AN1164

USB CDC Class on an Embedded Device

| Author: | Bud Caldwell |
| | Microchip Technology Inc. |

INTRODUCTION

The Universal Serial Bus (USB) has made it very simple for end users to attach peripheral devices to a personal computer, all but eliminating the myriad of different interconnects that used to be necessary. One such interconnect that is becoming increasingly more rare is the RS-232 serial COM port. In fact, many modern laptop computers no longer have one. This can pose a challenge, for the developer needs a serial communication channel from a peripheral to a host PC.

Fortunately, a device can use the USB Communication Device Class (CDC) and allow the user to take advantage of the simplicity of the USB while providing the functionality of a COM port. The CDC is how communication devices interface to the USB. Intended for devices such as MODEMS and network interfaces, a subset of the CDC features can be used to emulate a serial port providing a "virtual" UART.

The overall flexibility and power of the USB requires managing protocols for device identification, configuration, control and data transfer. The Microchip PIC32 CDC serial driver handles the USB so the developer designing a device doesn't have to.

This document describes the Microchip PIC32 USB CDC serial driver and acts as a programmer's guide for developers wishing to adapt it to their own application. The CDC serial driver provides a simple "UART-like" firmware interface for transmitting and receiving data to and from the host, hiding most of the USB details away from the application. The sample code provided is easily customizable, reducing the amount of effort and learning that might otherwise be necessary when adding a USB interface to a device.

If the Microchip USB CDC serial driver does not provide the USB-related functionality required by the application, Microchip provides sample implementations of other frequently requested USB device classes. These sample implementations are built upon the Microchip PIC32 USB device firmware stack (see **Appendix D: "USB Firmware Stack Architecture"**).

If no sample is available that suits the desired application, the designer can develop his or her own vendor or class-specific function driver using the Microchip USB stack (refer to AN1176, *"USB Device Stack for PIC32 Programmer's Guide"*) and still reduce the amount of USB detail that the developer must deal with directly.

ASSUMPTIONS

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

FEATURES

- Supports USB peripheral device applications
- Emulates a serial COM port on personal computers (PCs) that support the CDC Abstract Control Model
- Provides a simple firmware interface for data transfer to/from 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/)
- Handles CDC-specific requests
- Simplifies definition of USB descriptors and configuration information
- Event-driven system (interrupt or polled)

LIMITATIONS

- Uses control and data endpoints
- Uses interrupt transfer protocol for status notifications
- Uses bulk transfer protocol for data transfer
- Theoretical max data throughput: 1,216,000 bytes/second

> **Note:** Max data throughput figure assumes an otherwise quiet bus and that each packet transfers maximum-sized data payload of 64 bytes. (Refer to Section 5.8 of the *"USB 2.0 Specification"* for additional details on bulk transfers.)

AN1164

SYSTEM HARDWARE

This application was developed for the following hardware:

- PIC32 Family Microcontroller PIM (Processor Interface Module), supporting USB
- Microchip Explorer 16 Development Board
- USB PICtail™ Plus Daughter Board

PIC® MCU MEMORY RESOURCE REQUIREMENTS

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

PIC MCU HARDWARE RESOURCE REQUIREMENTS

The USB CDC serial 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)	Senses USB power (does not operate bus-powered)
VUSB (Input)	Power input for the USB D+/D- transceivers
RD6 (Input)	Monitors the state of switch 3 on the Explorer 16 development board

INSTALLING SOURCE FILES

The Microchip PIC32 USB CDC serial driver source is available for download from the Microchip web site (see **Appendix G: "Source Code for the Microchip USB CDC Serial Driver"**). The source code is distributed in a single Windows® installation file. Perform the following steps to complete the installation:

1. Execute the installation file. A Windows installation wizard will guide you through the installation process.
2. Before continuing with the installation, you must accept the software license agreement by clicking **I Accept**.
3. After completion of the installation process, you should see the "Microchip USB CDC Serial" 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 CDC serial device USB stack contains the following source and header files:

TABLE 2:   SOURCE FILES

File	Directory	Description
usb_device.c	Microchip\USB	USB device layer (device abstraction and *"Universal Serial Bus Specification, Revision 2.0, Chapter 9"* protocol handling)
usb_hal.c	Microchip\USB	USB Hardware Abstraction Layer (HAL) interface support
usb_hal_core.c	Microchip\USB	USB controller functions, used by HAL interface support
usb_device_local.h	Microchip\USB	Private definitions for USB device layer
usb_hal_core.h	Microchip\USB	Private definitions for HAL controller core
usb_hal_local.h	Microchip\USB	Private definitions for HAL
usb.h	Microchip\Include\USB	Overall USB header (includes all other USB headers)
usb_ch9.h	Microchip\Include\USB	USB device framework (*"Universal Serial Bus Specification, Revision 2.0, Chapter 9"*) definitions
usb_common.h	Microchip\Include\USB	Common USB stack definitions
usb_device.h	Microchip\Include\USB	USB device layer interface definition
usb_hal.h	Microchip\Include\USB	USB HAL interface definition
usb_device_cdc_serial.h	Microchip\Include\USB	CDC serial function driver API header
usb_func_serial.c	Microchip\USB\ cdc_serial_device_driver	CDC serial function driver implementation
usb_func_serial_local.h	Microchip\USB\ cdc_serial_device_driver	Private definitions for CDC serial function driver
HardwareProfile.h	usb_cdc_serial_device_demo	Hardware configuration parameters
io_cfg.h	usb_cdc_serial_device_demo	Macros supporting use of GPIO bits connected to switches
main.c	usb_cdc_serial_device_demo	Primary application source file
usb_config.h	usb_cdc_serial_device_demo	Application-specific USB configuration options (see**"USB Firmware Stack Configuration"**)
usb_app.c	usb_cdc_serial_device_demo	Application-specific USB support

AN1164

DEMO APPLICATION

The demo application shows how the PIC32 family CDC serial function driver provides a "virtual UART" to the PIC32 firmware application and emulates a serial COM port on the host PC.

The firmware application provides two services:

- Data Echo Service
  This service receives any data sent to it and echoes it back to the host, demonstrating basic two-way data transfer.
- Hello Message Service
  This service sends a text string to the host when SW3 is pressed on the Explorer 16 development board.

To test these services, use any terminal emulator program on the host PC. The installation process will register the PIC32 USB CDC serial demo as a standard serial COM port on the PC when it installs the source files. Once the firmware is programmed into the PIC32 (see programming instructions, below), connect the board to the host's USB and follow the on-screen instructions to install the default (OS-provided) driver. The demo application will show up as a new COM port (number assigned by the OS). Select this COM port from within the terminal emulator application to test the services.

> **Note:** Configure the terminal emulator to append line-feeds to incoming end-of-line characters if it does not advance to the next line when the "Return"/"Enter" key is pressed. Also, ensure that the terminal emulator is NOT configured for "local echo" of characters typed into it.

Once connected to the PIC32, you can type into the terminal emulator and see the text echoed back by the demo application. You can also press switch (S3) to see the hello message generated.

Programming the Demo Application

To program a target with the demo application, you must have access to a REAL ICE in-circuit emulator programmer. 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 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 the file 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.

The demo application contains necessary configuration options required for the Explorer 16 development 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.

1. Select the Program menu option from the MPLAB IDE programmer menu to begin programming the target.
2. After a few seconds, you should see the message "Programming successful". If not, double check your board and your MPLAB REAL ICE connection. Refer to MPLAB IDE online help for further assistance.
3. Remove power from the board and disconnect the MPLAB REAL ICE cable from the target board.
4. Reapply power to the board and make sure that the LCD displays a message identifying the CDC Demo. If not, double check your programming steps and repeat, if necessary.

The Main Application

The application's "main" function must call the USBInitialize API once, before any other USB activity takes place, to initialize the USB firmware stack. Then, it must call the USBTasks API in a "polling" loop (see Figure 1).

FIGURE 1: **MAIN APPLICATION LOGIC**

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

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

    // Manage "Echo Data Service"

    // Manage "Hello Message Service"

}
```

:

> **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 in the usb_app.c file, 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 be used in a polled, cooperative manner as demonstrated. If so, nothing in the main loop should block for more than a few microseconds or events may be lost. Alternatively, this routine may be called from the Interrupt Service Routine (ISR) whenever a USB interrupt occurs. If it is used this way, the entire USB firmware stack (the non-user API portion of the CDC serial driver) operates in an interrupt context (including the application's event-handler callback routine).

AN1164

THE DATA ECHO SERVICE

The Data Echo service provides an example of how to read and write data across the USB as if it was a UART. The code below maintains a simple state machine to manage a global data buffer. Whenever data is sent by the host, it is received into the buffer and echoed back to the host.

> **Note:** The state variable `gState` starts out in the `BUFFER_EMPTY` state.

FIGURE 2: DATA ECHO SERVICE

```
            switch (gState)
            {
            case BUFFER_EMPTY:      // Buffer is empty & available to receive data.

