MICROCHIP **AN1140**

USB Embedded Host Stack

Author: Kim Otten
Microchip Technology Inc.

INTRODUCTION

USB has become the standard method for devices to communicate with a PC. From general purpose devices, such as Flash drives and mice, to special purpose devices for specific applications, this popular standard has almost totally replaced other serial communication protocols.

Under the USB standard, USB devices may not communicate directly with each other. They may only communicate with a USB host which controls the bus on which one or more devices communicate. The most common USB host is a PC. With the introduction of Microchip's microcontrollers with the USB On-The-Go (OTG) module, it is now possible for embedded applications to utilize the wide range of USB devices as a USB embedded host.

USB OVERVIEW

There are many excellent references on USB, including the USB 2.0 Specification itself, that provide detailed information on USB operation. This section is intended only to provide a brief definition of the terms referenced in this application note or required to understand Stack operation.

USB Hosts and Peripheral Devices

A typical USB system consists of one host and one or more peripheral devices, often referred to as simply "devices". Each device can communicate only with the host; devices may not communicate directly with each other. The host initiates all communication on the bus. A device may send data to the host only when the host requests it, and it must be able to receive the data that the host sends. A device normally uses a Type-B receptacle or has a captive cable.

Most USB peripheral devices are divided into categories, called classes. Each class has special requirements regarding its communication format. The host must be able to recognize a device's class and meet the class's requirements or the host cannot communicate with the device. Two example classes are HID (Human Interface Device), such as found on a mouse, and Mass Storage, such as found on a Flash drive. Client drivers provide application level support for classes. Some USB peripheral devices are vendor-specific and do not fall into one of the predefined classes. Client drivers must be specifically written for these devices.

The number of devices that can attach to a host can be expanded through the use of hubs. USB is a tiered star network. Typically, a hub allows four or seven devices to attach to a single port. A maximum of five hubs can be chained together, creating up to five tiers. A maximum of 127 devices (including the hubs) can be connected on the bus.

A full USB host uses a Type-A receptacle, and must be able to communicate with any device. This support may be provided via special drivers that must be installed on the host prior to attaching the device. Hubs must be supported, and each port must be able to deliver a minimum of 100 mA.

Host vs. Embedded Host

A USB embedded host differs from a USB host in several small but important aspects. A USB embedded host:

- Supports only specific peripheral devices and/or classes of devices.
- Supports only transfer types required by the supported devices.
- Hub support is optional.
- Has relaxed power requirements.

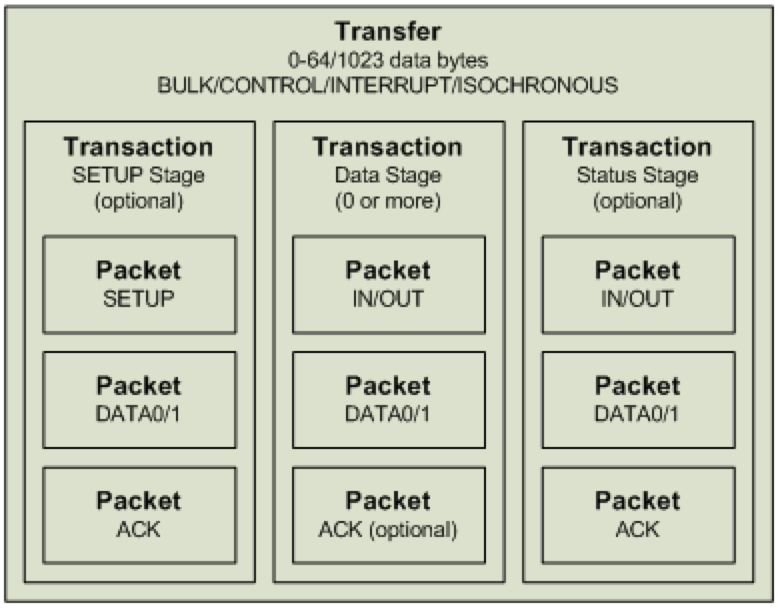
These restrictions allow an embedded host to be implemented on a device with fixed, limited memory.

Host Mode Operation

In a USB system, the host controls all traffic on the bus. A device may only respond to requests from the host; it may not initiate a data transfer. The USB OTG module may be used in either Host or Device mode. The exact method of operation differs between the two modes.

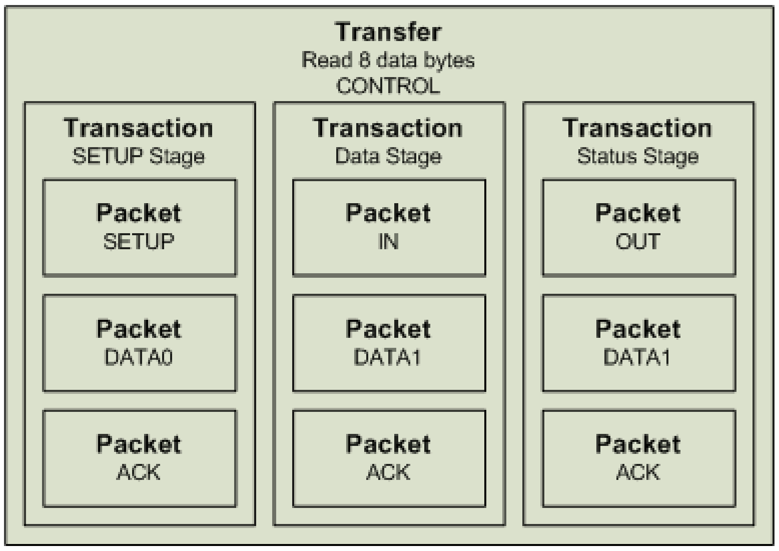
USB transfers can consist of multiple transactions, and transactions can consist of multiple packets. In addition, a single transfer can contain multiple data stage transactions. Figure 1 shows the generic format of a single USB transfer.

