MICROCHIP

Regional Training Centers

COM 3202

Designing a USB Embedded Host Application

Lab Manual (v1.00)

Initial Installation/Set-Up

Purpose:

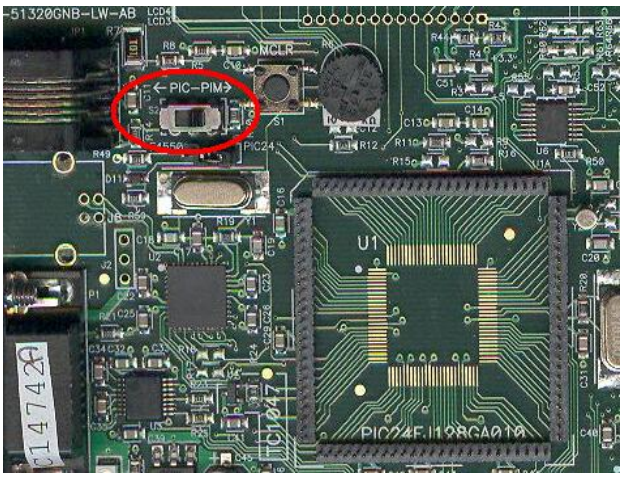
- Set-up your PC to run the labs

Equipment:

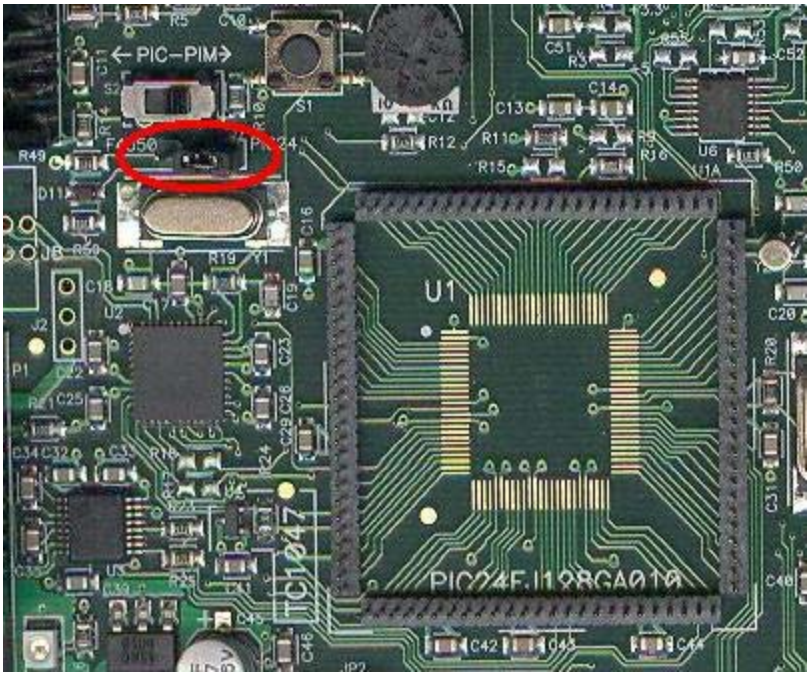
- PC running Windows XP Professional with Service Pack2 (SP2) and Internet Access
- PC should also have at least 3 available **High Speed** USB ports and/or a powered hub.
- Adobe PDF reader.
- Explorer 16 (DM240001) + PIC24F USB PIM (MA240014) or PIC32MX USB PIM (MA320002) + PICTail USB Card (AC164131)
- PICDEM FS USB Board (DM163025), programmed with factory default application "USB Device - MCHPUSB - Generic Driver - C18 - PICDEM FSUSB - MCHPUSB Bootload.hex"
- MPLAB REAL ICE (DV244005)
- MPLAB IDE v8.40
- MPLAB C30 v3.12 (do *not* use v3.20b), C32 v 1.05
- Microchip Application Libraries v2009-08-31 (MCHPFSUSB v2.5b (included in the class CDROM)
- 1 RS-232 –to- USB Converter Cable
- 1 Type A → Type Mini-B USB Cable
- 1-FAT16/32 Formatted USB Thumb Drive
- Recommended: 1GB Kingston Data Traveler (P/N: DTI/1GB)
- Pre-loaded with contents of "\Development Tools\Thumb Drive Contents" folder
- COM3202 Class CDROM – Installed (see next section)

Hardware Setup:

1. Before attaching the PIC24FJ256GB110/PIC32MX460F512L PIM to the Explorer 16 board, insure that the processor selector switch (S2) is in the "PIM" position as seen in the image below.

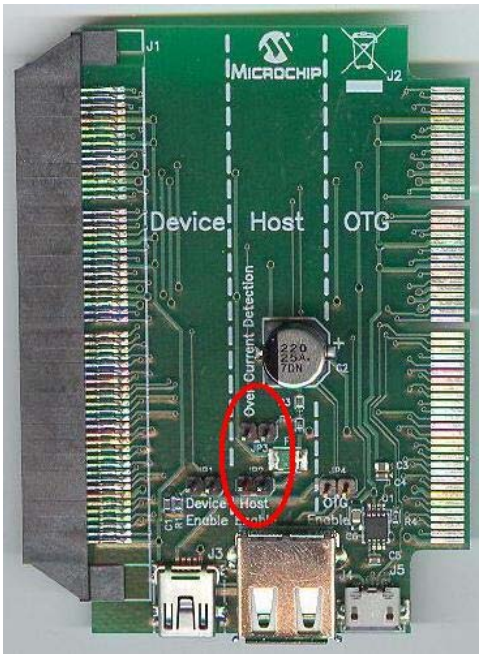


2. Short the J7 jumper to the “PIC24” setting:



3. Before inserting the PIM into the Explorer 16 board, remove all attached cables from both boards. Now, insert the PIM. Be careful when inserting the PIM to insure that no pins are bent or damaged during the process. Also insure that the PIM is not shifted in any direction and that all of the headers are properly aligned.

4. On the USB PICTail Plus board, short jumpers JP2 and JP3. Remove all other shorts on the board.



Courseware Installation/Restoration

First-time installation:

1. Install the class labs by copying the class CDROM contents to C:\RTC\COM3202
2. (Optional) Install the MCHPFSUSB v2.5b USB Framework by running the "Microchip Application Libraries v2009-08-31.exe" installer in C:\RTC\COM3202. Direct the installer to install into the default folder (C:\Microchip Solutions).
3. Copy the contents of "\Development Tools\Thumb Drive Contents" onto a FAT32-formatted USB Thumb Drive.
4. If necessary, re-program the PICDEM FS USB board with the .hex file "USB Device - MCHPUSB - Generic Driver - C18 - PICDEM FSUSB - MCHPUSB Bootload.hex" found in C:\RTC\COM3202

Restoring the COM3202 project files during the training:

1. Run "Restore Labs.bat" in C:\RTC\COM3202\Development Tools
2. Copy the contents of "\Development Tools\Thumb Drive Contents" onto a FAT32-formatted USB Thumb Drive.

Class Folder Structure (C:\RTC\COM3202)

C:\RTC\COM3202
\Microchip
\Lab1..Lab6
\USB Host - Mass Storage - Simple Demo
\USB Host - Mass Storage - Thumb Drive Data Logger
\USB Host - MCHPUSB - Generic Driver Demo
\Presentation & Handouts
\Users Guides & Data Sheets
\Development Tools

Key Folders:

\Microchip

USB Stack files – copied from C:\Microchip Solutions

\Presentation & Handouts

PDF files of the slides and this Student Handout

\Users Guides & Data Sheets

Repository for helpful documents discussed earlier.

\Development Tools

Contains all projects/labs and lab restore batch file, as well as other various tools used in the class

Running the Demos

Overview:

The following MCHPFSUSB Embedded Host framework demonstration projects are included in the COM3202 presentation:

`USB Host - MCHPUSB - Generic Driver Demo` (Part 1)
`USB Host - Mass Storage - Simple Demo` (Part 3)
`USB Host - Mass Storage - Thumb Drive Data Logger` (Part 3)

Instructions for running these demos are provided in the "Getting Started" HTML help pages that are included with the demos. They are available in the following folders:

```
C:\RTC\COM3202\USB Host - MCHPUSB - Generic Driver Demo\USB Host -
MCHPUSB - Generic Driver Demo.html

C:\RTC\COM3202\ USB Host - Mass Storage - Simple Demo\Getting Started -
Running the Host - Mass Storage - Simple Demo.html

C:\RTC\COM3202\ USB Host - Mass Storage - Thumb Drive Data Logger\
Getting Started - Running the Host - Mass Storage - Thumb Drive Data
logger.html
```

Detailed Instructions: "USB Host – Mass Storage – Thumb Drive Data Logger (Part 3):

Overview:

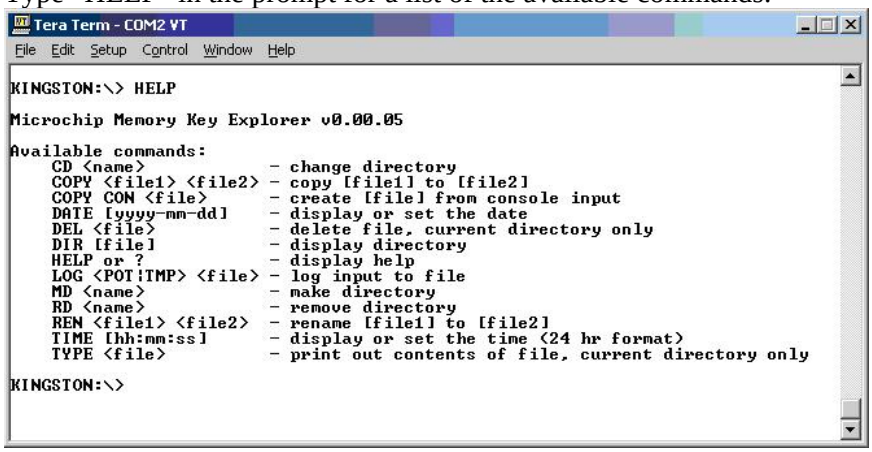
This is a demonstration of a simple application (terminal shell) that uses the Microchip Memory Disk Drive File System to manipulate files on a USB Thumb Drive. In Labs 4-5, you will learn how to configure/use this file system with the embedded host framework to manipulate files on a USB Thumb Drive.

Procedure:

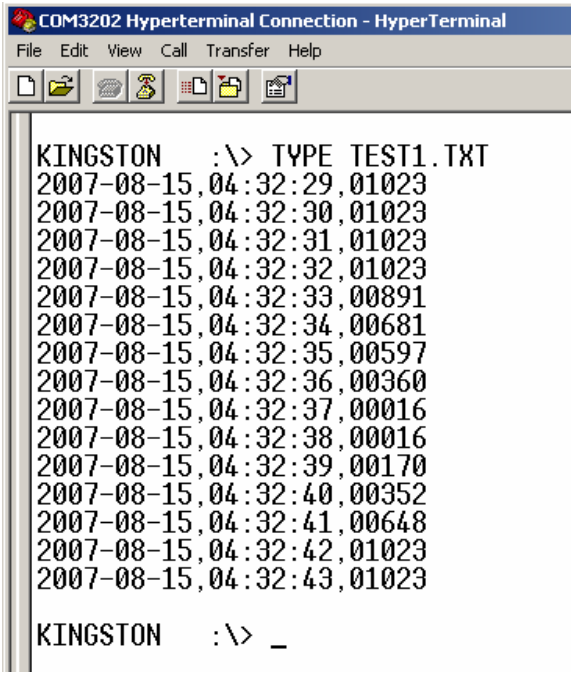
1. Ensure the basic hardware is set up (see steps 1-4 in **Hardware Setup**) and that you have inserted a FAT32-formatted a thumb drive into the USB PICTail Plus. You will also need to connect a serial cable from the Explorer16 to the computer.
2. Open/Build/Run the "**USB Host - Mass Storage - Thumb Drive Data Logger**" workspace in C:\RTC\COM3202\USB Host - Mass Storage - Thumb Drive Data Logger that is appropriate for the MCU you selected:
    a. PIC24: USB Host - Mass Storage - Thumb Drive Data Logger - C30.mcw
    b. PIC32: USB Host - Mass Storage - Thumb Drive Data Logger - C32.mcw
3. Open a HyperTerminal window. The HyperTerminal file **COM3202 Hyperterminal Connection.ht** (in **C:\RTC\COM3202**) contains the correct settings: 57600 baud, 8 data bits, no parity, 1 stop bit, no flow control. This console will be used to communicate with the shell program on the PIC.

4. Connect power to the Explorer 16. If power was already connected to the board before the serial port was connected, either cycle power, press reset, or press enter. This should give a ">" prompt. If the prompt does not appear then verify that you programmed the firmware correctly and that you have the correct serial port settings.

   Type "HELP" in the prompt for a list of the available commands.



5. Experiment with data-logging the potentiometer setting to a file ("LOG POT Test1.txt"). One sample per second is saved to the thumb drive.
6. Terminate the logging by pressing "Control-C"
7. Display the data by typing "TYPE TEST1.TXT"



Part 2 - Designing a Custom Class, Full-Speed USB Embedded-Host Application

Overview:

The lab exercises in this section focus on the steps needed to "connect" the application features to the USB client driver and the client driver to the host layer, not on implementing large amounts of new code.

The level of difficulty of each exercise starts out low and increases in the last few steps of the lab. There are three primary levels of difficulty.

Beginner:

Beginners should have no difficulty following the initial step-by-step instructions to learn the key concepts. This level of completion requires little or no original code development. Attendees will need to make minor changes, "comment out" temporary code, or "un-comment" code that has already been implemented.