                // If the buffer is free, read any available data.
                if (USBUSARTRxIsReady())
                {
                    gState = RECEIVING_DATA;
                    USBUSARTRx(gBuffer, BUFFER_SIZE);
                }
                break;

            case RECEIVING_DATA:    // App is waiting to receive data from USB.

                // Check to see if we have data yet.
                if ( (gSize = USBUSARTRxGetLength()) > 0 )
                {
                    gState = DATA_AVAILABLE;
                    // Intentional Fall Through!  Avoids unnecessary loop iteration.
                }
                else
                {
                    break;
                }

            case DATA_AVAILABLE:    // Data has been received, app can act on the data.

                // If the transmitter is ready, echo the data back to the host.
                if (USBUSARTTxIsReady())
                {
                    gState = SENDING_DATA;
                    USBUSARTTx(gBuffer, gSize);
                }
                break;

            case SENDING_DATA:      // App is waiting to finish sending data on USB.

                // If we're done transmitting, free up the buffer to receive new data.
                if (USBUSARTTxIsReady()) {
                    gSize  = 0;
                    gState = BUFFER_EMPTY;
                }
                break;

            default:
                while(1);           // Invalid state, hang the application.
            }
```

The first iteration of the state machine will start an "Rx" transfer to receive any available data from the host. To do this it calls the USBUSARTRx API routine, passing the address and size of the data buffer. However, before calling USBUSARTRx, it must call the USBUSARTRxIsReady API routine to ensure that the "virtual" UART is not currently busy receiving other data. It certainly wouldn't be busy on the first iteration of the loop, but it is quite likely that it will be on subsequent iterations. When a new "Rx" transfer is started, state machine transitions from the BUFFER_EMPTY state to the RECEIVING_DATA state.

When in the RECEIVING_DATA state, the application calls the USBUSARTRxGetLength API routine to check to see how much (if any) data has been received. This is important, since host is under no obligation to provide exactly the amount of data requested (although the firmware stack will not allow it to receive any more). It stores this amount into the gSize variable. Once data is received, the state machine transitions to the DATA_AVAILABLE state (falling directly through to the next "case" in the switch statement to avoid a potentially unnecessary iteration of the loop).

> **Note:** Once any quantity of data is reported by the USBUSARTRxGetLength routine, the CDC serial driver will not accept additional data from the host until the USBUSARTRx routine is called again.
>
> Since this simple application is only designed to be "half duplex", either transmitting or receiving (but, not both at the same time), it will not call USBUSARTRx again immediately. A more advanced application could do so using a different buffer, allowing the system to appear to receive data at the same time it was transmitting data, providing "full-duplex" functionality.

Once data is available in the buffer, the state machine calls the USBUSARTTxIsReady API routine to see if the virtual UART is ready to transmit data. If so, it calls the USBUSARTTx routine to start the transmission of the amount identified by the USBUSARTRxGetLength and transitions to the SENDING_DATA state.

In the SENDING_DATA state, the application checks to see if the virtual UART is done transmitting data by calling the USBUSARTTxIsReady routine. Once that routine returns TRUE, the state machine resets the size of the data in the buffer and transitions back to the BUFFER_EMPTY state, starting the process over again. Using this method, the application continuously reads data from the host and "echoes" it back.

THE HELLO MESSAGE SERVICE

The Hello Message service sends a text string to the host whenever switch 3 is pressed on the Explorer 16 development board. This demonstrates the put-string API provided by the Microchip PIC32 USB CDC serial driver.

The following code snippet from the application-specific tasks section of the application's main loop provides this service.

FIGURE 3: HELLO MESSAGE SERVICE

```
// Display message if button is pressed.
if(Switch3IsPressed())
{
    if(USBUSARTTxIsReady())
    {
        USBUSARTPuts(gTestStr);
    }
}
```

The Switch3IsPressed routine returns TRUE when SW3 on the Explorer 16 board is pressed.

> **Note:** No debouncing of SW3 is performed, so it may return TRUE multiple times for a single button press.

SW3 is checked every time through the loop. If it is pressed, the code checks to see if the virtual UART is ready to transmit data by calling the routine. If it returns TRUE, then it is safe to call the USBUSARTPuts routine to PUT (transmit) a text string contained in the gTestStr array through the "virtual" UART to the host.

> **Note:** There is an analogous GET string API routine (USBUSARTGets) to receive a text string, but it is not used in the demo application.

AN1164

Application-Specific USB Support

Since the CDC serial demo uses the Microchip USB peripheral device firmware stack, it defines three application-specific tables, listed below.

Application-specific tables:

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

These three tables and the functions used by the stack to access them, are defined in the `usb_app.c` file.

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 and this document discusses key fields that may need to be changed for different applications (see **"Modifying the Application-Specific USB Support"**).

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 uniquely identifies the 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. (See Figure 4 and refer to **Appendix E: "USB Descriptor Table"** for a complete definition.)

FIGURE 4:   DESCRIPTOR GROUPS
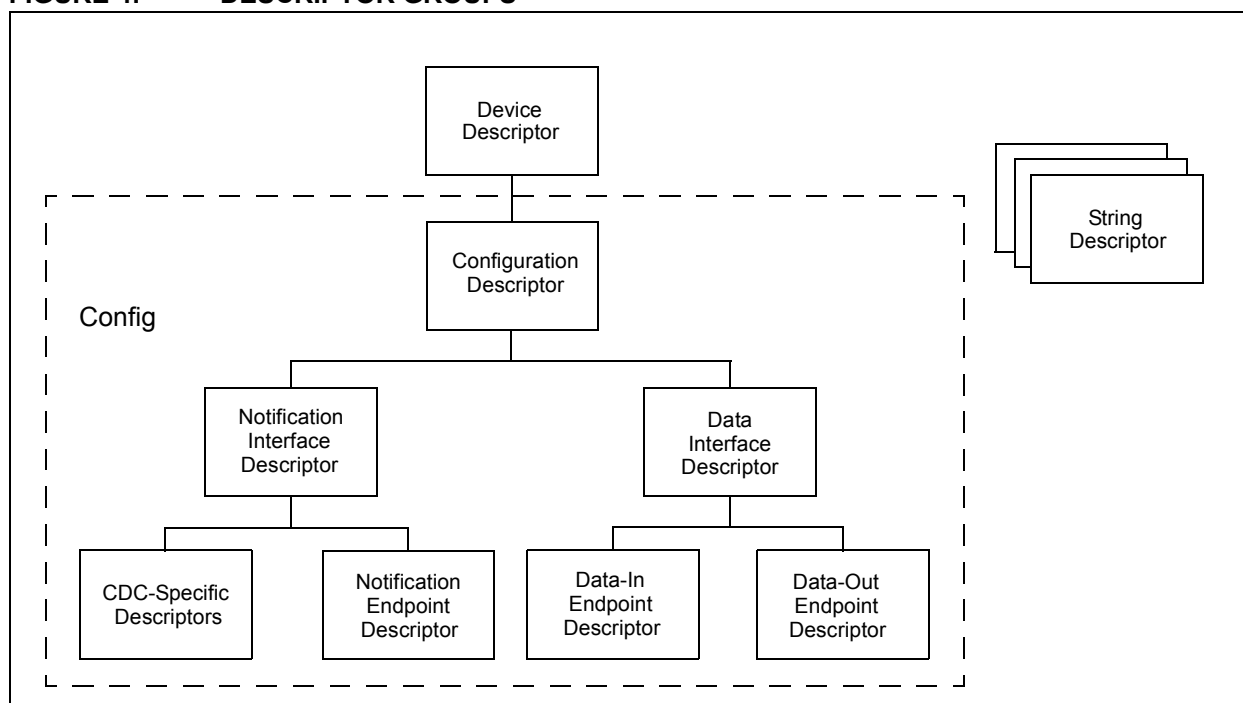
In order for the host to read to these descriptors, the USB firmware stack must have access to them. To provide this access, the application defines a `USBDevGetDescriptor` routine. This routine receives an ID value identifying the descriptor type.

In the configuration and string cases, the ID also contains an index number identifying which instance of the descriptor is being requested, along with a language ID for string descriptors. It then provides the length of the requested descriptor and a pointer to it. (See **Appendix F: "Get Descriptor Routine"**.)

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 CDC serial driver only has one configuration.

The endpoint configuration table (below) identifies which endpoints belong to which interface (for configuration 1) along with the data transfer direction and protocol features for each endpoint.

EXAMPLE 1:   ENDPOINT CONFIGURATION TABLE

