

---

---

## USB Generic Function on an Embedded Device

---

---

<i>Author: Bud Caldwell Microchip Technology Inc.</i>
---

### INTRODUCTION

The Universal Serial Bus (USB) is a simple and common interface for connecting peripheral devices to a personal computer or other host. To harness its flexibility and power with minimal effort, Microchip provides the USB Generic Function firmware. The generic driver provides a very simple interface for reading and writing data exchanged with the USB host. Sample code is provided that is easily customized for the designer's application.

If the generic driver does not provide all of the functionality required by a particular application (see "**USB Generic Function API**"), Microchip provides sample implementations of other frequently-requested USB device classes. These sample implementations are built upon the Microchip PIC32 USB peripheral firmware stack. If no sample is available that suits the desired application, the designer can develop his or her own function driver using the Microchip USB stack (refer to Microchip Application Note AN1176, "*USB Device Stack for PIC32 Programmer's Guide*").

This document describes the USB generic function on an embedded device. It also acts as a programmer's guide for developers who wish to provide a simple read/write data interface to a host over the USB and it describes how to incorporate the Microchip generic function driver into the developer's own application.

### ASSUMPTIONS

1. Working knowledge of C programming language
2. Some familiarity with the USB 2.0 protocol
3. Familiarity with Microchip MPLAB<sup>®</sup> IDE and MPLAB<sup>®</sup> REAL ICE<sup>™</sup> in-circuit emulator

### FEATURES

- Supports USB peripheral device applications
- Provides simple read/write interface for data exchange with the host
- Handles standard USB device requests, as stated in Chapter 9 of the "*Universal Serial Bus Specification, Revision 2.0*" (<http://www.usb.org/developers/docs/>)
- Simplifies definition of USB descriptors and configuration information
- Event-driven system (interrupt-based or polled)

### LIMITATIONS

- Uses a single endpoint
- Uses interrupt transfer protocol
- Theoretical max throughput: 64,000 bytes/sec., in accordance with the USB 2.0 specification

### SYSTEM HARDWARE

This application was developed for the following hardware:

- PIC32 USB PIM (Processor Interface Module)
- Microchip Explorer 16 Development Board
- USB PICTail<sup>™</sup> Plus Daughter Board

# AN1166

---

## PIC32 MCU MEMORY RESOURCE REQUIREMENTS

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

## PIC MCU HARDWARE RESOURCE REQUIREMENTS

The USB generic demo uses the following I/O pins:

**TABLE 1: PIC® MCU I/O PIN USAGE**

I/O Pin	Usage
D+ (IO)	USB D+ differential data signal
D- (IO)	USB D- differential data signal
VBUS (Input)	USB power
VUSB (Input)	Power input for the USB D+/D- transceivers
AN4 (Input)	Monitor Temperature Sensor via A/D Converter
AN5 (Input)	Monitor Potentiometer via A/D Converter

## PICtail Plus™ Jumper Settings

The Microchip USB PICtail Plus Daughter Board requires the following jumper settings:

**TABLE 2: JUMPER SETTINGS**

Jumper	Setting
JP1	Open
JP2	Short
JP3	Open

## INSTALLING SOURCE FILES

The Microchip generic function firmware source is available for download from the Microchip web site (see **Appendix G: “Source Code for the USB Generic Function on an Embedded Device”**). The source code is distributed in a single Windows® installation, as part of the PIC32 device install.

Perform the following steps to complete the installation:

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

## SOURCE FILE ORGANIZATION

The Microchip USB generic device firmware consists of several files that are organized in multiple directories. Table 3 shows the directory structure.

**TABLE 3: SOURCE FILES**

File	Directory	Description
usb_device.c	Microchip\USB	USB Device layer (device abstraction and protocol handling, as in Chapter 9 of the “ <i>Universal Serial Bus Specification, Revision 2.0</i> ”)
usb_hal.c	Microchip\USB	USB Hardware Abstraction Layer (HAL) interface support
usb_hal_core.c	Microchip\USB	USB controller functions, used by HAL interface support
usb_device_local.h	Microchip\USB	Private definitions for USB device layer
usb_hal_core.h	Microchip\USB	Private definitions for HAL controller core
usb_hal_local.h	Microchip\USB	Private definitions for HAL
usb.h	Microchip\Include\USB	Overall USB header (includes all other USB headers)
usb_ch9.h	Microchip\Include\USB	USB device framework definitions (Chapter 9 of the “ <i>Universal Serial Bus Specification, Revision 2.0</i> ”)
usb_common.h	Microchip\Include\USB	Common USB stack definitions
usb_device.h	Microchip\Include\USB	USB device layer interface definition
usb_hal.h	Microchip\Include\USB	USB HAL interface definition
usb_device_generic.h	Microchip\Include\USB	Generic function driver API header
usb_func_generic.c	Microchip\USB\generic_device_driver	Generic function driver implementation
usb_func_generic_local.h	Microchip\USB\generic_device_driver	Private definitions for generic function driver
HardwareProfile.h	usb_generic_device_demo	Hardware configuration parameters
local_typedefs.h	usb_generic_device_demo	Application specific data type definitions
main.c	usb_generic_device_demo	Primary application source file
usb_config.h	usb_generic_device_demo	Application-specific USB configuration options (see <b>Appendix A: “USB Firmware Stack Configuration”</b> )
usb_demo1_app.c	usb_generic_device_demo	Application-specific USB support
user.c	usb_generic_device_demo	“User” procedures/IO-processing code
user.h	usb_generic_device_demo	“User” procedures/IO-processing code definitions

## DEMO APPLICATION

The demo application consists of host-side PC (Personal Computer) software and device-side PIC32 firmware. Together, the two demonstrate a simple way to transfer data between a USB host and device. The PC application provides a Graphical User Interface (GUI) that allows the user to interact with the PIC32. The PC application supports reading the current setting of a potentiometer, reading temperature data, and controlling two LEDs on the Explorer 16 development board. The PIC32 firmware demonstrates how to support these services.

**Notes:** The PIC32 generic driver demo uses the existing PICDEM™ FS USB Demo Tool host PC application. Please refer to the *"PICDEM™ FS USB Demonstration Board User's Guide"* (DS51526) for details on installing and using this application.

The PICDEM™ Demo Tool has two modes: Boot mode and Demo mode. The PIC32 generic demo only uses Demo mode. Only that mode of the Demo Tool will be operational when using the tool with the PIC32 custom driver firmware.

## Programming the Demo Application

To program a target with the demo application, you must have access to an MPLAB REAL ICE in-circuit emulator. The following procedure assumes that you will be using MPLAB IDE. If not, please refer to your specific programmer's instructions.

1. Connect MPLAB REAL ICE to the Explorer 16 development board or your target board.
2. Apply power to the target board.
3. Launch MPLAB IDE.
4. Select the PIC device of your choice (required only if you are importing a hex file previously built).
5. Enable MPLAB REAL ICE as a programmer.
6. If you want to use a previously built hex file, import it into MPLAB.
7. If you are rebuilding the hex file, open the project file and follow the build procedure to create the application hex file.
8. The demo application contains necessary configuration options required for the Explorer 16 board. If you are programming another type of board, make sure that you select the appropriate oscillator mode from the MPLAB IDE configuration settings menu.
9. Select the "Program" menu option from the MPLAB REAL ICE programmer menu to begin programming the target.
10. After a few seconds, the message "Programming successful" is displayed. If not, check the board and MPLAB REAL ICE connections. Refer to MPLAB REAL ICE online help for further assistance.
11. Remove power from the board and disconnect the MPLAB REAL ICE cable from the target board.
12. Reapply power to the board and make sure that the LCD reads "PIC32 Generic Demo". If it does not, check your programming steps and repeat, if necessary.

## Using the Demo Application

Connect the PIC32 USB PIM and USB PICtail™ Plus Daughter Board to the Explorer 16 Development Board. Open the project file in the MPLAB IDE, build the firmware, and program it into the PIC32. Reset the microcontroller and start the firmware running. Then, attach the device connection to the host.