Intermediate:

Most attendees should find a reasonable challenge and have the opportunity to explore the key concepts by following step-by-step instructions on how to implement small code segments in well documented sections of the existing application or driver.

Advanced:

Those who are already quite familiar with Microchip PIC microcontrollers, the MPLAB IDE, and USB will find a challenge implementing missing features with minimal direction. Resource materials are provided in this document and portions of the code may already be implemented, but instructions are limited to high-level descriptions of the intended feature. The attendee must determine the necessary steps to implement the feature and implement larger sections of code from scratch.

Beginners should still be able to gain an understanding of the key concepts, but even the most advanced students will find it difficult to finish all of the exercises in the time allotted. To re-enforce the concepts, attendees are encouraged to purchase the lab materials and complete the exercises once they have returned to their own lab environments.

Do not open the lab projects and jump right in making changes to the code. To obtain the benefit of doing the lab exercises, you must follow the directions given in this manual.

Lab 1 – Implement Application Using Microchip Generic Client Driver

Purpose:

The purpose of this lab is to learn how to use the Microchip-provided "Generic" client driver to create a simple USB Host application and how to use the "USBConfig.exe" program to configure the Microchip USB Embedded Host firmware stack.

Basic Objectives:

1. Assemble the hardware
2. Generate the USB configuration files
3. Build the application
4. Debug the application

Intermediate Objective:

5. Add missing "To Do" items

Advanced Objective:

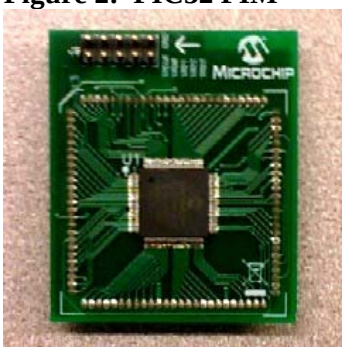
6. Add unimplemented features

Procedure:

1. Assemble the hardware.
   a. Select a PIC24 (see Figure 1) or PIC32 PIM (see Figure 2)

Figure 1: PIC24 PIM

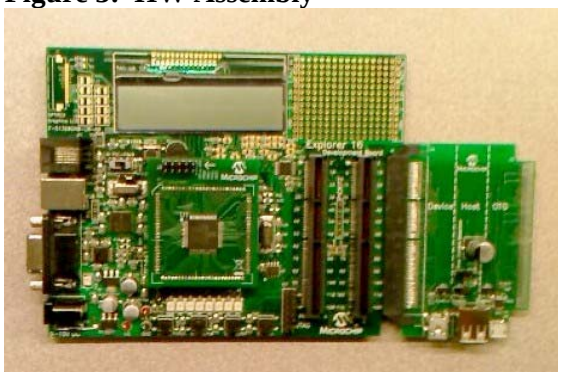
Figure 2: PIC32 PIM



b. Attach the PIM you selected and the USB PICTail™ Plus Daughter Board to the Explorer 16 board as shown in Figure 3.

Note: Be sure that pin 1 on the PIM is oriented to the upper Left corner, near the LCD display and the debugger connector.

Figure 3: HW Assembly



c. Ensure that the following jumper settings are correct on the USB PICtail™ Plus Daughter Board, as shown below:

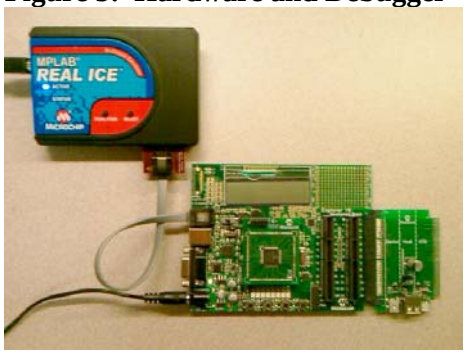
Figure 4: USB PICtail Jumpers

Over-Current Detection	– Shorted
Device Enable	– Open
Host Enable	– Shorted
OTG Enable	– Open

Note: Jumper numbers may vary depending on board revision.

d. Attach the power connector and the REAL ICE™ In-Circuit Emulator to the Explorer 16 board as shown in Figure 5.

Figure 5: Hardware and Debugger

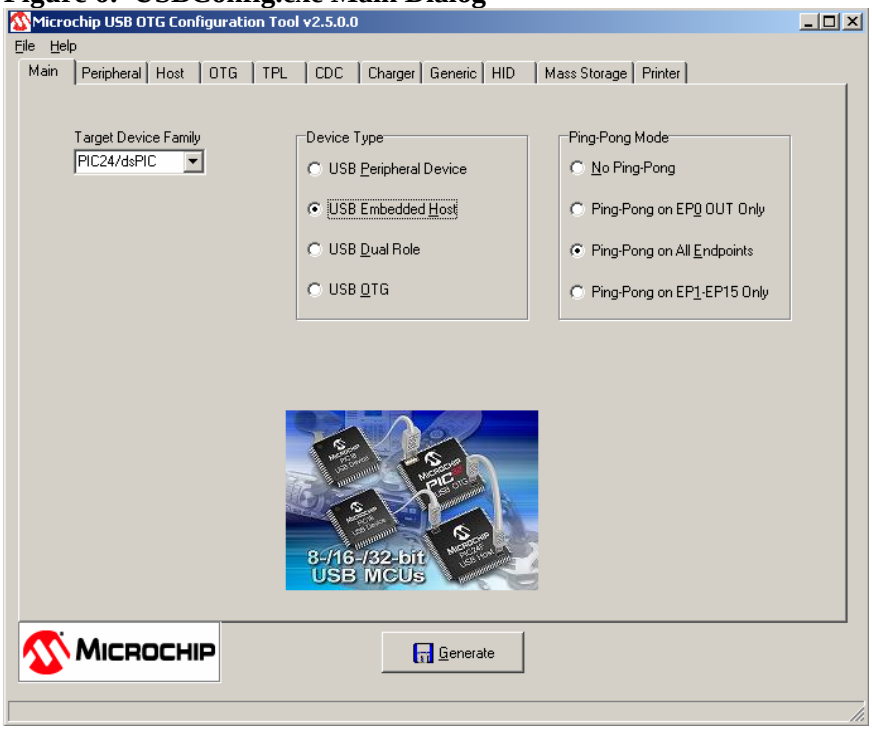


2. Generate the USB configuration files.

   a. Run the “USBConfig.exe” program (see Figure 6).

      By default, the “USBConfig.exe” program is installed in “C:\Microchip Solutions\USB Tools\USBConfig Tool”. For this class, it is provided in C:\RTC\COM3202

Figure 6: USBConfig.exe Main Dialog



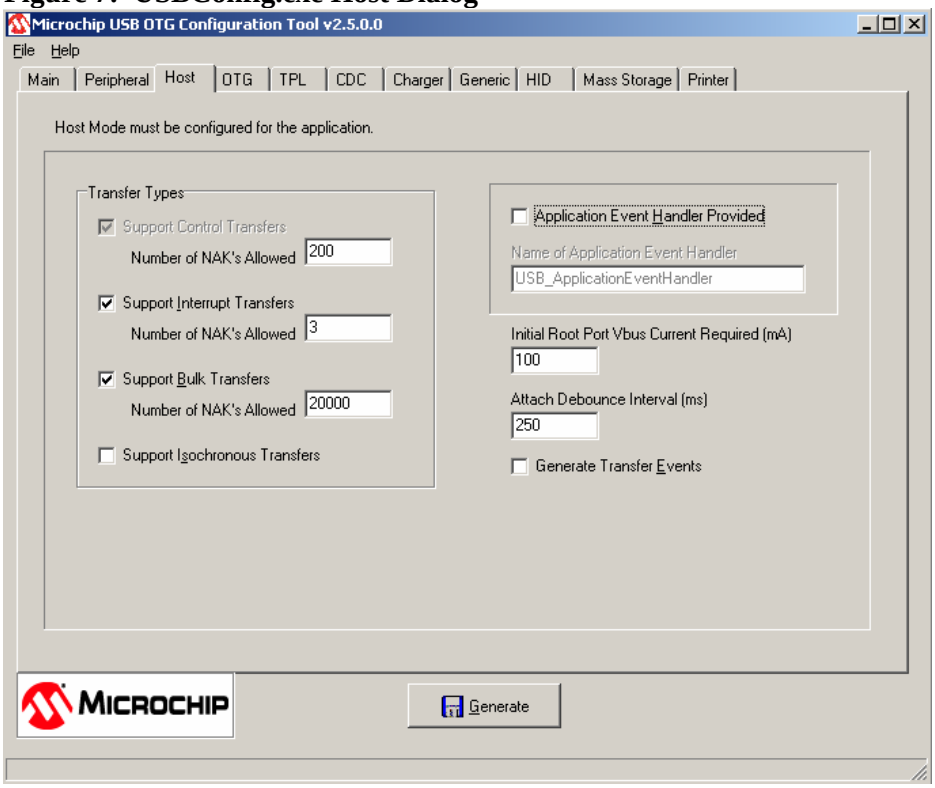
   b. Under the “Main” dialog tab (see Figure 6), make the following selections:

      - Under “Target Device Family”, choose the correct processor for the PIM you selected.
      - In the “Device Type” group, choose “USB Embedded Host”.
      - In the “Ping-Pong Mode” group, choose “Ping-Pong on All Endpoints”.

   c. Click the “Host” tab and make the following selections (see Figure 7):

      - Under the “Transfer Types” group, de-select “Support Isochronous Transfers”, leaving “Support Interrupt Transfers” and “Support Bulk Transfers” selected. (Leave the default number of “NAK’s” allowed for both.)
      - De-select the “Application Event Handler Provided” checkbox.
      - Ensure that the “Default Initial Root Port Vbus Current” is 100 mA.
      - Ensure that the “Attach Debounce Interval” is 250 ms.
      - Ensure that the “Generate Transfer Events” checkbox is de-selected.

Figure 7: USBConfig.exe Host Dialog

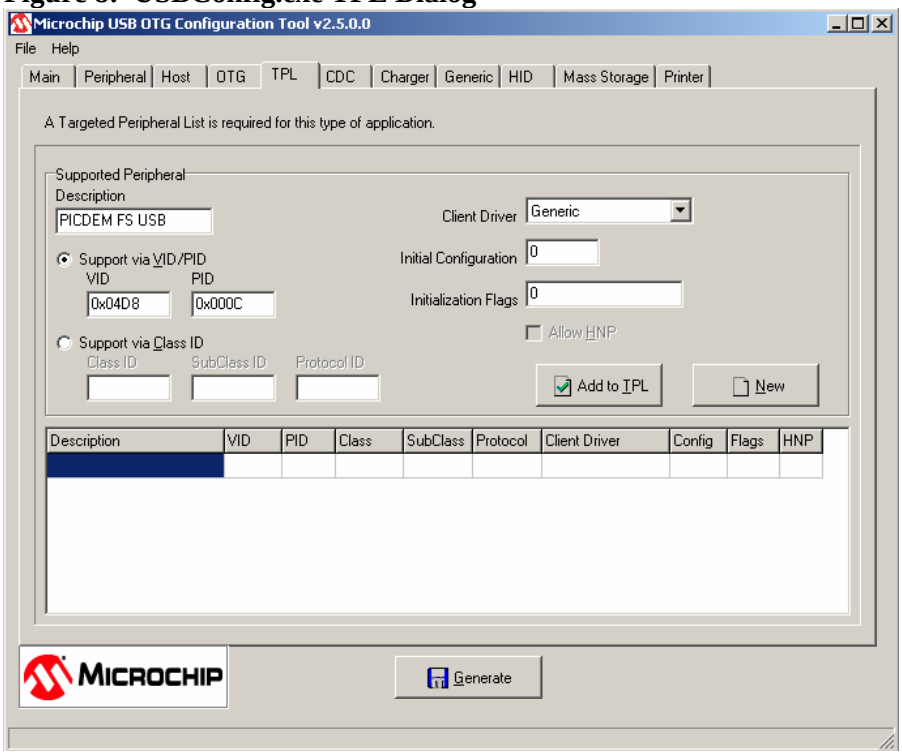


d. Click the “TPL” tab and fill in the TPL data for the PICDEM™ FS USB demo board (see Figure 8):

In the “Supported Peripheral” group, fill in the PICDEM™ FS USB demo board’s TPL information (see Figure 8):