FIGURE 1: GENERIC TRANSFER STRUCTURE



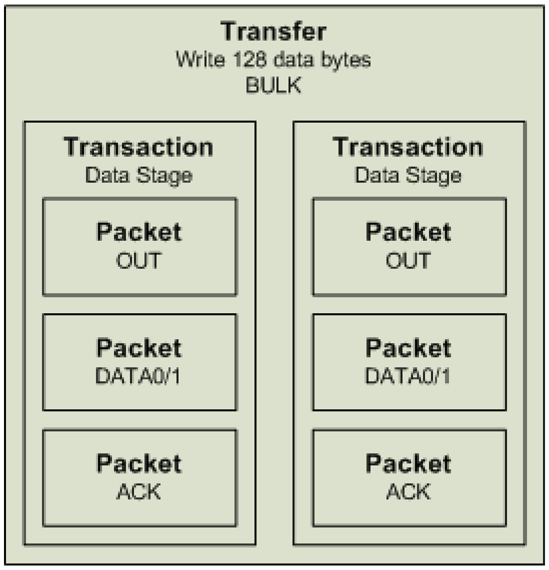
Control transfers often require all three transactions. A control transfer that reads 8 data bytes has the structure shown in Figure 2.

FIGURE 2: CONTROL TRANSFER STRUCTURE



Bulk, interrupt and isochronous transfers do not utilize the SETUP token or the status transaction. With bulk transfers, a maximum of 64 bytes may be transferred in a single data stage transaction. A bulk transfer that writes 128 data bytes requires two data stage transactions, and has the structure shown in Figure 3.

FIGURE 3: BULK TRANSFER STRUCTURE



The USB Embedded Host Stack interfaces with the USB OTG module in Host mode at the packet level, with the exception that ACK packets are handled automatically by the module. When the embedded host issues the bulk transfer shown in Figure 3, it must explicitly issue commands to transfer the two OUT and two DATA0/1 packets. When the embedded host issues the control transfer shown in Figure 2, it must explicitly issue commands to transmit all SETUP, DATA0/1, IN and OUT packets, for a total of 6 packets.

USB EMBEDDED HOST

Targeted Peripheral List

The Targeted Peripheral List (TPL) is the list of peripheral devices that the USB embedded host supports. Entries in the targeted peripheral list may refer to specific products (using the VID and PID of the product), or may refer to a class of products (using class, subclass and protocol). For example, a TPL for an embedded host that supports USB Flash drives and Microchip's PICDEM™ FS USB demonstration would look like Table 1:

TABLE 1: EMBEDDED HOST TPL

Description	Class Name	Class Code	Subclass Code	Protocol
Flash Drive	Mass Storage	0x08	0x06	0x50

Description	Manufacturer	Model	VID	PID
Full-Speed Demo	Microchip	PICDEM™ FS USB	0x04D8	0x000C

Dual Role Device or On-The-Go?

A device can be a dual role device by supporting both embedded host and USB device functionality via two receptacles. For example, a digital camera can act as a peripheral device when downloading pictures to a PC, and act as an embedded host when transferring pictures to a printer. The camera would determine its role as device or host depending on what cable was attached. If a cable was attached to the Type-A receptacle, the camera would act as a host. If a cable was attached to the Type-B receptacle, the camera would act as a device. The Type-A and Type-B receptacles must both operate concurrently unless they are only accessible one at a time.

A USB On-The-Go device is a device that can dynamically change its role from an embedded host to a peripheral device without changing cables. OTG devices use a micro-AB receptacle. The initial configuration is determined by the cable orientation. If the micro-A plug of the cable is inserted, the device will initially act as an embedded host. If the micro-B plug of the cable is inserted, the device will initially act as a peripheral device.

The choice of whether or not to utilize OTG depends on the need for the two devices to dynamically switch modes. In most cases, the host-peripheral role is constant for a pair of communicating devices. For example, the dual role digital camera described above would never need to swap roles with the PC, or with the printer, so it would not require OTG functionality.

There are also restrictions placed on the Targeted Peripheral List of an On-The-Go device. The TPL of the OTG device must list only specific devices; it may not list supported classes. For example, an OTG device cannot generically support USB Flash drives. An OTG device may support a non OTG peripheral, but it must do so by specifying the VID and PID of that device. A TPL for an OTG device that supports Microchip's PICDEM FS USB demonstration would look like Table 2:

TABLE 2: OTG TPL

Description	Manufacturer	Model	VID	PID
Full-Speed Demo	Microchip	PICDEM™ FS USB	0x04D8	0x000C

USB EMBEDDED HOST STACK

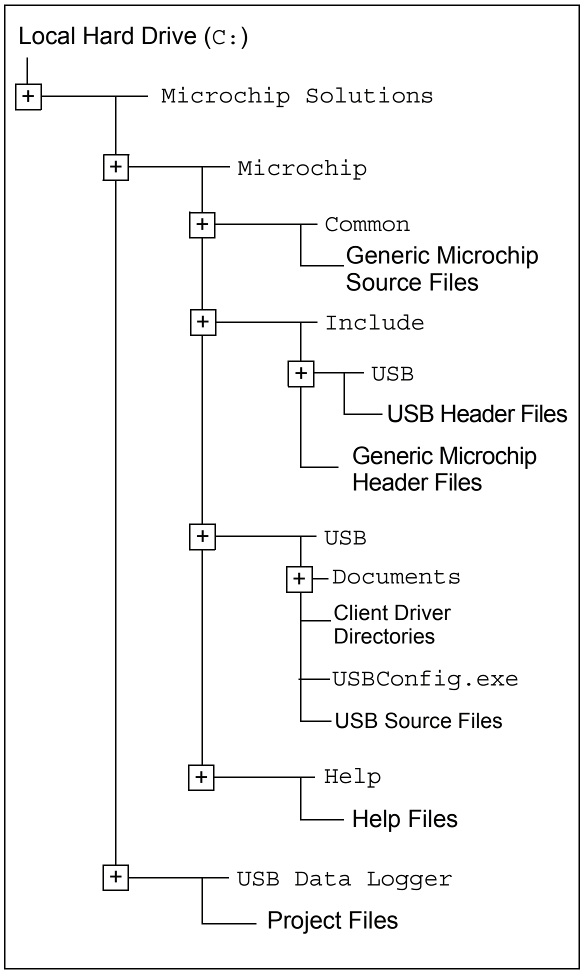
Microchip provides a royalty-free USB Embedded Host Stack for use with Microchip microcontrollers. This Stack is designed to run on all Microchip devices that have the USB OTG module. Stack operation can be configured through the use of various compile-time options to optimize both speed and size for a particular application.