**Notes:** Refer to the MPLAB IDE online help for instructions on how to build and program the firmware.

After the firmware is programmed into the PIC32, and it is connected to the host the first time, refer to the installation instructions for details on installing the USB device driver on the PC

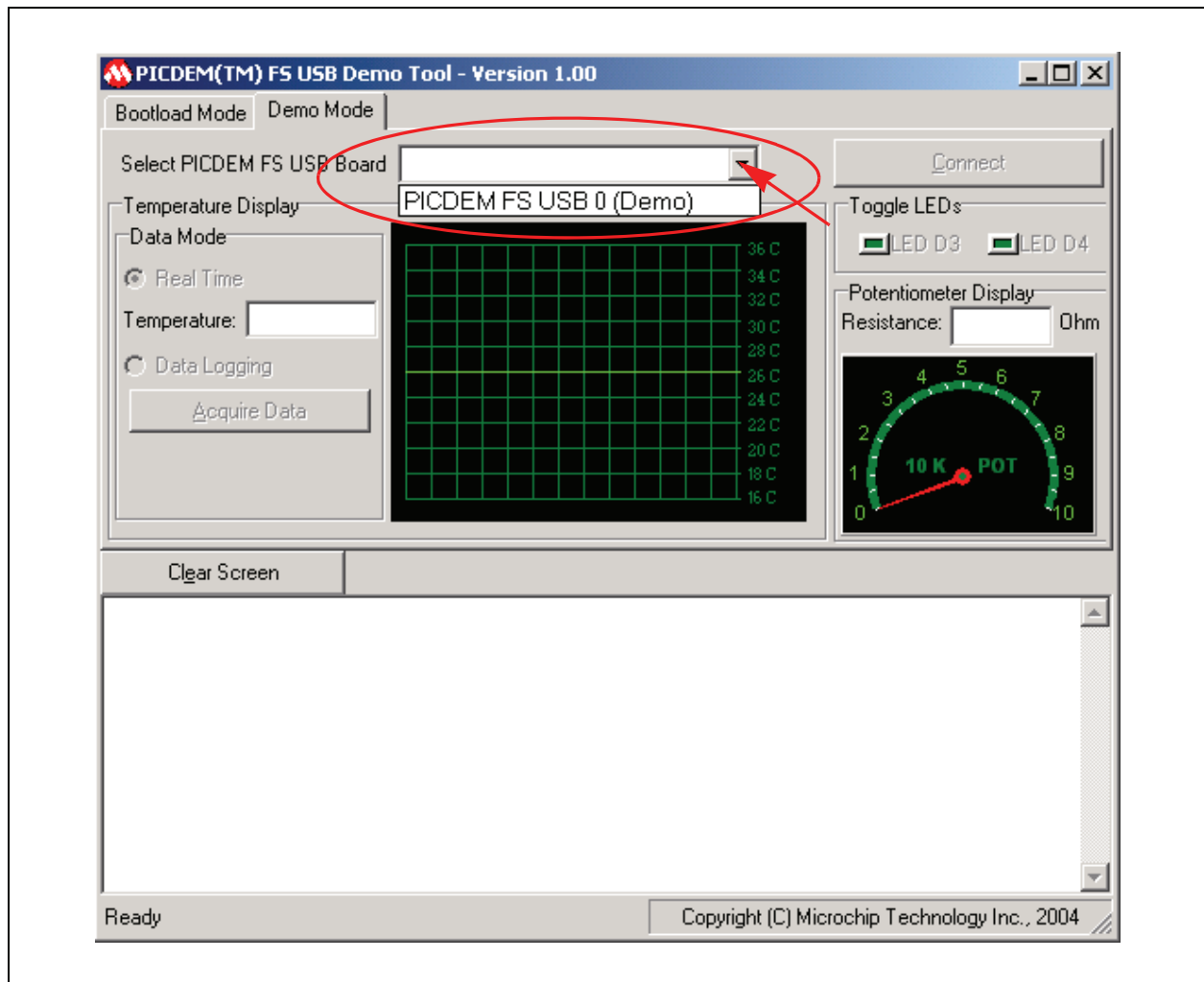
When the PIC32 is programmed and connected to the host, it is available in the pull-down box labeled "Select PICDEM FS USB Board" (see Figure 1).

Select the board and click **Connect** to start receiving temperature and potentiometer data. Click on the "Toggle LEDs" buttons to control the LEDs on the Explorer board.

**Note:** The demo tool LED buttons are labeled for the PICDEM FS USB demonstration board, not the Explorer 16 development board. So, the button labeled "LED D3" actually controls the LED labeled "D9", and the button labeled "LED D4" actually controls the LED labeled "D10" on the Explorer 16 board.

The PICDEM Demo Tool application uses a device driver to access the PIC32 over the USB and a separate DLL to provide a simple API to access the device driver. For additional details and example code, refer to the source code and Readme files installed with the PICDEM Demo Tool application.

**FIGURE 1: SELECTING THE DEMO APPLICATION**



## The Firmware

The generic demo firmware includes the following features:

1. Managing the USB
2. IO Processing

### MANAGING THE USB

The application's main logic calls the `USBInitialize` routine once, before any other USB activity takes place, to initialize the USB firmware stack. Then, it calls the `USBTasks` routine in a "polling" loop, as shown in Example 1.

#### EXAMPLE 1: MAIN APPLICATION LOGIC

```
USBInitialize(0);
UserInit();
while(1)
{
    USBTasks();
    ProcessIO();
}
return 0;
```

**Note:** Until `USBInitialize` is called, the USB interface module is disabled and the PIC32 will not connect to the USB.

The `USBInitialize` routine, helped by the application-specific USB support, handles everything necessary to initialize the USB firmware stack. The `USBTasks` routine manages the state of the USB firmware stack and performs the necessary steps required by events that occur on the bus.

**Note:** The `USBTasks` routine may also be called from the Interrupt Service Routine (ISR) whenever a USB interrupt occurs, instead of in a polling loop. If it is used this way, the entire USB firmware stack (the non user API portion of the generic driver) will operate in an interrupt context.

## IO PROCESSING

The two routines called from the main logic (see Example 1) that have not yet been discussed are the `UserInit` and `ProcessIO` routines. These two routines manage the IO to the non-USB portions of the demo application, read commands from the PC application, and write data back to the PC.

The `UserInit` routine initializes the PIC32 pins that control the LEDs on the Explorer 16 board. It also starts a timer that is used to control the sampling of the temperature sensor and initializes the temperature logging facility.

The process IO routine uses the `ServiceRequests` helper routine to read and respond to requests from the PC application and maintain the timing and acquisition of temperature samples. To do this, it uses the "USB Generic Function API" to read command packets from the host, as shown below:

#### EXAMPLE 2: PROCESS IO ROUTINE

```
if(USBGenRead((BYTE*)&dataPacket,sizeof(
dataPacket)))
{
    // Handle commands
}
```

The first byte of data read from the host identifies a command, as shown in Example 3. The rest of the packet contains command-specific data used to execute the given command.

### EXAMPLE 3: FIRST BYTE OF DATA PACKET FROM HOST

```

counter = 0;
switch(dataPacket._byte[0])
{
    case READ_VERSION:
        // Provide firmware version data
        break;

    case UPDATE_LED:
        // Update LEDs as specified
        break;

    case SET_TEMP_REAL:
        // Reset the temperature log;
        break;

    case RD_TEMP:
        // Acquire a temperature sample.
        break;

    case SET_TEMP_LOGGING:
        // Start temperature logging;
        break;

    case RD_TEMP_LOGGING:
        // send the temperature log data to the host and reset it.
        break;

    case RD_POT:
        // Read the potentiometer
        break;

    default:
        break;
}

```

In each case, once the command has been completed, a counter variable is updated to hold the count of data bytes sent back to the host. The firmware application then uses the USB Generic Function API to send the data back to the host (as shown below).

Using this method of command and response, the firmware demo application supports the services required by the PC application, demonstrating the ability to transfer data between the PIC32 and the host PC using the USB Generic Function API.

### EXAMPLE 4: SENDING DATA TO HOST

```

if(counter != 0)
{
    if(!mUSBGenTxIsBusy())
        USBGenWrite((BYTE*)
            &dataPacket, counter);
}

```

## Application-Specific USB Support

Since the demo uses the Microchip USB device firmware stack, it implements three application-specific tables, listed below.

Application-specific tables:

1. USB Descriptor Table
2. Endpoint Configuration Table
3. Function-Driver Table

These tables, and the functions used by the USB stack to access them, are defined in `usb_demo1_app.c`.