```
const EP_CONFIG gEpConfigTable[] =
{
    // EP2 Com Class Notification Endpoint
    {
        CDC_INT_EP_SIZE,        // Maximum packet size for this endpoint
        USB_EP_TRANSMIT|        // Configuration flags for this endpoint (see below)
        USB_EP_HANDSHAKE,
        CDC_CONFIG_NUM,         // Configuration number
        CDC_INT_EP_NUM,         // Endpoint number.
        0,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table (see below)
    },

    // EP3 Data Class Endpoints
    {
        CDC_BULK_OUT_EP_SIZE,   // Maximum packet size for this endpoint
        USB_EP_RECEIVE |        // Configuration flags for this endpoint (see below)
        USB_EP_TRANSMIT|
        USB_EP_HANDSHAKE,
        CDC_CONFIG_NUM,         // Configuration number (start at 1)
        CDC_BULK_EP_NUM,        // Endpoint number.
        1,                      // Interface number
        0,                      // Alternate interface setting (default=0)
        0                       // Index in device function table (see below)
    }
};
```

The endpoint configuration table identifies that the CDC serial driver uses two different interfaces, each with one endpoint. Interface 0 is the device management interface. It provides the host with a mechanism to control the device and to receive notification of events. Interface 1 is the data interface. It provides the data transfer mechanism for the "virtual" UART. Endpoint 2 (`CDC_INT_EP_NUM`) belongs to Interface 0. It is a USB "IN" endpoint, transmitting notifications to the host. Control data is received on Endpoint 0 (the USB control endpoint used for enumeration), making it a shared endpoint with Interface 0. Endpoint 3 (`CDC_BULK_EP_NUM`) belongs to Interface 1. It is used in both directions as an "IN" and "OUT" endpoint. Virtual UART data is transmitted or received through this endpoint. The table also associates both endpoints with the function driver at index zero in the function table (see **"The Function Driver Table"**).

> **Note:** In USB terminology, the device transmits data "IN" to the host and receives data "OUT" of the host.

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

EXAMPLE 2: STACK ACCESS ROUTINE

```
inline 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;

} // USBDEVGetEpConfigurationTable
```

This routine provides a pointer to the endpoint configuration table (as well as the number of entries it contains) to the USB stack. It is identified to the stack by the `USB_DEV_GET_EP_CONFIG_TABLE_FUNC` macro (see **"USB Stack Options"**).

THE FUNCTION DRIVER TABLE

The Microchip peripheral device FW 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 serial demo only implements one USB function, its table only contains one entry as shown below.

EXAMPLE 3:   FUNCTION DRIVER TABLE

```
const FUNC_DRV gDevFuncTable[] =
{
    {
        USBUARTInit,
        USBUARTEventHandler,
        0
    }
};
```

This table provides pointers to the serial function driver's initialization and event-handling routines, as well an initialization value, which is reserved for the CDC driver and is defined as a zero (0). This is all the information that the USB firmware stack needs to manage the CDC serial driver and make sure it is aware of events that occur on the bus.

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

EXAMPLE 4:   STACK ACCESS TO FUNCTION DRIVER TABLE

```
inline const FUNC_DRV *USBDEVGetFunctionDriverTable ( void )
{
    // Index into array and provide interface pointer.
    return gDevFuncTable;

} // USBDEVGetFunctionDriverTable
```

This routine returns the pointer to the base of the "get function driver table" routine. 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.

AN1164

USB Stack Options

The Microchip PIC32 USB device firmware stack supports a number of configuration options. These options are defined by the application in the `usb_config.h` file. This section discusses several options that are important to (or specific to) the CDC serial demo. (Refer to Section 9 of the *"USB Firmware Stack Configuration"* for details on all available options.)

IMPORTANT STACK OPTIONS

USB_DEV_HIGHEST_EP_NUMBER:

> This option affects how much memory the USB firmware stack allocates for tracking data transfers and DMA purposes. The CDC serial function driver uses two endpoints (Endpoints 2 and 3, as shown in the Endpoint Configuration and Descriptor tables, above), so it defines this macro as 3.

USB_DEV_EP0_MAX_PACKET_SIZE:

> This macro defines how much buffer space the USB firmware stack allocates for Endpoint 0 and must be defined as 8, 16, 32, or 64. The CDC 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 (below).

```
#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 match those of the access routines shown in **"Application-Specific USB Support"**.

In addition to the above routines, which provide access to the data required by the USB firmware stack, the CDC serial function driver defines a mechanism for the application to receive notification serial-related USB events (specifically changes to the line control settings and reception of encapsulated command strings). To do this, the application implements a routine that matches the following function signature:

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

To allow the application to name the routine as desired, the following macro identifies it to the USB firmware stack.

```
#define \
CDC_APP_EVENT_HANDLING_FUNC \
DemoEventHandler
```

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

CDC_CONFIG_NUM:

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

CDC_COMM_INTF_ID:

> This macro identifies the USB Interface number of the communication's class interface. 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.

CDC_INT_EP_NUM:

This macro defines the USB endpoint number used for the "Interrupt" endpoint, as part of the Communication's Class definition. It is defined as two for this demonstration, but can be changed to eliminate conflicts if the application is modified.

CDC_INT_EP_SIZE:

This macro defines the size of the Interrupt endpoint's buffer. It only needs to be large enough to send a Communication's Device Class (CDC) notification, which 8-bytes.

CDC_DATA_INTF_ID:

This macro identifies the USB Interface number of the data class interface used to transfer the actual serial data. It is defined as one for this demo and can be left at that value unless it needs to be changed for a different implementation.

CDC_BULK_EP_NUM:

This macro defines the USB endpoint number used for the "bulk" endpoint as part of the data class interface. It is defined as three for this demonstration, but can be changed to eliminate conflicts if necessary.

CDC_BULK_OUT_EP_SIZE:

This macro defines the size of the bulk data endpoint's receive (USB "OUT") buffer. It is set at the maximum size of 64 bytes to provide maximum potential throughput, but it can be changed to any of 8, 16, 32, or 64 bytes if needed.

CDC_BULK_IN_EP_SIZE:

This macro defines the size of the bulk data endpoint's transmit (USB "IN") buffer. It is set at the maximum size of 64 bytes to provide maximum potential throughput, but it can be changed to any of 8, 16, 32, or 64 bytes if needed.

The following macros define the default line coding settings that the CDC serial driver reports in response to the host's GET_LINE_CODING request. Keep in mind that, in a USB environment, communication occurs at speeds defined by the USB protocol and most hosts will set the line coding parameters as desired. None the less, the default values reported can be changed by changing the following macros.

```
// Bit Rate
CDC_DEFAULT_BPS          115200

// 1 stop bit
CDC_DEFAULT_FORMAT       0

// No parity
CDC_DEFAULT_PARITY       0

// 8-bits per word
CDC_DEFAULT_NUM_BITS     8
```

AN1164

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 PIC32 CDC serial function driver to develop a custom application. In general terms, customizing 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 custom 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 on implementing a function driver.

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 (see **"Demo Application"**, above) add the required USB support. Alternately, copy the demo application, rename the project and application files as desired, and add any non-USB code required.

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 then 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 CDC serial driver's virtual UART API routines as necessary to read data from or write data to the host. The usage of these routines is demonstrated by the demo application, discussed in **"The Main Application"** (above), and described in detail in **"USB CDC Serial Function API"** (below).

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 serial function 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 previously discussed, the 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 for the standard descriptor types and in the `cdc_serial_device.h` header file for the CDC specific descriptor types.

> **Note:** Refer to Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* and the *"USB Class Definitions for Communication Devices"* documents for additional details on the descriptors used by this application note.

This section will discuss changes that will 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 (defined in the `usb_ch9.h` header file).