The Stack is state machine based, and utilizes both interrupts and a polling mechanism. Interrupts are used for all time critical operations, while the polling mechanism is used to handle operations that are not time critical. Both mechanisms must be used to ensure correct operation of the Stack. The Stack can also be configured to provide notification of some system events rather than using the polling mechanism.

Installing the Stack

USB support packages are available from the Microchip web site (http://www.microchip.com/usb). Download the desired installation package from the web site and run the installation. Note that some USB demos utilize libraries from other application notes. The installations for those libraries will also be executed. By default, the USB Host Stack files will be installed in the directory structure shown in Figure 4.

FIGURE 4: INSTALLATION DIRECTORY STRUCTURE



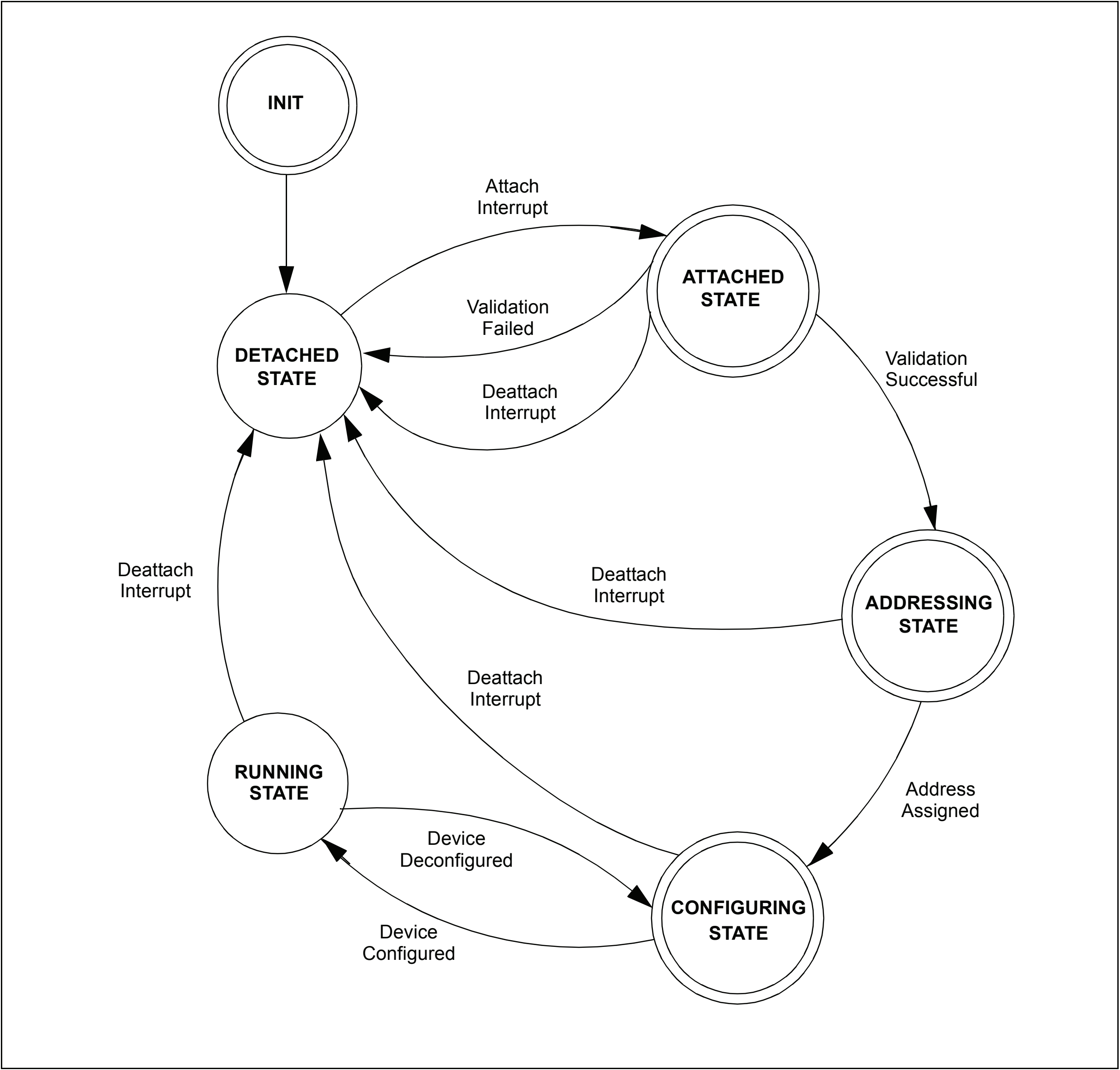
Stack Architecture

The USB Embedded Host Stack is modular, non-blocking and RTOS independent. The Stack consists of two main sections:

- State machine – for background processing, such as device enumeration
- Interrupt handler – for time critical processing to utilize the bus in an efficient manner

The state machine has the structure shown in Figure 5:

FIGURE 5: USB EMBEDDED HOST STATE MACHINE



When the device enters the running state, the Stack informs the client driver of the newly enumerated device by calling the initialization event handler. See the **“Defining the Event Handlers”** section for more information about this event handler. If the client driver successfully initializes, normal communication begins.

