USB Embedded Host Stack Programmer's Guide

Author: Bud Caldwell
Microchip Technology Inc.

INTRODUCTION

The Universal Serial Bus (USB) provides a common interface that greatly simplifies how an end user connects many types of peripheral devices to a personal computer (PC). Beyond just the PC, many embedded systems can take advantage of the USB as a way to connect to a wide variety of peripherals.

Unlike a PC, an embedded host is only required to support a predefined set of peripherals. Microchip provides sample firmware that enables hosts, using supported Microchip microcontrollers, to control some of the most commonly requested types of USB peripheral devices (see **"References"**).

For cases in which host firmware is not available to control the type of device required, the Microchip USB embedded host firmware stack provides an easy-to-use framework that simplifies the development of USB 2.0 compliant embedded hosts.

This application note describes how to implement a "client" driver for a USB peripheral using the Microchip host framework. Use of this framework simplifies implementation of firmware for an embedded host and makes it much easier to control almost any type of peripheral device desired.

ASSUMPTIONS

- Working knowledge of C programming language
- Familiarity with the USB 2.0 protocol
- Familiarity with the USB class or device to be hosted.
- Familiarity with Microchip MPLAB® IDE

FEATURES

- Supports USB embedded host applications
- Handles device enumeration and configuration
- Supports multiple class or "client" drivers
- Support for hosting multi-function devices
- Support for root-port power control
- Provides a simple Application Program Interface (API)
- Provides a simple Client Driver Interface (CDI)
- Uses a table-driven method to implement the host's Targeted Peripheral List (TPL)
- Support for control, interrupt, bulk, and isochronous transfers.

LIMITATIONS

- Does not support hubs
- Supports a single USB root port
- Number of client drivers supported limited only by available memory

SYSTEM HARDWARE

The USB firmware stack was developed for the following hardware:

USB variants of the PIC24 and PIC32 families of microcontrollers.

PIC® MCU MEMORY RESOURCE REQUIREMENTS

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

PIC® MCU HARDWARE RESOURCE REQUIREMENTS

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

TABLE 1: HARDWARE RESOURCE REQUIREMENTS

PIC® MCU 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
VBUSON (Output)	Enables or disables VBus power supply

INSTALLING SOURCE FILES

The USB host firmware stack source is available as part of Microchip's complete USB Embedded Host Support Package.

Perform the following steps to complete the installation:

1. Download the installation file from the Microchip corporate web site: www.microchip.com/usb.
2. Execute the installation file. A Windows® installation wizard will guide you through the installation process.
3. Before continuing with the installation, you must accept the software license agreement by clicking **I Accept**.
4. After completion of the installation process, you should see a new entry in the Microchip program group. The complete source code will be copied into the selected directory.
5. Refer to the release notes for a complete file manifest, the latest version-specific features, and limitations.

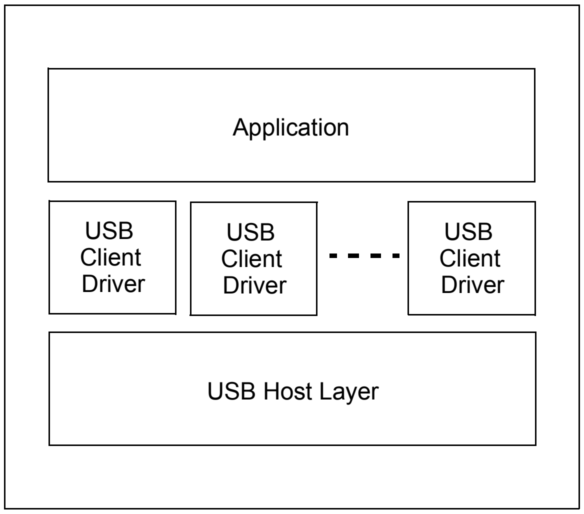
APPLICATIONS

This application note is a programmer's guide. It describes how to use the USB embedded host stack firmware when a sample application is not available to perform the desired task. However, several Microchip sample applications are noted in **"References"**. These applications are available for download from www.microchip.com.

USB EMBEDDED HOST FIRMWARE ARCHITECTURE

The USB embedded host firmware stack can be thought of as consisting of 3 layers, as shown in Figure 1.

FIGURE 1: USB EMBEDDED HOST FIRMWARE STACK



Application Layer

The application layer is the firmware necessary to implement the device's desired behavior. It is customer designed and implemented code, although it may be based on Microchip supplied sample code. The application layer communicates with a USB device through one or more USB client drivers, and uses any other firmware in the system, as necessary.

USB Client Driver

Each USB peripheral device implements a particular function (printer, mouse, mass storage device, etc.). Some devices may have multiple functions. A USB client driver enables the embedded host's application firmware to control a single function of a USB peripheral device that is connected to the host. Multi-function devices will usually be controlled by multiple client drivers. The client driver should model the function in an abstract way, so that the host application does not need to comprehend the details of how the device works.

USB Host Layer

The host layer provides an abstraction of the USB, supplying the following services:

- Performs device identification
- Performs device enumeration
- Manages client drivers
- Provides a simple interface to communicate with a USB peripheral device

When first connected to the bus, the host layer will read the descriptors (data structures defined by the USB 2.0 and its associated supplements) from the device to determine what type of device it is and what function(s) it supports. Then, it will check the TPL to see if the device can be supported. If it can be, the host layer will initialize the appropriate client driver (or drivers).

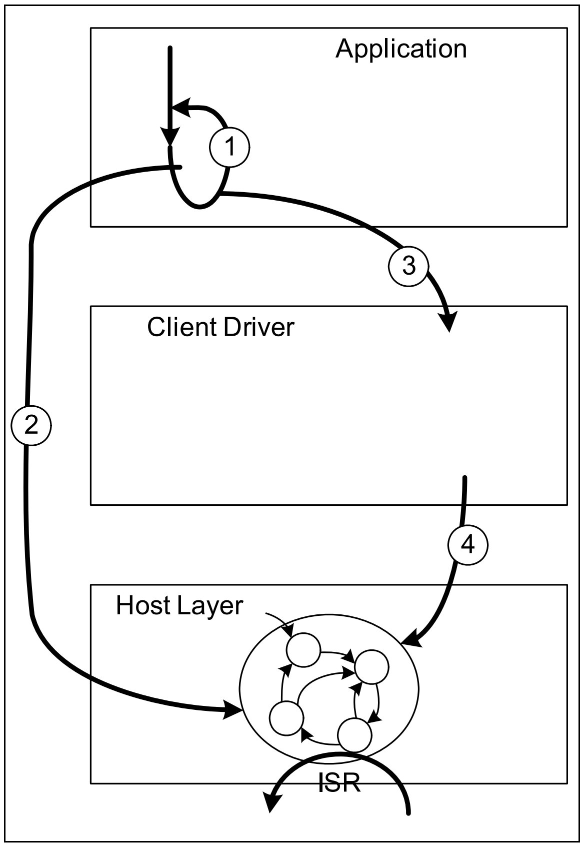
Client Driver Architecture

This section provides an overview of the client driver architecture.

CLIENT DRIVER API

A client driver provides a set of functions, data structures, and definitions that allow the application to control the device. This interface is the API (Application Program Interface). The exact design of the client driver's API is specific to the peripheral (or class of peripherals) to be controlled, and is determined by the driver's designer.

FIGURE 2: CALLING CLIENT DRIVER API ROUTINES



As shown in Figure 2, the application usually contains a main loop (arrow #1) that controls the overall state of the firmware stack .

From within this loop, it must call a USB tasks routine to maintain the state of the host layer (arrow #2). There is also an Interrupt Service Routine (ISR) contained within the host layer that services interrupts as they occur on the bus.

The ISR communicates with the host layer's state machine. To communicate with the USB device, the application would call one or more of the client's API routines (arrow #3).