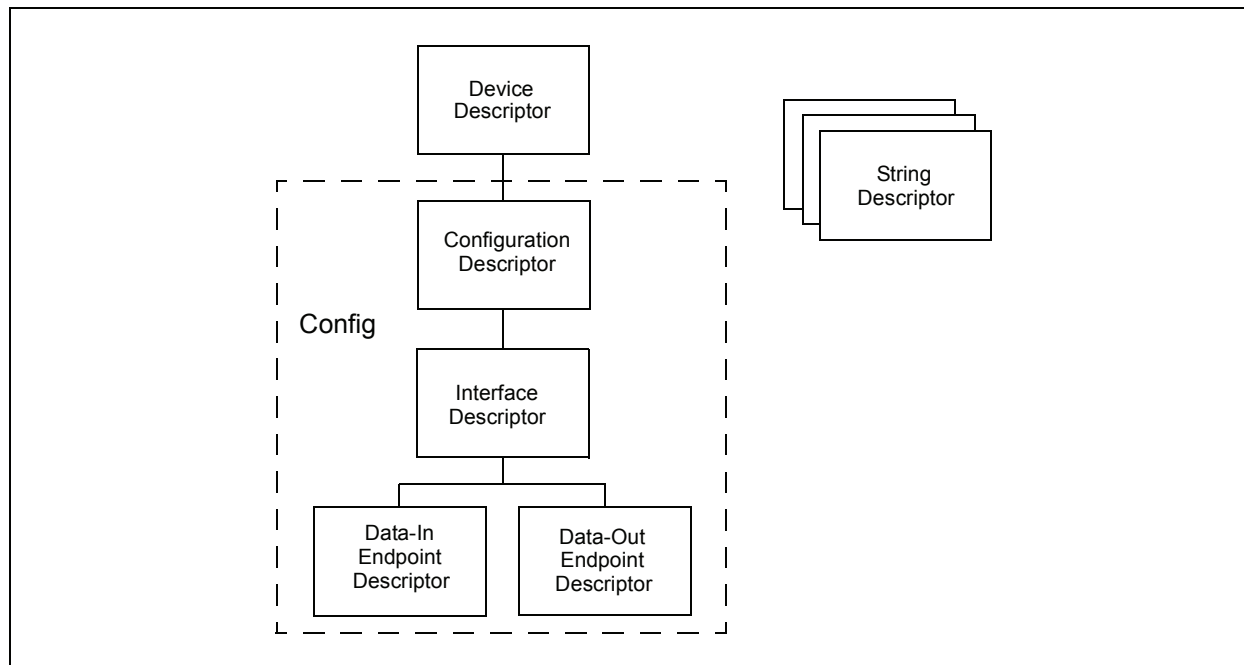
### THE USB DESCRIPTOR TABLE

Every USB device must provide a set of data structures called “descriptors” that give details to the host about how to use it. Exactly how these descriptors are provided and what information they contain is defined in Chapter 9 of the “*Universal Serial Bus Specification, Revision 2.0*” and its class-specific supplements. Please refer to these documents for complete details. The demo application defines sample descriptors. Key fields that may need to be changed for different applications are discussed later.

In general terms, the USB descriptors can be thought of as belonging to one of three different groups: those describing the overall device, those describing possible device configurations, and those providing user-readable information. Each USB device has one, and only one, descriptor in the first group – the device descriptor. It identifies the type of device and gives the number of possible configurations. Each configuration (the second group) has its own set of descriptors, describing the details of that configuration. User-readable information is kept in the string descriptors, making up the third group. String descriptors are optional, but helpful to the end user. For additional information, see Figure 2, and also, refer to **Appendix E: “USB Descriptor Table”**.

In order for the host to read to these descriptors, the USB firmware stack must have access to them. To provide this access, the application defines a `USBDevGetDescriptor` routine. This routine is called by the lower-layers of the stack and passed a value identifying the descriptor type. For string descriptors, the routine is also passed a string index number and language ID. In all cases, the routine must provide a pointer to the requested descriptor and its size in bytes. For details, see **Appendix F: “Get Descriptor Routine”**.

**FIGURE 2: DESCRIPTOR GROUPS**





## THE ENDPOINT CONFIGURATION TABLE

Software on the host PC communicates to functions on USB devices through logical “interfaces” containing one or more “endpoints”. Endpoints and interfaces are identified by numbers, starting at zero. USB devices can have one or more configurations of these endpoints and interfaces, identified by a number starting at one. Which configuration is used is selected by the host during a process called “enumeration”. However, the generic function driver only has one configuration.

The endpoint configuration table in Example 5 identifies which endpoints belong to which interface along with the data transfer direction and protocol features for each endpoint.

### EXAMPLE 5: ENDPOINT CONFIGURATION TABLE

```
const EP_CONFIG gEpConfigTable[] =
{
    { // EP1 - In & Out
      EP_MAX_PKT_INTR_FS,
      USB_EP_TRANSMIT |
      USB_EP_RECEIVE |
      USB_EP_HANDSHAKE,
      USBGEN_EP_NUM,
      USBGEN_CONFIG_NUM,
      USBGEN_INTF_NUM,
      0,
      0
    }
};
```

This table configures endpoint one as bidirectional (USB\_EP\_TRANSMIT | USB\_EP\_RECEIVE), supporting handshaking (USB\_EP\_HANDSHAKE) as required by the interrupt protocol. It also associates this endpoint with the function driver at index zero in the function table (see “**The Function Driver Table**”). To access the configuration table, the application defines the following routine.

### EXAMPLE 6: GET ENDPOINT CONFIGURATION TABLE ROUTINE

```
const EP_CONFIG *USBDEVGetEpConfigurationTable ( int *num_entries )
{
    // Provide the number of entries
    *num_entries = sizeof(gEpConfigTable)/sizeof(EP_CONFIG);

    // Provide the table pointer.
    return gEpConfigTable;
}
```

This routine provides a pointer to the endpoint configuration table as well as the number of entries it contains to the USB firmware stack. It is identified to the stack by the macro USB\_DEV\_GET\_EP\_CONFIG\_TABLE\_FUNC (see “**USB Stack Options**”).

## THE FUNCTION DRIVER TABLE

The Microchip device firmware stack uses a table to manage access to function drivers, as it is capable of supporting multi-function devices. Each entry in the table contains the information necessary to manage a single function driver. Since the demo only implements a single USB function, its table only contains one entry as shown below.

### EXAMPLE 7: FUNCTION DRIVER TABLE

```
const FUNC_DRV gDevFuncTable[] =
{
    { // Generic Function Driver
      USBGenInitialize,
      USBGenEventHandler,
      USBGEN_EP_NUM
    }
};
```

This table provides pointers to the generic function driver's initialization and event-handling routines, as well as an initialization value identifying, in this case, the endpoint used by the driver. This is all the information that the USB firmware stack needs to manage the generic driver and make sure it is aware of events that occur on the bus.

To provide the USB firmware stack with access to this table, the application defines the following routine.

### EXAMPLE 8: FUNCTION DRIVER TABLE ACCESS ROUTINE

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

This routine returns the pointer to the base of the function driver table. The size of the table is not needed because the endpoint configuration table contains the indices into the function driver table. As long as these indices are correct, no access violation will occur.

## USB Stack Options

The Microchip USB device firmware stack supports a number of configuration options. These can be options defined in the `usb_config.h` file to configure the stack as desired. This section discusses several options that are important to (or specific to) the generic demo. See “**USB Firmware Stack Configuration**” for details on all available options.

### IMPORTANT STACK OPTIONS

The following is a list of options that are important to the generic driver:

- `USB_DEV_HIGHEST_EP_NUMBER`
- `USB_DEV_EP0_MAX_PACKET_SIZE`

The options are described below:

`USB_DEV_HIGHEST_EP_NUMBER`:

This option affects how much memory the USB firmware stack allocates for tracking data transfers and DMA purposes. The generic function driver uses one endpoint: endpoint one, as shown in the Example 5, so it defines this macro as 1.

`USB_DEV_EP0_MAX_PACKET_SIZE`:

This macro defines how much buffer space the USB firmware stack allocates for endpoint zero and must be defined as 8, 16, 32, or 64. The generic demo defines it as 8, to reduce RAM usage.

## APPLICATION-SPECIFIC OPTIONS

As discussed in “**Application-Specific USB Support**”, the application must define three routines to provide access to the three application-specific tables required by the USB firmware stack. These routines are identified to the FW stack by three macros:

**FIGURE 3: TABLE ACCESS ROUTINE MACROS**

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

Notice that the function names (on the right) match those of the access routines shown in “**Application-Specific USB Support**”.

The following macros allow the application to define several generic driver-specific options.

- USBGEN\_CONFIG\_NUM
- USBGEN\_INTF\_NUM
- USBGEN\_EP\_NUM
- USBGEN\_EP\_SIZE

The macros are described below:

USBGEN\_CONFIG\_NUM:

This macro identifies the configuration number used for the generic function. Only one configuration is available so this is defined as one.