FIGURE 5: **USB DEVICE DESCRIPTOR 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;
```

AN1164

The following are key fields that may need to be changed when designing a new CDC serial device.

`bMaxPacketSize0:`

If the size of the Endpoint zero buffer is changed, this field must be changed as well.

Note:	The example code initializes this field 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:`

The `iManufacturer`, `iProduct`, and `iSerialNum` fields 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.

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

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 6:   CONFIGURATION DESCRIPTOR STRUCTURE

```
typedef struct
{
    BYTE    bLength;
    BYTE    bDescriptorType;
    WORD    wTotalLength;
    BYTE    bNumInterfaces;
    BYTE    bConfigurationValue;
    BYTE    iConfiguration;
    BYTE    bmAttributes;
    BYTE    bMaxPower;

} USB_CONFIGURATION_DESCRIPTOR;
```

There is only one field that is likely to need to be changed 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 increment 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 Communication Management Interface Descriptor

The communication management interface descriptor is a normal USB interface descriptor. It 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 7: INTERFACE DESCRIPTOR 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 two fields discussed below may be of interest.

bInterfaceNumber:

No two USB interfaces may have the same interface number unless an alternate interface setting is used (and the CDC serial driver doesn't). However, if additional USB functionality is integrated with this application, you may need to change the interface number for the CDC communication management 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 index in the string descriptor table will need to be placed in this location.

CDC Class-Specific Descriptors

The CDC-specific descriptors indicate to the host that this device supports the CDC, Abstract Control Model (ACM) with the line control feature and no call management features. The only changes to these descriptors that may be necessary are changes to the values that indicate which interface numbers are used for the CDC function. These changes are only necessary if changes were made to the interface ID numbers for either the communication management interface or the data class interface. Any other changes may cause the host to expect behavior that is not supported by the CDC function driver and may induce errors.

The Union Functional Descriptor and the Call Management Functional Descriptor are defined by the following data types.

FIGURE 8: CDC-SPECIFIC DESCRIPTOR STRUCTURE

```
/* Union Functional Descriptor */
typedef struct _USB_CDC_UNION_FN_DSC
{
    BYTE bFNLength;
    BYTE bDscType;
    BYTE bDscSubType;
    BYTE bMasterIntf;
    BYTE bSlaveIntf0;
} USB_CDC_UNION_FN_DSC;

/* Call Management Functional Descriptor */
typedef struct _USB_CDC_CALL_MGT_FN_DSC
{
    BYTE bFNLength;
    BYTE bDscType;
    BYTE bDscSubType;
    BYTE bmCapabilities;
    BYTE bDataInterface;
} USB_CDC_CALL_MGT_FN_DSC;
```

bMasterIntf:

This number indicates to the host which interface will be used as the master communication interface, primarily used for device notifications. It will need to be changed if the ID of the communication management interface was changed.

`bSlaveIntf0`:

The slave interface is the bulk transfer data interface. If the ID of the data interface was changed then this number will need to be changed as well.

Note:	The communication device class definition allows for devices with multiple data interfaces. Thus, this descriptor can be expanded to include additional slave interface ID values. However, this implementation only uses slave Interface zero (0).

`bDataInterface`:

This value should be the same as the `bSlaveIntf0` value. It is duplicated in the call management functional descriptor to indicate that call management is embedded into the data stream. However, no call management capabilities are identified.

Note:	The communication management and data interfaces are identified by numbers defined by the `CDC_COMM_INTF_ID` and `CDC_DATA_INTF_ID` macros, respectively. If any changes to these interface ID numbers are made by changing the values of these macros, then the changes to the union functional and call management functional descriptors will happen automatically.

Modifying the Notification Endpoint Descriptor

The notification endpoint descriptor identifies the type of transfer supported by the endpoint, its direction, buffer size and polling period. The descriptor may need to be changed if there is some reason to change which endpoint is used. The values identified below are the ones most likely to be changed. Changing others may cause the endpoint to stop functioning as required.

Endpoint descriptors are defined by the following data type.

FIGURE 9: **ENDPOINT DESCRIPTOR STRUCTURE**

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

`bEndpointAddress`:

This value identifies which endpoint is used for notifications to the host. As mentioned above, it may be changed if there is a conflict with any additional USB functions integrated with this application.

Notes:	The `bEndpointAddress` is initialized using the `CDC_INT_EP_NUM` macro. If the endpoint number is changed by changing the value of this macro, then the `bEndpointAddress` value in the notification endpoint descriptor will be changed automatically.
	It is important that the direction and transfer type do not change or the CDC function driver will not work.

`wMaxPacketSize`:

This value indicates to the host what the size of the buffer is that is associated with the notification endpoint. This buffer only needs to be large enough to send a CDC notification (8 bytes). However, if some application has a need to send larger notifications, then this value could be increased to 16, 32, or 64.

`bInterval`:

This value determines the polling interval (in milliseconds, from 1 to 255) for the notification endpoint. As an interrupt transfer endpoint, it is regularly polled by the host for data. This polling frequency could be modified to match the needs of the application.

Modifying the Data Interface Descriptor

The data interface descriptor provides the number identifying the data interface, the class information, and the number of endpoints.

> **Note:** Refer to Figure 7 for the data type used to define this descriptor.

Normally, there will be no need to change any fields in the data interface descriptor. However, if additional USB functionality is being integrated with the application, the following fields may need to be changed.

bInterfaceNumber:

No two USB interfaces may have the same interface number (unless one is an alternate setting of the other). So, if additional USB functionality is integrated with this application, you may need to change the interface number for the data 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 the data interface. If you desire to add one, its index in the string descriptor table will need to be placed this value.

Modifying the Data Endpoint Descriptor

The data endpoint is used in both directions to transmit and receive data. Thus, it has two endpoint descriptors. Like the notification endpoint descriptor, the data endpoint descriptors identify the type of transfer, direction, and buffer size. Unlike the notification endpoint, the data endpoints support the bulk transfer protocol. Thus, there is no polling period (it is specified as zero).

The values identified below are the ones most likely to be changed. Changing others may cause the endpoint to stop functioning as required. Refer to Figure 8 for the data type used to define the data endpoint descriptors.

> **Note:** It is important that the transfer type not be changed or the CDC function driver will not work.

bEndpointAddress:

This value identifies which endpoints are used for data transfer to-and-from the host. This may be changed if there is a conflict with any additional USB functions integrated with this application. The bEndpointAddress values are initialized using the CDC_BULK_EP_NUM macro. If the endpoint numbers are changed by changing the value of this macro, then the bEndpointAddress values in the data endpoint descriptors will be changed automatically.

> **Note:** Since there is a separate descriptor for each of the data endpoints (the "transmit" or "IN" endpoint and the "receive" or "OUT" endpoint), you could potentially specify different endpoint numbers for transmit and receive. However, this would be ill advised as the host's device driver may assume that the same endpoint number was used for "IN" transfers and "OUT" transfers.

wMaxPacketSize:

This value indicates to the host what the sizes of the buffers are that are associated with the data endpoints. These values could be 8, 16, 32, or 64, according to the needs of the application. Smaller buffers would consume less memory space on the device and larger buffers would provide greater data throughput efficiency.

> **Note:** The USB firmware stack does not allocate buffers for the USB endpoints. Instead, it uses the application-defined buffers for all transfers via data endpoints.

AN1164

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, which is 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 ID's to the first descriptor.

> **Note:** Refer to the USB IF Language Identifiers document 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:** Every device should have a unique serial number, or only one such device can be attached to a given host at a time. Also, the host system may require the user to re-install the driver software every time the device is connected to a different USB port; rather then 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 device stack and is not included in the endpoint configuration 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. The `EP_CONFIG` structure and flags are defined in the `usb_device.h` header file. Each entry in the table consists of the following data structure:

FIGURE 10: ENDPOINT CONFIGURATION 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;
```

`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:

- `USB_EP_TRANSMIT` – Enable endpoint for transmitting data

- `USB_EP_RECEIVE` – Enable endpoint for receiving data

- `USB_EP_HANDSHAKE` – Enable generation of handshaking (ACK/NAK) packets (non-isochronous endpoints only)

- `USB_EP_NO_INC` – Used only for direct DMA to another device's FIFO

`ep_num`:

This field identifies which endpoint the structure describes.

config, intf, and alt_intf:

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

function:

This field identifies which function driver uses the endpoint identified in ep_num. It does this by providing the index into the function driver table.

> **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 then one class or vendor-specific function. To support this, the Microchip PIC32 USB FW stack uses the function driver table to manage access to function drivers. Each entry in the table contains the information necessary to manage a single function driver. If a device (like the CDC serial demo) only implements one USB function, the table will only contain one entry.

The only reason to modify the function driver table is if the application is modified to integrate another USB function. Each new driver supported will require an entry of its own in the table. Of course, the CDC function must have an entry as well (if it is used).

The following data structure defines an entry in the function-driver table.

FIGURE 11: FUNCTION DRIVER TABLE ENTRY

```
struct _function_driver_table_entry
{
  USBDEV_INIT_FUNCTION_DRIVER Initialize;
  USB_EVENT_HANDLER      EventHandler;
    BYTE                        flags;
};
typedef struct _function_driver_table_entry
  FUNC_DRV, *PFUNC_DRV;
```

Initialize and 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 (which is ignored by the CDC driver).

EventHandler:

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

> **Note:** For additional details, refer to Microchip Application Note AN1176, *"USB Device Stack for PIC32 Programmer's Guide"*.

AN1164

Modifying the USB Stack Options

This section highlights several key configuration options necessary to ensure proper operation of the USB peripheral device stack. Refer to **Appendix A: "USB Firmware Stack Configuration"** 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 so that it behaves like a USB device, be sure this macro is defined (no value required). Otherwise, the behavior of the USB stack will not be appropriate for a USB 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 USB-DEVHandleBusEvent.

