Class Objectives

After taking this class, you will be able to

- Describe the electrical, mechanical, protocol and compliance requirements for a USB embedded host design,
- Apply the Microchip Embedded Host frameworks to:
  - Design an embedded host application using an existing client driver
  - Create your own "generic" client driver on a PIC24/PIC32 based USB embedded host.
  - Add USB thumb drive capability to your application (datalogging, file manipulation & bootloading)

Agenda

- Part 1:
  - Introduction to USB Embedded Host
- Part 2:
  - Designing a Custom Class, Full-Speed USB Embedded-Host Application
- Part 3:
  - Designing a Mass-Storage Class, Full-Speed USB Embedded-Host Application
- Part 4:
  - Using the USB Thumb Drive Boot Loader Application

Class Folders
- After Installing the Class CD -

```
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
```


Resources Used

Hardware

- PIC24 (MA240014) or PIC32 (MA320002) USB PIM
- Explorer 16 Board (DM240001)
- USB PICtail™ Plus Daughter Board (AC164131)
- PICDEM™ FS USB Demo Board (DM163025) pre-programmed with default factory application

Tools

- MPLAB® IDE w/C30 or C32
- MPLAB REAL ICE™ Emulator (DV244005) or ICD3 (DV164035)
- USB Protocol Analyzer (Optional)

Software

- Microchip Application Framework v2009-08-31 (MCHPFSUSB v2.5b + MDD v1.2.3), available from
  - www.microchip.com/usb

MICROCHIP

Regional Training Centers

Part 1

Introduction to USB Embedded Host

Objectives (Part 1)

- To know what USB hosting options are available
- To understand how Embedded Host is different from Full Function (Standard) Host
- To know where to go next to get more information, tools, training, etc. to get a design going

Agenda (Part 1)

- Overview of USB Hosting Options
- Connectors
- Certification Considerations For USB Embedded Host
- Development Resources

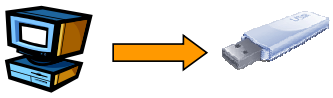
MICROCHIP

Regional Training Centers

Overview of USB Hosting Options

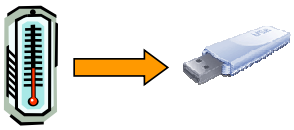
Overview

- Full-Function Host
  - Always a host, never a USB device
  - Standard A connector
  - Must always supply power
  - Has sufficient hardware to support almost all USB devices
- Example: Personal Computer



Overview

- Embedded Host
  - Always a host, never a USB device
  - Standard A connector
  - Must always supply power
  - Restricted ability to add new device support
- Example: Data Logger



Overview

- Dual Role Devices (DRDs)
  - 2 connectors (Standard A & Standard B/miniB)
  - Wants to be either embedded host or USB device but doesn't need to dynamically switch
- Example: Data Logger with field update via PC

Overview

- On-The-Go (OTG)
  - Mobile, simple hosts
  - Want to be host sometimes but device sometimes
  - Power consumption
  - Micro A/B connector
- Example: PDA

OTG vs. DRD

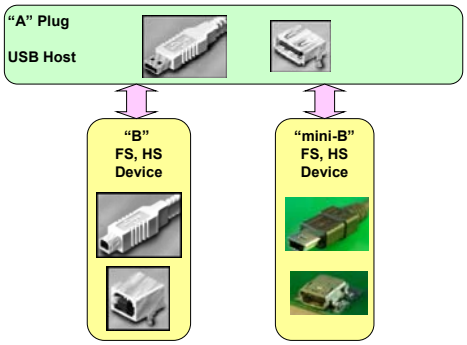
- Both are Host and Peripheral in one
- Full Speed Peripheral / Host Capable
- DRD (Dual Role Device)
  - Contains both a Host (type-A) and a Peripheral (type-B) connector
  - Peripheral or Host role is determined through which Physical connector is used
  - If both are accessible, both must be functional
- OTG (On-The-Go)
  - Contains Micro A/B connector
  - Cable connection decides who is host initially
  - Host Negotiation Protocol (HNP) used to dynamically and temporarily swap roles

MICROCHIP

Regional Training Centers

Connectors

Embedded Host/DRD
- Standard USB 2.0 Connectors -



OTG

- OTG Plugs and Receptacles
  - Micro-B plug and receptacle
  - Micro-A/B receptacle
    - Only allowed on OTG products
  - Micro-A plug
    - Indicates who is initially the host

MICROCHIP
Regional Training Centers

Certification Considerations for USB Embedded Host

Agenda
- Certification Considerations -

- Electrical
- Targeted Peripheral List
- Power
- Speed
- Transfer Types
- Hub Support
- Indications to the User

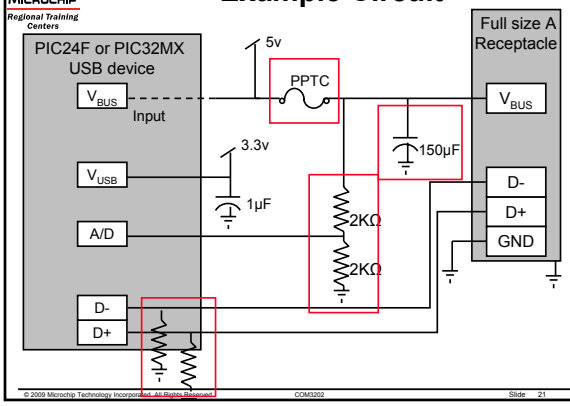
Electrical

- The Type A port must pass standard host electrical compliance tests, per "USB Compliance Systems Checklist sec 3.1"