The USB bus operates with a 1 ms frame. Within that frame, multiple messages can be sent. Therefore, interrupts are used to respond quickly enough to effectively control USB traffic. The Stack uses two key interrupts:

- Start-Of-Frame (SOF Interrupt) – Generated by the USB OTG module when it is about to begin a new 1 ms frame.
- Transaction Complete Interrupt – Generated by the USB OTG module when the last requested transaction is complete.

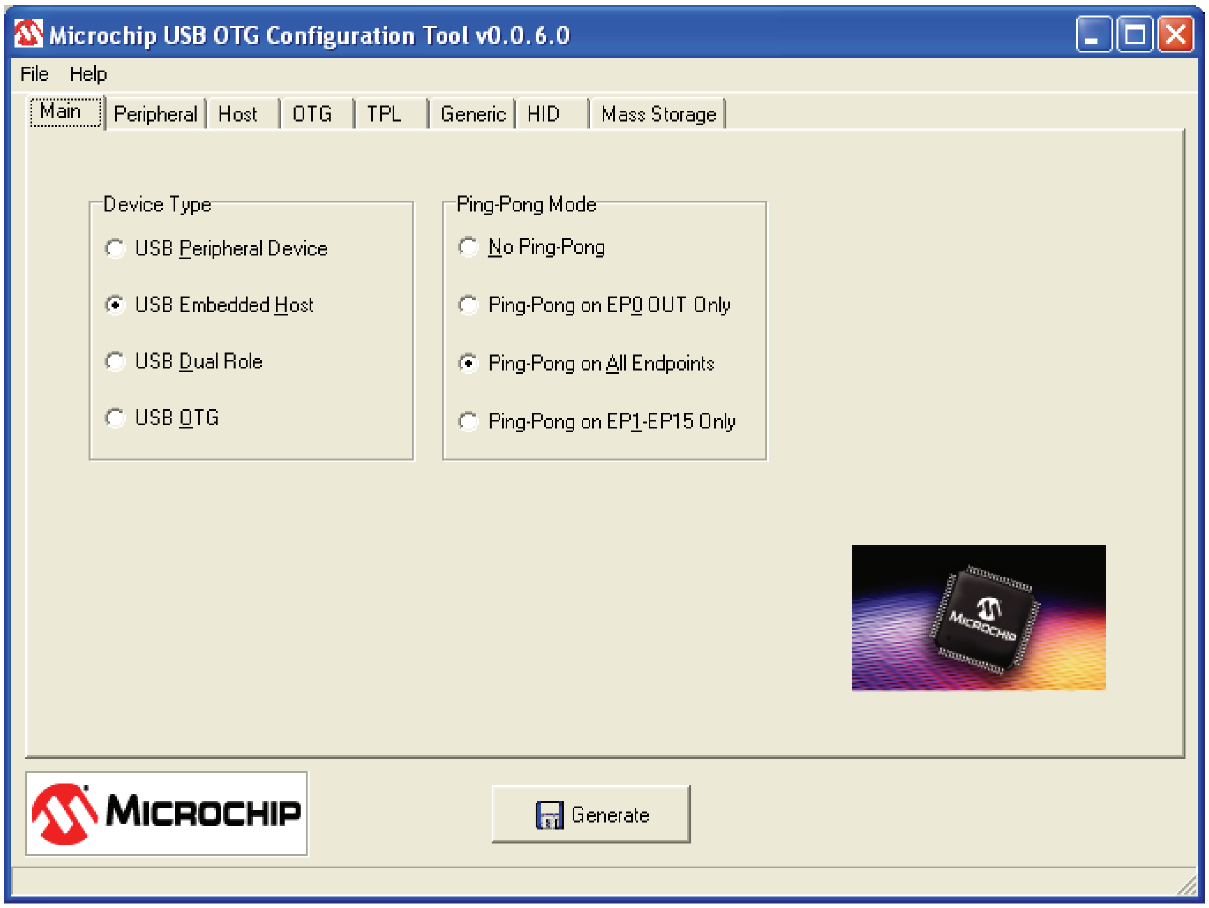
Upon receiving these interrupts, the Interrupt Service Routine (ISR) determines the next transaction, if any, to send on the bus.

Configuring the Stack

The USB Host Stack and its client drivers can be configured for a specific application by using the configuration tool, `USBConfig.exe`, located in the directory structure shown in Figure 4.

As shown in Figure 6, the Device Type and Ping-Pong Mode first need to be specified on the Main tab. After the device type is specified, appropriate configuration parameters are enabled on other tabs to configure the host/device and any required client drivers.