USBGEN\_INTF\_NUM:

This macro identifies the USB interface number used by the generic driver. It is defined as zero for this demo and can be left at that value unless a more complex implementation requires it to be changed.

USBGEN\_EP\_NUM:

This macro defines the USB endpoint number used for the “interrupt” endpoint to transfer data to and from the host. It is defined as one for this demonstration, but can be changed to eliminate conflicts if the application is modified.

USBGEN\_EP\_SIZE:

This macro defines the maximum packet size for the Interrupt endpoint as well as the minimum size of the buffer that must be available to receive the data. It can be set to 8, 16, 32, or 64 as required by the application.

## CUSTOMIZING THE USB APPLICATION

In addition to demonstrating how to transfer data across the USB, this application is intended to serve as a starting point for USB peripheral device designs using supported Microchip microcontrollers. This section describes how to use the Microchip custom driver to develop a new application.

At a high level, modifying the demo application is a three-step process.

1. Modify the main application.
2. Modify the application-specific USB support.
3. Configure USB stack options.

**Note:** If greater customization is required, the developer can design and implement a new USB function driver. However, doing so is beyond the scope of this document. Please refer to AN1176, “*USB Device Stack for PIC32 Programmer’s Guide*” for details.

## Modifying the Main Application

Using MPLAB IDE, create a new application for the supported microcontroller. (Refer to the MPLAB IDE online help for instructions on how to create a project.) Implement and test any non-USB application-specific support desired. Then, using the provided demo application as an example, add the required USB support. Alternately, copy the demo application, create a new project and application files as desired, and add any non-USB code required.

**Note:** You should add the following include-file search paths to your project's build options.

```
.\
..\Microchip\Include
..\..\Microchip\Include
```

Then, from within any file needing to access the USB and/or Generic Driver API routines, add the following include statements.

```
#include "usb/usb.h"
#include "usb/generic_device.h"
```

In the main application be sure to call `USBInitialize` to initialize the USB stack before calling any other USB routines. Then, call `USBTasks` in a loop as shown in the example application. It is vital that no code executed within the main polling loop blocks or waits on anything taking longer than a few microseconds. If so, the `USBTasks` interface will not be called quickly enough to service USB events as they occur. If blocking behavior is required (or if interrupt-driven behavior is preferred), the `USBTasks` routine may be linked directly to the USB device's Interrupt Service Routine. (Refer to the MPLAB IDE online help for details on how to use interrupts.)

From within the application, call the `USBGenRxIsBusy`, `USBGenGetRxLength`, and `USBGenRead` API routines as necessary to read data from the host. Call `USBGenTxIsBusy` and `USBGenWrite` as necessary to write data to the host. The usage of these routines is demonstrated below and in the demo application.

### EXAMPLE 9: READING DATA FROM HOST

```
if (!USBGenRxIsBusy())
{
    // Get the previous read's size
    PrevSize = USBGenGetRxLength();

    // Start the next read
    USBGenRead(&gData, sizeof(gData));
}
```

If `USBGenRxIsBusy` returns `FALSE`, `PrevSize` will be assigned the actual number of bytes received since the last call to `USBGenRead` (if any) and a new read (Rx) transfer will be started. After `USBGenRead` has been called, `USBGenRxIsBusy` will return `TRUE` until the transfer has completed.

#### EXAMPLE 10: WRITING DATA TO HOST

```
if (!USBGenTxIsBusy())
{
    USBGenWrite(&gData[, sizeof(data));
}
```

If `USBGenTxIsBusy` returns `FALSE`, this logic will start a write (Tx) transfer. After calling `USBGenWrite`, `USBGenTxIsBusy` will return `TRUE` until all of the data has been transferred to the host. If necessary, the program will need to keep track of the transfer size on its own as there is no Tx equivalent to `USBGenGetRxLength`.

**Note:** USB data transfer direction terminology can be a bit confusing. In USB terms, data transfer is always relative to the host. So, an `IN` transfer means data flows from the device to the host and an `OUT` transfer means data flows from the host to the device. This document and the PIC32 firmware generally refer to data transfer relative to the PIC32. So, data is transmitted (Tx) by the PIC32 to the host and received (Rx) from the host to the PIC32.

The following table summarizes.

USB Term	Firmware Term	Description
IN	Transmit	Data flows from the device to the host.
OUT	Receive	Data flows from the host to the device.

## Modifying the Application-Specific USB Support

Very little modification of the application-specific USB support is necessary to implement a new application that uses the Microchip generic driver. The most important changes are related to the descriptor table. Unless additional USB-function behavior is added, no additional descriptors should be needed.

### MODIFYING THE USB DESCRIPTOR TABLE

As discussed in “**Managing the USB**”, every USB device has one device descriptor, one or more sets of descriptors describing possible configurations, and a number of string descriptors. The data types used for these descriptors are defined in the `usb_ch9.h` header file.

This section discusses changes that need to be made to these descriptors when modifying the application.

## Modifying the Device Descriptor

The device descriptor provides information that applies to the overall device. This includes the device class, vendor and product ID numbers, the number of configurations, and endpoint zero information.

The device descriptor is created for the USB firmware stack using the following data type (as defined in `usb_ch9.h`).

**FIGURE 4: DEVICE DESCRIPTOR DATA STRUCTURE**

```
typedef struct
{
    BYTE bLength;
    BYTE bDescriptorType;
    WORD bcdUSB;
    BYTE bDeviceClass;
    BYTE bDeviceSubClass;
    BYTE bDeviceProtocol;
    BYTE bMaxPacketSize0;
    WORD idVendor;
    WORD idProduct;
    WORD bcdDevice;
    BYTE iManufacturer;
    BYTE iProduct;
    BYTE iSerialNum;
    BYTE bNumConfigurations;
} USB_DEVICE_DESCRIPTOR;
```

The following key fields may need to be changed when designing a new device:

- `bMaxPacketSize0`
- `idVendor`
- `idProduct`
- `bcdDevice`
- String Indices
  - `iManufacturer`
  - `iProduct`
  - `iSerialNum`

All of the other fields should remain the same, unless very major changes are being made to the application (such as adding additional configurations).

The following items describe the key fields:

`bMaxPacketSize0`:

If the size of the endpoint zero buffer is changed (by changing the value of the `USB_DEV_EP0_MAX_PACKET_SIZE` macro), `bMaxPacketSize0` must be changed, as well.

**Note:** In the example code, this field is initialized using the `USB_DEV_EP0_MAX_PACKET_SIZE` macro. So, if the example code is used, this field will automatically change when the macro is changed.

`idVendor`:

The Vendor ID (VID) value must be changed to match the ID code allocated to your company by the USB Implementor's Forum (USB IF). If you do not have a VID allocated by the USB IF, contact your Microchip representative about the possibility of using the Microchip vendor ID (0x04D8) and leasing an unused Microchip product ID.

`idProduct`:

The Product ID (PID) value must be changed to match the PID allocated to the product being developed. Each vendor is responsible for allocating and tracking PIDs for products it produces. If you have leased a PID from Microchip, this value must be placed here and the VID must match the Microchip ID.

`bcdDevice`:

This value is a Binary Coded Decimal (BCD) representation of the product revision number. It should be changed to match the revision of the product design.

String Indices:

`iManufacturer`, `iProduct`, and `iSerialNum` contain indices into the string descriptor table to string descriptors that describe the manufacturer, product, and serial number in Unicode strings. Those string descriptors will need to be changed to provide appropriate descriptions for the product, but the index numbers placed in the device descriptor do not need to change unless the positions of these descriptors in the table are changed.

## Modifying the Configuration Descriptor

The sample code implements a single configuration. Thus, there is only one set of configuration-specific descriptors, beginning with a single configuration descriptor.

The configuration descriptor is defined using the following data type:

**FIGURE 5: CONFIGURATION DESCRIPTOR DATA STRUCTURE**

```
typedef struct
{
    BYTE    bLength;
    BYTE    bDescriptorType;
    WORD    wTotalLength;
    BYTE    bNumInterfaces;
    BYTE    bConfigurationValue;
    BYTE    iConfiguration;
    struct
    {
        BYTE    reserved_zero: 5;
        BYTE    remote_wakeup: 1;
        BYTE    self_powered: 1;
        BYTE    reserved_one: 1;
    }bmAttributes;
    BYTE    bMaxPower;
} USB_CONFIGURATION_DESCRIPTOR;
```

bMaxPower is the only field that is likely to need changing in the configuration descriptor.

bMaxPower:

This field indicates the amount of current required for the device to operate in this configuration. The value placed in the descriptor is one-half of the desired current. So a value of 50 represents a maximum draw of 100 mA for this configuration of the device to operate properly. Each increment in this value indicates in increments of 2 mA in the maximum current draw.