MODIFYING OPTIONS EFFECTING RAM USAGE

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

USB_DEV_HIGHEST_EP_NUMBER:

This macro indicates the highest endpoint number used by the function. In the case of the CDC serial driver, it is defined as three (3), since Endpoint 3 is used for the data interface. This value may be changed to integrate additional USB functionality that required additional endpoints.

Note:	Increasing this number will increase the amount of RAM used by the USB stack to allocate additional entries in the BDT 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 CDC serial driver does not use alternate interface settings, this should not be changed unless this application is integrated with another application that does require alternate interface settings.

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 CDC serial driver, this value is defined as eight (8) bytes. A very 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 USB firmware stack only allocates buffer space for Endpoint 0.

MODIFYING APPLICATION-SPECIFIC USB
SUPPORT OPTIONS

To ensure that the USB stack can call the three user-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`

If the names of any of the routines change, the corresponding macro will need to be updated. The macros are described below:

`USB_DEV_GET_DESCRIPTOR_FUNC:`

> This macro identifies the name of the application-specific "get descriptor" routine (see **"The USB Descriptor Table"**) to the USB stack. This is the routine that provides the address and size of a requested descriptor.

EXAMPLE 5: 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 (see **"The Endpoint Configuration Table"**) 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 6: 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 (see **"The Function Driver Table"**) to the USB stack. This is the routine that provides the address of the function driver table.

EXAMPLE 7: 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.

AN1164

MODIFYING CDC SERIAL FUNCTION OPTIONS

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

- CDC_CONFIG_NUM
- CDC_COMM_INTF_ID
- CDC_INT_EP_NUM
- CDC_INT_EP_SIZE
- CDC_DATA_INTF_ID
- CDC_BULK_EP_NUM
- CDC_BULK_OUT_EP_SIZE
- CDC_BULK_IN_EP_SIZE

The macros are described below:

CDC_CONFIG_NUM:

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

CDC_COMM_INTF_ID:

> This macro defines the interface ID value of the communication management interface. Its value should not change unless additional USB functionality is integrated with the application.

CDC_INT_EP_NUM:

> This macro defines the endpoint number of the notification endpoint used as part of the communication management interface. It could be changed to optimize endpoint and memory usage or if this application is integrated with another USB function and there are conflicts in the endpoints used.

CDC_INT_EP_SIZE:

> This macro defines the size (in bytes) of the endpoint buffer used for CDC notifications. It should not change unless there is a need for larger notification data.

CDC_DATA_INTF_ID:

> This macro defines the ID of the data interface. Its value should not change unless additional USB functionality is integrated with the application.

CDC_BULK_EP_NUM:

> This macro defines the endpoint number used by the bulk data interface. It could be changed to optimize endpoint and memory usage or to resolve endpoint conflicts if this application is integrated with other USB functionality.

CDC_BULK_OUT_EP_SIZE:

> This macro defines the packet size of the USB bulk data "OUT" endpoint. It affects the size of data packets received from host. It is set at the maximum of 64 bytes to maximize data throughput. However, it could be reduced to 8, 16, or 32 in order to reduce RAM requirements for data buffering.

CDC_BULK_IN_EP_SIZE:

> This macro defines the packet size of the USB bulk data "IN" endpoint. It affects the size of data packets sent (transmitted) to the host. It is set at the maximum of 64 bytes to maximize data throughput. However, it could be reduced to 8, 16, or 32 in order to reduce RAM requirements for data buffering.

> **Note:** The USB firmware stack does not allocate the buffer space for any endpoint except Endpoint zero. Instead, it dynamically switches the memory target of the USB transfer to the application's buffers whenever the read or write API routines are called. This means that changing these macros (CDC_BULK_OUT_EP_SIZE and CDC_BULK_IN_EP_SIZE) may not directly affect the amount of RAM used. It will change the packet sizes in which data is transferred to or from the host.

Default Line Coding

The following macros define the default line coding parameters (as defined by the sample code, provided).

```
// Bit Rate
CDC_DEFAULT_BPS          115200

// 1 stop bit
CDC_DEFAULT_FORMAT       0

// No parity
CDC_DEFAULT_PARITY       0

// 8-bits per word
CDC_DEFAULT_NUM_BITS     8
```

These values will be reported to the host when requested. However, the use of these parameters is completely up to the application. They do not effect the communication over USB. Also, the host will most likely send new line coding parameters as part of enumeration.

CDC_APP_EVENT_HANDLING_FUNC

This macro identifies the name of the application's event-handling routine. This routine is called by the CDC serial driver asynchronously when the line coding changes or when encapsulated communications commands are received from the host. If the name of the application's event-handling routine is changed then this macro definition must change to match it.

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.

AN1164

CONCLUSION

This document has shown how to use the Microchip PIC32 CDC serial function driver and USB firmware stack to emulate a UART over the Universal Serial Bus on supported Microchip microcontrollers.

Usually, implementing a USB device would require the development of firmware to handle USB protocols for device identification, control, and data transfer. The CDC serial driver and USB firmware stack has taken care of the details of the USB protocol so that the device developer doesn't have to. The sample code provided only requires minor modifications to adapt it to most applications that would otherwise use a traditional UART. This allows the device designer to provide a solution that lets the end user enjoy the benefits of the USB with minimal effort.

REFERENCES

- Microchip Application Note AN1176, *"USB Device Stack for PIC32 Programmer's Guide"* www.microchip.com
- Microchip MPLAB® IDE In-circuit Development Environment, available free of charge, by license, from 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
- *"Class Definitions for Communication Devices"* http://www.usb.org/developers/devclass_docs

APPENDIX A: USB FIRMWARE STACK CONFIGURATION

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

USB_SUPPORT_DEVICE

USB_DEV_EVENT_HANDLER

USB_DEV_HIGHEST_EP_NUMBER

USB_DEV_EP0_MAX_PACKET_SIZ

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

> **Note:** Refer to AN1176, *"USB Device Stack for PIC32 Programmer's Guide"* for details on the usage of the above configuration options.

CDC_CONFIG_NUM

CDC_COMM_INTF_ID

CDC_INT_EP_NUM

CDC_INT_EP_SIZE

CDC_DATA_INTF_ID

CDC_BULK_EP_NUM

CDC_BULK_OUT_EP_SIZE

CDC_BULK_IN_EP_SIZE

CDC_DEFAULT_BPS

CDC_DEFAULT_FORMAT

CDC_DEFAULT_PARITY

CDC_DEFAULT_NUM_BITS

CDC_APP_EVENT_HANDLING_FUNC

> **Note:** The following options are all defined by the application. Thus, the example for each is also the default as defined by the demo application.

CDC_CONFIG_NUM

Purpose: This macro defines the configuration ID number for the CDC serial driver.

Precondition: None

Valid Values: Device configuration numbers must begin at one (1).

Example: `#define CDC_CONFIG_NUM        1`

CDC_COMM_INTF_ID

Purpose: This macro defines the USB interface ID number for the CDC serial driver's communication management interface.

Precondition: None

Valid Values: Interface ID numbers must begin at zero (0) and must not conflict with any other active interface on the same device.

Example: `#define CDC_COMM_INTF_ID        0`

CDC_INT_EP_NUM

Purpose: This macro defines the USB endpoint number for the CDC serial driver's interrupt endpoint used for asynchronous notifications to the host.

Precondition: None

Valid Values: Endpoint numbers must be between one (Endpoint 0 is dedicated) and 15, inclusive, and must not be used more then once in per direction.

> **Note:** The USB firmware stack allocates memory for every endpoint, starting from zero 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.

Example: `#define CDC_INT_EP_NUM        2`

CDC_INT_EP_SIZE

Purpose: This macro defines the maximum packet size allowed for the CDC driver's notification endpoint. Normal use of this endpoint dictates it be defined as eight (8) bytes.

Precondition: None

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

Example: `#define CDC_INT_EP_SIZE        8`

CDC_DATA_INTF_ID

Purpose: This macro defines the USB interface ID number for the CDC serial driver's data interface.

Precondition: None

Valid Values: Interface ID numbers must begin at zero, and must not conflict with any other active interface on the same device.

Example: `#define CDC_DATA_INTF_ID        1`

CDC_BULK_EP_NUM

Purpose:	This macro defines the USB endpoint number for the CDC serial driver's bulk endpoint used for data transfer to-or-from the host.
Precondition:	None
Valid Values:	Endpoint numbers must be between one (Endpoint 0 is dedicated) and 15, inclusive, and must not be used more then once in per direction..

> **Note:** The USB Firmware stack allocates memory for every endpoint, starting from zero 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.