In response, the client driver will most likely call into the host layer to start the tasks necessary to implement the request (arrow #4).

After the client driver API routine returns, the application must continue to call the USB tasks routine (arrow #2) to allow the task to complete.

CLIENT DRIVER'S INTERFACE TO THE HOST LAYER

In addition to calling host-layer interface routines (see **“Host Layer API and Client Driver Interface”**); the client driver must provide two “callback” functions to interface with the host layer.

The first function is required to initialize the client driver. The host layer will call it when a device of the appropriate type has been connected and configured.

The host layer will call the other routine when events occur on the USB about which the client driver may need to know. A code identifying the event, along with any additional data required, will be passed into the “event handling” routine.

These two “callback” functions, along with the other functions and definitions provided by the host layer, make up the CDI by which client drivers access the USB and communicate with their associated devices.

CLIENT DRIVER STATE MACHINE

A client driver will normally include some form of state machine to manage the device.

This state machine can be maintained in either of the following ways:

- Event-driven
- Polled

To support a fully event-driven implementation, the application must enable transfer events and define an event handling routine (refer to the `USB_HOST_APP_EVENT_HANDLER` and `USB_ENABLE_TRANSFER_EVENT` configuration options). Sections **“Event-Driven Client Drivers”** and **“Polling-Based Client Drivers”** describe these two methods in more detail. Sections **“The Client Driver’s Event-Handling Routine”** through **“Implementing a Polled Client Driver”** describe how to implement each method. The main differences between the two methods are the direction and order in which the calls are performed and in how the tasks are split up.

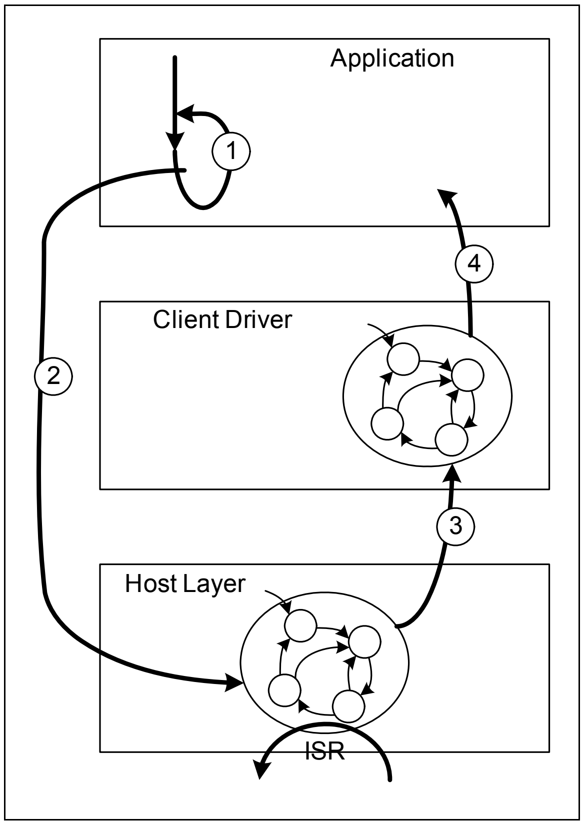
As mentioned above, the application will normally contain a main loop that controls the over-all state of the firmware stack from which it will call a USB tasks routine that maintains the state of the host layer. This is the same in both the polled and event-driven cases. Also in both cases, the ISR, contained within the host layer, services interrupts as they occur on the bus and communicates with the host layer’s state machine.

To start some activity, the application will normally call one or more of the client driver’s API routines as described in **“Client Driver API”**. To complete this activity, the state machine must be maintained using either the polled or event-driven methods.

EVENT-DRIVEN CLIENT DRIVERS

When using the event-driven method, the state machine of the client driver is managed by the client driver's event-handling routine so actions that require some time to complete can continue while the application is busy doing other things.

FIGURE 3: EVENT DRIVEN CLIENT DRIVER



In Figure 3, the application's main loop (arrow #1) must regularly call the host layer's USB tasks routine (arrow #2).

When some activity has completed on the USB, the host layer will call the client driver's event-handling call-back routine to notify it of the event (arrow #3). It will also provide any data necessary to correctly interpret the event.

If necessary (and supported), the client driver can then call the applications optional event-handling routine to notify the application (arrow #4).

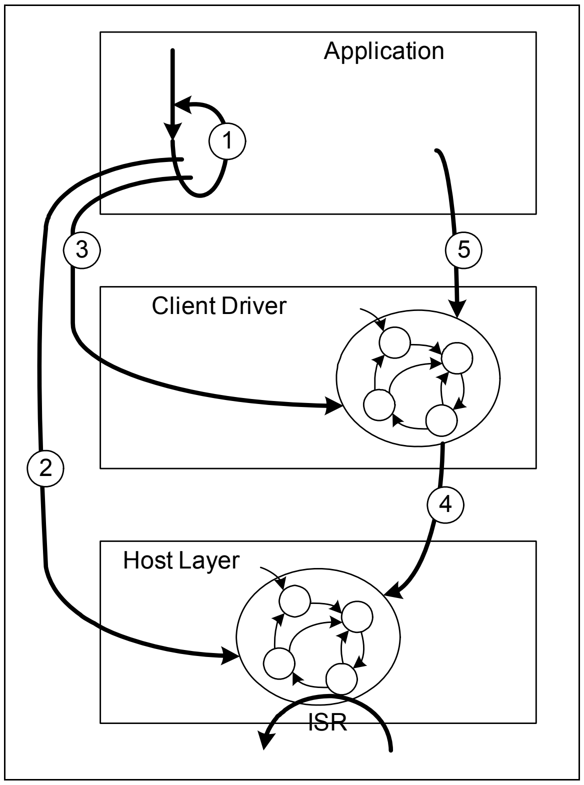
Client-specific events (passed to the application or to another driver layer for multi-layered clients) may or may not correspond one-to-one with USB events that are passed to the client driver by the host layer. In some cases the host layer may pass many events to the client driver before the client driver passes a single event to the application, if it does so at all. In other cases a call to a client driver's API routine may immediately result in a call back to the application's event-handling routine. The exact usage is up to the client driver's designer and the needs of the USB peripheral to be controlled.

The key feature of an event-driven client driver is that transitions from one state to another occur in response to an event on the USB and are managed by the client driver's event-handling routine. This results in tasks being split up between events, so that each event will start the next portion of some activity that will result in another event or the completion of the activity. It also results in calls occurring back up the stack, from the lower layers toward the application in response to a call to the USB tasks routine.

POLLING-BASED CLIENT DRIVERS

When using the polled method, the client driver's state machine is maintained by the driver's own tasks routine that should be considered part of the client driver's API.

FIGURE 4: POLLING-BASED CLIENT DRIVER



As shown in Figure 4, the application's main loop (arrow #1) must regularly call both the host layer's tasks routine (arrow #2) and the client driver's tasks routine (arrow #3).

The client's tasks routine will manage transitions in the driver's state machine by calling host layer (CDI) routines to check the status of the bus (arrow #4). As actions complete, the client driver's tasks routine will update state data to reflect events on the USB.

The application must then call one of the driver's API routines (arrow #5) to check on the status of whatever activity on which it is waiting to find out when actions have been completed.

The key feature of this method is that calls are directed down the stack. Actions can be started by API routines, when called by the application or actions can be started later by the client's state machine. The state machine must then have states that wait for some activity to be started or that start the activity themselves. Either way, it must also have states that check for the activity to be completed, usually by calling host layer CDI routines.