**Notes:** A USB device may request a maximum of 500 mA from the bus, but low power hosts (or hubs) may only be able to supply a maximum of 100 mA from the bus.

USB "On The Go" or embedded hosts may be able to supply as little as 8 mA. If your device is intended to operate with such a host, be sure that it only draws the amount of current supported by that host.

## Modifying the Interface Descriptor

The interface descriptor provides a number identifying the interface, the class information for the interface, and the number of endpoints required for the interface.

The interface descriptor is defined using the following data type:

**FIGURE 6: INTERFACE DESCRIPTOR DATA STRUCTURE**

```
typedef struct
{
    BYTE    bLength;
    BYTE    bDescriptorType;
    BYTE    bInterfaceNumber;
    BYTE    bAlternateSetting;
    BYTE    bNumEndpoints;
    BYTE    bInterfaceClass;
    BYTE    bInterfaceSubClass;
    BYTE    bInterfaceProtocol;
    BYTE    iInterface;
} USB_INTERFACE_DESCRIPTOR;
```

Normally, there will be no need to change any fields in the interface descriptor. However, the following two fields may be of interest.

bInterfaceNumber

No two USB interfaces in a single device configuration may have the same interface number unless an alternate interface setting is used (which the custom driver does not). However, if additional USB functionality is integrated with this application, you may need to change the interface by changing this value to allow the host to uniquely identify each interface in the device.

iInterface

No sample string descriptor was provided (or required) for this interface. If you desire to add one, its string-descriptor index will need to be placed in this location.

## Modifying the Endpoint Descriptors

The generic driver uses a single endpoint to transmit and receive data. This requires two endpoint descriptors. The endpoint descriptors identify the type of transfer (the generic driver uses interrupt transfers), the direction and buffer sizes as well as the polling period.

Endpoint descriptors are defined by the following data type:

**FIGURE 7: ENDPOINT DESCRIPTORS DATA STRUCTURE**

```
typedef struct
{
    BYTE bLength;
    BYTE bDescriptorType;
    BYTE bEndpointAddress;
    BYTE bmAttributes;
    WORD wMaxPacketSize;
    BYTE bInterval;
} USB_ENDPOINT_DESCRIPTOR;
```

The values identified in the following items are the ones most likely to be changed. Changing others may cause the endpoint to stop functioning.

### bEndpointAddress:

This value identifies to which endpoint the descriptor refers. This may be changed if there is a conflict with any additional USB functions integrated with this application. This value is initialized using the `USBGEN_EP_NUM` macro. If the endpoint number is changed by changing the value of this macro, then the endpoint descriptors will be changed automatically.

### wMaxPacketSize:

This value identifies the size of the buffer that is associated with the endpoint to the host. This value could be 8, 16, 32, or 64 according to the needs of the application. A smaller buffer would consume less memory space on the device and a larger buffer would provide greater data throughput efficiency.

**Note:** The USB firmware stack does not allocate buffers for the USB endpoints (other than endpoint 0). Instead, it uses the application-defined buffers for all transfers via data endpoints.

## Modifying the String Descriptors

The string descriptor table provides human-readable information in Unicode strings that help the host represent the device to the user. It also provides the device's serial number, represented as a string.

Strings may be supported in many different languages. The first entry in the string descriptor table identifies the list of languages supported. The example only supports English (United States). Additional languages may be supported by adding additional language IDs to the first string descriptor.

**Notes:** Refer to “*Universal Serial Bus (USB) Language Identifiers (LangIDs)*”, on the Internet at [http://www.usb.org/developers/docs/USB\\_LANGIDs.pdf](http://www.usb.org/developers/docs/USB_LANGIDs.pdf) for the list of available language IDs.

When adding additional languages, be sure to increase the size of the first string descriptor (string descriptor zero) by increasing the value of the `NUM_LANGS` macro.

In the example code, string descriptors are provided for the vendor description, product description, and serial number. Each of these should be changed to represent the application being developed.

**Note:** Although it is not required, every device should have a unique serial number. If it doesn't, you may not be able to connect two of the same type of device into the same host. Also, the host may require the user to re-install the driver software every time the device is connected to a different USB port, rather than just once when the device is first connected.



## MODIFYING THE ENDPOINT CONFIGURATION TABLE

The endpoint configuration table identifies direction and protocol features for every endpoint used on the USB device. The table also identifies which function driver will service events that occur for each endpoint. The only exception is that endpoint zero is configured automatically by the USB stack and is not included in the endpoint configuration table.

Each entry in the table consists of the following data structure:

**FIGURE 8: ENDPOINT CONFIGURATION DATA STRUCTURE**

```
typedef struct
_endpoint_configuration_data
{
    UINT16  max_pkt_size;
    UINT16  flags;
    BYTE    config;
    BYTE    ep_num;
    BYTE    intf;
    BYTE    alt_intf;
    BYTE    function;
} EP_CONFIG, *PEP_CONFIG;
```

The EP\_CONFIG structure and flags are defined in the usb\_device.h header file.

max\_pkt\_size:

This field defines how many bytes this endpoint can transfer in a single packet.

flags:

This field provides the information used to configure the behavior of the endpoint. The following flags are defined.

Endpoint Configuration Flags:

USB\_EP\_TRANSMIT

Enable EP for transmitting data

USB\_EP\_RECEIVE

Enable EP for receiving data

USB\_EP\_HANDSHAKE

Non-isochronous endpoints use ACK/NACK

USB\_EP\_NO\_INC

Use for DMA to another device FIFO

config, intf, and alt\_intf:

These fields identify which device configuration, interface and alternate interface setting uses the configuration described in this structure.

ep\_num:

This field identifies which endpoint the structure describes.

function:

This field identifies which function driver uses the endpoint identified in ep\_num. It does this by providing the index into the Supported-Function-Drivers Table.

Normally, the endpoint configuration table will not need to be modified. However, if additional USB functionality is integrated with this application then additional entries will need to be added, as described above.

**Warning:** Some of the information contained in the endpoint configuration table duplicates information defined in the descriptor table. This redundancy is required to eliminate the additional code that would otherwise need to parse the descriptor table to retrieve the information. However, it does place a burden on the programmer to ensure the two tables are coherent.

## MODIFYING THE SUPPORTED FUNCTION DRIVERS TABLE

A USB device may implement more than one class or vendor-specific function. To support this, the Microchip USB firmware stack uses the function driver table to manage access to supported function drivers. Each entry in the table contains the information necessary to manage a single function driver. If an application (like the generic demo) only implements a single USB function, the table will only contain one entry. The following data structure defines an entry in the function driver table.

**FIGURE 9: FUNCTION DRIVER TABLE STRUCTURE**

```
struct _function_driver_table_entry
{
    USBDEV_INIT_FUNCTION_DRIVERInitialize;
    USB_EVENT_HANDLER                 EventHandler;
    BYTE                               flags;
};
```

### Initialize & flags:

The `Initialize` field holds a pointer to the function driver's initialization routine. The initialization routine is called when the host chooses the device configuration appropriate to the function driver identified by the entry given in the table. When called, the initialization routine is passed the `flags` parameter.

### EventHandler:

This field holds a pointer to the function driver's routine for handling class or vendor-specific USB events.

The generic function driver uses the `flags` field to identify which endpoint it should use. (Note: This must be the same endpoint reported in the descriptors.) The other two fields are links that the lower-layer of the USB firmware stack uses to call the generic function driver.

The function driver table may need modification if the application integrates another USB function with the generic driver function.

## Modifying the USB Stack Options

This section highlights several key configuration options necessary to ensure proper operation of the USB device stack. Refer to **Appendix A** for full descriptions of all available configuration options.

### REQUIRED OPTIONS

The following options must be defined as described below.

#### USB\_SUPPORT\_DEVICE:

To ensure that the USB stack is built for peripheral-device mode, be sure this macro is defined (no value required). Otherwise, the behavior of the USB stack will not be appropriate for a USB peripheral device application.

#### USB\_DEV\_EVENT\_HANDLER:

This macro allows the user to replace the "Device" layer of the USB firmware stack (see **Appendix D: "USB Firmware Stack Architecture"**). However, doing so is beyond the scope of this document so the application should ensure that this macro is defined as the name of the device layer's event-handling routine, `USBDEVHandleBusEvent`.