Example:	`#define CDC_BULK_EP_NUM        3`

CDC_BULK_OUT_EP_SIZE

Purpose:	This macro defines the maximum packet size (in bytes) allowed for the CDC driver's Data-Out (receive) endpoint. Larger values will increase data throughput. Smaller values will reduce throughput.
Precondition:	None
Valid Values:	This macro must be defined as 8, 16, 32, or 64.
Example:	`#define CDC_BULK_OUT_EP_SIZE  64`

CDC_BULK_IN_EP_SIZE

Purpose:	This macro defines the maximum packet size (in bytes) allowed for the CDC driver's Data-In (transmit) endpoint. Larger values will increase data throughput. Smaller values will reduce throughput.
Precondition:	None
Valid Values:	This macro must be defined as 8, 16, 32, or 64.
Example:	`#define CDC_BULK_IN_EP_SIZE  64`

CDC_DEFAULT_BPS

Purpose:	This macro defines the default UART data rate (in bits per second) reported to the host.
Precondition:	None
Valid Values:	This macro may be defined as any 32-bit number, the commonly accepted bit-rate values are: 110, 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 460800, and 921600.
Example:	`#define CDC_DEFAULT_BPS        115200`

CDC_DEFAULT_FORMAT

Purpose:	This macro defines the default UART character format reported to the host.
Precondition:	None
Valid Values:	0 = 1 Stop bit per character
	1 = 1.5 Stop bits character
	2 = 2 Stop bits per character
Example :	`#define CDC_DEFAULT_FORMAT        0`

CDC_DEFAULT_PARITY

Purpose:	This macro defines the default UART parity type reported to the host.
Precondition:	None
Valid Values:	0 = None
	1 = Odd Parity
	2 = Even Parity
	3 = Mark Parity
	4 = Space Parity
Example:	`#define CDC_DEFAULT_PARITY      0`

CDC_DEFAULT_NUM_BITS

Purpose:	This macro defines the default number of bits per UART word reported to the host.
Precondition:	None
Valid Values:	This may be defined as 5, 6, 7, 8, or 16 bits per word.
Example:	`#define CDC_DEFAULT_NUM_BITS    8`

CDC_APP_EVENT_HANDLING_FUNC

Purpose:	This macro identifies the name of the application's event-handling routine to the CDC serial function driver.
Precondition:	This routine must be defined by the application.
Valid Values:	The value of this macro must equate to the name of the application's event-handling routine.
Example:	`#define CDC_APP_EVENT_HANDLING_FUNC CdcDemoEventHandler`

APPENDIX B: USB CDC SERIAL FUNCTION API

This section describes the CDC serial function driver API. This API provides a means for the application to transfer data on the USB as if it were using a UART. Most of the USB details are hidden.

.

Table C-1 summarizes the CDC serial function driver API.

TABLE B-1: USB GENERIC FUNCTION API SUMMARY

Operation	Description
USBUSARTTxIsReady	Determines if the driver is ready to send data on the USB.
USBUSARTTxIsBusy	Determines if the driver is currently busy sending data.
USBUSARTRxIsReady	Determines if the driver is ready to receive data from the USB.
USBUSARTRxIsBusy	Determines if the driver is currently busy receiving data.
USBUSARTRxGetLength	Provides the current number of bytes that have been received from the USB and placed into the caller's buffer.
USBUSARTTx	Starts a transfer, sending data on the USB.
USBUSARTRx	Starts a transfer, receiving data from the USB.
USBUSARTGets	Gets a string of bytes from the USB (uses USBUSARTRx).
USBUSARTPuts	Sends a string of bytes from the USB, including the NULL terminator (uses USBUSARTTx).
USBUSARTGetLineCoding	Provides the current line-coding information (bits-per-second, number of Stop, parity, and data bits per word).
USBUSARTGetCmdStr	Retrieves an encapsulated command string from host.
USBUSARTSendRespStr	Sends a response to an encapsulated command to the host.
USBUSARTSendNotification	Sends an asynchronous notification packet to the host.
CDC_APP_EVENT_HANDLING_FUNC	This is a call-back routine, called by the CDC serial function driver when an application-specific event occurs.

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

USB CDC Serial Function API - USBUSARTTxIsReady

This routine determines if the CDC function driver is ready to send data on the USB.

Syntax

```
BOOL USBUSARTTxIsReady ( void )
```

Parameters

None

Return Value

TRUE if the system is ready to transmit data on the USB

FALSE if not.

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
if (USBUSARTTxIsReady ())
{
    USBUSARTTx (&buffer, sizeof(buffer));
}
```

USB CDC Serial Function API – USBUSARTTxIsBusy

This routine determines if the CDC serial function driver is currently busy sending data on the USB.

Syntax

```
BOOL USBUSARTTxIsBusy ( void )
```

Parameters

None

Return Value

TRUE if the system is busy transmitting data on the USB

FALSE if not

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
if (!USBUSARTTxIsBusy ())
{
    USBUSARTTx (&buffer, sizeof(buffer));
}
```

USB CDC Serial Function API - USBUSARTRxIsReady

This routine determines if the CDC serial function driver is ready to receive data on the USB.

Syntax

```
BOOL USBUSARTRxIsReady ( void )
```

Parameters

None

Return Value

`TRUE` if the system is ready to receive data from the USB

`FALSE` if not.

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
if (USBUSARTRxIsReady())
{
    USBUSARTRx(&buffer, sizeof(buffer));
}
```

USB CDC Serial Function API – USBUSARTRxIsBusy

This routine determines if the CDC serial function driver is currently busy receiving data on the USB.

Syntax

```
BOOL USBUSARTRxIsBusy ( void )
```

Parameters

None

Return Value

TRUE if the system is busy receiving data on the USB

FALSE if not

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
if (!USBUSARTRxIsReady())
{
    USBUSARTRx(&buffer, sizeof(buffer));
}
```

USB CDC Serial Function API - USBUSARTRxGetLength

This routine provides the current number of bytes that have been received from the USB and placed into the caller's buffer since the most recent call to `USBUSARTRx`.

Syntax

```
BYTE USBUSARTRxGetLength ( void )
```

Parameters

None

Return Value

The current number of bytes available in the caller's buffer

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
Size = USBUSARTRxGetLength();
```

USB CDC Serial Function API - USBUSARTTx

This routine starts a USB transfer to transmit data on the USB. It is non-blocking, so the transfer will have been started but not completed. The caller will have to monitor `USBUSARTTxIsReady` or `USBUSARTTxIsBusy` to determine when the transfer has completed.

Syntax

```
void USBUSARTTx ( BYTE *pData, BYTE len )
```

Parameters

`pData` – Pointer to the starting location of the data to transmit

`len` – Length of the data, in bytes

Return Value

None

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

`USBUSARTTxIsReady` must return `TRUE` (or `USBUSARTTxIsBusy` must return `FALSE`) before this routine is called or unexpected behavior may result.

Side Effects

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

Example

```
if (USBUSARTTxIsReady ())
{
    USBUSARTTx (&buffer, sizeof(buffer));
}
```

AN1164

USB CDC Serial Function API – USBUSARTRx

This routine starts a USB transfer to receive data from the USB. It is non-blocking, so the transfer will have been started but not completed. The caller will have to monitor `USBUSARTRxIsReady` or `USBUSARTRxIsBusy` to determine when the transfer has completed.

Syntax

```
void USBUSARTRx( BYTE *pData, BYTE len )
```

Parameters

`pData` – Pointer to the buffer in which to receive data

`len` – Length of the buffer, in bytes

Return Value

None

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

`USBUSARTRxIsReady` must return `TRUE` (or `USBUSARTRxIsBusy` must return `FALSE`) before this routine is called or unexpected behavior may result.

Side Effects

A transfer from the USB of the given size and data has been started.

Example

```
if (USBUSARTRxIsReady ())
{
    USBUSARTRx (&buffer, sizeof(buffer));
}
```

USB CDC Serial Function API - USBUSARTPuts

This routine writes a string of data to the USB, including the null character. It is non-blocking, so the transfer will have been started but not completed. The caller will have to monitor `USBUSARTTxIsReady` or `USBUSARTTxIsBusy` to determine when the transfer has completed.

Syntax

```
void USBUSARTPuts ( char *data )
```

Parameters

`data` – Pointer to a null-terminated string of data. If a null character is not found, 255 bytes of data will be transferred to the host.

Return Value

None

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

`USBUSARTTxIsReady` must return `TRUE` (or `USBUSARTTxIsBusy` must return `FALSE`) before this routine is called or unexpected behavior may result.

Side Effects

A transfer onto the USB of the given string has been started.

Example

```
if(USBUSARTTxIsReady())
{
    USBUSARTPuts(gTestStr);
}
```

USB CDC Serial Function API - USBUSARTGets

This routine reads a string of data of the given length from the USB, not including the null character. It is non-blocking, so the transfer will have been started but not completed. The caller will have to monitor USBUSARTRxIsReady or USBUSARTRxIsBusy to determine when the transfer has completed.

Syntax