CLIENT DRIVER ARCHITECTURE SUMMARY

As described in the preceding sections, a client driver consists of the following:

- Device-specific (or device class-specific) API
- Logic necessary to implement the API and manage the driver's state machine
- Two call-back functions that are used by the host layer to initialize the driver and provide notification of events that occur on the bus

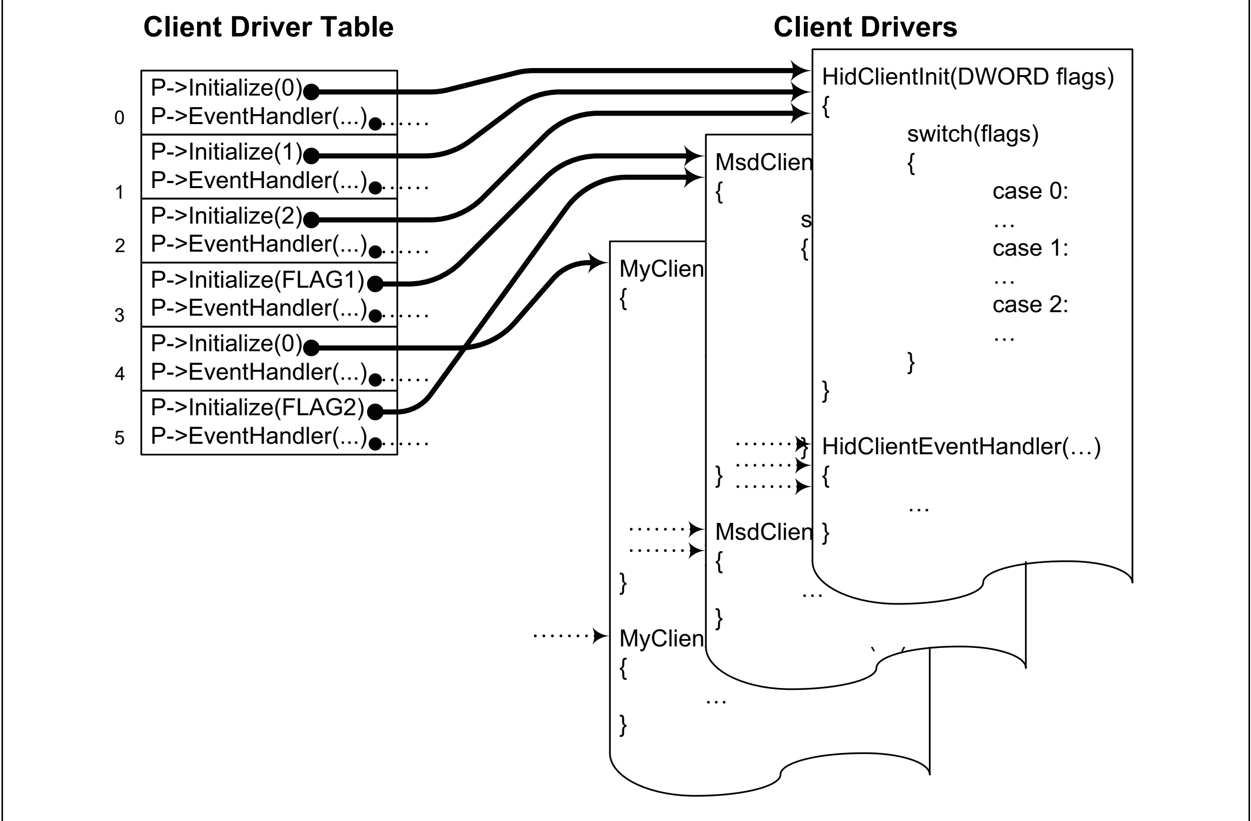
Client Driver Table

Since an embedded USB host may need to support several different types of devices, it may need several different client drivers. This may be the case even if the embedded host has only a single USB host port. In fact, some peripheral devices can have multiple functions, so more then one client driver may be active at one time. In order for the host layer to be able to manage multiple client drivers, it must be able to call multiple routines using the same function signature (one for each driver). To support this, a table driven method is used. Since the set of client drivers supported will almost certainly be different for each embedded host, the application must implement this table. Each entry in the table corresponds to a single client driver and contains pointers to the driver's initialization and event-handling call-back routines (see **"Client Driver Architecture"**). For additional flexibility, each table entry also contains an initialization value that can be used to modify the driver's behavior.

Figure 5 illustrates the relationship between the client driver table and the client drivers.

Note:	The dotted arrows showing the `EventHandler` pointers have been partially removed to avoid cluttering the diagram.

FIGURE 5: CLIENT DRIVER TABLE



When a device is attached, the host layer reads its descriptors and determines whether the device can be supported. If it can be supported, the device will be configured and made ready for the driver to use. Then the host layer indexes into the appropriate entry in the client driver table and calls the client driver's initialization routine using the "Initialize" pointer, and passing to it the initialization value given in that entry of the table.

The driver can then perform any initialization activities that are necessary. Later, when events occur on the USB, the host layer calls the event-handling routine using the `EventHandler` pointer in the same entry in the client driver table, passing it data that identifies the event (as described in **"Client Driver Architecture"**).

More than one entry in the table can correspond to a single client driver. The initialization value can be used to modify the behavior of the driver, depending on which entry in the client driver table was used. This is useful for writing an adaptive driver with behavior that varies according to the specific device or type of device.

For example, a Human Interface Device (HID) client driver may need to support a keyboard, a mouse, or a joy stick. The host layer may use a different entry in the client driver table, depending on which of those three devices is detected. If a different initialization value is used for each entry (e.g., 0, 1, and 2 in Figure 5), the client driver can behave appropriately for the type of device.

Targeted Peripheral List

A full USB host, such as a PC, must be able to install the client drivers for USB devices for which the host was not originally designed. However, an embedded host is only required to support a fixed set of USB peripheral devices or classes of devices. This set is defined by the embedded host's TPL (Targeted Peripheral List).

USB peripheral devices are identified in the TPL in one of two ways*:

- VID-PID combination
- Class-Subclass-Protocol combination

VID is the vendor ID number (provided by the USB Implementer's Forum to identify the device maker). PID is the product ID number (provided by the maker of the device).

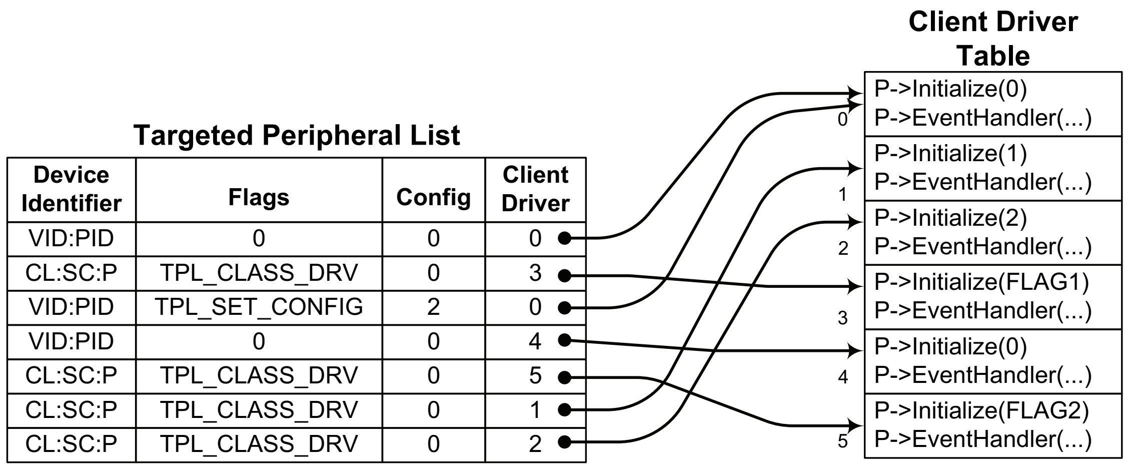
USB peripheral devices are all assigned to a particular class of devices (or identified as vendor-specific). Each device class can have a number of subclasses and each subclass can support one or more protocols that it may use.