## MODIFYING OPTIONS EFFECTING RAM USAGE

To ensure that the USB stack does not allocate any more RAM than is required, define the following macros carefully.

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

The macros are described below:

`USB_DEV_HIGHEST_EP_NUMBER`:

This macro indicates the highest endpoint number used by the device. In the case of the custom driver, it is defined as one, since Endpoint 1 is the only endpoint used, beyond endpoint zero. This value may be changed to integrate additional USB functionality if needed.

**Note:** Increasing this number will increase the amount of RAM used by the USB stack to allocate additional BDT entries and to track state data.

`USB_DEV_SUPPORTS_ALT_INTERFACES`:

This macro must be defined if the application supports alternate settings for any of its USB interfaces. Since the custom driver does not use alternate interface settings, this should not be changed unless this application is integrated with another application that does.

`USB_DEV_EP0_MAX_PACKET_SIZE`:

Endpoint zero can support buffer sizes of 8, 16, 32, or 64 bytes. The RAM for this buffer is allocated based upon how the `USB_DEV_EP0_MAX_PACKET_SIZE` macro is defined. For the generic driver, this value is defined as 8 bytes. A small decrease in the time necessary to enumerate the device could be obtained by increasing this to one of the larger sizes at the cost of additional RAM dedicated to the endpoint zero buffer.

**Note:** The device layer (see **Appendix D**) allocates buffer space for endpoint 0.

## MODIFYING APPLICATION-SPECIFIC USB SUPPORT OPTIONS

To ensure that the USB stack can call the three application-defined routines to access the descriptor, endpoint configuration, and function driver tables, the following macros must be defined correctly:

- `USB_DEV_GET_DESCRIPTOR_FUNC`
- `USB_DEV_GET_EP_CONFIG_TABLE_FUNC`
- `USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC`

# AN1166

---

The macros are described below:

USB\_DEV\_GET\_DESCRIPTOR\_FUNC:

This macro identifies the name of the application-specific get-descriptor routine to the USB stack. This is the routine that provides the address and size of a requested descriptor.

**EXAMPLE 11: IDENTIFYING THE “GET DESCRIPTION” FUNCTION**

```
#define USB_DEV_GET_DESCRIPTOR_FUNC          USBDEVGetDescriptor
```

If the name of the get-descriptor routine is changed, then the definition of this macro must change to match the new routine name.

USB\_DEV\_GET\_EP\_CONFIG\_TABLE\_FUNC

This macro identifies the name of the application-specific get-endpoint-configuration-table routine to the USB stack. This is the routine that provides the address of the endpoint configuration table as well as the number of entries it contains.

**EXAMPLE 12: IDENTIFYING THE “GET ENDPOINT CONFIGURATION TABLE” FUNCTION**

```
#define USB_DEV_GET_EP_CONFIG_TABLE_FUNC    USBDEVGetEpConfigurationTable
```

If the name of the get-endpoint-configuration-table routine is changed, then the definition of this macro must change to match the new routine name.

USB\_DEV\_GET\_FUNCTION\_DRIVER\_TABLE\_FUNC

This macro identifies the name of the application-specific get-function-driver-table routine to the USB stack. This is the routine that provides the address of the function driver table.

**EXAMPLE 13: IDENTIFYING THE “GET FUNCTION DRIVER TABLE” FUNCTION**

```
#define USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC  USBDEVGetFunctionDriverTable
```

If the name of the get-function-driver-table routine is changed, then the definition of this macro must change to match the new routine name.

Refer to the “**Application-Specific USB Support**” and “**USB Firmware Stack Configuration**” sections for additional information.

## MODIFYING GENERIC FUNCTION OPTIONS

The generic function driver has several options that affect how it uses resources. These options may be changed depending on the needs of the intended application.

- USBGEN\_CONFIG\_NUM
- USBGEN\_INTF\_NUM
- USBGEN\_EP\_NUM
- USBGEN\_EP\_SIZE

The macros are described below:

### USBGEN\_CONFIG\_NUM:

This macro defines the configuration ID value of the generic function. Since the generic function only supports a single configuration, this value should not need to be changed unless the application is integrated with additional USB functionality with multiple configurations.

### USBGEN\_INTF\_NUM:

This macro defines the interface ID value of the custom/generic driver interface. Its value should not change unless additional USB functionality is integrated with the application.

### USBGEN\_EP\_NUM:

This macro defines the number of the endpoint used to send and receive data to and from the host. Its value should not need to be changed, unless this application is integrated with another USB function and there are conflicts in the endpoints used.

### USBGEN\_EP\_SIZE:

This macro defines the size (in bytes) of the endpoint buffer used for data transfer. Since it is defined by the application it may be changed to any legal size for interrupt endpoints: 8, 16, 32, or 64.

## MISCELLANEOUS OPTIONS

There are two additional options that may need to be changed, depending on the application.

### USB\_DEV\_SELF\_POWERED:

Defining this macro informs the USB stack that the device is self powered. If the device is intended to be bus powered, this macro should not be defined.

### USB\_DEV\_SUPPORT\_REMOTE\_WAKEUP:

Defining this macro informs the USB stack that the device supports remotely waking up the host. If it does not, this macro should not be defined.

## CONCLUSION

This document and the associated demo application consisting of both PC software and PIC32 firmware have demonstrated a simple “generic” method of communicating between a host PC and a USB peripheral device.

Normally, managing the Universal Serial Bus requires that a developer handle complex protocols for device identification, control, and data transfer. However, Microchip has taken care of the USB details and provided a simple generic function driver to make implementing applications simple for developers who use supported Microchip microcontrollers.

## REFERENCES

- Microchip Application Note AN1176, “*USB Device Stack for PIC32 Programmer’s Guide*”  
[www.microchip.com](http://www.microchip.com)
- Microchip MPLAB® IDE  
In-circuit development environment, available free of charge, by license, from [www.microchip.com/mplabide](http://www.microchip.com/mplabide)
- “*Universal Serial Bus Specification, Revision 2.0*”  
<http://www.usb.org/developers/docs>
- “*OTG Supplement, Revision 1.3*”  
<http://www.usb.org/developers/onthego>
- “*Universal Serial Bus (USB) Language Identifiers (LangIDs)*”  
[http://www.usb.org/developers/docs/usb\\_langids.pdf](http://www.usb.org/developers/docs/usb_langids.pdf)

## APPENDIX A: USB FIRMWARE STACK CONFIGURATION

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

The following configuration options can be used to customize the stack:

- `USB_SUPPORT_DEVICE`
- `USB_DEV_EVENT_HANDLER`
- `USB_DEV_HIGHEST_EP_NUMBER`
- `USB_DEV_EP0_MAX_PACKET_SIZE`
- `USB_DEV_SUPPORTS_ALT_INTERFACES`
- `USB_DEV_GET_DESCRIPTOR_FUNC`
- `USB_DEV_GET_EP_CONFIG_TABLE_FUNC`
- `USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC`
- `USB_DEV_SELF_POWERED`
- `USB_DEV_SUPPORT_REMOTE_WAKEUP`
- `USB_SAFE_MODE`
- `USBGEN_CONFIG_NUM`
- `USBGEN_INTF_NUM`
- `USBGEN_EP_NUM`
- `USBGEN_EP_SIZE`

# AN1166

---

## USB\_SUPPORT\_DEVICE

- Purpose:** This macro determines that the application being implemented is for a USB peripheral device.
- Precondition:** None
- Valid Values:** This macro does not need to have a value assigned to it. Defining it is sufficient to select the USB role of the application.
- Default:** Not defined
- Example:** `#define USB_DEVICE_ONLY`

## USB\_DEV\_EVENT\_HANDLER

- Purpose:** This macro defines the name of the bus-event-handling routine for the peripheral device support layer of the USB peripheral FW stack. The device support layer handles all standard requests (see Chapter 9 of the “*Universal Serial Bus Specification, Revision 2.0*”) requests. The macro should always be defined as shown in the example unless the user wishes to handle standard device requests directly.
- Precondition:** None
- Valid Values:** This macro needs to be equal to the name of a routine capable of handling all USB device requests.
- Default:** `USBDevHandleBusEvent`
- Example:** `#define USB_DEV_EVENT_HANDLER USBDEVHandleBusEvent`

## USB\_DEV\_HIGHEST\_EP\_NUMBER

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

**Note:** The USB device SW stack will use additional RAM on a per-endpoint basis to manage data transfer.

- Precondition:** None
- Valid Values:** Valid values are any integer between 1 and 15, inclusive.
- Default:** 15
- Example:** `#define USB_DEV_HIGHEST_EP_NUMBER 15`

## USB\_DEV\_EP0\_MAX\_PACKET\_SIZE

- Purpose:** This macro defines the maximum packet size allowed for endpoint 0.