```
BYTE USBUSARTGets ( char *buffer, BYTE len )
```

Parameters

buffer – Pointer to where bytes received are to be stored

len – The number of bytes expected

Note:	No more then "len" bytes will be received.

Return Value

The number of bytes sent.

Note:	Since the transfer has not started yet when this routine returns, this value is always zero (0). The caller should use the USBUSARTRxGetLength routine to get the number of bytes sent.

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

USBUSARTRxIsReady must return TRUE (or USBUSARTRxIsBusy must return FALSE) before this routine is called or unexpected behavior may result.

Side Effects

A transfer from the USB of the given size and data has been started.

Example

```
if(USBUSARTRxIsReady())
{
    USBUSARTGets(&test_str, sizeof(test_str));
}
```

USB CDC Serial Function API – USBUSARTGetLineCoding

This routine provides the current line-coding (BPS, # Stop bits, parity, and data bits-per-word) information.

> **Note:** If the line coding data is currently being updated by the host, this function will return FALSE and the caller should retry it later.

Syntax

```
BOOL USBUSARTGetLineCoding ( LINE_CODING *pLCData )
```

Parameters

pLCData – Pointer to where bytes received are to be stored

The following structure is used to hold the line coding data:

```
typedef union _LINE_CODING
{
    struct
    {
        BYTE _byte[LINE_CODING_LENGTH];
    };
    struct
    {
        DWORD   dwDTERate;
        BYTE    bCharFormat;
        BYTE    bParityType;
        BYTE    bDataBits;
    };
} LINE_CODING;
```

The LINE_CODING_LENGTH macro is defined as 7. The dwDTERate field should contain the number of bits-per-second. The bCharFormat field should be 0, 1, or 2 indicating 1, 1.5, or 2 stop bits. The bParityType field should be 0, 1, 2, 3 or 4 indicating no, odd, even, mark, or space parity. Finally, bDataBits should be 5, 6, 7, 8, or 16 indicating how many bits in a data word.

Return Value

TRUE if successful

FALSE if the line-coding data was currently in the process of being updated

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

None

Example

```
if (USBUSARTGetLineCoding (&line_coding) == FALSE)  {
    // Handle Error
}
```

USB CDC Serial Function API - USBUSARTGetCmdStr

This routine gets an encapsulated command string from the host.

> **Note:** This routine should only be called when an EVENT_CDC_CMD event notification has been received or the caller may interfere with normal control-request processing.

Syntax

```
BOOL USBUSARTGetCmdStr ( char *buffer, BYTE len )
```

Parameters

buffer – Pointer to where bytes received are to be stored

len – The number of bytes expected

Return Value

TRUE if successful

FALSE if not

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

The application has received an EVENT_CDC_CMD event notification in response to the host sending a SEND_ENCAPSULATED_COMMAND request.

Side Effects

The next len bytes read from EP0 will be read into the memory located at the address in "buffer".

Example

```
switch (event) {
    case EVENT_CDC_CMD:
        return USBUSARTGetCmdStr (command, sizeof(command));
//… Handle other events as needed.
```

USB CDC Serial Function API - USBUSARTSendRespStr

This routine writes a string of data to the host in response to a GET_ENCAPSULATED_RESPONSE request.

> **Note:** This routine should only be called once per GET_ENCAPSULATED_RESPONSE request.

Syntax

```
BOOL USBUSARTSendRespStr ( char *data )
```

Parameters

data – Pointer to a null-terminated string of response data

Return Value

TRUE if successful

FALSE if not

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

The application must have received an EVENT_CDC_RESP event in response to the host sending a GET_ENCAPSULATED_RESPONSE request.

Side Effects

A transfer of the response data will be started using the USB control endpoint.

Example

```
switch (event) {
    case EVENT_CDC_RESP:
        return USBUSARTSendRespStr (&response_string);
//… Handle other events as needed.
```

AN1164

USB CDC Serial Function API - USBUSARTSendNotification

This routine sends a notification packet to the host through the notification endpoint.

Syntax

```
BOOL USBUSARTSendNotification ( CDC_NOTIFICATION *notification )
```

Parameters

`notification` – Pointer to the notification data to send to the host

The "`CDC_NOTIFICATION`" data type is defined as follows:

```
typedef SETUP_PKT CDC_NOTIFICATION;
```

The "`SETUP_PKT`" data type is defined in the `usb_ch9.h` file as follows:

```
typedef struct SetupPkt
{
    union                           // offset   description
    {                               // ------   ------------------------
        BYTE bmRequestType;         //   0      Bit-map of request type
        struct
        {
            BYTE    recipient: 5; //             Recipient of the request
            BYTE    type:      2; //             Type of request
            BYTE    direction: 1; //             Direction of data X-fer
        };
    }requestInfo;

    BYTE    bRequest;               //   1      Request type
    UINT16  wValue;                 //   2      Depends on bRequest
    UINT16  wIndex;                 //   4      Depends on bRequest
    UINT16  wLength;                //   6      Depends on bRequest
} SETUP_PKT, *PSETUP_PKT;
```

This structure is used to send application or class-specific notifications to the host. It can be statically initialized using the following macro.

```
CDC_INIT_NOTIFICATION(n,v,i,l)
```

Where: n = Notification number (above)

v = `wValue` (notification-specific)

i = `wIndex` (notification-specific)

l = `wLength` (notification-specific) Length of data to follow, if any.

Return Value

`TRUE` if able to start the transfer

`FALSE` if not

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

A transfer of the response data will be started using the USB control endpoint.

Example

```
if (USBUSARTSendNotification(&notification) == FALSE) {
     // Handle failure
}
```

USB CDC Serial Function API – CDC_APP_EVENT_HANDLING_FUNC

This routine is implemented by the application. It is called by the CDC serial function driver to allow the application to handle CDC-specific events as they occur.

> **Note:** The demo application names this routine `CdcDemoEventHandler`. It is identified to the CDC serial driver by defining the `CDC_APP_EVENT_HANDLING_FUNC` macro to equal this function name.

Syntax

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

Parameters

`event` – CDC-specific event ID

`data` – Pointer to event-specific data

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

The following CDC-Specific events are defined.

EVENT_CDC_LINE_CTRL

This event indicates that a change in the line-control status occurred. It has no associated data. When this event occurs, the application may call `USBUSARTGetLineCoding` to identify the new line-control settings and take any appropriate action.

EVENT_CDC_CMD

This event indicates that an encapsulated command is about to be sent from the host. The `data` parameter points to a 16-bit value (`size = 2`) that identifies the size of the expected command (in bytes). When this event occurs, the application should call `USBUSARTGetCmdStr` (using a buffer of the given size) to receive the protocol-specific command.

EVENT_CDC_CMD_RCVD

This event indicates that an encapsulated command has been received. The `data` parameter points to a 32-bit (`size = 4`) value that identifies the actual size of the command received. (Note this should equal the value given with the EVENT_CDC_CMD event, even though the data size is different.) This event is used to indicate to the application that the `USBUSARTGetCmdStr` request has completed and the command has been received.

EVENT_CDC_RESP

This event indicates that an encapsulated response has been requested by the host. The `data` parameter points to a 16-bit (`size = 2`) value that identifies the expected size of the response. When this event occurs, the application may call `USBUSARTSendRespStr` to send a protocol-specific response string.

Return Value

The application should return `TRUE` to indicate that the event was handled and `FALSE` to indicate that it was not handled.

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

Side effects will depend on how the application implements the routine.

Example

```
PUBLIC BOOL CdcDemoEventHandler ( USB_EVENT event, void *data, int size )
{
    UINT16      cmd_size;
    LINE_CODING line_coding;
```

```
    // Handle specific events.
    switch (event)
    {
    // The host has changed the line-control status.
    case EVENT_CDC_LINE_CTRL:

        // Read the new line coding.
        //
        // (This demo doesn't do anything with the new line
        // coding, but it could after reading it, if desired.)
        return USBUSARTGetLineCoding (&line_coding);


    // The host is about to send an encapsulated command.
    case EVENT_CDC_CMD:

        // Read the command string.
        if (size == sizeof(UINT16) && data != NULL)
        {
            cmd_size = (UINT16)min(sizeof(gCommand), size);
            return USBUSARTGetCmdStr (gCommand, cmd_size);
        }
        break;


    } // switch (event)


    // Event not handled.
    return FALSE;

} // CdcDemoEventHandler
```

APPENDIX C: USB CDC SERIAL FUNCTION DRIVER INTERFACE

This section describes the routines that make up the interface between the CDC serial function driver and the lower-level USB device firmware stack. This interface consists of two routines, one to initialize the function driver and another to handle CDC class-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 **"The Function Driver Table"**) to identify them to the lower-level USB stack. This mechanism allows support for multi-function devices.

.

TABLE C-1:   USB CDC SERIAL FUNCTION DRIVER INTERFACE SUMMARY

Operation	Description
USBUARTInit	Initializes the CDC serial driver
USBHandleEvents	Identifies and handles bus events

USB CDC Serial Function Driver Interface - USBUARTInit

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

Syntax

```
BOOL USBUARTInit ( unsigned long flags )
```

Parameters

`flags` - Initialization Flags (reserved)

Return Value

`TRUE` if successful

`FALSE` if not

Preconditions

None

Side Effects

The USB CDC function driver has been initialized and is ready to handle CDC-specific events.

Example

```
const FUNC_DRV gDevFuncTable[] =
{
    // USB CDC Serial Emulation Function Driver
    {
        USBUARTInit,            // Init routine
        USBUARTEventHandler,    // Event routine
        0                       // Init flags
    }
};
```

USB CDC Serial Function Driver Interface – USBUARTEventHandler

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

Syntax

```
BOOL USBUARTEventHandler (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

> **Note:** Events are defined by the lower-level USB firmware stack and handled by the CDC serial function driver. Refer to the application's event-handling routine (see **"CDC_APP_EVENT_HANDLING_FUNC"**) for events that may be propagated to the application.