Both the VID/PID and Class-Subclass-Protocol (CL-SC-P) numbers are provided to the host in the peripheral device's descriptors (tables of data contained on the device). Refer to the *"Universal Serial Bus Specification, Revision 2.0"* for details about the USB device framework (see **"References"**).

Note:	* Embedded hosts can support client drivers for specific devices, and for classes of devices, as well. However, a true USB "On-The-Go" (OTG) device must specify supported devices individually by VID and PID. Refer to the USB On-The-Go Supplement for details on USB OTG devices (see **"References"**).

The Microchip USB embedded host firmware models the TPL as a table that associates the device identifier (either VID-PID or CL-SC-P combination) with an entry in the client driver table. When a device is attached to the USB, the TPL table is searched to determine if a device is supported, and to identify which client driver will be used to control the device. Figure 6 illustrates the relationship between the TPL and client driver table.

FIGURE 6: TARGETED PERIPHERAL LIST TABLE



Targeted Peripheral List

Device Identifier	Flags	Config	Client Driver
VID:PID	0	0	0
CL:SC:P	TPL_CLASS_DRV	0	3
VID:PID	TPL_SET_CONFIG	2	0
VID:PID	0	0	4
CL:SC:P	TPL_CLASS_DRV	0	5
CL:SC:P	TPL_CLASS_DRV	0	1
CL:SC:P	TPL_CLASS_DRV	0	2

Client Driver Table

0	P->Initialize(0)
	P->EventHandler(...)

1	P->Initialize(1)
	P->EventHandler(...)

2	P->Initialize(2)
	P->EventHandler(...)

3	P->Initialize(FLAG1)
	P->EventHandler(...)

4	P->Initialize(0)
	P->EventHandler(...)

5	P->Initialize(FLAG2)
	P->EventHandler(...)

A bit (`TPL_CLASS_DRV`) in the "Flags" field of each TPL table entry indicates if the Device Identifier field contains a Class-Subclass-Protocol combination (if set) or a VID-PID combination (if not set). Associated with each Device Identifier is an index into the client driver table. This index is used to locate the corresponding entry in the client driver table and access the client driver as described in **"Client Driver Table"**.

The TPL also contains other information providing an optional ability to select the initial configuration of the peripheral device if the `TPL_SET_CONFIG` flag is set in the flags field. (Otherwise, the "Config" number is ignored and the initial configuration is chosen starting at the lowest configuration number (1) and stopping at the first configuration that can be supported.)

Referring to Figure 6, notice that more than one entry in the TPL table can reference a single entry in the client driver table (for example, the first and third entries). This allows multiple, specific devices of the same class to use a single client driver for that class by specifying each device's VID-PID combination. Alternately, an entire class of devices can be supported by specifying a CL-SC-P combination (for example, the second entry). Also, if more then one entry in the client driver table points to a single client driver (as shown in Figure 5 in **"Client Driver Table"**), a single class driver can be used to support several specific devices by VID-PID combination or various classes (or subclasses) of devices by CL-SC-P combination. Any required changes in driver behavior based on variations between devices and subclasses or protocol differences can be indicated using the client driver's initialization value, given in the client driver table entry. Together, the TPL and client driver tables provide a highly flexible mechanism through which an embedded host can support practically any combination of peripheral devices and client drivers desired.

Note:	The TPL is searched starting at the top so that the first matching entry found will be given priority if more then one entry might match a single device. This can be useful for supporting multiple configurations of a single device or device-specific behavior with a fall-back to general class behavior. A client driver's initialization routine has an opportunity to fail, causing the search to continue.

IMPLEMENTING AN EMBEDDED HOST'S FIRMWARE

This section describes the steps necessary to design and implement the firmware for an embedded USB host using the Microchip framework.

Overview:

1. Implement the main application
2. Implement the USB client driver(s)
3. Implement the TPL and Client Drivers Tables
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 application-specific non-USB functionality 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 its main loop.

EXAMPLE 1: MAIN APPLICATION ROUTINE

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

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

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

As described in **"USB Embedded Host Firmware Architecture"**, the interface between the application and the client driver is completely up to the designer of the client driver. However, if an event-driven implementation is chosen, the application should implement an event-handling routine to receive events from the USB stack and any client drivers (see the `USB_EVENT_HANDLER` data type). If this is done, client drivers can be designed to call the application and pass events to it similar to the way the driver receives events from the USB stack. If not, the client driver must contain API routines to provide any status information required. The application can also receive events from the host layer, such as the VBus events shown in Example 2.

EXAMPLE 2: APPLICATION EVENT-HANDLER

```
BOOL MyApplicationEventHandler ( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    // Handle specific events.
    switch (event)
    {
    case EVENT_MY_CLIENT_ATTACH:
        // Do anything necessary when your device attaches.
        break;
    case EVENT_MY_CLIENT_DETACH:
        // Do anything necessary when your device detaches.
        break;
    case EVENT_VBUS_SES_END:
        // Turn off VBus.
        break;
    case EVENT_VBUS_SES_VALID:
        // Turn on VBus.
        break;
    default:
        return FALSE
    }
    return TRUE;
}
```

To identify this function to the host layer (and to Microchip-supplied client drivers) define the `USB_HOST_APP_EVENT_HANDLER` macro in the USB configuration header (see **"Configuring the USB Stack Options"**) to equate to the function's name.

EXAMPLE 3: IDENTIFYING THE APPLICATIONS EVENT HANDLER

```
#define USB_HOST_APP_EVENT_HANDLER  MyApplicationEventHandler
```

Refer to the `USB_EVENT` data type full details on which events are pre-defined and what data (if any) is associated with each event.

The application must also implement the TPL and Client Drivers Tables and make calls to the API routines of client drivers as necessary to control any supported USB peripheral devices. However, this is most easily done after all client drivers have been implemented.

Implementing the USB Client Driver(s)

As described in **"USB Embedded Host Firmware Architecture"**, the purpose of a client driver is to provide a simple, abstract model of the function of a USB peripheral device, and implement an API by which the application may control it.

The design of this API is completely dependent on the device (or class) to be controlled. However, to implement the API, the client driver must interface with the USB Host layer through its CDI. (Refer to **"Host Layer API and Client Driver Interface"**.)

The following examples show how simple read and write API routines might be implemented.

EXAMPLE 4: CLIENT DRIVER API ROUTINES

```
BYTE MyClientRead( BYTE DevAddr, void *Buffer, UINT32 Len )
{
    BYTE RetVal;

    // Make sure we're in an initialized state
    if (myClientData.Initialized != TRUE)
        return USB_INVALID_STATE;

    // Make sure the right device address was given.
    if (DevAddr != myClientData.DevAddr)
        return USB_INVALID_STATE;

    // Make sure we're not busy already
    if (myClientData.RxBusy)
        return USB_BUSY;

    // Set the busy flag, clear the count and start a new IN transfer.
    myClientData.RxBusy = TRUE;
    myClientData.RxLen  = 0;
    RetVal = USBHostRead( DevAddr, USB_IN_EP|MY_EP_NUM, (BYTE *)Buffer, Len );
    if (RetVal != USB_SUCCESS) {
        myClientData.RxBusy = FALSE;    // Clear flag to allow re-try
    }

    return RetVal;

}

BYTE MyClientWrite( BYTE DevAddr, void *Buffer, UINT32 Len )
{
    BYTE RetVal;

    // Make sure we're in an initialized state
    if (myClientData.Initialized != TRUE)
        return USB_INVALID_STATE;

    // Make sure the right device address was given.
    if (DevAddr != myClientData.DevAddr)
        return USB_INVALID_STATE;

    // Make sure we're not busy already
    if (myClientData.RxBusy)
        return USB_BUSY;

    // Set the busy flag and start a new OUT transfer.
    myClientData.TxBusy = TRUE;
    RetVal = USBHostWrite( DevAddr, USB_OUT_EP|MY_EP_NUM, (BYTE *)Buffer, Len );
    if (RetVal != USB_SUCCESS) {
        myClientData.TxBusy = FALSE;    // Clear flag to allow re-try
    }

    return RetVal;

}
```

THE CLIENT DRIVER'S INITIALIZATION ROUTINE

The purpose of the client driver's initialization routine is to initialize (or re-initialize) the client driver when a supported device is attached and configured. It must implement the function signature defined by the `USB_CLIENT_INIT` data type.