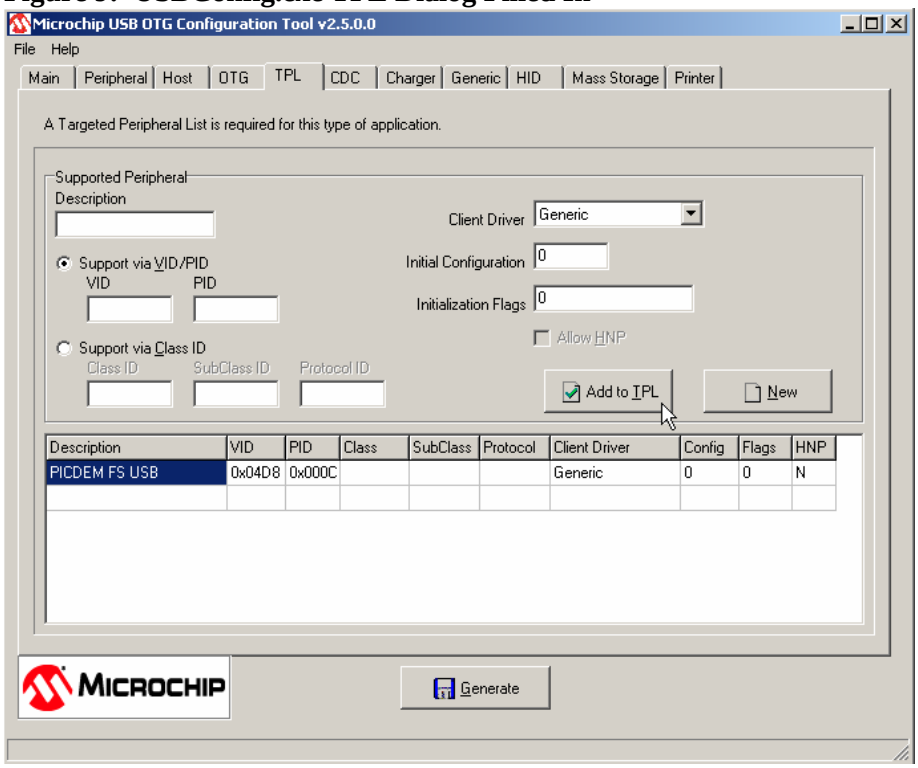
- Description: PICDEM FS USB
- Check “Support via VID/PID” and enter:
  - VID: 0x04D8
  - PID: 0x000C
- Ensure that “Support via Class ID” is not checked
- Select “Generic” in the “Client Driver” pull-down box.
- Leave the default “Initial Configuration” value of ‘0’
- Leave the default “Initialization Flags” value of ‘0’

Figure 8: USBConfig.exe TPL Dialog



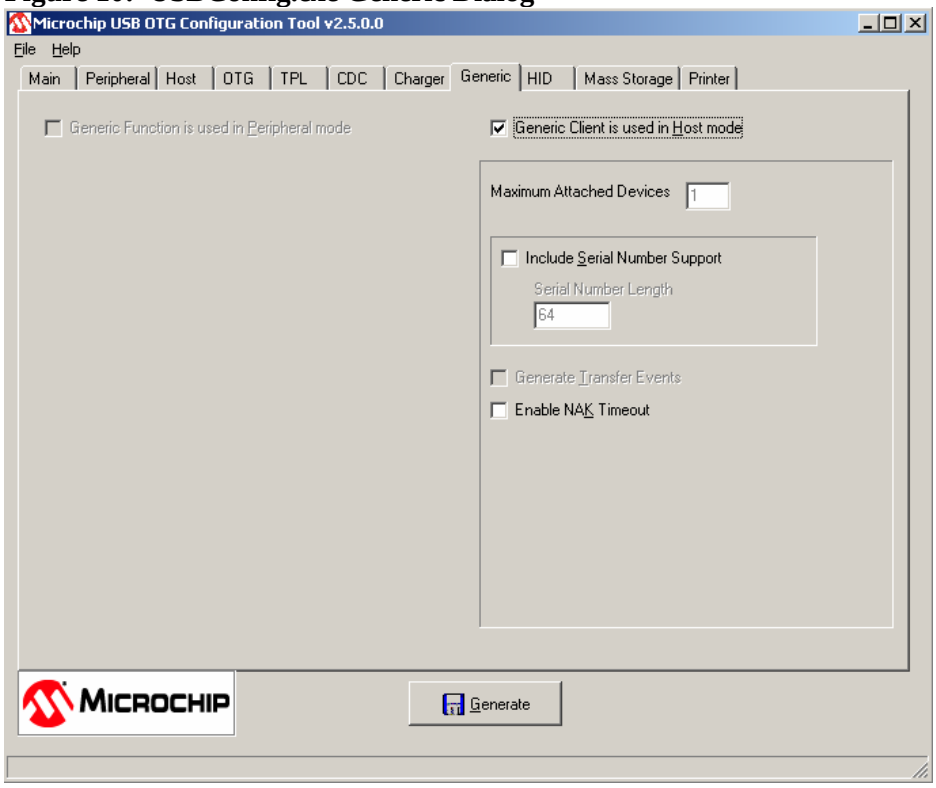
e. Click “Add to TPL” and observe that the data from the “Supported Peripheral” group gets cleared and is transferred to the TPL table below, (see Figure 9).

Figure 9: USBConfig.exe TPL Dialog Filled In



f. Click the “Generic” tab and select “Generic Client is used in Host mode” (see Figure 10).

Figure 10: USBConfig.exe Generic Dialog

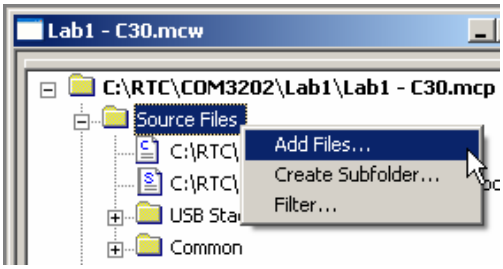


Note: “Include Serial Number Support” may be checked or not checked; it will not be used in this lab.

g. Click the “Generate” button and save the configuration files generated to the “Lab 1” project directory C:\ RTC\COM3202\Lab 1

h. Close the “USBConfig.exe” program.

3. Build the application.
    a. Open the Lab 1 workspace that is appropriate project for the MCU you selected.
        - PIC24: C:\ RTC\COM3202\Lab1\Lab1 – C30.mcw
        - PIC32: C:\ RTC\COM3202\Lab1\Lab1 – C32.mcw
    b. Ensure that REAL ICE™ is not selected as the programmer by selecting “None” under the “Select Programmer” choice in the “Programmer” top-level menu.
    c. Select REAL ICE™ as the debugger by selecting “REAL ICE” under the “Select Tool” choice in the “Debugger” top-level menu.
    d. In MPLAB, select “Build All” from the “Project” menu or click the  button on the project manager toolbox.
    e. Observe the error messages indicating that “USB_HOST_FW_MAJOR_VER” and “USB_HOST_FW_MINOR_VER” are undefined.
    f. Open the “main.c” file and search for the “LAB1 Step 3f” instructions in the comment block. Follow the instructions (steps 1-4) to include the correct USB header files.
    g. After attempting to rebuild the program, observe that there are several error messages indicating that “usbClientDrvTable” and “usbTPL” are undefined.
    h. Add the “usb_config.h” and “usb_config.c” files you created in step 2 to the project by right-clicking on the “Header Files” and “Source Files” folders in the project window and selecting “Add Files” as shown at the right. Browse to the Lab 1 project directory to locate the files.



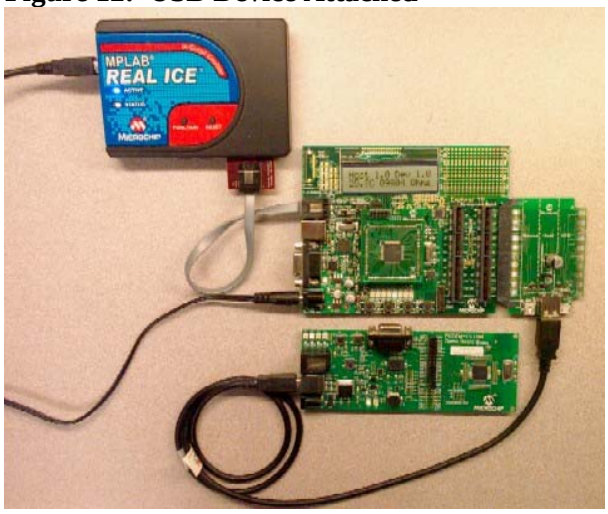
    i. Re-build the project. It should build successfully at this time.
4. Debug the application
    a. Select “Program” from the “Debugger” menu or click the  button on the REAL ICE™ tool bar.
    b. Select “Run” from the “Debugger menu or click the  button on the Debug tool bar.

    The initial message should now be displayed on the Explorer 16’s LCD (see Figure 11). This message gives the version number of the Host firmware and has dashes (‘-‘) as space holders for the information it reads from the device: device firmware version number, temperature in degrees Celsius, and resistance of the potentiometer in Ohms.

Figure 11: Initial LCD Message

c. Connect the PICDEM™ FS USB Demo Board to the full-sized Type-A Host connector on the USB PICtail™ Plus Daughter Board using standard Type-A-to-Type-B USB cable as shown in Figure 12.

Figure 12: USB Device Attached



d. Observe that the message on the LCD screen does not change when the device is attached. This indicates that the application did not recognize the device being attached to the host. Also observe that the device "panics", flashing LEDs D1 & D2 together rapidly. This indicates that the device was not properly enumerated by the host.

e. Follow the instructions in the "LAB1 Step 4e"comment block in "main.c" to initialize and maintain the USB stack. Re-build, program and run the application.

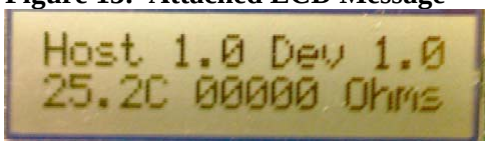
Observe that, once the USB stack has been initialized and maintained that the device now enumerates (as indicated by LEDs D1 and D2 on the PICDEM™ FS USB Demo Board flashing alternately). However, the messages shown in the LCD on the Explorer 16 have not changed so the application has not yet recognized that the device is attached.

f. Follow the instructions in the "LAB1 Step 4f" comment block in "main.c" to allow the application to recognize when the device has been attached by calling the Generic driver's "USBHostGenericGetDeviceAddress()" API routine.

g. Next, follow the instructions in the "LAB1 Step 4g" comment block to enable the application to send commands to the device. (Note: the "receive response" support is already operational.)

You should now see the device's firmware version number and temperature appear on the LCD screen along with zeroes ("00000") for the potentiometer resistance value (see Figure 13).

Figure 13: Attached LCD Message



You can also touch the temperature sensor, U4 on the PICDEM™ FS USB Demo Board (near the middle of the board by the potentiometer), and (at normal room temperatures) see the temperature change on the host.

5. Add missing “To Do” items. (Intermediate)
    a. Follow the instructions in the “LAB1 Step 5a” comment block in the “main.c” file to add the missing feature to read the potentiometer data.

       If you’ve added the missing feature correctly into the application, you should be able to turn the potentiometer and watch the resistance reading change on the LCD.

       Use the debugger to step through the code as directed in the comments. Pay attention to the sequence execution of the state transitions and to the command-and-response method of communicating with the device.
6. Add unimplemented features. (Advanced)
    a. Follow the instructions in the “LAB1 Step 6a” comment block in the “main.c” file to add a new feature to the application (monitor switches S3 and S6 and send commands to toggle PICDEM FS USB LEDs D3 + D4 respectively when pressed).

       Hints:

       1. You might observe that there are state transitions defined by the DEMO_STATE enumeration that are not being used.
       2. The command-response protocol for the PICDEM™ FS USB Demo Board is described in **Appendix B – MCHPFSUSB “Demo” Firmware Command Reference**.
       3. You **must** update the previous state transition value from “DEMO_STATE_GET_TEMPERATURE” to “DEMO_STATE_SEND_SET_LED” before this section of code will even be called.
7. Solution workspaces are provided:
    - PIC24: C:\RTC\COM3202\Lab1\Solution\Lab1 – C30.mcw
    - PIC32: C:\RTC\COM3202\Lab1\Solution\Lab1 – C32.mcw

Summary:

This lab demonstrates an easy way to design a simple USB Host application. It uses the Microchip “Generic” client driver by configuring the Microchip USB Embedded Host stack using simple “USBConfig.exe” graphical tool.

The application initializes the USB framework by calling “USBInitialize()”. It then waits, polling for the device to attach. All the while, it must maintain the USB framework by calling “USBTasks()” in its main loop. Once the device has attached to the host, the application switches between non-blocking “tasks” in sequence, using the Microchip “Generic” driver to send command packets to the device and receive response packets from the device.

During this exercise, you’ve had the opportunity to use the PIC24 or PIC32 microcontroller of your choice along with the Explorer 16 board and the USB PICtail™ Plus Daughter Board to run a host application for the Microchip PICDEM™ FS USB Demo Board. You had the opportunity to debug the application and observe some of the potential issues that might arise. If time permitted, you may also have had the opportunity to expand the application with new features.

Lab 2 – Implement Driver (Polled)

Purpose:

The primary objective of this lab is to understand the polled implementation of the Generic client driver and how it relates to the TPL and Client Driver tables. This lab uses its own implementation of the Generic client driver, rather than the pre-defined driver distributed with the Microchip USB Framework. The basic framework of the Generic client driver has been implemented for you. Your task is to complete key sections to make the driver operational.

Basic Objectives:

1. Build the application
2. Define the Client Driver table
3. Define the TPL table
4. Implement the Generic client driver
5. Test the driver

Intermediate Objective:

6. Expand the Client Driver table

Advanced Objective:

7. Expand the API & use the new capability

Procedure:

1. Build the application
   a. Ensure that the hardware is connected as described in Lab 1 step 1.
   b. Open the Lab 2 MPLAB workspace that is appropriate project for the MCU you selected.
      - PIC24: C:\ RTC\COM3202\Lab2\Lab2 – C30.mcw
      - PIC32: C:\ RTC\COM3202\Lab2\Lab2 – C32.mcw
   c. Attempt to build the application and observe the errors reported. You should see the following message:

      "Application must define support mode in usb_config.h".

      Note: This message is generated by a test placed in the "usb_common.h" header file to protect you from attempting to build the USB framework without selecting host, device or OTG mode of operation.
   d. Open the "usb_config.h" file and search for the instructions for "LAB2 Step 1d" in the comment block beginning at line 42. Follow the directions to define the support mode and several other USB framework parameters.
   e. Attempt to build the application and observe the errors reported. You should see several undefined references to the "usbClientDrvTable" symbol.