FIGURE 6: USB CONFIGURATION – MAIN

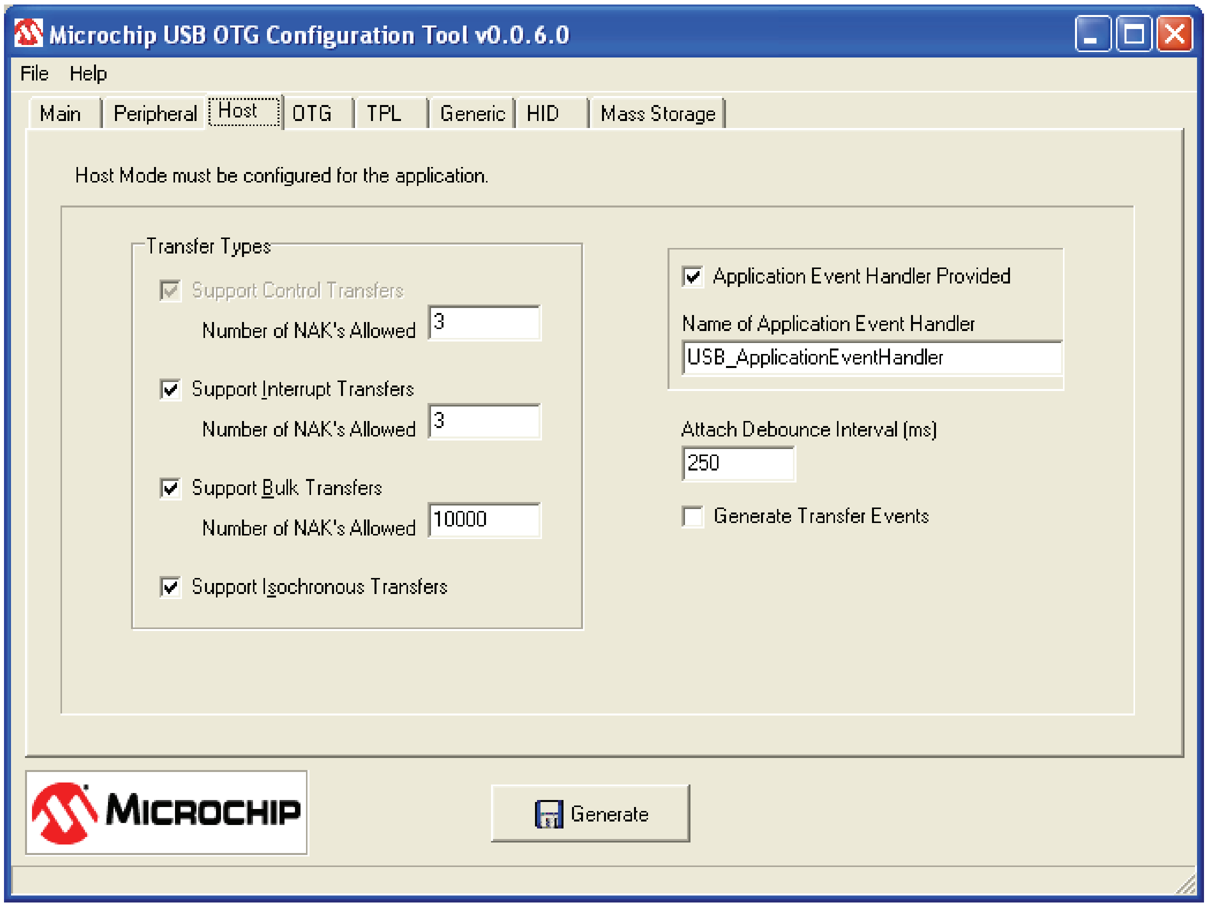


Device Type
- USB Peripheral Device
- [✓ USB Embedded Host]
- USB Dual Role
- USB OTG

Ping-Pong Mode
- No Ping-Pong
- Ping-Pong on EP0 OUT Only
- [✓ Ping-Pong on All Endpoints]
- Ping-Pong on EP1-EP15 Only

If embedded host, dual role or OTG is specified, Host mode can be configured on the Host tab, as shown in Figure 7. The endpoint support can be tailored to conserve program memory. The attach debounce interval can be increased from the USB specification value of 100 ms to allow support of slower devices. The name of the function in the main source file that serves as the application level event handler must be entered. If transfer events are going to be used, the appropriate checkbox must be checked. See the **"Using Transfer Events"** section for more information about using transfer events.

FIGURE 7: USB CONFIGURATION – HOST



Host Mode must be configured for the application.

Transfer Types

[✓] Support Control Transfers
Number of NAK's Allowed: 3

[✓] Support Interrupt Transfers
Number of NAK's Allowed: 3

[✓] Support Bulk Transfers
Number of NAK's Allowed: 10000

[✓] Support Isochronous Transfers

[✓] Application Event Handler Provided

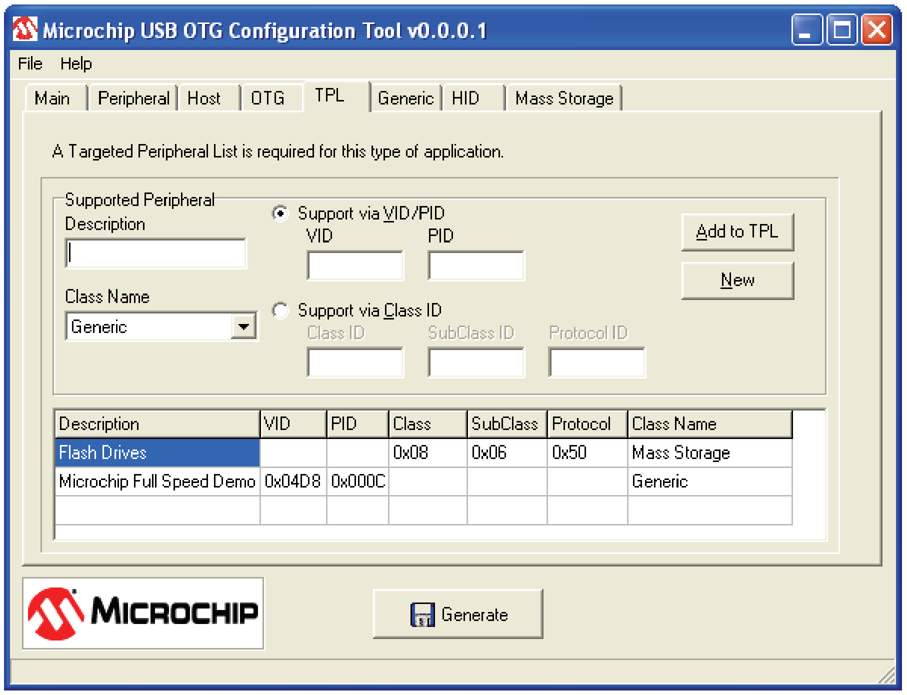
Name of Application Event Handler: USB_ApplicationEventHandler

Attach Debounce Interval (ms): 250

[ ] Generate Transfer Events

The TPL can be specified on the TPL tab, as shown in Figure 8. Either VID/PID or class specification can be used. See the **"Targeted Peripheral List"** section for more information about the TPL.

FIGURE 8: USB CONFIGURATION – TPL



A Targeted Peripheral List is required for this type of application.