**Note:** The USB device SW stack will use additional RAM equal to the value of this macro.

- Precondition:** None
- Valid Values:** This macro must be defined as 8, 16, 32, or 64.
- Default:** 8
- Example:** `#define USB_DEV_EP0_MAX_PACKET_SIZE 8`



## USB\_DEV\_SUPPORTS\_ALT\_INTERFACES

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

<b>Note:</b> The USB device SW stack will use additional RAM and Flash to manage alternate interfaces when this macro is defined.
---

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_SUPPORTS_ALT_INTERFACES`

## USB\_DEV\_GET\_DESCRIPTOR\_FUNC

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

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_GET_DESCRIPTOR_FUNC USBDEVGetDescriptor`

## USB\_DEV\_GET\_EP\_CONFIG\_TABLE\_FUNC

**Purpose:** This macro defines the name of the routine that provides a pointer to the endpoint configuration table used to configure endpoints as desired.

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_GET_EP_CONFIG_TABLE_FUNC \`  
`USBDEVGetEpConfigurationTable`

## USB\_DEV\_GET\_FUNCTION\_DRIVER\_TABLE\_FUNC

**Purpose:** This macro defines the name of the routine that provides the pointer to the function driver table.

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_GET_FUNCTION_DRIVER_TABLE_FUNC \`  
`USBDEVGetFunctionDriverTable`

# AN1166

---

## USB\_DEV\_SELF\_POWERED

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

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

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_SELF_POWERED`

## USB\_DEV\_SUPPORT\_REMOTE\_WAKEUP

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

**Precondition:** None

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

**Default:** Not defined

**Example:** `#define USB_DEV_SUPPORT_REMOTE_WAKEUP`

## USB\_SAFE\_MODE

**Purpose:** Define this macro to enable parameter and bounds checking throughout the USB SW stack.

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

**Precondition:** None

**Valid Values:** This macro does not need to have a value assigned to it; defining it is sufficient to enable safe mode.

**Default:** Not defined

**Example:** `#define USB_SAFE_MODE`

## USBGEN\_CONFIG\_NUM

**Purpose:** This macro defines the configuration ID number for the generic driver. By default, the driver only supports a single configuration.

**Precondition:** None

**Valid Values:** Device configuration numbers must begin at '1'.

**Default:** None – must be defined by the application.

**Example:** `#define USBGEN_CONFIG_NUM 1`

## USBGEN\_INTF\_NUM

**Purpose:** This macro defines the USB Interface ID number for the generic driver's communication management interface.

**Precondition:** None

**Valid Values:** Interface ID numbers must begin at '0' and must not conflict with any other active interface in the same configuration.

**Default:** None – must be defined by the application.

**Example:** `#define USBGEN_INTF_NUM 0`

## USBGEN\_EP\_NUM

**Purpose:** This macro defines the USB endpoint number for the generic driver's interrupt endpoint used for data transfer to and from the host.

**Precondition:** None

**Valid Values:** Endpoint numbers must be between 1 (Endpoint 0 is dedicated) and 15, inclusive, and must not be used more than once in each direction.

<p><b>Note:</b> The USB Firmware stack allocates memory for tracking every endpoint, starting from '0' and ending at the highest endpoint used (see "USB_DEV_HIGHEST_EP_NUMBER:"). Allocating unused endpoints in this range will cause unused memory to be allocated.</p>
--

**Default:** None – must be defined by the application.

**Example:** `#define USBGEN_EP_NUM 1`

## USBGEN\_EP\_SIZE

**Purpose:** This macro defines the maximum packet size allowed for the custom driver's interrupt endpoint. It is also used by the demo application to define the buffer size.

**Precondition:** None

**Valid Values:** This macro must be defined as 8, 16, 32, or 64.

**Default:** None – must be defined by the application.

**Example:** `#define USBGEN_EP_SIZE 64`

## APPENDIX B: USB GENERIC FUNCTION API

This section describes the generic function driver API. The API provides a means for the application to transfer data on the USB as if it were a “generic” device with read/write capability. USB details are hidden from the application.

Table B-1 summarizes the generic function driver API

**TABLE B-1: USB GENERIC FUNCTION API SUMMARY**

Operation	Description
USBGenRxIsBusy	Checks to see if the system is currently busy receiving data over the USB from the host
USBGenTxIsBusy	Checks to see if the system is currently busy transmitting data over the USB to the host
USBGenGetRxLength	Provides the number of bytes received from the most recent transfer from the host
USBGenWrite	Prepares to send data to the host over the USB
USBGenRead	Prepares to receive data from the host over the USB

Detailed descriptions of the API routines are presented on the following pages.

## USB Generic-Function API - USBGenRxIsBusy

This routine determines if the USB interface is currently busy receiving data from the host.

### Syntax

```
BOOL USBGenRxIsBusy ( void )
```

### Parameters

None

### Return Value

TRUE if the USB interface is currently busy receiving data

FALSE if it is available to receive new data

### Precondition

USBInitialize must have been called and returned a success indication and the device must have been successfully enumerated by the host as a Microchip generic USB device.

### Side Effects

None

### Example

```
if (!USBGenRxIsBusy())  
{  
    USBGenRead(&buffer, sizeof(buffer));  
}
```

# AN1166

---

## USB Generic-Function API - USBGenTxIsBusy

This routine determines if the USB interface is currently busy transmitting data to the host.

### Syntax

```
BOOL USBGenTxIsBusy ( void )
```

### Parameters

None

### Return Value

TRUE if the USB interface is currently busy transmitting data

FALSE if it is available to send new data

### Precondition

USBInitialize must have been called and returned a success indication and the device must have been successfully enumerated by the host as a Microchip generic USB device.

### Side Effects

None

### Example

```
if (!USBGenTxIsBusy())
{
    USBGenWrite(&gData[gTail], size);
}
```

## USB Generic-Function API - USBGenGetRxLength

This routine identifies how much data has been received from the host.

### Syntax

```
BYTE USBGenGetRxLength ( void )
```

### Parameters

None

### Return Value

Returns the number of bytes copied to the caller's buffer by the most recent call to `USBGenRead`

### Precondition

`USBInitialize` must have been called and returned a success indication and the device must have been successfully enumerated by the host as a Microchip generic USB device.

### Side Effects

None

### Example

```
USBGenRead(&buffer, sizeof(buffer));  
while (!USBGenRxIsBusy())  
    ; // Wait for read to finish  
count = USBGenGetRxLength(); // Get the actual number of bytes read.
```

# AN1166

---

## USB Generic-Function API - USBGenWrite

This routine starts a new transmission of data to the host over the USB.

### Syntax

```
void USBGenWrite ( BYTE *buffer, BYTE len )
```

### Parameters

*buffer* – Pointer to the starting location of the data bytes

*len* – Length of the caller's buffer in bytes

### Return Value

None

### Precondition

USBInitialize must have been called and returned a success indication and the device must have been successfully enumerated by the host as a Microchip generic USB device. Also, USBGenTxIsBusy() must return FALSE before this routine is called or unexpected behavior may result.

### Side Effects

A transmission onto the USB of the given size and data has been started.

### Example

```
if (!USBGenTxIsBusy())
{
    USBGenWrite(&buffer, sizeof(buffer));
}
```



## USB Generic-Function API - USBGenRead

This routine prepares to receive data over the USB from the host into the caller's buffer.

### Syntax

```
BYTE USBGenRead ( BYTE *buffer, BYTE len )
```

### Parameters

*buffer* – Pointer to the starting location of the buffer to receive the data

*len* – Length of the caller's buffer in bytes

### Return Value

The number of bytes that have been currently received from the host.

**Note:** This will always be zero when the Rx transfer has just been started. After that, if a transfer is in progress, calling this routine will return the number of bytes currently available in the caller's buffer.

### Precondition

USBInitialize must have been called and returned a success indication and the device must have been successfully enumerated by the host as a Microchip generic USB device.

### Side Effects

The USB interface has been prepared to receive data from the host into the caller's buffer. (See the note under "Return Value", above.)