2. Define the client driver table.
    a. Open the “main.c” file and follow the “LAB2 Step 2a” instructions in the comment block at line 212 to define the client driver table.
    b. Attempt to build the application again and observe the errors reported. You should see several undefined references to the “usbTPL” symbol.
3. Define the TPL table.
    a. Open the “main.c” file and follow the “LAB2 Step 3a” instructions in the comment block at line 244 to define the TPL table.
    b. Attempt to build the application again. At this time, it should build cleanly.
4. Implement the Generic client driver (4a-4d: fix the client driver initialization call back function; 4e-4i: enable the client driver’s tasks routine to be called)
    a. Program the application to the microcontroller and run it. Then connect the device and observe that it appears to be enumerated (as indicated by the alternate flashing of LEDs D1 and D2 on the PICDEM™ FS USB Demo Board), but the host does not appear to recognize that it has been attached.
    b. Look at the “CheckForNewAttach()” routine in the “main.c” file at line 416 and observe that this routine calls the Generic API routine “USBHostGenericGetDeviceAddress()” to look for a non-zero device address to identify that the device has been attached.
    c. Open the “usb_client_generic.c” file and look at the “USBHostGenericGetDeviceAddress()” routine at line 307 and observe that this routine gets the address from the Generic driver’s state variable “gc_DevData.ID.deviceAddress” (which itself is initialized during the driver’s initialization)
    d. Go to line 121 and follow the “LAB2 Step 4d” instructions in the comment block to complete the driver’s initialization routine (“USBHostGenericInit( )”), which saves the device address, VID, PID and initializes the driver’s state flags. This will allow the application to identify that the device has been attached.
    e. Re-build and program the application to the microcontroller and run it. Observe that although the application now recognizes that the device has been attached, it does not appear to be exchanging any data with the device (Explorer LEDs D5+D6 = ON – the state machine is stuck waiting for device firmware verification. See function “BlinkStatus ( )” for details)
    f. Place a breakpoint in the “USBHostGenericTasks()” routine on line 526 in the “usb_client_generic.c” file. Reset/run the application.
    g. Observe that the breakpoint is not hit. This means that “USBHostGenericTasks()” is not being called.
    h. Follow the “LAB2 Step 4h” instructions in the comment block starting at line 88 of “usb_config.h” to support calling of the “USBHostGenericTasks()” routine.
    i. Build the application, program it to the microcontroller, and run it. (Leave the breakpoint in place). Observe that the “USBHostGenericTasks()” routine breakpoint stops the program.

5. Test the driver

    a. Delete all breakpoints, reset and run the program. At this point, the application should work as expected.

    b. Verify that the device firmware version number is displayed correctly (it should read "Dev 1.0") and test the temperature sensor and potentiometer displays. Also test to see if buttons S3 and S6 on the Explorer 16 board control LEDs D3 and D4 on the PICDEM™ FS USB Demo Board.

6. Expand the Client Driver table. (Intermediate)

    a. Follow the "LAB2 Step 6a" directions in the comment block located at line 275 of "main.c" to expand the client driver table. Build/program the application.

    Hint: Don't forget to add a comma (',') after the original entry in the table.

    b. Set a breakpoint at line 118 in function USBHostGenericInit() in "usb_client_generic.c" Observe that the "flags" value from the second entry in the Client Driver table has been passed into the initialization routine parameter `"flags"`

    This method can be used to make a single client driver support multiple devices (or types of devices). Instead of just changing the Client Driver index number in the TPL table, you would normally add a second entry in the TPL with the VID and PID of the second device (or the class information for the second type of device). This second TPL entry would then use the index to the second entry in the Client Driver table. The client driver would then use the "flags" parameter to identify the second device.

    Alternately (and perhaps more normally), you could include two Client Drivers in your application (one for each type of device). Then, the second entry in the Client Driver table would contain the names of the initialization and event handling routines of the second driver. You would also need a second TPL entry, identifying the second device (or class of devices) that contained the index to the second Client Driver table entry.

7. Expand the API & use the new capability. (Advanced)

    The objective of this step is to add a new feature to the Generic driver and expand the API to support this feature. Then, test and demonstrate this feature from the application.

    a. Follow the "LAB2 Step 7a" instructions at line 121 in "usb_client_generic.h" (the Generic driver's API header) to add a new data item to the driver context data structure.

    b. Follow the "LAB2 Step 7b" instructions at line 147 in "usb_client_generic.c" to retrieve the transfer type for the data endpoints in the Generic device.

    Hints:

    - Appendix C: USB Descriptors shows the structure of the USB descriptors.
    - The "Endpoint" descriptor contains the "bmAttributes" field that contains the transfer type information.
    - You may find useful definitions of Endpoint Transfer Types in the "Microchip\Include\USB\usb_ch9.h" file.

c. Follow the “LAB2 Step 7c” instructions near line 575 in “usb_client_generic.h” to add a new API macro to allow the application access to the endpoint attributes data.

   Hint: Be sure to use the “API_VALID” macro to validate the device address.

d. Follow the “LAB2 Step 7d” instructions near line 625 in “main.c” to access the transfer type data and display either “B” for bulk, or “I” for interrupt type to character 16, row 1 of the LCD.

8. Solution workspaces are provided:
   - PIC24: C:\RTC\COM3202\Lab2\Solution\Lab2 – C30.mcw
   - PIC32: C:\RTC\COM3202\Lab2\Solution\Lab2 – C32.mcw

Summary:

The host layer enumerates the device, looks up the appropriate driver in the TPL table, and uses the Client Driver table to call the driver’s “Initialize()” routine. In this lab, you’ve gotten a chance to explore the relationship between these two tables and how the host layer uses them to initialize the driver.

You have also had the opportunity to see how the application accesses the device, using the client driver’s API routines, as well as how the client driver provides access to the device using the Host layer’s CDI routines.

This lab has demonstrated that, using table-driven methods and a polling-based implementation, the host layer can manage multiple client drivers to provide access to any type of USB device for which a client driver has been written.

Lab 3 – Event-Driven Driver

Purpose:

The primary objective of this lab is to understand the event-driven implementation of the Generic client driver and how using it is different from using a polled driver. The lab includes modified versions of both the client driver and application. Your task will be to implement the event-driven features.

Basic Objectives:
1. Build the application
2. Implement the application's event-handling routine
3. Implement the event-driven Generic client driver

Intermediate Objective:
4. Add missing feature

Advanced Objective:
5. Add unimplemented feature

Procedure:

1. Build the application
   a. Ensure that the hardware is connected as described in Lab 1 step 1.
   b. Open the Lab 3 MPLAB workspace that is appropriate for the MCU you selected.
      - PIC24: C:\ RTC\COM3202\Lab3\Lab3 – C30.mcw
      - PIC32: C:\ RTC\COM3202\Lab3\Lab3 – C32.mcw
   c. Attempt to build the application and observe the errors reported. You should see the following message:

      "Generic client driver requires transfer event (USB_ENABLE_TRANSFER_EVENT) support."

      Note: This error message is generated by a build-time test purposely placed into the Generic client driver's API header ("usb_client_generic.h").
   d. Open the "usb_config.h" file and follow "LAB3 Step 1d" the directions in the comment block beginning at line 76 to enable transfer events.
   e. Attempt to build the application again and observe the errors reported. You should see references to the following undeclared identifiers:
      - EVENT_GENERIC_DETACH
      - EVENT_GENERIC_TX_DONE
      - EVENT_GENERIC_RX_DONE

      These are Generic-driver-specific events.

      Note: There is one more (EVENT_GENERIC_ATTACH). However, its use is "commented out" at this time.

f. Open the Generic client driver's API header ("usb_client_generic.h") and follow the "LAB3 Step 1f" directions in the comment block beginning at line 75 to extend the USB_EVENT enumeration with the Generic-driver-specific events.

g. Attempt to build the application again. At this time, it should build cleanly.

h. Program the application to the microcontroller and run it, then connect the device. Observe that it appears to be enumerated (as indicated by the alternate flashing of LEDs D1 and D2), but the host application does not appear to recognize that it has been attached (as indicated by the un-lit LEDs D5 and D6 on the Explorer16 board – per BlinkStatus( ), this indicates the state machine is IDLE).

2. Implement the application's event-handling routine

a. Place a breakpoint on line 1266 of main.c at the top of the switch statement in the "usb_custom_host_demo_EventHandler()" routine.

b. Restart and run the application and observe that the breakpoint is never hit. This indicates that the application's event-handling routine is not being called.

c. Open "usb_config.h" and follow the "LAB3 Step 2c" directions at line 93 to identify the application's event-handling routine to the USB Embedded Host stack.

Note: Leave the breakpoint from step 2a in place.

d. Re-build, program, and run the application. Observe that the application's event-handling routine is now being called.

e. Remove the breakpoint from line 1266 of main.c, reset the application and run it again. Observe that the application still does not appear to recognize a device connection.

3. Implement the event-driven Generic client driver

a. Open "usb_client_generic.c" and follow the "LAB3 Step 3a" directions in the comment block starting at line 120 to enable the client driver to call the application's event-handling routine and pass it the "EVENT_GENERIC_ATTACH" event when the device has been attached to the host.

b. Re-build, program and run the application. Observe that the application still does not appear to recognize a device connection.

c. Open "main.c" and follow the "LAB3 Step 3c" directions in the comment block starting at line 1274 to enable the application's event-handling routine to respond to the "EVENT_GENERIC_ATTACH" event. Re-build and program the application.

d. Place a breakpoint in "main.c" on line 1279, the call to the "DemoAttachEventHandler()" routine in the application's event-hander. Run the application.

e. When debugger hits the breakpoint, open the call stack window (select "Call Stack" from "View" on the main MPLAB menu). You should see the following call stack:

- "usb_custom_host_demo_EventHandler()", called from…
- "USBHostGenericInit()", called from…
- "USBHostTasks()", called through the "USBTasks()" macro from…
- main()

This illustrates how the application's event-handling routine is called "up" the layers of theUSB framework from the USB Host layer, through the client driver. It also illustrates that this is ultimately the result of the main application calling the "USBTasks()" macro, which includes a call to "USBHostTasks()".

f. Remove the breakpoint from line 1279. Step into the “DemoAttachEventHandler()” routine. Observe that, if the PICDEM™ FS USB device is identified, the device’s USB address is recorded (because it’s needed by the Generic driver’s API routines) and the application state is changed to the “DEMO_STATE_GET_DEV_VERSION” state.

   Note: The application does not necessarily need to check the device’s VID & PID unless it needs to identify different devices. The Embedded Host layer has already verified the device’s VID & PID by looking it up in the TPL table. This is just done for demonstration purposes to illustrate that the information is provided with the attach event.

g. Run the application. Note now that the host application recognizes the attachment as indicated by the alternate flashing of LEDs D5 & D6 on the Explorer16 board. Note however that there appears to be no data from the PICDEM FS USB being displayed on the LCD.

h. In “main.c” find the state transition code for the “DEMO_STATE_GET_DEV_VERSION” state at line 907. Observe that, when this case is executed, the “ManageDemoState()” routine calls the “SendDeviceCommand()” routine (defined at line 373), which ultimately calls the Generic client driver “USBHostGenericWrite()” API routine.

i. Open “usb_client_generic.c” and find the implementation of the “USBHostGenericWrite()” routine at line 245. Observe that this routine calls the “USBHostWrite()” CDI (Client Driver Interface) routine (at line 280) implemented by the USB Embedded Host layer.

   **In event-driven implementations (when “USB_ENABLE_TRANSFER_EVENT” is defined as we did in step 1, above), calls to host-layer read or write routines will result in an “EVENT_TRANSFER” event being sent to the client driver when the transfer has finished.

j. In “usb_client_generic.c”, follow the “LAB3 Step 3j “ directions in the comment block starting at line 192 to call the application’s event handler when either a “Tx” or “Rx” transfer has finished.

k. Re-build, program, and run the application. The application should now correctly display the device’s firmware version number and the temperature data.

   Note The potentiometer resistance value will not be shown yet.

4. Add missing feature

   a. Open “main.c” and follow the “LAB3 Step 4a “ directions in the comment block at line 922 to send the command to the device to read the potentiometer.

   b. Re-build, program, and run the application and observe that the application displays a “Bad Command Sent” error message on the LCD.

      Note: If a USB analyzer is available, you can use it to capture the bus traffic and verify that the command sent was correct. The command values and format are given in Appendix B: PICDEM FS USB “Demo” Firmware Command Reference.

   c. Set a breakpoint in “main.c” at line 1293 where the “TxDoneEventHandler()” routine is called in response to the application receiving a “EVENT_GENERIC_TX_DONE” event. Reset and run the application.

   d. Step into the “TxDoneEventHandler()” routine and observe that it was the “READ_VERSION” command that just finished being transmitted. Let the application run again and identify the next command to finish. Repeat this process to identify that the “RD_POT” command is not currently supported by the “TxDoneEventHandler()”.

e. Follow the “LAB3 Step 4e” directions in “main.c” in the comment block at line 1088 to enable the application to correctly respond to the “EVENT_GENERIC_TX_DONE” event for the “RD_POT” command.

f. Remove the breakpoint from “main.c” line 1293, re-build, program, and run the application. Observe that the application now gives an “Unknown Resp Err” message on the LCD.

g. Follow the “LAB3 Step 4g” directions in the comment block at line 1188 in “main.c” to enable the application to correctly respond to the “EVENT_GENERIC_RX_DONE” event and display the POT data.

h. Re-build, program, and run the application. It should now correctly display the potentiometer data on the LCD.