Supported Peripheral

Description:

Class Name: Generic

[✓ Support via VID/PID] VID PID

Support via Class ID: Class ID, SubClass ID, Protocol ID

Description	VID	PID	Class	SubClass	Protocol	Class Name
Flash Drives			0x08	0x06	0x50	Mass Storage
Microchip Full Speed Demo	0x04D8	0x000C				Generic

Application-specific class support can be configured with other tabs. Refer to the associated USB application notes for more information about the classes and their client drivers.

PIC24F Device Requirements

The USB OTG module requires a 48 MHz clock. Since this is beyond the maximum CPU clock speed, there is a method to provide both the CPU and USB clocks from a single oscillator source.

The oscillator source must operate at a frequency that is a multiple of 4 MHz. This frequency is divided down to 4 MHz by the USB PLL prescaler. The USB PLL prescaler does not automatically sense the incoming oscillator frequency. The user must manually set the PLL divider appropriately to generate the required 4 MHz prescaler output, using the PLLDIV2:PLLDIV0 Configuration bits. This limits the choices for primary oscillator frequency to a total of 8 possibilities, shown in Table 3.

Note: The tolerance of the FRC oscillator is greater than the USB specification allows; therefore, FRC is not recommended.

TABLE 3: VALID PRIMARY OSCILLATOR CONFIGURATIONS FOR PIC24F USB OPERATION

Input Oscillator Frequency	Clock Mode	PLL Division (PLLDIV<2:0>)
48 MHz	ECPLL	÷12 (`111`)
40 MHz	ECPLL	÷10 (`110`)
24 MHz	HSPLL, ECPLL	÷6 (`101`)
20 MHz	HSPLL, ECPLL	÷5 (`100`)
16 MHz	HSPLL, ECPLL	÷4 (`011`)
12 MHz	HSPLL, ECPLL	÷3 (`010`)
8 MHz	HSPLL, ECPLL	÷2 (`001`)
4 MHz	HSPLL, ECPLL, XTPLL	÷1 (`000`)

The USB PLL output is used to drive an on-chip 96 MHz PLL frequency multiplier, which drives two clock branches. One branch uses a fixed divide-by-2 frequency divider to generate the 48 MHz USB clock. The other branch uses a fixed divide-by-3 frequency divider and a configurable PLL prescaler/divider (CLKDIV<CPDIV1:0>) to generate a range of system clock frequencies. The available system clock options are listed in Table 4. Refer to the microcontroller data sheet for more information about oscillator modes.

TABLE 4: SYSTEM CLOCK OPTIONS DURING USB OPERATION

MCU Clock Division (CPUDIV<1:0>)	Microcontroller Clock Frequency
None (`00`)	32 MHz
÷2 (`01`)	16 MHz
÷4 (`10`)	8 MHz
÷8 (`11`)	4 MHz

Note: Check the device-specific data sheet for other possible clocking options or restrictions.

PIC32 Requirements

The USB OTG module requires a 48 MHz clock. There are three methods for providing this clock:

- Use the 8 MHz FRC internal oscillator
- Provide a 48 MHz oscillator on the OSC1/OSC2 pins
- Use the 96 MHz USB PLL from OSC1/OSC2

The USB PLL is enabled via the UPLLEN Configuration bit. The oscillator source on the OSC1/OSC2 pins must operate at a frequency that is a multiple of 4 MHz. This frequency is divided down to 4 MHz by the USB PLL prescaler. The USB PLL prescaler does not automatically sense the incoming oscillator frequency. The user must manually set the PLL divider appropriately to generate the required 4 MHz output, using the UPLLIDIV Configuration bits. The 96 MHz PLL output is then fed into a fixed divide-by-2 frequency divider to generate the 48 MHz USB clock. The system and peripheral bus clocks are not affected by the use of the USB PLL.

Alternately, the FRC can be used when the USB module needs a clock source upon exit from Suspend mode. The FRC source is selected by setting the OSCCON<UFRCEN> bit.

Note: Check the device-specific data sheet for other possible clocking options or restrictions.

Project Requirements

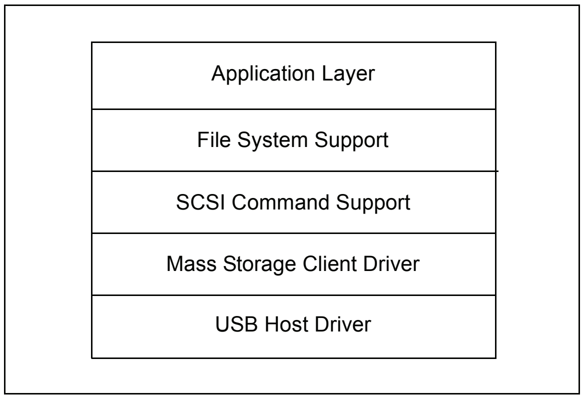
The USB Embedded Host Stack stores information about the attached device in dynamic memory. Enough heap space must be allocated for the USB peripheral's device descriptor, all configuration descriptors and 64 bytes of overhead. Client drivers may have additional dynamic memory requirements. If there is not enough dynamic memory to hold the device information, the device will fail to enumerate.

Heap size can be specified in the MPLAB® IDE project by selecting *Project>Build Options…>Project* from the MPLAB IDE main menu. For 16-bit microcontrollers, select the MPLAB LINK30 tab, and enter a value in the heap size edit box. Then, click **OK**.

Using the Stack

Most applications will not interface directly with the USB Embedded Host Stack. Instead, they will use a client driver, which in turn will use the Host Stack. Some applications have additional layers that interface with the client driver. For example, the application described by *AN1145, "Using a USB Flash Drive on an Embedded Host"* has five layers, including the application layer, as shown by Figure 9.

FIGURE 9: USB FLASH DRIVE DATA LOGGER ARCHITECTURE



The information presented here is primarily intended for client driver developers.

Note: For detailed information about the USB Embedded Host Stack API, please refer to the API documentation provided in the Help directory and *AN1141, "USB Embedded Host Stack Programmer's Guide"*.

Defining a TPL

The Targeted Peripheral List (TPL) must be defined so the Stack knows what USB peripheral devices it can support. If the application is using USB OTG, the TPL may consist only of specific VID and PID pairs for the supported devices. If the application is using USB embedded host, then the TPL may consist of supported classes as well as specific VID and PID pairs.