### Example

```
if (!USBGenRxIsBusy())
{
    USBGenRead(&buffer, sizeof(buffer));
}
```

## APPENDIX C: USB GENERIC FUNCTION DRIVER INTERFACE

This section describes the routines that make up the interface between the generic function driver and the lower-level USB device firmware stack. This interface consists of two routines, one to initialize the function driver and the other to handle generic-driver-specific events.

Neither of these two routines should ever be called directly by the application. They are called by the lower-level USB firmware stack at the appropriate time. Pointers to these routines are placed in the function driver table (see “**Application-Specific USB Support**”) to identify them to the lower-level USB stack. This mechanism allows support for multi-function devices.

**TABLE C-1: USB GENERIC FUNCTION DRIVER INTERFACE SUMMARY**

Operation	Description
USBGenInitialize	Initializes the generic driver
USBGenEventHandler	Identifies and handles bus events

## USB Generic Function Driver Interface - USBGenInitialize

This routine is called by the lower-level USB firmware stack. It is called when the system has been configured as a Microchip custom device by the host. Its purpose is to initialize and activate the generic function driver.

### Syntax

```
BOOL USBGenInitialize ( unsigned long flags )
```

### Parameters

`flags` – Initialization Flags, bits 3-0 identify the endpoint used. All other bits are reserved and should be given as zero.

### Return Value

TRUE if successful

FALSE if not

### Precondition

None

### Side Effects

The Microchip generic function driver has been initialized and is ready to handle events.

### Example

```
const FUNC_DRV gDevFuncTable[] =  
{  
    { // Generic Function Driver  
      USBGenInitialize, // Init routine  
      USBGenEventHandler, // Event routine  
      USBGEN_EP_NUM // Endpoint Number (bottom 4 bits)  
    }  
};
```

# AN1166

---

## USB Generic Function Driver Interface - USBGenEventHandler

This routine is called by the lower-level USB firmware stack to notify the generic function driver of events that occur on the USB. Its purpose is to handle these events as necessary to support the generic driver API.

### Syntax

```
BOOL USBGenEventHandler ( USB_EVENT event, void *data, unsigned int size )
```

### Parameters

event – Event ID

data – Pointer to event-specific data

size – Size (in bytes) of the event-specific data, if any

<b>Note:</b> Events are defined by the lower-level USB firmware stack and handled by the generic function driver.
---

### Return Value

TRUE if the event was handled

FALSE if not (or if additional processing is required)

### Precondition

The system has been enumerated as a Microchip generic device on the USB and the generic driver has been initialized.

### Side Effects

The side effects vary greatly depending on the event. In general, the event has been handled appropriately.

### Example

<b>Note:</b> Refer to AN1176, “USB Device Stack for PIC32 Programmer’s Guide” for a complete list of possible USB device-layer events and descriptions of their associated data.
--

```
const FUNC_DRV gDevFuncTable[] =
{
    { // Generic Function Driver
      USBGenInitialize,      // Init routine
      USBGenEventHandler,   // Event routine
      USBGEN_EP_NUM        // Endpoint Number (bottom 4 bits)
    }
};
```

## **APPENDIX D: USB FIRMWARE STACK ARCHITECTURE**

For a description of the PIC32 USB Device Firmware Stack's architecture, refer to AN1176, "*USB Device Stack for PIC32 Programmer's Guide*".

# AN1166

## APPENDIX E: USB DESCRIPTOR TABLE

The generic demo application defines its descriptor table, as shown in the “**Application-Specific USB Support**” and the “**Customizing the USB Application**” sections, with the values shown in the following tables:

**TABLE E-1: DEVICE DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	12
bDescriptorType	USB_DESCRIPTOR_TYPE_DEVICE	1	01
bcdUSB	USB spec version, in BCD	2	0200
bDeviceClass	Device class code	1	00
bDeviceSubClass	Device sub-class code	1	00
bDeviceProtocol	Device protocol	1	00
bMaxPacketSize0	EP0, max packet size	1	08
idVendor	Vendor ID (VID)	2	04d8
idProduct	Product ID (PID)	2	000C
bcdDevice	Device release number, in BCD	2	0000
iManufacturer	Manufacturer name string index	1	01
iProduct	Product description string index	1	02
iSerialNum	Product serial number string index	1	03
bNumConfigurations	Number of supported configurations	1	01

**TABLE E-2: CONFIGURATION DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	09
bDescriptorType	USB_DESCRIPTOR_TYPE_CONFIGURATION	1	02
wTotalLength	Total size of all descriptors in this configuration	2	0020
bNumInterfaces	Number of interfaces in this configuration	1	01
bConfigurationValue	ID value of this configuration	1	01
iConfiguration	Index of string descriptor describing this configuration	1	00
bmAttributes	Bitmap of attributes of this configuration	1	80
bMaxPower	1/2 Maximum current (in mA)	1	32

**TABLE E-3: INTERFACE DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	09
bDescriptorType	USB_DESCRIPTOR_TYPE_INTERFACE	1	04
bInterfaceNumber	Interface ID number	1	00
bAlternateSetting	ID number of alternate interface setting	1	00
bNumEndpoints	Number of endpoints in this interface	1	02
bInterfaceClass	USB interface class ID	1	00
bInterfaceSubClass	USB interface sub-class ID	1	00
bInterfaceProtocol	USB interface protocol ID	1	00
iInterface	Interface description string index	1	00

**TABLE E-4: DATA (OUT) ENDPOINT DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	07
bDescriptorType	USB_DESCRIPTOR_TYPE_INTERFACE	1	05
bEndpointAddress	Address and direction of the endpoint	1	01
bmAttributes	Interrupt transfer endpoint	1	03
wMaxPacketSize	Largest packet this EP can handle	2	0040
bInterval	Polling period (in mS)	1	20

**TABLE E-5: INTERRUPT (IN) ENDPOINT DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	07
bDescriptorType	USB_DESCRIPTOR_TYPE_ENDPOINT	1	05
bEndpointAddress	Address and direction of the endpoint	1	81
bmAttributes	Interrupt transfer endpoint	1	03
wMaxPacketSize	Largest packet this EP can handle	2	0040
bInterval	Polling period (in mS)	1	20

**TABLE E-6: LANGUAGE ID STRING (0) DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	04
bDescriptorType	USB_DESCRIPTOR_TYPE_STRING	1	03
wLangID	Language ID code	2	0409

**TABLE E-7: VENDOR DESCRIPTION STRING (1) DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex/String)
bLength	Size of this descriptor	1	34
bDescriptorType	USB_DESCRIPTOR_TYPE_STRING	1	03
bString	Serial number string	50	Microchip Technology Inc.

**TABLE E-8: DEVICE DESCRIPTION STRING (2) DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex/String)
bLength	Size of this descriptor	1	34
bDescriptorType	USB_DESCRIPTOR_TYPE_STRING	1	03
wLangID	Language ID code	54	PIC32 PICDEM Demo Emulation

**TABLE E-9: SERIAL NUMBER STRING (3) DESCRIPTOR**

Field	Description	Size (Bytes)	Value (Hex/String)
bLength	Size of this descriptor	1	16
bDescriptorType	USB_DESCRIPTOR_TYPE_STRING	1	03
bString	Serial number string	20	0000000000

## APPENDIX F: GET DESCRIPTOR ROUTINE

The following “get descriptor” routine (and helpers) provides access to the descriptors (which are application specific) to the lower-level USB stack.

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

        default:
            return NULL;
    }
} // GetConfigurationDescriptor

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

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

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

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

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

        default:
            return NULL;
    }
} // GetStringDescriptor

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

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

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



```
// Fail all un-supported descriptor requests:  
  
default:  
    return NULL;  
}  
  
} // USBDEVGetDescriptor
```

The helper routines are “inline” functions used to make the code more readable without incurring the overhead of a function call.

USBDEVGetDescriptor is identified to the USB firmware stack by the `USB_DEV_GET_DESCRIPTOR_FUNC` macro (see **Appendix A: “USB Firmware Stack Configuration”**).

## ***Software License Agreement***

The software supplied herewith by Microchip Technology Incorporated (the "Company") is intended and supplied to you, the Company's customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

## **APPENDIX G: SOURCE CODE FOR THE USB GENERIC FUNCTION ON AN EMBEDDED DEVICE**

The complete source code for the Microchip USB custom driver 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](http://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.

# AN1166

---

NOTES:

---

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

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

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

---

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

**Trademarks**

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC<sup>32</sup> logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2008, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

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

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



---

---

## WORLDWIDE SALES AND SERVICE

---

---

### AMERICAS

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