Values

This routine must return a pointer to the first element in the function-driver table.

Preconditions

`USBInitialize` must have been called and returned a success indication.

Side Effects

None

Example

```
// This is a sample implementation. The function is not called by the application.
const FUNC_DRV *USBDEVGetFunctionDriverTable (void)
{
    // Index into the array and provide the interface pointer.
    return gDevFuncTable;

}
```

APPENDIX C: USB FUNCTION DRIVER INTERFACE

This section describes class-or-vendor-specific Function Driver Interface (FDI) to the USB firmware stack. The FDI provides a way for one or more class-specific USB “functions” to communicate with the host. This includes initializing the function when appropriate, providing data transfer capabilities and providing a way to receive events.

Since the function is not initialized until the host selects the device configuration and since USB events are by nature asynchronous, some of the FDI functions are “callouts” from the USB Firmware stack. Others are calls into the USB Firmware stack. Table 6 summarizes the FDI and identifies which calls are into and which are out of the USB stack.

TABLE 6: USB DEVICE LAYER INTERFACE SUMMARY

Operation	Call Type	Description
`USBDEVTransferData`	`IN`	Starts a data transfer (transmit or receive).
`USBDEVGetLastError`	`IN`	Provides information about bus errors.
`<Func-Driver Initialization Routine>` (This routine is called through a pointer in the function-driver table.)	`OUT`	Implemented by function driver(s). Called by the device layer when the device is configured with the function..
`<Func-Event Handling Routine>` (This routine is called through a pointer in the function-driver table.)	`OUT`	Implemented by function driver(s). Handles all function-specific bus events and class or vendor requests.

Detailed descriptions of the USB FDI routines follow.

USB FDI - USBDEVTransferData

This routine initiates a data transfer on the USB from the given endpoint in a given direction. The caller provides the buffer from which to transfer the data for transmission or in which to place the data when receiving.

Syntax

```
BOOL USBDEVTransferData ( TRANSFER_FLAGS flags, void *buffer, unsigned int size )
```

Parameters

`flags` – Used to indicate both endpoint and direction.

`buffer` – Pointer to the buffer from/to which the data will be transferred.

`size` – Number of bytes of data to transfer.

Return Values

`TRUE` if the data transfer was successfully started,

`FALSE` if not.

Preconditions

USBInitialize must have been called successfully and the system must have been connected to a host on the USB.

Side Effects

A USB data transfer has been prepared. The actual transfer of the data will occur under control of the USB controller via DMA later when requested by the host. An `EVENT_TRANSFER` event will be sent to the function driver's `<Func-Event Handling Routine>` routine when the transfer has completed.

Example

```
switch(bRequest)
{
case SET_LINE_CODING:
    // Start an Rx transaction on EP0 to get the line coding.
    gCdcSer.flags |= CDC_FLAGS_LINE_CTRL_BUSY;
    return USBDEVTransferData(XFLAGS(USB_EP0|USB_RECEIVE), &line_coding, sizeof(line_coding));


// Handle other requests
}
```


USB FDI – USBDEVGetLastError

This routine provides a bit mapped representation of the most recent error conditions.

Syntax

unsigned long `USBDEVGetLastError` (void)

Parameters

None

Return Values

A bit mapped* representation of the most recent error conditions:

USBDEV_PID_ERR	Indicates an error in the packet ID field of a packet.
USBDEV_CRC16	Indicates that there was a CRC error in a data packet.
USBDEV_DFN8	Indicates that the data field size of a data packet was not an integer multiple of 8 bits.
USBDEV_BTO_ERR	Indicates a bus turn-around time-out error has occurred.
USBDEV_DMA_ERR	Indicates that the DMA engine was unable to read/write memory.
USBDEV_BTS_ERR	Indicates a bit-stuffing error.
USBDEV_XFER_ID	Indicates that the HAL was unable to identify the given transfer EP.
USBDEV_NO_EP	Indicates that an invalid endpoint number was given.
USBDEV_DMA_ERR2	Indicates that there was an error trying to start a DMA transaction during transfer processing.

* Values may be ORd together if more then one error has occurred since last call to `USBDEVGetLastError`.

Preconditions

`USBInitialize` must have been called successfully.

Side Effects

The internal record of the error has been cleared. However, nothing has been done to fix the error condition. The Caller may need to take appropriate steps.

Example

```
switch(event)
{
case EVT_BUS_ERR:
    error = USBDEVGetLastError();
    HandleBusError(error);// Routine to check for each error and handle it appropriately.
    break;
// Handle other events...
}
```

USB FDI – <Func-Driver Init Routine>

This routine is a “call out” from the USB firmware and must be implemented by the USB peripheral device function driver. The actual name is defined by the driver. The USB firmware will call it using a pointer, in the function driver table. The routine must perform basic initialization of the function driver that implements it. The USB Firmware will call it when the host sets the configuration, before sending any events to the function driver.