5. Add unimplemented feature (SET_LED)

   a. Follow the “LAB3 Step 5a” instructions in the comment block at line 955 in “main.c” to add a new feature to the application.

      Hints:

      b. Before starting, you will need to edit the RD_POT state transition on line 1195 in main.c to correctly advance the main state variable to the new state (i.e. change the state transition to DEMO_STATE_SET_LED, which has already been defined)

      c. Function UpdateDeviceLEDs( ) is provided for you, however, you will need to add it to the ManageDemoState( ) function as an “event-driven” state (i.e. call the function; set the state to DEMO_STATE_RUNNING; let the events happen; use the Tx & Rx event handlers to advance to the next state).

      d. The ‘Tx’ and ‘Rx’ transfer-done event handling routines (UPDATE_LED) have already been implemented for you.

6. Solution workspaces are provided:

   - PIC24: C:\RTC\COM3202\Lab3\Solution\Lab3 – C30.mcw
   - PIC32: C:\RTC\COM3202\Lab3\Solution\Lab3 – C32.mcw

Summary:

This lab demonstrated an event-driven implementation of the “Generic” client driver and the application that uses it. During this lab, you had the opportunity to observe that the event-driven method performs calls “up” the firmware-layer stack, from the Host layer, through the Client Driver, to the application. This eliminates the “double polling” need for both the application and the driver to poll for status changes on the USB.

Although this lab only demonstrated a single client driver, the event-driven method will “scale” more efficiently since it only needs to call a client driver when its associated device is attached to the host. This is different from the polled method, which must call each client driver through the “USBTasks()” macro (and potentially the “USBInitialize()” macro as well) even if the device is not attached.

Finally, you are encouraged at this time to compare the solutions to all three labs. Observe that each implementation (whether polled or event-driven) has an event-handling routine. This is because there are events that sent to the application and/or the client driver that are not associated the completion of a specific transfer. Observe that the application’s event-handling routine is optional (although normal operation of a host will require it), but the client driver requires an event-handling routine to receive the “detach” event as a minimum.

Part 3 – Designing a Mass Storage Class, Full-Speed USB Embedded Host Application

Overview:

You will be completing a "terminal shell" application on the PIC which will incorporate various Microchip MDD (Memory Disk Drive) File system APIs to manipulate files on a USB Thumb drive and display the results in a PC terminal emulator (HyperTerminal or other).

Lab 4 – Creating and Deleting Files

Purpose:

In this lab, you will add three commands to the shell program:

Name	Function	Syntax	ARG1	ARG2
MKFILE	Creates an empty file	**MKFILE ARG1**	The name of the file to be created, in 8.3 format	-
REN	Renames an existing file	**REN ARG1 ARG2**	The old file name, in 8.3 format	The new file name, in 8.3 format
DEL	Deletes a file	**DEL ARG1**	The name of the file to be deleted, in 8.3 format	-

Procedure:

1. Insert a FAT16/32 formatted Thumb Drive into connector J4 on the USB PICTail Plus Board.

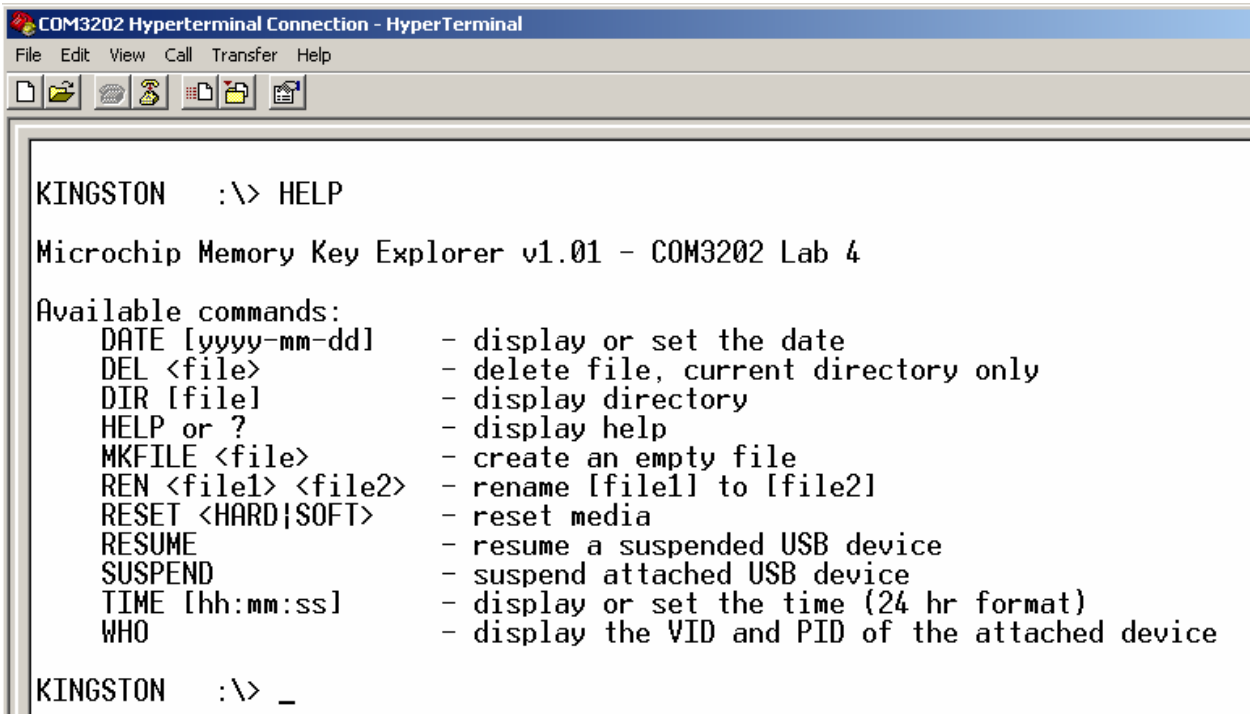
2. Open a HyperTerminal window. The HyperTerminal file **COM3202 Hyperterminal Connection.ht** included with your lab folders (in **C:\RTC\COM3202**) contains the correct settings: 57600 baud, 8 data bits, no parity, 1 stop bit, no flow control. This console will be used to communicate with the shell program on the PIC.

3. Start MPLAB.

4. Open the Lab 4 MPLAB workspace that is appropriate for the MCU you selected.
   - PIC24: C:\ RTC\COM3202\Lab4\Lab4 – C30.mcw
   - PIC32: C:\ RTC\COM3202\Lab4\Lab4 – C32.mcw

5. Compile, build, and program your Explorer 16 board.

6. Familiarize yourself with the console program. The **HELP** command will provide information about commands and syntax. Note that the **MKFILE**, **REN**, and **DEL** commands won't work correctly until you complete the lab.



7. Open the source file **Lab4.c** in MPLAB.

8. Modify each function where indicated by comment blocks. You can test each function and command separately. The functions you'll be modifying are:

    a) Lab4PartA: This function is called by the **MKFILE** command. It will create a file based on the path name passed in by the user.
    b) Lab4PartB: This function is called by the **REN** command. It will rename an existing file.
    c) Lab4PartC: This function is called by the **DEL** command. It will delete a file based on the path name passed in by the user.

Refer to the next section "Useful Functions (Lab4)" for a listing of MDD API functions you can use to complete these sections.

9. Re-compile, build, and program your Explorer 16 board.

10. Verify that the **DEL**, **MKFILE**, and **REN** functions work correctly. You can use the **DIR** function to view the contents of your thumb-drive.

11. Solution workspaces are provided:
    - PIC24: C:\RTC\COM3202\Lab4\Solution\Lab4 – C30.mcw
    - PIC32: C:\RTC\COM3202\Lab4\Solution\Lab4 – C32.mcw

Useful Functions (Lab4):

FSfopen

Opens a file on the device and associates an FSFILE structure (stream) with it.

Syntax

```
FSFILE * FSfopen (const char * fileName, const char *mode )
```

Parameters

filename: A null terminated char string specifying the file name. The file name must be fewer than 8 characters, followed by a radix (.) followed by an extension containing three of fewer characters. The file name cannot contain any directory or drive letter information.

mode: A null terminated string specifying the file operation.
The valid strings are:

r Read Only

w Write
If a file with the same name exists, it will be overwritten
No reads allowed

a Append
If the file exists, the current location will be set to the end of the file.
Otherwise, the file will be created.
No reads allowed

Return Values

A pointer to an FSFILE structure to identify the file in subsequent operations
NULL if the specified file could not be opened

Precondition

FSInit is called

Example

```
FSFILE * fPtr;
fPtr = FSfopen("FILE.TXT", "w" );
if (fPtr == NULL)
{
       // Error handling
}
```

Useful Functions (Lab4) Continued:

FSfclose

Closes an opened file. Saves root directory and FAT information for that file.

Syntax

```
int FSfclose( FSFILE *stream )
```

Parameters

stream: A pointer to a FSFILE structure obtained from a previous call of FSfopen

Return Values

Returns 0 on success
Returns EOF (-1) on failure

Precondition

stream is a valid FSFILE pointer.

Example

```
if( FSfclose( stream ) != 0 )
{
	// Failed to close the file.
}
```

FSrename

The FSrename function will allow the user to rename files and directories

Syntax

```
int FSrename (const char *fileName, FSFILE * fo)
```

Parameters

fileName: The new name of the file
fo: The file to rename

Return Values

Returns 0 on success
Returns EOF (-1) on failure

Precondition

FSInit is called successfully. The file to be renamed is opened successfully.

Example

```
FSFILE * fptr;
fptr = FSfopen ("FILE.TXT", "r");
if( FSrename("NEWNAME.TXT", fptr) != 0 )
{
	// Error Handling
}
```

Useful Functions (Lab4) Continued:

FSremove

The FSremove function deletes the file identified by filename. If the file is opened with FSfopen, it must be closed before calling FSremove.

Syntax

```
int FSremove (const char * filename)
```

Parameters

filename: A pointer to a null terminated string

Return Values

Returns 0 on success
Returns EOF (-1) on failure

Precondition

FSInit is called successfully

Example

```
if( FSremove("FILE.TXT") != 0 )
{
       // Error Handling
}
```

Lab 5 – Reading and Writing Files

Purpose:

In this lab, you will add three commands to the shell program:

Name	Function	Syntax	ARG1	ARG2
TYPE	Displays the contents of a file, in ASCII text.	**TYPE ARG1**	The file to display information from	-
COPY	Copies data from one file into a second file, or from the console to a file.	**COPY ARG1 ARG2**	The name of the file that will be copied	The destination file. This file will be created or overwritten.
		COPY CON ARG2	"**CON**" specifies that the input comes from the console.	
LOG	Logs data from a potentiometer or temperature sensor to a file. Specify the file name as a *.CSV file to easily view the data. The LOG command is already complete for the temperature logging function.	**LOG POT ARG2**	"**POT**" specifies that the input comes from the potentiometer	The file the data will be stored in. If a file with this name exists, it will be overwritten.
		LOG TMP ARG2	"**TMP**" specifies that the input comes from the temperature sensor.	

Procedure:

1. Close the previous workspace by selecting *File> Close Workspace…* from the main menu.

2. Open the Lab 5 MPLAB workspace that is appropriate for the MCU you selected.
   - PIC24: C:\ RTC\COM3202\Lab5\Lab5 – C30.mcw
   - PIC32: C:\ RTC\COM3202\Lab5\Lab5 – C32.mcw

3. Open the source file **Lab5.c.**

4. Modify each function where indicated by comment blocks. You can test each function and command separately. The functions you'll be modifying are:

    a) Lab5PartA: This function is called by the **TYPE** command. It will read one character from a file passed in by the user.
    b) Lab5PartB: This function is called by the **COPY CON** function. It will write one character at a time into a file based on input from the user from the console.
    c) Lab5PartC: This function is called by the **COPY** function. It will read data from a file specified by the user and copy it into another file specified by the user.
    d) Lab5PartD: This function is called by the **LOG POT** function. It will write text and data based on the readings from a potentiometer to a file.

Refer to the next section "Useful Functions (Lab5)" for a listing of MDD API functions you can use to complete these sections.

5. Compile, build, and program your Explorer 16 board.

6. Verify that the **COPY**, **LOG**, and **TYPE** functions work correctly.

7. Solution workspaces are provided:
    - PIC24: C:\RTC\COM3202\Lab5\Solution\Lab5 – C30.mcw
    - PIC32: C:\RTC\COM3202\Lab5\Solution\Lab5 – C32.mcw

Useful Functions (Lab5): **See Next Page**

Useful Functions (Lab5):

FSfread

Reads data from the previously opened file. FSfread reads 'n' data objects, each of length 'size' bytes from the given file 'stream.' The data is copied to the buffer pointed by 'ptr.' The total number of bytes transferred is n * size .

Syntax

```
size_t FSfread( void *ptr, size_t size, size_t n, FSFILE *stream )
```

Parameters

```
ptr:          pointer to buffer to hold the data read
size:         length of object in bytes
n:            number of objects to read
stream:       stream pointer to file opened with read (r) mode
```

Return Values

```
On success FSfread returns the number of objects (not bytes) actually read
On End-Of-File or error it returns 0
```

Precondition

```
File is opened is read mode
```

Example

```
//Read 100 objects of size 10 bytes each
NumberOfObjects = FSfread( bfr, 10, 100, pFile );
if( NumberOfObjects < 100 )
{
       // Function did not read all 100 objects.  Either an error
       // occurred or we reached the end of the file (EOF).
}
else
{
       //read all 100 objects
}
```

Useful Functions (Lab5) Continued:

FSfwrite

Writes data to the previously opened file. FSfwrite writes 'n' data objects, each of length 'size' bytes to the given file 'stream.' The data is copied from the buffer pointed to by 'ptr.' The total number of bytes transferred is n * size.