The TPL can be defined using the configuration tool, `USBConfig.exe`, provided with the Stack. See the **"Configuring the Stack"** section.

Defining the Event Handlers

CLIENT LEVEL EVENTS

The Stack requires two event handlers in the client driver. The first is the device initialization handler, which is called during device enumeration, after the device's configuration has been set. The device initialization handler should be of the type defined by the `typedef`:

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

This function performs device initialization specific to the device's class. If initialization occurs with no error, this routine should return `TRUE`. If errors are encountered, this routine should return `FALSE`, and the device will not be enumerated. Example 1 shows a sample initialization handler.

EXAMPLE 1: INITIALIZATION HANDLER

```
BOOL USBHostSampleInitialize( BYTE address, DWORD flags )
{
    // This handler supports one attached device.
    if (deviceHandle != 0)
    {
        return FALSE; // We cannot support this device.
    }
    deviceHandle = address;

    // Initialize something based on flags.
    if (flags & 0x01)
    {
        // Do something
    }
    else
    {
        // Do something else
    }
    return TRUE; // Device successfully initialized
}
```

The second event handler is required to handle events that occur during normal operation. This event handler should be of the type defined by the `typedef`:

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

For example, one of the events that can occur is `EVENT_DETACH`. This occurs when a device has detached from the bus. In this case, the client driver will need to update its status by doing operations, such as removing the device from its list of attached devices that utilize the client driver. Example 2 shows a sample event handler.

EXAMPLE 2: EVENT HANDLER

```
BOOL USBHostSampleEventHandler( BYTE address, USB_EVENT event,
                      void *data, DWORD size )
{
    switch (event)
    {
        case EVENT_DETACH:
            deviceHandle = 0;
            return TRUE;
            break;

        // More events handled here…
    }
    return FALSE; // We did not handle the event
}
```

Several events are provided for OTG operations. In addition, the Stack can be configured to provide transfer events (`EVENT_TRANSFER`). Refer to the **"Using Transfer Events"** section.

See the API documentation in the Help directory for a complete list of events.

The Stack requires a list of all initialization and event handlers for every supported client driver. This list can be defined using the configuration tool, `USBConfig.exe`, provided with the Stack. See the **"Configuring the Stack"** section.

APPLICATION LEVEL EVENTS

Some events are intended for the top level application rather than the client driver. The primary example of this type of event is `EVENT_REQUEST_POWER`. This event is generated when a device requests power while in the process of enumerating. If the application needs to monitor or control how much power is allocated to the attached device, it must implement an application event handler. This event handler must also be of the type, `USB_CLIENT_EVENT_HANDLER`, shown above. The name of the event handler must be entered into the Name of Application Event Handler field on the Host tab of the configuration tool, `USBConfig.exe` (see the **"Configuring the Stack"** section). Example 3 shows a minimum event handler for controlling power.

EXAMPLE 3: APPLICATION LEVEL EVENT HANDLER

```
BOOL USB_ApplicationEventHandler( BYTE address, USB_EVENT event,
                void *data, DWORD size )
{
    switch( event )
    {
        case EVENT_REQUEST_POWER:
            // data points to a byte that represents the amount
            // of power requested in mA, divided by two.
            if (*(BYTE*)data > 50)  // We will allow up to 100mA
            {
                return FALSE;
            }
            break;
    }
    return TRUE;
           // Allow all other events to pass.
}
```

If no event handler is implemented, the Embedded Host Stack will respond as if the event handler returned `TRUE`.

Stack Initialization

USB embedded host operation is initialized by a single function:

`BYTE USBHostInit( void );`

This function initializes all internal variables for Stack operation. It should only be called once during the application's execution. The USB OTG module on the microcontroller is not configured in this function.

The USB configuration tool will create a macro, `USBInitialize()`, to call all of the initialization routines required by the USB host driver and the supported client drivers.

Note:	Applications should use the `USBInitialize()` macro instead of the lower level function call.

Normal Stack Operation

After initialization, USB OTG module configuration and USB embedded host operation are triggered by a single function:

`void USBHostTasks( void );`

This function implements the state machine required to enumerate a device. It should be called at regular intervals to ensure timely enumeration.

The USB configuration tool will create a macro, `USBTasks()`, to call all of the task routines required by the USB host driver and the supported client drivers.

Note:	Applications should use the `USBTasks()` macro instead of the lower level function call.

Communicating with a Peripheral Device

Note:	This section describes the routines required for a client driver to communicate with a peripheral device. Application level code will not call these functions; instead, they will utilize the appropriate routine(s) in the client driver(s).

Normal communication with a device is initiated by two functions:

`BYTE USBHostRead( BYTE deviceAddress, BYTE endpoint, BYTE *data, DWORD size );`

`BYTE USBHostWrite( BYTE deviceAddress, BYTE endpoint, BYTE *data, DWORD size );`

A return code of USB_SUCCESS (0x00) indicates that the operation was started successfully.

After initiating communication, take care that `USBTasks()` (`USBHostTasks()` and any required client tasks) are performed while waiting for the operation to complete. The status of the operation can be determined by calling the function:

```
BOOL USBHostTransferIsComplete( BYTE
deviceAddress, BYTE endpoint, BYTE
*errorCode, DWORD *byteCount );
```

If the function returns `FALSE`, the transfer is not complete, and the returned error code and byte count are not valid. If the function returns `TRUE`, the returned error code indicates the status of the operation, and the returned byte count indicates how many bytes were transferred.

A transfer of data from the host to the device might look like Example 4.