Syntax

`BOOL <Func-Driver Init Routine> (unsigned long flags)`

Where `<Func-Driver Table Routine>` is a driver-defined function name

Parameters

`flags` – Function-driver initialization flags (driver specific).

Return Values

`TRUE` if successful,

`FALSE` if not

Preconditions

`USBInitialize` must have been called and returned a success indication.

Side Effects

Side effects are dependent on the function driver that implements the routine. However, in all cases the driver must be initialized and ready to receive class and/or vendor specific events.

Example

```
// This is a sample implementation. The function is not called by the application.
BOOL USBUARTInit (unsigned long flags)
{
    memset(&gCdcSer, 0, sizeof(gCdcSer));
    // Initialize an state necessary
    gCdcSer.flags  = flags & CDC_FLAGS_INIT_MASK;
    // Perform other initialization as necessary
    return TRUE;
}

// Sample Function Driver Table Entry:

// USB CDC Serial Emulation Function Driver
{
    USBUARTInit,// Init routine
    USBUARTEventHandler,// Event routine
    0                           // Init flags
}, // Additional entries may follow.
```

USB FDI – <Func-Driver Event Handling Routine>

This routine is a "call out" from the USB firmware and must be implemented by the USB peripheral device function driver. The actual name is defined by the driver. The USB firmware will call using a pointer, placed in the function driver table.

Syntax

`BOOL <Func-Driver Event Handling Routine>` (`USB_EVENT` event, void *data, int size)

Where `< Func-Driver Event Handling Routine >` is a driver-defined function name

Parameters

`event` – Enumerated data type identifying the event that has occurred (see **"Predefined Events:"**)

`data` – A pointer to event-dependent data (if available, see **"Predefined Events:"**)

`size` – The size of the event-dependent data, in bytes.

Return Values

`TRUE` if successful,

`FALSE` if not (or if not finished handling the event)

Preconditions

`USBInitialize` must have been called and returned a success indication.

Side Effects

Side effects are dependent on the function driver that implements the routine. However, in all cases the driver must handle the event appropriately and prepare for the next event expected to occur (such as starting a new data transfer if appropriate).

Example

Refer to **"Event Handling"** for an example of how to implement this function.

Predefined Events:

`EVENT_NONE` False event trigger (should never happen).

`EVENT_TRANSFER` A previous USB transfer has completed. This event provides a pointer to the following data:

```
typedef struct _transfer_event_data
{
    UINT32          size;           // Actual number of bytes transferred
    USB_XFER_DIR    direction;      // Direction of endpoint
    BYTE            ep_num;         // Endpoint Number
} USB_TRANSFER_EVENT_DATA;
```

`EVENT_SOF` A start of frame has occurred.

`EVENT_RESUME` A resume signal has been received on the bus.

`EVENT_SUSPEND` A suspend signal (3 ms idle) has occurred on the bus.

`EVENT_RESET` A Reset signal has been received on the bus.

`EVENT_DETACH` The USB cable has been detached.

`EVENT_ATTACH` A USB cable has been attached.

`EVENT_STALL` A stall has occurred on one or more endpoints. This event provides a pointer to a 16-bit word providing a bitmap of which endpoints have stalled (bit 0 = EP0 stalled, bit 1 = EP1 stalled, etc.).

`EVENT_SETUP` A device or function-specific setup packet has been received. This event provides a pointer to the setup packet data:

```
typedef struct SetupPkt
{
    union                           // offset   description
    {                               // ------   -------------------------
        BYTE bmRequestType;         //   0      Bit-map of request type
```

```
            struct
            {
                BYTE    recipient:  5;  //          Recipient of the request
                BYTE    type:       2;  //          Type of request
                BYTE    direction:  1;  //          Direction of data X-fer
            };
        }requestInfo;
        BYTE    bRequest;               //   1      Request type
        UINT16  wValue;                 //   2      Depends on bRequest
        UINT16  wIndex;                 //   4      Depends on bRequest
        UINT16  wLength;                //   6      Depends on bRequest
    } SETUP_PKT;
```

`EVENT_USER_BASE` This is the first event available for application definition. Add integer values to this base to define application-specific events.

`EVENT_BUS_ERROR` An error has occurred on the bus. Call `USBHALGetLastError()` to identify it and clear the error-record.

APPENDIX D: SOURCE CODE FOR THE USB DEVICE STACK PROGRAMMER'S GUIDE

The source code for the Microchip USB device stack firmware is offered under a no-cost license agreement. It is available for download as a single archive file from the Microchip corporate web site, at:

www.microchip.com.

After downloading the archive, check the release notes for the current revision level and a history of changes to the software.

REVISION HISTORY

Rev. A Document (02/2008)

This is the initial released version of this document.

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08