Syntax

```
size_t FSfwrite( const void *ptr, size_t size, size_t n, FSFILE *stream )
```

Parameters

```
ptr:        pointer to buffer holding data to write
size:       length of object in bytes
n:          number of objects to write
stream:     stream pointer to file opened with write (w) or append (a) mode
```

Return Values

On success FSfwrite returns the number of objects (not bytes) actually written.
On error it returns a short count or 0

Precondition

File is opened in write (w) or append (a) mode

Example

```
// Write twenty five-byte objects from ptr to pFile
if( FSfwrite( ptr, 5, 20, pFile ) != 20 )
{
      // not all items were written
}
```

FSfprintf

The FSfprintf function will write a formatted string to a file.

Syntax

```
int FSfprintf (FSFILE *fptr, const char * fmt, ...)
```

Parameters

```
fptr:       Pointer to a file to write to.
fmt:        The string to write (specified in ROM).
...:        Format specifier arguments.
```

Return Values

Returns the count of characters written on success.
Returns -1 otherwise.

Precondition

The file to be written to has been opened successfully.

Useful Functions (Lab5) Continued:

Remarks

The FSfprintf function formats output, passing the characters to the specified stream. The format string is processed one character at a time and the characters are output as they appear in the format string, except for format specifiers. A format specifier is indicated in the format string by a percent sign, %; following that, a well-formed format specifier has the following components. Except for the conversion specifier, all format specifiers
are optional. Depending on the format specifier used, an argument may be read from the function call, formatted as specified by the user, and inserted in the string.

1. Flag Characters:
   - '-' : The result of the format conversion will be left justified.
   - '+' : By default, a sign is only prefixed to a signed conversion if the result is negative. If this flag is included, a + sign will be prefixed if the result of a signed conversion is positive.
   - '0' : This flag will prefix leading zeros to the result of a conversion until the result fills the field width. If the - flag is specified, the 0 flag will be ignored. If a precision is specified, the 0 flag will be ignored.
   - ' ' : The space flag will prefix a space to the result of a signed conversion if the result is positive. If the space flag and the + flag are both specified, the space flag will be ignored.
   - '#' : This flag indicates the .alternate form. of a conversion. For the '0' conversion, the result will be increased in precision, such that the first digit of the result will be 0. For the 'x' conversion, a "0x" will be prefixed to the result. For the 'X' conversion, a "0X" will be prefixed to the result. For the 'b' conversion, a "0b" will be prefixed to the result. For the 'B' conversion, a "0B" will be prefixed to the result.

2. Field Width:

   The field width specifier follows the flag specifiers. It determines the minimum number of characters that result from a conversion. If the result is shorter than the field width, the result is padded with leading spaces until it has the same size as the field width. If the '0' flag specifier is used, the result will be padded with leading zeros. If the '-' flag specifier is used, the result will be left justified and will be followed by trailing spaces.

   The field width may be specified as an asterisk character (*). In this case, a 16-bit argument will be read from the list of format specifiers to specify the field width. If the value is negative, it is as if the '-' flag is specified, followed by a positive field width.

3. Field Precision:

   The field precision specifies the minimum number of digits present in the converted value for integer conversions, or the maximum number of characters in the converted value for a string conversion. It is indicated by a period (.), followed by an integer value or by an asterisk (*).

   If the field precision is not specified, the default precision of 1 will be used. If the field precision is specified by an asterisk character, a 16-bit argument will be read from the list of format specifiers to specify the field precision.

Useful Functions (Lab5) Continued:

4. Size Specification:

The size specification applies to any integer conversion specifier or pointer conversion specifier. The integer conversion specifiers are as follows: the size specifier will determine what type of argument is read from the format specifier list. For the n conversion, the size specifier for each pointer type corresponds to the specifier for that data type. So, to convert something to a long long pointer, you would use the specifier for a long long data type with the n conversion.

TABLE B-3: SIZE SPECIFIERS

Argument Type	C18	C30
signed char, unsigned char	hh	hh
short int, unsigned short int	h	h
short long, unsigned short long	H	-
intmax_t, uintmax_t	j (32-bit)	j (64-bit)
long, unsigned long	l	l
long long, unsigned long long	-	q
size_t	z	z
sizerom_t	Z	-
ptrdiff_t	t	t
ptrdiffrom_t	T	-

5. Conversion Specifiers:

- c: The int argument will be converted to an unsigned char value and the character represented by that value will be written.
- d,i: The int argument is formatted as a signed decimal.
- o: The unsigned int argument will be converted to an unsigned octal.
- u: The unsigned int argument will be converted to an unsigned decimal.
- b,B: The unsigned int argument will be converted to an unsigned binary.
- x: The unsigned int argument will be converted to an unsigned hexadecimal. The characters, a, b, c, d, e and f, will be used to represent the decimal numbers, 10-15.
- X: The unsigned int argument will be converted to an unsigned hexadecimal. The characters, A, B, C, D, E and F, will be used to represent the decimal numbers, 10-15.
- s: Characters from the data memory array of char argument are written until either a terminating '\0' character is seen ('\0' is not written) or the number of chars written is equal to the precision.
- S: Characters from the program memory array of char arguments are written until either a terminating '\0' character is seen ('\0' is not written) or the number of chars written is equal to the precision. In C18, when outputting a far rom char *, make sure to use the H size specifier (%HS).
- p: The pointer to the (data or program memory) argument is converted to an equivalent size unsigned integer type and that value is processed as if the x conversion operator had been specified. In C18, if the H size specifier is present, the pointer is a 24-bit pointer; otherwise, it is a 16-bit pointer.
- P: The pointer to void the (data or program memory) argument is converted to an equivalent size unsigned integer type and that value is processed as if the X conversion operator had been specified. In C18, if the H size specifier is present, the pointer is a 24-bit pointer; otherwise, it is a 16-bit pointer.

Useful Functions (Lab5) Continued:

```
      n:     The number of characters written so far shall be stored in the location
             referenced by the argument, which is a pointer to an integer type in data
             memory. The size of the integer type is determined by the size specifier
             present for the conversion, or a 16-bit integer if no specifier is
             present.
      %:     A literal percent sign will be written.

If the conversion specifier is invalid, the behavior is undefined.
```

Example

```
unsigned long long hex = 0x123456789ABCDEF0;
FSfprintf (fileptr, .This is a hex number:%#20X%c%c., 0x12ef, 0x0D, 0x0A);
FSfprintf (fileptr, .This is a bin number:%#20b\r\n., 0x12ef);
FSfprintf (fileptr, .%#26.22qx., hex);
// Output:
// This is a hex number:              0X12EF
// This is a bin number:     0b1001011101111
//   0x000000123456789abcdef0
```

Part 4 – Using the USB Thumb Drive Bootloader Application

Lab 6 – Update firmware using the Thumb Drive Bootloader

Overview:

The basic operation of the Thumb Drive Boot Loader is to allow a USB Host-enabled device to enumerate a formatted USB Thumb Drive, access a specifically named file containing properly prepared application firmware, and program it into executable Flash memory. **See Appendix D for a detailed description of the thumb drive boot loader architecture and core APIs.**

For this lab, you will first build/run an existing application (the "USB Generic Device Demo"). Then, you will take the necessary steps to prepare this application for use with the USB thumb drive boot loader. Finally, you will program the boot loader into the device, and attempt to load/run the modified application from the thumb drive.

At this writing, the boot-loaders are sourced from 2 different USB frameworks (see below).

Folder Structure:

```
C:\RTC\COM3202\Lab6

     .\PIC24
          \Bootloader Application
          \USB Generic Device Demo
     .\PIC32
          \Bootloader Application
               \Microchip
          \USB Generic Device Demo
     .\USB Generic Device Demo Support
          \Pdfsusb
          \Driver and inf
```

Sources For The Demo Projects:

1. PIC24 “Bootloader Application” is based on MCHPFSUSB v2.5a project “USB Host – Bootloaders”
2. PIC24 & PIC32 “USB Generic Device Demo” is based on MCHPFSUSB v2.5a project “USB Device - MCHPUSB - Generic Driver Demo”
3. PIC32 “Bootloader Application” is based on beta release v0.04 and comes with it’s own local copy of PIC32 USB stack files (“\Microchip”)

Boot Loader Restrictions:

FS Format:	FAT 16 or FAT 32
File Format:	Intel Hex record
File Name:	Build-time programmable
USB Support:	Boot loader has its own USB Host stack
Configuration Bits:	Must be defined in the boot loader
Program Flash:	Must be shared with boot loader code

Basic Operation Pseudo Code:

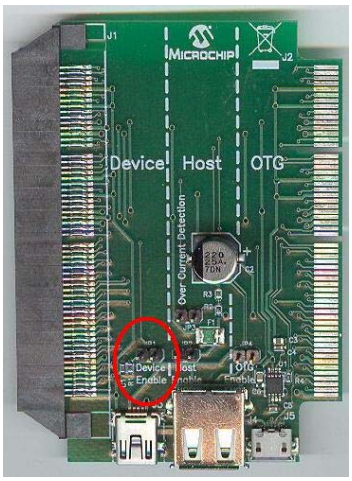
```
execute boot loader startup code
if ( bootstrap condition met)
{
    initialize boot loader USB host stack
    find application firmware file
    parse file and program it to Flash
}

Jump to application startup code
```

Demo Procedure:

Step 1. Develop/Run the "Application" (“USB Generic Device Demo”)

1. Configure the board jumpers for USB Peripheral Operation and connect using USB cable to PC.



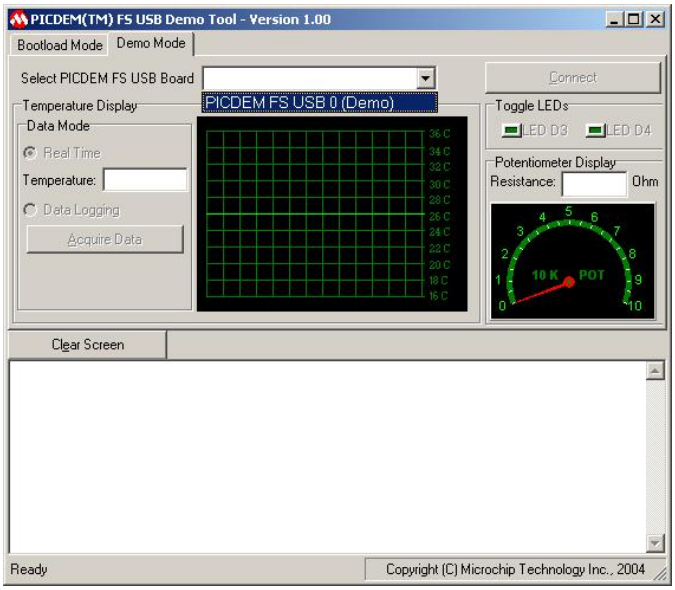
2. Open the MPLAB workspace that is appropriate for the MCU you selected.

- PIC24: C:\RTC\COM3202\Lab6\PIC24\USB Generic Device Demo\USB Generic Device Demo.mcw
- PIC32: C:\RTC\COM3202\Lab6\PIC32\USB Generic Device Demo\USB Generic Device Demo.mcw

3. Program/Run the application, and (if necessary) direct windows to install the custom class driver found in “C:\RTC\COM3202\Lab6\USB Generic Device Demo Support\Driver and inf”

4. Start the PC application "Pdfsusb.exe" in " C:\RTC\COM3202\Lab6\USB Generic Device Demo Support\Pdfsusb".

5. Click “Demo Mode” tab and select” PICDEM FS USB 0 (Demo)”. Then select “Connect”. Verify operation.



Step 2. Rebuild The Application for use with the Bootloader

PIC32 Boot Loader

1. If there is a linker script already attached to the project, remove it from the project by right clicking on it and selecting “remove”.

2. Copy the bootloader application linker file “procdefs.ld” to the demo project folder. The linker file can be found in the “C:\RTC\COM3202\Lab6\PIC32\Bootloader Application\Application Files\Linker Files” folder.

Note: The “procdefs.ld” file is designed to work with the default settings in the PIC32 boot loader’s “boot_config.h” file. If the application’s entry point (physical + virtual address) and/or size is modified, “procdefs.ld” will also need to be modified appropriately.

3. Rebuild the application.

4. Copy the application’s .hex file ("USB Generic Device Demo.hex") to a FAT32 formatted thumb drive and rename it "image.hex".

PIC24 Boot Loader

1. In MPLAB, if there is a linker script already attached to the project, remove it from the project by right clicking on it and selecting “remove”.

2. If the “PIC24 HID Bootloader Remapping.s” file is attached to the project then also remove this file by right clicking on it and selecting remove.

3. Copy the bootloader application linker file “p24FJ256GB110_Host_MSD_Bootloader.gld” and to the application’s project folder. The linker file can be found in the “C:\RTC\COM3202\Lab6\PIC24\Bootloader Application\Application Files\Linker Files” folder.

4. In MPLAB, add this file to the project by right clicking on the “Linker script” folder in the project window, select “Add Files…”. Select the correct file.

5. In MPLAB, add the “MSD Bootloader Remapping.c” file to the project by right-clicking on the “Source Files” folder in the project window. This file can be found in the “C:\RTC\COM3202\Lab6\PIC24\Bootloader Application\Application Files\Interrupt Remapping” folder.

Note: The “p24FJ256GB110_Host_MSD_Bootloader.gld” + “MSD Bootloader Remapping.c” files re-map the application’s entry point + interrupt vectors.