EXAMPLE 5: CDI INITIALIZATION ROUTINE

```
BOOL MyClientInit ( BYTE address, DWORD flags )
{
    BYTE *pDesc;

    // Initialize state data
    memset(&myClientData, 0, sizeof(myClientData));

    // Save device the address, VID, & PID
    myClientData.DevAddr = address;
    pDesc  = USBHostGetDeviceDescriptor(address);
    pDesc += 8;
    myClientData.vid  =  (UINT16)*pDesc;       pDesc++;
    myClientData.vid |= ((UINT16)*pDesc) << 8; pDesc++;
    myClientData.pid  =  (UINT16)*pDesc;       pDesc++;
    myClientData.pid |= ((UINT16)*pDesc) << 8; pDesc++;

    // Set Client Driver Init Complete.
    myClientData.Initialized = TRUE;

    // Notify the application that my device has been attached.
    USB_HOST_APP_EVENT_HANDLER(address, EVENT_MY_CLIENT_ATTACH, NULL, 0);

    // Do anything else necessary.

    return TRUE;

}
```

For some types of peripheral devices, the client driver's initialization routine will need to do something to start the interaction with the peripheral device, such as starting a control transfer or reading the first block of data. For other peripherals, it will be sufficient to notify the application that the device has been attached and the application will perform the initial action. In some cases, a later event (possibly unrelated to the USB) will start the interactions with the peripheral. Exact actions taken will depend entirely on the peripheral device being used and the application being implemented.

THE CLIENT DRIVER'S EVENT-HANDLING ROUTINE

As described in **"USB Embedded Host Firmware Architecture"**, client drivers can be designed to be event driven or polled. For an event-driven client driver, the event-handling routine maintains the state of the driver by performing any necessary actions to transition from one state to the next. The following example of a simple event-handling routine checks to make sure that the client driver is in an initialized state then handles either the "detach" (`EVENT_DETACH`) or "transfer done" (`EVENT_TRANSFER`) events.

In both cases, the routine updates state variables and notifies the application.

This example also demonstrates the use of event-specific data.

EXAMPLE 6: EVENT-HANDLING ROUTINE

```
BOOL MyClientEventHandler ( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    // Make sure we're in an initialized state
    if (myClientData.Initialized != TRUE)
        return FALSE;

    // Make sure it was for our device
    if ( address != myClientData.DevAddr)
        return FALSE;

    // Handle specific events.
    switch (event)
    {
    case EVENT_DETACH:
        // Notify the application that the device has been detached.
        USB_HOST_APP_EVENT_HANDLER(myClientData.DevAddr, EVENT_MY_CLIENT_DETACH,
                                   &myClientData.DevAddr, sizeof(BYTE) );
        myClientData.Initialized = FALSE;
        return TRUE;

    case EVENT_TRANSFER:
        if ( (data != NULL) && (size == sizeof(HOST_TRANSFER_DATA)) )
        {
            HOST_TRANSFER_DATA *pTrans = (HOST_TRANSFER_DATA *)data;

            if (pTrans->bEndpointAddress  & USB_IN_EP )
            {
                myClientData.RxBusy   = FALSE;
                myClientData.RxLength = pTrans->dataCount;
                USB_HOST_APP_EVENT_HANDLER(myClientData.DevAddr, EVENT_MY_CLIENT_RX_DONE,
                                           &pTrans->dataCount, sizeof(DWORD) );
            }
            else
            {
                myClientData.TxBusy = FALSE;
                USB_HOST_APP_EVENT_HANDLER(gc_DevData.DevAddr, EVENT_MY_CLIENT_TX_DONE,
                                           &pTrans->dataCount, sizeof(DWORD) );
            }
            return TRUE;
        }
        break;

    default:
        break;
    }

    return FALSE;
}
```

The “detach” event (`EVENT_DETACH`) occurs when the associated device is disconnected from the USB. When this happens, this client’s event handling routine notifies the application by sending a client-specific event (`EVENT_MY_CLIENT_DETACH`) to the application’s event-handling routine. It then clears its “Initialized” flag to indicate that this device is no longer valid. It clears the flag after notifying the application because the application may need to call other API routines to process the detach event and other API routines would normally test the “Initialized” flag to ensure that the driver is in a valid state before allowing an operation to proceed.

Note:	Instead of defining its own detach event (`EVENT_MY_CLIENT_DETACH`), the event-handling routine could also have propagated the `EVENT_DETACH` event to the application. The choice us up to the designer of the client driver’s API.

The “transfer done” event (`EVENT_TRANSFER`) occurs when a previously started transfer on a given endpoint has completed. In the case of this client driver, it is assumed that only one endpoint is used so the actual endpoint number is not checked. However, the routine does check the direction bit (which is part of the endpoint “address”) to determine if it was a transmit (Tx) or receive (Rx) transfer that just completed. It then updates state variables and notifies the application. It is up to the application to start another transfer if and when it is required. In this case, the routine updated state variables before notifying the application, because the “Tx” and “Rx” API routine may very well be called in response to the client-specific events.

The routine returns `TRUE` when an event was handled and `FALSE` in all other cases. This indicates to the host layer that the client driver has successfully handled that event.

Note 1:	Code executed within the context of the event-handling routine must not block.
2:	Refer to the `USB_EVENT` data type for a complete list and description of all predefined events. (See **“Host Layer API and Client Driver Interface”**.)

DEFINING CLIENT-SPECIFIC EVENTS

In the examples shown, both the client driver’s initialization and the event-handling routines called the application’s event-handling routine to send it client-specific events. To define client-specific events, the `USB_EVENT` data type must be extended. This data type is a C language `enum` with several predefined values. All of the predefined values are less then the `EVENT_USER_BASE` value (except `EVENT_BUS_ERR`, which is defined as `UINT_MAX` to make it the highest possible value). This allows new event values to be easily defined using the following technique.

EXAMPLE 7: DEFINING CLIENT-SPECIFIC EVENTS

```
#ifndef EVENT_MY_CLIENT_OFFSET          // The application can add a non-zero offset
    #define EVENT_MY_CLIENT_OFFSET 0    // to my client's events to resolve conflicts
#endif                                  // in event number.

#define EVENT_MY_CLIENT_ATTACH  (EVENT_USER_BASE + EVENT_MY_CLIENT_OFFSET + 0)
        // Indicates that my device has been attached.
        // data: NULL
        // size: 0

#define EVENT_MY_CLIENT_DETACH  (EVENT_USER_BASE + EVENT_MY_CLIENT_OFFSET + 1)
        // Indicates that the device has been detached from the USB.
        // data: Points to a BYTE that contains the device address.
        // size: sizeof(BYTE)

#define EVENT_MY_CLIENT_TX_DONE (EVENT_USER_BASE + EVENT_MY_CLIENT_OFFSET + 2)
        // Indicates that a previous write request has completed.
        // data: Pointer to a variable containing the actually number bytes written.
        // size: sizeof(DWORD)

#define EVENT_MY_CLIENT_RX_DONE (EVENT_USER_BASE + EVENT_MY_CLIENT_OFFSET + 3)
        // Indicates that a previous read request has completed.
        // data: Pointer to a variable containing the actually number bytes read.
        // size: sizeof(DWORD).
```

There are two key features of this technique.