Return Value

TRUE if the event was handled

FALSE if not (or if additional processing is required).

Preconditions

The system has been enumerated as a CDC serial emulation device on the USB and the CDC function driver has been initialized.

Side Effects

The side effects vary greatly depending on the event. In general, the CDC-specific event has been handled or passed to the application for handling.

Example

```
const FUNC_DRV gDevFuncTable[] =
{
    // USB CDC Serial Emulation Function Driver
    {
        USBUARTInit,                // Init routine
        USBUARTEventHandler,        // Event routine
        0                           // Init flags
    }
};
```

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"*.

APPENDIX E: USB DESCRIPTOR TABLE

The CDC serial demo application defines the USB_DESC_TABLE data type in usb_app.c and uses that data type to define its descriptor table, as shown below.

TABLE E-1: DEVICE DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	12
bDescriptorType	Type, always USB_DESCRIPTOR_DEVIC	1	1
bcdUSB	USB spec version, in BCD	2	0200
bDeviceClass	Device class code	1	2
bDeviceSubClass	Device sub-class code	1	0
bDeviceProtocol	Device protocol	1	0
bMaxPacketSize0	EP0, max packet size	1	8
idVendor	Vendor ID (VID)	2	04d8
idProduct	Product ID (PID)	2	000A
bcdDevice	Device release num, in BCD	2	0000
iManufacturer	Manufacturer name string index	1	1
iProduct	Product description string index	1	2
iSerialNum	Product serial number string index	1	3
bNumConfigurations	Number of supported configurations	1	1

TABLE E-2: CONFIGURATION DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	09
bDescriptorType	Type, USB_DESCRIPTOR_CONFIGURATION	1	02
wTotalLength	Total size of all descriptors in this configuration	2	0043
bNumInterfaces	Number of interfaces in this configuration	1	02
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	Type, USB_DESCRIPTOR_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	01
bInterfaceClass	USB interface class ID	1	02
bInterfaceSubClass	USB interface sub-class ID	1	02
bInterfaceProtocol	USB interface protocol ID	1	01
iInterface	Interface description string index	1	00

AN1164

TABLE E-4: CDC HEADER FUNCTIONAL DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bFunctionLength	Size of this descriptor	1	05
bDescriptorType	Type, CS_INTERFACE	1	24
bDescriptorSubtype	Header functional descriptor subtype	1	00
bcdCDC	BCD Release version of the CDC specification	2	0110

TABLE E-5: CDC ABSTRACT CONTROL MODEL FUNCTIONAL DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bFunctionLength	Size of this descriptor	1	04
bDescriptorType	Type, CS_INTERFACE	1	24
bDescriptorSubtype	Abstract Control Model functional descriptor subtype	1	2
bmCapabilities	Device supports set/get line coding and notification	1	02

TABLE E-6: CDC UNION FUNCTIONAL DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bFunctionLength	Size of this descriptor	1	05
bDescriptorType	Type, CS_INTERFACE	1	24
bDescriptorSubtype	Union functional descriptor	1	06
bmMasterInterface	Interface number of the master CDC controlling interface	1	00
bSlaveInterface0	Interface number of the first slave interface		01

TABLE E-7: CDC CALL MANAGEMENT FUNCTIONAL DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bFunctionLength	Size of this descriptor	1	05
bDescriptorType	Type, CS_INTERFACE	1	24
bDescriptorSubtype	Abstract Control Model functional descriptor subtype	1	01
bmCapabilities	Device handles call management itself	1	00
bDataInterface	Interface number of data class interface optionally used for call management		01

TABLE E-8: NOTIFICATION (IN) ENDPOINT DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	07
bDescriptorType	Type, USB_DESCRIPTOR_ENDPOINT	1	05
bEndpointAddress	Address and direction of the endpoint	1	82
bmAttributes	Interrupt transfer endpoint	1	03
wMaxPacketSize	Largest packet this EP can handle	2	0008
bInterval	Polling period (in mS)	1	02

TABLE E-9: DATA CLASS INTERFACE DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	09
bDescriptorType	Type, USB_DESCRIPTOR_INTERFACE	1	04
bInterfaceNumber	Interface ID number	1	01
bAlternateSetting	ID number of alternate Interface setting	1	00
bNumEndpoints	Number of endpoints in this interface	1	02
bInterfaceClass	USB interface class ID	1	0A
bInterfaceSubClass	USB interface sub-class ID	1	00
bInterfaceProtocol	USB interface protocol ID	1	00
iInterface	Interface description string index	1	00

TABLE E-10: DATA (OUT) ENDPOINT DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	07
bDescriptorType	Type, USB_DESCRIPTOR_ENDPOINT	1	05
bEndpointAddress	Address and direction of the endpoint	1	03
bmAttributes	Bulk transfer endpoint	1	02
wMaxPacketSize	Largest packet this EP can handle	2	0040
bInterval	Not Polled	1	00

TABLE E-11: DATA (IN) ENDPOINT DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex)
bLength	Size of this descriptor	1	07
bDescriptorType	Type, USB_DESCRIPTOR_ENDPOINT	1	05
bEndpointAddress	Address and direction of the endpoint	1	83
bmAttributes	Bulk transfer endpoint	1	02
wMaxPacketSize	Largest packet this EP can handle	2	0040
bInterval	Not Polled	1	00

TABLE E-12: LANGUAGE ID STRING (0) DESCRIPTOR

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

TABLE E-13: VENDOR DESCRIPTION STRING (1) DESCRIPTOR

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

TABLE E-14: DEVICE DESCRIPTION STRING (2) DESCRIPTOR

Field	Description	Size (Bytes)	Value (Hex/String)
bLength	Size of this descriptor	1	34
bDescriptorType	Type, USB_DESCRIPTOR_STRING	1	03
wLangID	Language ID code	50	"CDC RS-232 Emulation Demo"

TABLE E-15: SERIAL NUMBER STRING (3) DESCRIPTOR

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

APPENDIX F: GET DESCRIPTOR ROUTINE

The following "get descriptor" routine (and helpers) provide access to the descriptors (which are application-specific) from the lower-level USB stack.

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

    default:
        return NULL;
    }

} // GetConfigurationDescriptor


static inline const void *GetStringDescriptor( PDESC_ID desc, unsigned int *length )
{
    // Check language ID
    if (desc->index > 0 && desc->lang_id != LANG_1_ID) {
        return NULL;
    }

    switch(desc->index)
    {
    case 0: // String 0
        *length = sizeof(gDescTable.string_0)+sizeof(gDescTable.langid);
        return &gDescTable.string_0;

    case 1: // String 1
        *length = sizeof(gDescTable.string_1)+sizeof(gDescTable.string_1_data);
        return &gDescTable.string_1;

    case 2: // String 2
        *length = sizeof(gDescTable.string_2)+sizeof(gDescTable.string_2_data);
        return &gDescTable.string_2;

    case 3: // String 3
        *length = sizeof(gDescTable.string_3)+sizeof(gDescTable.string_3_data);
        return &gDescTable.string_3;

    default:
        return NULL;
    }

} // GetStringDescriptor


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

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

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

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

    default:
        return NULL;
    }

} // USBDEVGetDescriptor
```

The helper routines are "inline" functions. They are 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 **"USB Stack Options"**).

APPENDIX G: SOURCE CODE FOR THE MICROCHIP USB CDC SERIAL DRIVER

The complete source code for the Microchip PIC32 USB CDC serial 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.

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

AN1164

REVISION HISTORY

Rev. A Document (02/2008)

This is the initial released version of this document.

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

- Microchip products meet the specification contained in their particular Microchip Data Sheet.

- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.

- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.

- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

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

Trademarks

QUALITY MANAGEMENT SYSTEM

CERTIFIED BY DNV

ISO/TS 16949:2002

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