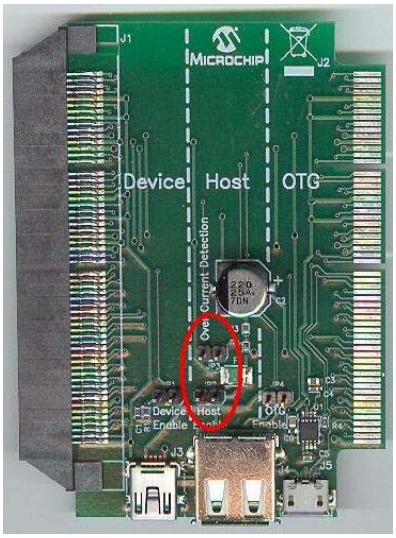
6. Edit the “MSD Bootloader Remapping.c” file as required for the application. Determine which ISRs are used in the application by inspecting the .ivt section of the application’s .map file. A solution file is provided for this application (“MSD Bootloader Remapping_Solution.c”)

7. Rebuild the application.

8. Copy the application's .hex file ("USB Generic Device Demo.hex") to a FAT32 formatted thumb drive and rename it "image.hex".

Step 3. Program the Bootloader Application into the Board & Load Application

1. Remove USB cable from the mini-B connector, and configure the USB Pictail jumper for USB Embedded Host Operation



2. Open the boot loader workspace file that is appropriate for the MCU you selected

- PIC24: C:\RTC\COM3202\Lab6\PIC24\Bootloader Application\PIC24 Thumb Drive Bootloader.mcw
- PIC32: C:\RTC\COM3202\Lab6\PIC32\ Bootloader Application\PIC32 Thumb Drive Bootloader.mcw

3. Build/program the bootloader application into the board

4. Enter Boot Loader Mode

- PIC24: Press/hold MCLR and S3, then release MCLR. LED D5 will illuminate to indicate that we are in bootloader mode.
- PIC32: Press/hold MCLR and S4, then release MCLR. LED D10 will flash to indicate that we are in bootloader mode.

5. Load & Boot the Application

- PIC24: Insert the thumb drive into the USB Type-A receptacle and the boot loader will load and boot the application image. LED D5 will turn off once the application has started.
- PIC32: Insert the thumb drive into the USB Type-A receptacle and the boot loader will load and boot the application image. If it is unable to find the file, D10 will flash rapidly to indicate an error. Otherwise, LED D10 will stop blinking once the application has started.

Step 4. Run the Application

1. Remove the thumb drive,

2. Change USB Pictail jumper configuration back to USB Peripheral (Step 1.1 above) & insert the USB cable into the mini-B plug.

3. Reset the board. Verify that the application enumerates/runs using the Pdfsusb.exe application.

Appendix A – Creating New USB Projects Based on COM3202 Labs/Demos

Project References to USB Framework Stack files are relative. Requires the \Microchip subfolder to be installed at the same level as the project folder.

Example: To create a new USB project based upon project “Lab1” in C:\RTC\COM3202:

1. Copy the sub-folder “\Lab1” to C:\MyUSBProjects
2. Copy the sub-folder “\Microchip” to C:\MyUSBProjects

The project will now build/run.

Appendix B – PICDEM FS USB "Demo" Firmware Command Reference

The source code for the demo may be found in "C:\Microchip Solutions\USB Device - MCHPUSB - Generic Driver Demo\Generic Driver Demo - Firmware\user.c". See the "ServiceRequests" function.

READ_VERSION

Command Byte:	0x00
Synopsis:	Returns the firmware Version
Format:	READ_VERSION, 0x02
Returns:	READ_VERSION, 0x02, MAJOR, MINOR

ID_BOARD

Command Byte:	0x31
Synopsis:	Sets a binary code on LEDs 3 & 4. ID_CODE is 0-3.
Format:	ID_BOARD, ID_CODE
Returns:	ID_BOARD

UPDATE_LED

Command Byte:	0x32
Synopsis:	Provides direct access to turn LED 3 or 4, on or off.
Format:	UPDATE_LED, LED_NUM, LED_STATUS LED_NUM is either 3 or 4. LED_STATUS is either 0 or 1.
Returns:	UPDATE_LED

SET_TEMP_REAL

Command Byte:	0x33
Synopsis:	Resets real time temperature logging and sets the temperature mode to Real Time.
Format:	SET_TEMP_REAL
Returns	SET_TEMP_REAL

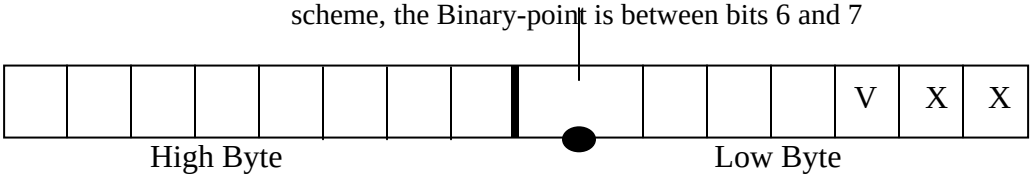
READ_POT

Command Byte:	0x37
Synopsis:	Returns the A2D conversion from the position of the potentiometer. Result is 10 bits, right justified.
Format:	READ_POT
Returns:	READ_POT, ADRESL, ADRESH

READ_TEMPERATURE

Command Byte: 0x34

Synopsis: Returns a single temperature conversion. Temperature returned is a 16 bit number representing degrees C. Bits 0 & 1 are not valid. Bit 2 indicates temperature is valid. The remaining 13 bits represent degrees Celsius, in 1/16th (0.0625) degree increments. In a fixed point coding scheme, the Binary-point is between bits 6 and 7



V – Validity flag. 1 indicates temperature reading is valid.

Format: READ_TEMPERATURE

Returns: READ_TEMPERATURE, TEMPL, TEMPH

SET_TEMP_LOGGING

Command Byte: 0x35

Synopsis: Resets real time temperature logging and sets the temperature mode to Logging.

Format: SET_TEMP_LOGGING

Returns SET_TEMP_LOGGING

READ_TEMP_LOGGING

Command Byte: 0x36

Synopsis: Returns the logged temperatures. When logging is enabled, the board will sample the temperature sensor once every second. This command will read the stored temperature values. a single temperature conversion. Temperature returned is a 16 bit number representing degrees C. Bits 0 & 1 are not valid. Bit 2 indicates temperature is valid. The remaining 13 bits represent degrees Celsius, in 1/16th (0.0625) degree increments. The log is a circular buffer, so once 30 samples are logged, new readings will start back at the beginning of the buffer.

Format: READ_TEMPERATURE

Returns: READ_TEMPERATURE, Sample count, TEMPL0, TEMPH0, TEMPL1, TEMPH1…

Appendix C – USB Descriptors

The following tables from the USB 2.0 specification (see references) show the contents of the descriptors used by the labs. They are reproduced here for reference.

Table 9-8. Standard Device Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	DEVICE Descriptor Type
2	*bcdUSB*	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

Table 9-8. Standard Device Descriptor (Continued)

Offset	Field	Size	Value	Description
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	*bMaxPacketSize0*	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	*idVendor*	2	ID	Vendor ID (assigned by the USB-IF)
10	*idProduct*	2	ID	Product ID (assigned by the manufacturer)
12	*bcdDevice*	2	BCD	Device release number in binary-coded decimal
14	*iManufacturer*	1	Index	Index of string descriptor describing manufacturer
15	*iProduct*	1	Index	Index of string descriptor describing product
16	*iSerialNumber*	1	Index	Index of string descriptor describing the device's serial number
17	*bNumConfigurations*	1	Number	Number of possible configurations

Table 9-10. Standard Configuration Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	CONFIGURATION Descriptor Type
2	*wTotalLength*	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	*bNumInterfaces*	1	Number	Number of interfaces supported by this configuration
5	*bConfigurationValue*	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration
6	*iConfiguration*	1	Index	Index of string descriptor describing this configuration

Table 9-10. Standard Configuration Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in *bMaxPower* to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request (see Section 9.4.5). If a device configuration supports remote wakeup, D5 is set to one.
8	*bMaxPower*	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100 mA). Note: A device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered. A device may not increase its power draw from the bus, when it loses its external power source, beyond the amount reported by its configuration. If a device can continue to operate when disconnected from its external power source, it continues to do so. If the device cannot continue to operate, it fails operations it can no longer support. The USB System Software may determine the cause of the failure by checking the status and noting the loss of the device's power source.

Table 9-12. Standard Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	INTERFACE Descriptor Type
2	*bInterfaceNumber*	1	Number	Number of this interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	*bAlternateSetting*	1	Number	Value used to select this alternate setting for the interface identified in the prior field
4	*bNumEndpoints*	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses the Default Control Pipe.
5	*bInterfaceClass*	1	Class	Class code (assigned by the USB-IF). A value of zero is reserved for future standardization. If this field is set to FFH, the interface class is vendor-specific. All other values are reserved for assignment by the USB-IF.
6	*bInterfaceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bInterfaceClass* field. If the *bInterfaceClass* field is reset to zero, this field must also be reset to zero. If the *bInterfaceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

Table 9-12. Standard Interface Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bInterfaceProtocol*	1	Protocol	Protocol code (assigned by the USB). These codes are qualified by the value of the *bInterfaceClass* and the *bInterfaceSubClass* fields. If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use a class-specific protocol on this interface. If this field is set to FFH, the device uses a vendor-specific protocol for this interface.
8	*iInterface*	1	Index	Index of string descriptor describing this interface

Table 9-13. Standard Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	ENDPOINT Descriptor Type
2	*bEndpointAddress*	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint

Table 9-13. Standard Endpoint Descriptor (Continued)

Offset	Field	Size	Value	Description
3	*bmAttributes*	1	Bitmap	This field describes the endpoint's attributes when it is configured using the *bConfigurationValue*. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved Refer to Chapter 5 for more information. All other bits are reserved and must be reset to zero. Reserved bits must be ignored by the host.

Table 9-15. String Descriptor Zero, Specifying Languages Supported by the Device

Offset	Field	Size	Value	Description
0	*bLength*	1	N+2	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*wLANGID[0]*	2	Number	LANGID code zero
...	...	...	...	...
N	*wLANGID[x]*	2	Number	LANGID code *x*

Appendix D – PIC32 USB Thumb Drive Bootloader Documentation

Here, reproduced in its entirety, is the file "USB Thumb Drive Boot Loader.doc" and "read_me.txt" documentation which is in the PIC32 Thumb Drive Bootloader (beta) v0.04 release package. This is the only current released documentation (other than source code). The Architecture/APIs are identical for both PIC24+PIC32

"USB Thumb Drive Boot Loader.doc"

The USB Thumb Drive Boot Loader enables updating of application firmware from a file located on a USB thumb drive.

Overview
The basic operation of the Thumb Drive Boot Loader is to allow a USB Host-enabled device to enumerate a formatted USB Thumb Drive, access a specifically named file containing properly prepared application firmware, and program it into executable Flash memory.

Restrictions

FS Format:	FAT 16 (FAT 32 later)
File Format:	Intel Hex record
File Name:	Build-time programmable
USB Support:	Boot loader has its own USB Host stack
Configuration Bits:	Must be defined in the boot loader
Program Flash:	Must be shared with boot loader code

Basic Operation Pseudo Code

```
execute boot loader startup code
if ( bootstrap condition met)
{
    initialize boot loader USB host stack
    find application firmware file
    parse file and program it to Flash
}

Jump to application startup code
```

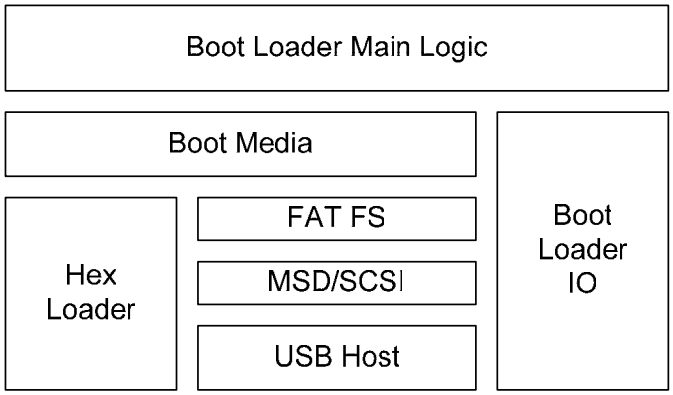
Demo Operation
This demo was designed to work with the Explorer-16 board with a USB PICTail+ daughter board. At the time this was written, it had only been tested using PIC32MX460F512L Processor Interface Module (PIM).