First, since the client-specific events are all defined by adding offsets to a pre-defined member of the `USB_EVENT` enumeration (`EVENT_USER_BASE`), they will all be given the data type associated with that value. This ensures that all client-specific events are of the correct data type.

Second, since a secondary offset with a default value was included in each event's definition, the entire set of events can be easily shifted to a different range of values within the `USB_EVENT` enumeration. This allows applications to easily resolve conflicts between the event definitions of different client drivers by defining the secondary offset (`EVENT_MY_CLIENT_OFFSET` in this example) as an appropriate value to prevent overlapping events. This is especially useful when a device includes more then one client driver or when a client driver will be shared across multiple products.

IMPLEMENTING A POLLED CLIENT DRIVER

If a polled model is preferred over the event-driven method, it can be easily implemented. Instead of using an event-handling routine like the one shown in **"The Client Driver's Event-Handling Routine"**, a polled client driver would implement a central tasks routine to manage state transitions. This routine would check the state of the host layer and take different actions, based on the current state of the client driver. It would need to be called regularly, along with the `USBTasks` routine, and would become part of the client driver's API.

Example 8 demonstrates simple read and write functionality, similar to the event-driven example shown in **"The Client Driver's Event-Handling Routine"**.

EXAMPLE 8: POLLED CLIENT-TASKS ROUTINE

```
void MyClientTasks ( void )
{
    BYTE deviceStatus;
    BYTE errorCode;

    // Make sure we're in an initialized state.
    if (myClientData.Initialized != TRUE)
        return;

    // Check device status.
    deviceStatus= USBHostDeviceStatus(myClientData.DevAddr);

    // Make sure our device hasn't been disconnected.
    if ( deviceStatus != USB_DEVICE_ATTACHED )
    {
        myClientData.Initialized = FALSE;
        return;
    }

    // Perform state-specific tasks.
    switch (myClientData.State)
    {
    case STATE_IDLE:
        break;

    case STATE_WAITING_TRANSFER_DONE:

        if (USBHostTransferIsComplete(myClientData.DevAddr, MY_RX_ENDPONT,
                                      &errorCode, &myClientData.RxLength)
        {
            if (errorCode)
            {
                 // handle errors
            }
            myClientData.RxBusy = FALSE;
        }

        if (USBHostTransferIsComplete(myClientData.DevAddr, MY_TX_ENDPONT,
                                      &errorCode, &myClientData.TxLength)
        {
            if (errorCode)
            {
                 // handle errors
            }
            myClientData.TxBusy = FALSE;
        }

        if ( !( myClientData.RxBusy || myClientData.$TxBusy) )
                        myClientData.State  = STATE_IDLE;
        break;

    default:// invalid state!
        myClientData.State  = STATE_IDLE;
        break;
    }

}
```

In both the polled and event-driven examples, the read and write API routines check the appropriate "busy" flag (`myClientDriver.TxBusy` or `myClientDriver.RxBusy`). If not busy, the API routine sets the appropriate flag and (in the polled case) transitions to the `STATE_WAITING_TRANSFER_DONE` state.

One important difference between the polled and event-driven techniques is that the event-handling routine is called directly when the transfer has finished, and the polling routine must call `USBHostTransferIsComplete` CDI routine to determine when the transfer has finished.

Implementing the TPL and Client Driver Tables

As described in **"USB Embedded Host Firmware Architecture"**, the TPL and client driver tables determine which devices (or classes of devices) an embedded host will support, and which client driver(s) will be used for each. Since this choice is application-specific, these tables are considered part of the application.

IMPLEMENTING THE CLIENT DRIVER TABLE

The client driver table is implemented as an array of structures of the `CLIENT_DRIVER_TABLE` data type, which is defined as follows:

FIGURE 7: CLIENT DRIVER TABLE STRUCTURE

```
typedef struct _CLIENT_DRIVER_TABLE
{
    USB_CLIENT_INIT          Initialize;
    USB_CLIENT_EVENT_HANDLER EventHandler;
    DWORD                    flags;
} CLIENT_DRIVER_TABLE;
```

The "Initialize" member is a pointer to the client driver's initialization routine and the `EventHandler` member is a pointer to the client driver's event-handling routine. These data types for these callback routine pointers are defined as follows:

EXAMPLE 9: CALLBACK POINTER DATA TYPES

```
typedef BOOL (*USB_CLIENT_INIT)          ( BYTE address, DWORD flags );

typedef BOOL (*USB_CLIENT_EVENT_HANDLER) ( BYTE address, USB_EVENT event,
                                           void *data, DWORD size );
```

A driver must implement routines that match these function signatures such as those shown in the examples in **"The Client Driver's Initialization Routine"** and **"The Client Driver's Event-Handling Routine"**. Prototypes for these routines should be given in the client driver's public API header file so that the application can use them in the Client Drivers Table, as shown by Example 10.

EXAMPLE 10: CLIENT DRIVER TABLE

```
CLIENT_DRIVER_TABLE usbClientDrvTable[] =
{
    {   // HID Client Driver: Mouse
        USBHostHidInit,                // Initialization Routine
        USBHostHidEventHandler,        // Event Handler Routine
        0                              // Initializaton Parameter
    },
    {   // HID Client Driver: Keyboard
        USBHostHidInit,                // Initialization Routine
        USBHostHidEventHandler,        // Event Handler Routine
        1                              // Initializaton Parameter
    },
    {   // HID Client Driver: Joystick
        USBHostHidInit,                // Initialization Routine
        USBHostHidEventHandler,        // Event Handler Routine
        2                              // Initializaton Parameter
    },
    {   // Mass Storage Client Driver: Bulk Only
        USBHostMsdInit,                // Initialization Routine
        USBHostMsdEventHandler,        // Event Handler Routine
        FLAG1                          // Initializaton Parameter
    },
    {   // My Client Driver
        USBHostMyClientInit,           // Initialization Routine
        USBHostMyClientEventHandler,   // Event Handler Routine
        0                              // Initializaton Parameter
    },
    {   // Mass Storage Client Driver: CBI
        USBHostMsdInit,                // Initialization Routine
        USBHostMsdEventHandler,        // Event Handler Routine
        FLAG2                          // Initializaton Parameter
    }
};
```

Example 10 demonstrates how to support six different types of devices using three client drivers. (See Figure 5, in **"Client Driver Table"**.)

1. The HID class driver in this example would be an adaptive driver that supports the following types of devices:
    a) keyboard when its initialization routine is passed 0
    b) mouse when its initialization routine is passed 1
    c) joystick when its initialization routine is passed 2
2. The mass storage class driver would support the following protocols:
    a) Bulk-only protocol when `FLAG1` is passed to its initialization routine
    b) CBI (Command Block Interface) when `FLAG2` is passed.
3. The third "My Client" driver uses the examples in this document.

Note: These examples are just for illustration. Refer to the appropriate USB device class specifications for details on HID, mass storage, and other device classes.

Refer to the application notes listed in the **"References"** section for details on the client drivers provided by Microchip.

By default, the host layer expects the client driver table array to be named `usbClientDriverTable`. If another name is used, it must be identified by defining the `USB_CLIENT_DRIVER_TABLE` macro (see Example 11).

EXAMPLE 11: IDENTIFYING THE CLIENT DRIVER TABLE

```
#define USB_CLIENT_DRIVER_TABLE \
        myClientDriverTable
```

Note: The client driver table is never searched. It is only accessed using the indices given in the TPL table. So the host layer never needs to know its exact length.

IMPLEMENTING THE TPL TABLE

The TPL table is an array of structures of the `USB_TPL` data type. Each structure contains members for the device identifier, the initial configuration, the client driver table index, and a `flags` field.