EXAMPLE 4: SEND DATA TO A PERIPHERAL DEVICE

```
error = USBHostWrite( device, EP1, buffer,  sizeof(buffer) );
if (error)
{
    // There was a problem
}
else
{
    while (!USBHostTransferIsComplete( device, EP1, &error, &count ))
    {
        USBHostTasks();
    }
    if (error)
    {
        // There was a problem
    }
    else
    {
        // The data was transferred successfully
    }
}
```

Advanced Stack Use

USING TRANSFER EVENTS

Example 4 shows a simple method of transferring data, but it has the disadvantage that execution is trapped in a loop while waiting for the transfer to complete. One way to avoid this is to utilize transfer events.

All client drivers must have an event handler, since some events must be supported for basic Stack operation. Transfer events, however, are optional, since they may not be needed for all applications and they increase required resources.

Transfer events operate differently than other events in the system. The completion status of a USB transfer is determined in the USB ISR, when the transaction complete interrupt fires. However, the transfer event is not generated at this point. In order to utilize the bus efficiently, USB interrupts must be as short as possible to allow as many transactions as possible within one USB frame (1 ms). The client driver's event handler may be rather large and require a great deal of time to execute, and it may generate another event that must be handled by another layer of the application, resulting in a very long delay. Therefore, a transfer event queue is used. Transfer events (`EVENT_TRANSFER`) are enqueued in the ISR along with a structure of the type, `HOST_TRANSFER_DATA`, so the application can determine what transfer completed and whether or not it was successful. The function, `USBHostTasks()`, dequeues the transfer events and sends them to the client driver's event handler.

In general, more program and data memory are required to support transfer events due to the transfer event queue. The code architecture is more sophisticated, and this architecture is more difficult for a beginning C programmer to design, develop, debug and maintain. However, transfer events allow the client driver to minimize or even eliminate a background task processing function, utilizing processing bandwidth more efficiently.

TERMINATING TRANSFERS

The Stack has the ability to automatically terminate transfers that the device is failing to respond to by counting the number of NAKs that the device sends. If the application requires a different time-out mechanism, it can terminate a transfer manually by calling the function:

```
void USBHostTerminateTransfer( BYTE
deviceAddress, BYTE endpoint );
```

ISSUING STANDARD DEVICE REQUESTS

All USB devices respond to certain requests from the host. The application may issue USB standard device requests by calling the function:

```
BYTE USBHostDeviceRequest( BYTE
deviceAddress, BYTE bmRequestType, BYTE
bRequest, WORD wValue, WORD wIndex, WORD
wLength, BYTE *data, BYTE dataDirection
);
```

The Stack will then issue the requested control transfer on Endpoint 0. Refer to the USB 2.0 Specification for more information about standard device requests.

CLEARING ENDPOINT ERRORS

If a transfer is unsuccessful because too many errors occurred, or because the device stalled the endpoint, the error must be cleared before another transfer can be attempted. The error condition may be cleared by calling the function:

```
BYTE USBHostClearEndpointErrors( BYTE
deviceAddress, BYTE endpoint );
```

If a stall occurred, `USBHostClearEndpointErrors()` must be called, plus the stall condition on the device must be cleared. This is done by issuing a `CLEAR FEATURE` device request (as described in the **"Issuing Standard Device Requests"** section) with the `ENDPOINT HALT` feature selector and the endpoint number that is currently stalled. Refer to the USB 2.0 Specification for more information about the `CLEAR FEATURE` device request.

ACCESSING THE DEVICE AND CONFIGURATION DESCRIPTORS

During enumeration, the Stack obtains and stores the device descriptor and the configuration descriptors of the device. The client driver can access the device descriptor by using the `USBHostGetDeviceDescriptor()` function. This function returns a pointer to the device descriptor.

```
BYTE * USBHostGetDeviceDescriptor( BYTE
deviceAddress );
```

The client driver can also access the configuration descriptor of the current configuration by using the function, `USBHostGetCurrentConfiguration-Descriptor()`. This function also returns a pointer to the descriptor.

```
BYTE * USBHostGetCurrentConfiguration-
Descriptor( BYTE deviceAddress )
```

GETTING STRING DESCRIPTORS

A USB peripheral device may contain one or more string descriptors that describe items, such as the product, the manufacturer and the device's serial number. The index of the desired string can be obtained by referencing the appropriate field in the device descriptor. Then, the desired string can be obtained by calling the function:

```
BYTE USBHostGetStringDescriptor ( BYTE
deviceAddress, BYTE stringNumber, BYTE
*stringDescriptor, BYTE stringLength )
```

Example 5 shows how to obtain an attached device's serial number.

Refer to the USB 2.0 Specification for more information about the device descriptor and string descriptors.

EXAMPLE 5: RETRIEVING A DEVICE'S SERIAL NUMBER

```
deviceDescriptor = USBHostGetDeviceDescriptor( device );
snIndex = deviceDescriptor[16];  //See Device Descriptor spec.

if (snIndex != 0)
{
    if (!USBHostGetStringDescriptor( device, snIndex, buffer, 50 ))
    {
        while(!USBHostTransferIsComplete( device, 0, &error, &count ))
        {
            USBHostTasks();
        }
        if (!error)
        {
            // Serial number is in buffer[] in UNICODE
        }
    }
}
```

CONCLUSION

The USB Embedded Host Stack for Microchip's 16-bit and 32-bit microcontrollers provides a platform for supporting various USB classes. With these solutions, applications can now utilize the wide variety of available USB peripheral devices, including mass storage and human interface devices. The USB embedded host capability of the USB OTG module, available on select 16-bit and 32-bit Microchip microcontrollers, opens a whole new world for embedded applications.

REFERENCES

- *AN1141, "USB Embedded Host Stack Programmers Guide"* (http://www.microchip.com)
- *AN1142, "USB Mass Storage Class on an Embedded Host"* (http://www.microchip.com)
- *AN1143, "USB Generic Client on an Embedded Host"* (http://www.microchip.com)
- *AN1144, "USB HID Class on an Embedded Host"* (http://www.microchip.com)
- *AN1145, "Using a USB Flash Drive on an Embedded Host"* (http://www.microchip.com)
- Universal Serial Bus web site (http://www.usb.org)
- Microchip Technology Inc. web site (http://www.microchip.com)

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