To invoke the boot loader, press S4 on the Explorer-16 board and reset the microcontroller. LED D10 will begin flashing to indicate the boot loader is in control. Ensure that the JP3 (Host Enable) and JP1 (Over Current Detection) jumpers are shorted. Insert a FAT-16 formatted USB thumb drive into J7 (USB Type-A receptacle) with the application firmware image file on it named "image.hex". The application

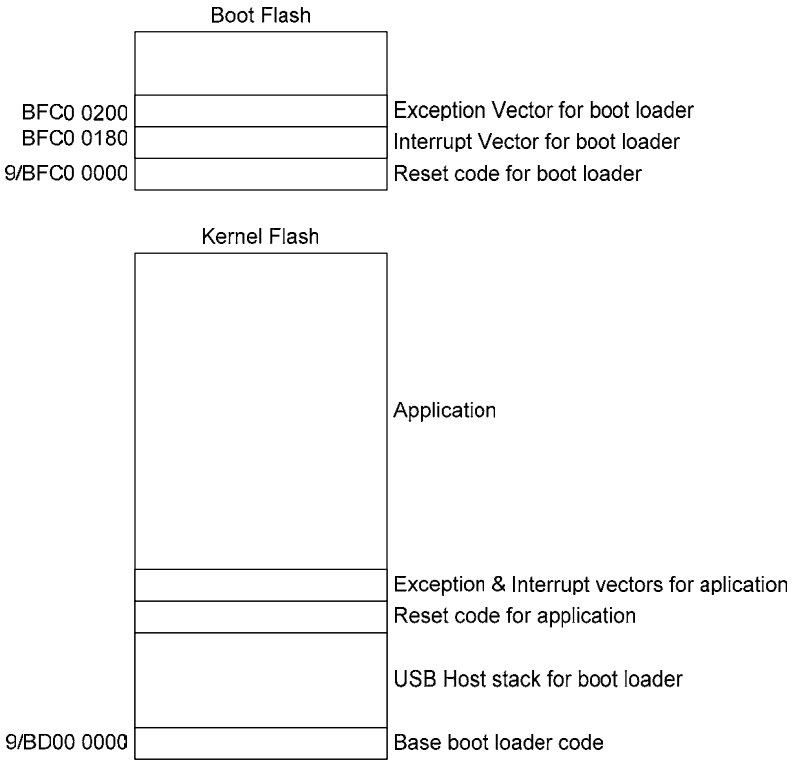
firmware's hex-record file must be built using the provided (application_link_script\procdefs.ld) linker script and must fit within the available program Flash memory.

If the boot loader finds a valid "image.hex" file, it will read the file and program it to the program Flash memory area and launch it. If it does not, LED D10 will begin flashing rapidly to indicate an error.

Architecture



Basic Memory Layout



Interfaces

The thumb drive boot loader was designed in layers so that each layer may be easily replaced if a different file format, boot medium or IO usage is desired. The interfaces for each of the above mentioned layers are briefly described below. Some of the interface "routines" are implemented as macros. (Improvement Note: They could all be given macro translations for easy porting.)

Boot Loader IO Interface
The boot loader IO interface provides the means for the boot loader to identify if the user wishes to invoke or abort the loader. It also provides a way to communicate boot loader status to the user.

Header File: boot_io.h

Routines	Description
BLIO_InitializeIO	Performs any initialization necessary to enable the required IO mechanisms.
BLIO_DeinitializeIO	Disables any IO mechanisms used. Note: It is vital that this routine disables and clears any possible interrupts that may have been enabled by any IO used.
BLIO_LoaderEnabled	This routine identifies if the user wants to invoke the boot loader when as the system is reset.
BLIO_AbortLoad	If the boot medium has not yet been inserted, this routine identifies if the user wants to abort the loader.
BLIO_ReportBootStatus	This routine reports boot loader status and errors to the user.

Build Parameter	Required?	Description
BOOT_SWITCH	Required	This macro identifies the switch used to invoke the boot loader.
BOOT_LED	Optional	This macro identifies the status LED used.
FOSC	Optional	Used to calculate the oscillator period
TOGGLES_PER_SEC	Optional	Identifies the blink rate of the status LED
CORE_TICK_PERIOD	Optional	Calculated tick period of the timer core timer

(Improvement Note: The build parameters could be abstracted better.)

Media Interface
The media interface provides access to and control of the medium used to access the application's image file. In this demo, it provides a wrapper around the MDD FAT FS implementation and the USB MSD stack and the Loader interface, described below. It could be replaced to access any desired medium

Header File: boot_media.h

Routines	Description
BLMedia_InitializeTransport	This routine performs any initialization required by the boot medium.
BLMedia_DeinitializeTransport	This routine de-initializes the boot medium. Note: It is vital that this routine disables and clears any media-related interrupts.
BLMedia_MonitorMedia	This routine is called repeatedly to maintain the boot medium (before the "LoadFile" routine is called).
BLMedia_MediumAttached	This routine is called to identify if the boot medium has been attached to the system or not.
BLMedia_LocateFile	This routine is called to attempt to locate the boot image file.
BLMedia_LoadFile	Once the file has been located, this routine is called to read, translate, and load the image.

Build Parameters	Required?	Description
BOOT_FILE_NAME	Optional	This identifies the name of the application's image file. Default: "image.hex"

MAX_LOCATE_RETRYS	Optional	Defines the number of attempts to locate the image file before failing. Default: 3
MAX_NUM_MOUNT_RETRIES	Optional	USB MSD specific parameter that identifies the number of attempts to mount the thumb drive. Default: 10
BL_READ_BUFFER_SIZE	Optional	USB MSD specific parameter that defines the size of the read buffer (in bytes). Default: 512

Loader Interface

The loader interface provides a for the media layer to translate and program the application's image file. It can be replaced or modified to support different file formats.

Header: boot_load.h

Routines	**Description**
Loader_Initialize	This routine performs any initialization required by the loader.
Loader_GetFlashBlock	This routine translates raw data read in chunks from the file into binary data ready to program to a block of Flash memory.
Loader_ProgramFlashBlock	Once the data has been translated, this routine programs the block of Flash memory.
Loader_PermitProgramming	This routine can be implemented to provide a way to enable and disable actual programming of Flash.
Loader_ValidateSerialNumber	This routine can be implemented to provide a way to validate a serial number embedded in the application image.
Loader_ValidateRevisionNumber	This routine can be implemented to provide a way to validate a revision number embedded in the application image.
Loader_CheckErrorDetection	This routine can be implemented to provide a way to detect if the image has been correctly received.

Build Parameters	**Required?**	**Description**
APPLICATION_ADDRESS	Required	Provides the virtual address of the application's entry point in memory (once it has been programmed to Flash).
PROGRAM_FLASH_BASE	Required	Provides the physical address of the first block of Flash memory available for the application image.
PROGRAM_FLASH_LENGTH	Required	Provides the length (in bytes) of the Flash memory available.
FLASH_BLOCK_SIZE	Required	Provides the size (in bytes) of one block of available Flash memory
BLOCK_FILL_DEFAULT	Optional	Provides a value used to fill any unused space in each Flash block. Default: 0xFF

Common Interface Elements
There are some elements that are common to all layers or that are not specific to a any of the above defined interfaces.

Header File: boot.h

Note: This file aggregates/includes all of the other boot loader header files (including boot_config.h) so it is the only one that needs to be included by source (.c) files.

Routines	Description
BL_ApplicationIsValid	This routine can be implemented to detect if the application image currently programmed into Flash memory is valid.
BL_ApplicationFailedToLaunch	This routine can be implemented to provide error behavior if the application image fails to launch and returns to the boot loader. (Note: This is highly improbable since an invalid application will most likely hang the system.)

Boot Status Codes
The boot loader source code contains calls to BLIO_ReportBootStatus in appropriate places. These calls report a status code (and, optionally, a message string) to advise the user of the boot loader's current status. The codes are defined by an expandable enumerated data type defined in the configuration header. (Refer to the definition of the BOOT_STATUS enum in the "boot_config.h" header.)

Notes
For convenience several interface routines are defined by macros in boot_config.h.
The boot loader has been implemented as sample code. However, elements of it may be later become libraries and be moved to the "Microchip" directory.

"read_me.txt"

```
This file contains last minute updates and information about the Microchip
USB Thumb Drive Boot Loader demo (here after referred to as just the "boot
loader").

This distribution is a beta release only.  All contents are provided
"as is", with no promise of support or suitability for any particular
purpose.  Also, please refer to copyright notices associated with
individual items included in this release.


Getting Started:

To use the boot loader, you can either open the workspace file
(usb_host_thumbdrive_bootloader_demo\usb_host_thumbdrive_bootloader_demo.mcw)
in MPLAB, build it, and program it to Flash on a PIC32MX4xxFxxxx part.
Alternately, you can import the pre-built bootloader.hex file (see
bootloader\precompiled\bootloader.hex) and program it to Flash.

Two pre-compiled application demos are provided (both hex file and
the associated map file so you can see the memory mapping used).  You
can use these pre-compiled applications to test the boot loader or you
```

```
can use the included "procdefs.ld" to build your own applications for
the boot loader.


Usage:

The boot loader provides a means of loading a properly-built application
hex-file image from a USB Thumb Drive, programming it to Flash on the
microcontroller and booting (launching) the application image.  To do
this, the application image must be built using the "procdefs.ld" linker
script (place it in the application's main project folder, but DO NOT
include it in the project from within MPLAB).  After the application
has been properly built, copy its hex file and name it image.hex to the
root of the thumb drive.  You then press S4 on the Explorer 16 board
while resetting the micro.  This will enter the boot loader mode, as
indicated by LED D10 flashing.  Insert the thumb drive into the USB
Type-A receptacle and the boot loader will load and boot the application
image.  If it is unable to find the file, D10 will flash rapidly to
indicate an error.  To abort the load, press S4 a second time.


Folders and Contents:

  Microchip
  |
  |  This directory contains the Microchip library code used by the
  |  usb_host_thumbdrive_bootloader_demo.
  |
  +--  Include
  |
  |      This directory contains the interface header files for the
  |      libraries used by the boot loader.
  |
  +--  MDD File System
  |
  |      This directory contains the file system support library for the
  |      boot loader.
  |
  +--  USB

         This directory contains the USB library support for the boot
         loader.

  usb_host_thumbdrive_bootloader_demo
  |
  |  Main boot loader distribution folder.  This folder also contains the
  |  boot loader's main MPLAB workspace file.
  |
  +--  application
  |    |
  |    |  This folder contains the linker script that must be used to build
  |    |  an application for the boot loader (procdefs.ld) as well as
  |    |  hex files for pre-built sample applications.
  |    |
  |    +--  usb_cdc_serial_device_demo
  |    |
```

```
|    |      This folder contains a sample USB CDC Serial device demo
|    |      pre-built for the boot loader.
|    |
|    +--  usb_hid_mouse_device_demo
|
|           This folder contains a sample USB HID-Mouse device demo
|           pre-built for the boot loader.
|
+--  bootloader
|    |
|    |  This folder contains the source code for the boot loader.
|    |
|    +--  Objects-PIC32MX-usb_host_thumbdrive_bootloader
|
|         This folder contains pre-build files for the boot loader.
|
|         NOTE:  These files will be over-written if you build the
|                boot loader, so copy them somewhere else if you wish
|                to keep them.
|
+-- documentation

       This directory contains a documenet describing the basic
       architecture and interfaces for each layer of the boot loader.
```

Notes:

Use the "boot_config.h" file to customize the boot loader. The following items can be easily modified.

APPLICATION_ADDRESS

This is the virtual address of the application's entry point. If the location of the application is changed, this must be changed so that the boot loader can correctly launch the application.

PROGRAM_FLASH_BASE

This is the physical address of the application in program Flash memory. This address should be the physical equivalent to APPLICATION_ADDRESS.

PROGRAM_FLASH_LENGTH

This is the length of the program Flash memory space (starting from PROGRAM_FLASH_BASE) that is available for the application.

If the program address, Flash base, or size is modified, the "procdefs.ld" file will also need to be modified appropriately.

FLASH_BLOCK_SIZE

This is the size of a single block (or page) of program Flash.

```
BOOT_FILE_NAME

  This defines a string that provides the expected name of application's
  image (hex) file.

BL_READ_BUFFER_SIZE

  This defines the size of the buffer used to read data from the
  thumb drive.

BOOT_SWITCH

  This identifies the switch used to activate the boot loader.


Refer to the file(s) available in the "documents" directory and the
source code for additional details.


Date:  11/11/2008
```

Appendix E – Application Event Handler Example for USB Embedded Host Certification

The following function is used in Microchip's EH certification project and is provided as a reference example showing a minimal set of USB events which the application should handle

```
BOOL USB_ApplicationEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    #ifndef MINIMUM_BUILD
        switch( event )
        {
            case EVENT_REQUEST_POWER:
                // The data pointer points to a byte that represents the amount of power
                // requested in mA, divided by two.  If the device wants too much power,
                // we reject it.
                if (*(BYTE*)data <= (MAX_ALLOWED_POWER / 2))
                {
                    return TRUE;
                }
                else
                {
                    UART2PrintString( "\r\n***** USB Error - device requires too much current
*****\r\n" );
                }
                break;

            case EVENT_RELEASE_POWER:
                // Since we have support for only one device, we do not need to
                // track power released.
                return TRUE;
                break;

            case EVENT_VBUS_OVERCURRENT:
                UART2PrintString( "\r\n***** USB Error - overcurrent detected *****\r\n" );
                return TRUE;
                break;

            case EVENT_HUB_ATTACH:
                UART2PrintString( "\r\n***** USB Error - hubs are not supported *****\r\n" );
                return TRUE;
                break;

            case EVENT_UNSUPPORTED_DEVICE:
                UART2PrintString( "\r\n***** USB Error - device is not supported *****\r\n" );
                return TRUE;
                break;

            case EVENT_CANNOT_ENUMERATE:
                UART2PrintString( "\r\n***** USB Error - cannot enumerate device *****\r\n" );
                return TRUE;
                break;

            case EVENT_CLIENT_INIT_ERROR:
                UART2PrintString( "\r\n***** USB Error - client driver initialization error
*****\r\n" );
                return TRUE;
                break;

            case EVENT_OUT_OF_MEMORY:
                UART2PrintString( "\r\n***** USB Error - out of heap memory *****\r\n" );
                return TRUE;
                break;
```

```
            case EVENT_UNSPECIFIED_ERROR:   // This should never be generated.
                UART2PrintString( "\r\n***** USB Error - unspecified *****\r\n" );
                return TRUE;
                break;

        }
    #endif

    return FALSE;
}
```