The `USB_TPL` structure is defined as follows:

FIGURE 8: TPL TABLE STRUCTURE

```
typedef struct _USB_TPL
{
    union
    {
        DWORD      val;
        struct
        {
            WORD   idVendor;
            WORD   idProduct;
        };
        struct
        {
            BYTE   bClass;
            BYTE   bSubClass;
            BYTE   bProtocol;
        };
    } device;

    BYTE           bConfiguration;

    BYTE           ClientDriver;

    union
    {
        BYTE       val;
        struct
        {
            BYTE   bfAllowHNP      : 1;
            BYTE   bfIsClassDriver : 1;
            BYTE   bfSetConfiguration : 1;
        };
    } flags;

} USB_TPL;
```

The “device” member is a DWORD-sized union that can hold either the VID-PID or CL-SC-P device-identifier number combinations. The other fields are each one byte in size, with the `flags` field being a union of the individual flags bits. The `bConfiguration` member allows the device’s initial configuration to be specified in the TPL table when the `bSetConfiguration` flag is set. The `ClientDriver` member is the index into the client driver table for the device identified and is how the TPL table associates a particular device (or a class of devices) with a particular client driver. The `bfAllowHNP` flag, when set, instructs the host layer to enable Host Negotiation Protocol (HNP) and is only valid for true USB OTG devices. The `bfIsClassDriver` flag is set to when a CL-SC-P ID combination is used and the `ClientDriver` index is for a class driver. When this flag is cleared, a VID-PID combination must be used in the `device` field. The driver can be device-specific or a class driver of the correct type for the device identified.

Since the TPL table is searched from the beginning whenever a device is newly attached to the USB, the length of the table must be identified to the host layer. This must be done by defining the `NUM_TPL_ENTRIES` macro.

EXAMPLE 12: DEFINING TABLE LENGTH

```
#define NUM_TPL_ENTRIES   7
```

By default, the host layer expects the TPL table array to be named `usbTPL`. If the array is named anything else, it must be identified to the host layer using the `USB_TPL_TABLE` macro.

EXAMPLE 13: IDENTIFYING THE TPL TABLE

```
#define USB_TPL_TABLE    MyUsbTplTable
```

The TPL table is usually initialized statically. To simplify initialization of the device identifier field, the `INIT_VID_PID` and `INIT_CL_SC_P` macros are available.

FIGURE 9: DEVICE IDENTIFIER FIELD INITIALIZATION MACROS

```
#define INIT_VID_PID(v,p)       {((v)|((p)<<16))}
#define INIT_CL_SC_P(c,s,p)     {((c)|((s)<<8)|((p)<<16))}
```

The following initialization macros (which can be ORd together) are also available to simplify initialization of the `flags` field.

FIGURE 10: TPL FLAGS MACROS

```
#define TPL_ALLOW_HNP    0x01     // Flag to enable Host Negotiation Protocol
#define TPL_CLASS_DRV    0x02     // Flag to identify a class driver
#define TPL_SET_CONFIG   0x04     // Flag for setting the configuration
```

Example 14 depicts the example TPL table that, when combined with the example client driver table (see **"Implementing the Client Driver Table"**), shows how a system could support any of the following:

- An application-specific HID-class device (VID = 0x04D8, PID = 0x00EF)
- A mass storage class device using the bulk-only protocol (CL = 0x08, SC = 0x06, P = 0x50)
- A second configuration (probably with reduced resource requirements) of the same application-specific HID-class device, as listed in the first line (VID = 0x04D8, PID = 0x00EF, Configuration = 2)
- A vendor-specific device (VID = 0x04D8, PID = 0x00EE)
- A mass-storage class device using the CBI (Control/Bulk/Interrupt) protocol (CL = 0x08, SC = 0x06, P = 0x01)
- A HID-class keyboard (CL = 0x03, SC = 0x00, P = 0x01)
- A HID-class mouse (CL = 0x03, SC = 0x00, P = 0x02)

EXAMPLE 14: TPL TABLE

```
USB_TPL MyUsbTplTable[NUM_TPL_ENTRIES] =
{
//    VID & PID or                           Client
//    Class, Subclass & Protocol      Config Driver  Flags
    { INIT_VID_PID( 0x04D8, 0x00EF ),   0,      0,    {0}              },
    { INIT_CL_SC_P(0x08, 0x06, 0x50),   0,      3,    {TPL_CLASS_DRV}  },
    { INIT_VID_PID( 0x04D8, 0x00EF ),   2,      0,    {TPL_SET_CONFIG} },
    { INIT_VID_PID( 0x04D8, 0x00EE ),   0,      4,    {0}              },
    { INIT_CL_SC_P(0x08, 0x06, 0x01),   0,      5,    {TPL_CLASS_DRV}  },
    { INIT_CL_SC_P(0x03, 0x00, 0x01),   0,      1,    {TPL_CLASS_DRV}  },
    { INIT_CL_SC_P(0x03, 0x00, 0x02),   0,      2,    {TPL_CLASS_DRV}  }
};
```

Note:	The VID number shown in the example is for Microchip. The PID numbers shown are just examples and do not necessarily correspond to any particular devices.

Configuring the USB Stack Options

This section highlights several key configuration options that must be used correctly to ensure proper operation of the USB Embedded Host firmware stack. These options, when required, must be defined in the USB configuration header file. This file must be in the project's include-file search path and should be considered part of the application (see **"Host Layer API and Client Driver Interface"**).

The following is a list of USB stack options:

- `USB_SUPPORT_HOST`
- `NUM_TPL_ENTRIES`
- `USB_HOST_APP_EVENT_HANDLER`
- `USB_ENABLE_TRANSFER_EVENT`
- `USB_EVENT_QUEUE_DEPTH`
- `USB_PING_PONG_MODE`
- `USB_SUPPORT_INTERRUPT_TRANSFERS`
- `USB_SUPPORT_ISOCHRONOUS_TRANSFERS`
- `USB_SUPPORT_BULK_TRANSFERS`
- `USB_NUM_BULK_NAKS`
- `USB_NUM_CONTROL_NAKS`
- `USB_NUM_INTERRUPT_NAKS`
- `USB_NUM_COMMAND_TRIES`
- `USB_NUM_ENUMERATION_TRIES`

USB_SUPPORT_HOST

Purpose	This macro identifies that the USB firmware stack is being used for an embedded host application.
Precondition	None
Valid Values	This macro does not need to have a value assigned to it. Defining it is sufficient.
Default:	Not defined
Example	`#define USB_HOST_ONLY`

NUM_TPL_ENTRIES

Purpose	This macro identifies the number of entries in the embedded host's Targeted Peripheral List (TPL) table. It is necessary to define this because the host firmware must search the TPL table whenever a device is attached to the bus to determine if it can be supported.
Precondition	None
Valid Values	This macro must be defined as an integer constant that identifies the number of entries in the TPL table.
Default:	Not defined
Example	`#define NUM_TPL_ENTRIES    7`

USB_HOST_APP_EVENT_HANDLER

Purpose	This macro identifies the name of the application's optional event-handing call-back routine. If the application does not implement the routine, this macro should be left undefined.
Precondition	None
Valid Values	This macro must equate to the name of the application's event call-back routine or be left undefined.
Default:	Not defined
Example	`#define USB_HOST_APP_EVENT_HANDLER  myClientEventHandler`

USB_ENABLE_TRANSFER_EVENT

Purpose	This macro causes the USB embedded host firmware stack to include support for the `EVENT_TRANSFER` event when built. If this macro is not defined, the `EVENT_TRANSFER` event will never be sent to any client drivers or to the application's event handling routine. Applications that do not require this event, and do not include any client drivers that require this event, can leave this macro undefined to reduce the size of the USB firmware.
Precondition	None
Valid Values	This macro does not need to have a value assigned to it. Defining it is sufficient.
Default:	Not defined
Example	`#define USB_ENABLE_TRANSFER_EVENT`

USB_EVENT_QUEUE_DEPTH

Purpose	This macro determines how many `EVENT_TRANSFER` events (or transfer error events) the USB host firmware can queue up before calling a client driver's event-handling routine. Increasing this value will allow the firmware to support greater latency between calls to the `USBTasks` routine at the potential expense of real-time response and a slight increase RAM usage.
Precondition	This macro is ignored if `USB_ENABLE_TRANSFER_EVENT` is not also defined.
Default Value	4
Valid Values	If used, this macro must be defined as a non-zero integer value. If not defined, this macro will be assigned the default value. It is recommended that it never be assigned a value less then the default value.
Default:	Not defined
Example	`#define USB_EVENT_QUEUE_DEPTH     6`

USB_PING_PONG_MODE

Purpose	Some families of Microchip microcontrollers support multiple methods of buffering USB data (refer to the appropriate data sheet). A USB endpoint may have a single buffer for data transferred to-or-from that endpoint; or it may have two buffers, between which the controller alternates in a ping-pong fashion when transferring data. This macro determines which method is used.
Precondition	The chosen microcontroller must support the method selected.
Valid Values	`USB_PING_PONG__NO_PING_PONG` – Disables ping-pong buffering `USB_PING_PONG__ALL_BUT_EP0` – Disables ping-pong buffering for Endpoint `0`, enabling it for all others `USB_PING_PONG__EP0_OUT_ONLY` – Enables ping-pong buffering for Endpoint `0`, disabling it for all others `USB_PING_PONG__FULL_PING_PONG` – Enables ping-pong buffering for all endpoints
Default:	Not defined
Example	`#define USB_PING_PONG_MODE     USB_PING_PONG__FULL_PING_PONG`

> **Note:** Since only Endpoint `0` is used when operating as an embedded host (when `USB_HOST_ONLY` is defined), the `USB_PING_PONG__FULL_PING_PONG` and `USB_PING_PONG__EP0_OUT_ONLY` options are equivalent (both enabling ping pong buffering) and the `USB_PING_PONG__NO_PING_PONG` and `USB_PING_PONG__ALL_BUT_EP0` options are equivalent (both disabling ping pong buffering). In general, it is recommended to enable full ping pong buffering unless it is not supported.
>
> For PIC32 processors, this value should always be set to `USB_PING_PONG__FULL_PING_PONG`.

USB_SUPPORT_INTERRUPT_TRANSFERS

Purpose	When defined, this macro includes support for interrupt transfers when the USB embedded firmware stack is built. If it is not defined, interrupt transfers may not be supported. It can be left undefined to reduce program size, if interrupt transfers are not required by any device, or class of devices, listed in the TPL table.
Precondition	None
Valid Values	This macro does not need to have a value assigned to it. Defining it is sufficient.
Default:	Not defined
Example	`#define USB_SUPPORT_INTERRUPT_TRANSFERS`

`USB_SUPPORT_ISOCHRONOUS_TRANSFERS`

Purpose	When defined, this macro includes support for isochronous transfers when the USB embedded firmware stack is built. If it is not defined, isochronous transfers may not be supported. It can be left undefined to reduce program size, if isochronous transfers are not required by any device, or class of devices, listed in the TPL table.
Precondition	None
Valid Values	This macro does not need to have a value assigned to it. Defining it is sufficient.
Default:	Not defined
Example	`#define USB_SUPPORT_ISOCHRONOUS_TRANSFERS`

`USB_SUPPORT_BULK_TRANSFERS`

Purpose	When defined, this macro includes support for bulk transfers when the USB embedded firmware stack is built. If it is not defined, bulk transfers may not be supported. It can be left undefined to reduce program size, if bulk transfers are not required by any device, or class of devices, listed in the TPL table.
Precondition	None
Valid Values	This macro does not need to have a value assigned to it. Defining it is sufficient.
Default:	Not defined
Example	`#define USB_SUPPORT_BULK_TRANSFERS`

`USB_NUM_BULK_NAKS`

Purpose	This macro determines how many times a device is allowed to NAK a bulk transfer before the USB host firmware will be considered an error.
Precondition	This macro is only valid if `USB_SUPPORT_BULK_TRANSFERS` is also defined.
Valid Values	This macro must be defined as non-zero integer value. Since some bulk devices can NAK thousands of bulk transfers before being ready to supply data, this number can be quite high.
Default:	10000
Example	`#define USB_NUM_BULK_NAKS    10000`

`USB_NUM_CONTROL_NAKS`

Purpose This macro determines how many times the USB host firmware will allow a device to NAK a control transfer before it will be considered an error.

Note:	For compatibility with the widest range of USB devices, it is recommended not to use this option.

Precondition	None
Valid Values	This macro must be defined as non-zero integer value. A value of 3 is recommended.
Default:	3
Example	`#define USB_NUM_CONTROL_NAKS    3`

USB_NUM_INTERRUPT_NAKS

Purpose	This macro determines how many times the USB host firmware will allow a device to NAK an interrupt `OUT` transfer before it will be considered an error. (When a device NAKs an interrupt `IN` transfer, it is just an indication that it does not have any data to send.)
Precondition	None
Valid Values	This macro must be defined as non-zero integer value. A value of 3 is recommended.
Default:	3
Example	`#define USB_NUM_INTERRUPT_NAKS    3`

USB_NUM_COMMAND_TRIES

Purpose	This macro determines how many times the USB host firmware will retry a control transfer to a device before it will be considered an error.
Precondition	None
Valid Values	This macro must be defined as non-zero integer value. A value of 3 is recommended.
Default:	3
Example	`#define USB_NUM_COMMAND_TRIES    3`

USB_NUM_ENUMERATION_TRIES

Purpose	This macro defines how many times the host will try to enumerate the device before giving up and setting the device's state to `DETACHED`.
Precondition	None
Valid Values	Any integer value, although small numbers are recommended.
Default:	3
Example	`#define USB_NUM_ENUMERATION_TRIES    2`

For the latest details on the USB stack's configuration options, refer to the API document included with the source files.

HOST LAYER API AND CLIENT DRIVER INTERFACE

The CDI provides functions to read and write data to or from a peripheral device's endpoints, perform USB control transfers, access device descriptors, control device configuration, perform general device control and receive status information. The CDI also defines the function signatures of the client driver's initialization and event-handling call-back routines.

Refer to the release notes distributed with the embedded host firmware source files for full details on the CDI.

CONCLUSION

Microchip provides sample firmware for the most commonly requested classes of devices (see **"References"**). However, even in cases where no sample implementation is available to control a specific USB device, the Microchip USB embedded host firmware stack provides a powerful and easy-to-use framework for developing embedded host applications using supported Microchip microcontrollers. This document explains how to implement client drivers for this framework to allow an embedded host design to take advantage of the power, flexibility, and availability of USB devices.

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
- Requirements and Recommendations for USB Products with Embedded Hosts and/or Multiple Receptacles http://www.usb.org/developers/docs/
- Microchip MPLAB® IDE In-circuit development environment, available free of charge, by license, from http://www.microchip.com/mplabide
- Microchip Application Note AN1140, “*USB Embedded Host Stack*”
- Microchip Application Note AN1142, “*USB Mass Storage Class on an Embedded Host*”
- Microchip Application Note AN1143, “*Generic Client on an Embedded Host*”
- Microchip Application Note AN1144, “*USB HID Class on an Embedded Host*”
- Microchip Application Note AN1145, “*Using a USB Flash Drive on an Embedded Host*”

APPENDIX A: SOURCE CODE FOR THE USB EMBEDDED HOST STACK

The source code for the Microchip USB embedded host 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.

NOTES:

MICROCHIP

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