3.1 Power Delivery

P1	Can the system supply 0 to 500mA on each of its downstream ports, regardless of whether or not the system or USB is suspended?	yes☐ no☐	7.2.1 7.2.3
P2	Does the system implement overcurrent protection to prevent more than 5A from being drawn from any downstream port?	yes☐ no☐	7.2.1.2.1
P3	Is the system's overcurrent protection resettable without user mechanical intervention?	yes☐ no☐	7.2.1.2.1
P4	Can the system maintain V_{BUS} between 4.75 at 5.25V at all of its downstream connectors for DC loads between 0 and 500mA per downstream port?	yes☐ no☐	7.2.2
P5	Does the system have at a total of at least 120µF of low ESR bypass capacitance at its ports?	yes☐ no☐	7.2.4.1
P6	Does the system's port bypassing limit the maximum voltage droop at any of its downstream ports to 330mV, even when subjected to hot-plug inrush currents with peaks of 7.5A or more? (As of this writing, the highest inrush current the USB-IF has observed from a within spec configuration is 7.40A.)	yes☐ no☐	7.2.4.1
P7	Are overcurrent events reported to the host controller?	yes☐ no☐	10.2

Example Circuit



Targeted Peripheral List (TPL)

- List of supported devices for that embedded host
  - Devices not on that list will not be able to enumerate
- Devices are identified via specific VID/PID, or class code (class, subclass, protocol)
- TPL may be specified in 2 forms

Example TPL
- Listing Specific Products (VID/PID) -

Manufacturer	Model	VendorID	ProductID	Description	Speed
Logitech	M-BJ58	0x046D	0xC00E	USB Wheel Mouse	LS
Hewlett-Packard	D125X1	0x03F0	0x2311	All-in-one Printer/Scanner	FS

Example TPL
- Listing Supported Device Classes -

Class Name	Description	Class Code	Sub-Class Code	Protocol	Speeds Supported
Mass Storage	Support for USB Floppy drives	0x08	0x04	0x50	LS, FS

Devices Tested

Manufacturer	Model	VendorID	ProductID	Description	Speed
TEAC Corp.	FD-05PUB	0x0644	0x0000	USB Floppy Drive	FS

Power

- Embedded Hosts must be capable of supplying 8mA (min.)
- They must also be capable of supplying the max. current that any specific device on the TPL requires (up to 500mA max).
- For class support, Embedded Hosts must be capable of supplying up to 500mA
- Embedded Hosts must report a failure to the user when peripherals consume more current than the host supplies.

Speed

- Embedded Hosts only need to support the speeds required by the devices on it's TPL

Transfer Types

- All Embedded Hosts must support control transfers in order to enumerate the selected peripherals
- Embedded Hosts may support one or more of the remaining three transfer types as required by the TPL

Hub Support

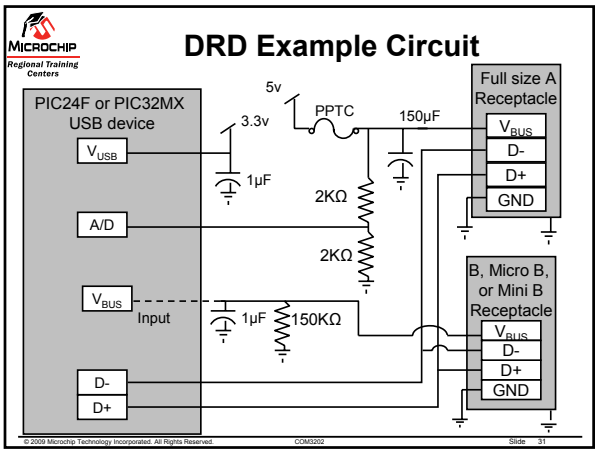
- Hub support is not required for Embedded Host ports
- The Embedded Host must provide an indication to the user of any unsupported Hub configuration.
- Our embedded host stack currently does not provide hub support

Indications to the User

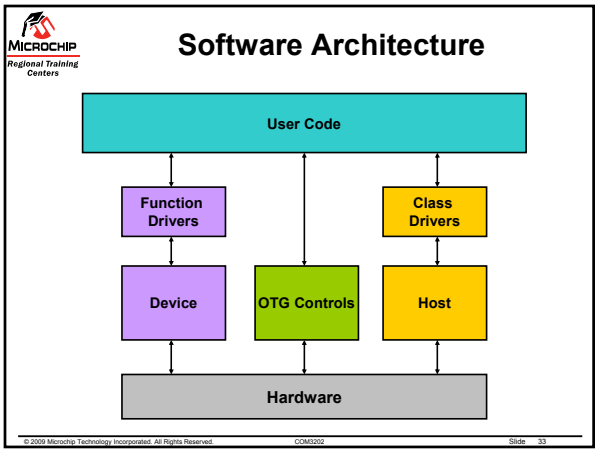
- On connection of a peripheral, an Embedded Host must indicate to the user whether the peripheral is supported
  - Indicator may be as simple as a "success/failure" LED
  - Textual messages are preferred for Embedded Hosts which contain such a display

Additional Considerations For Dual Role Devices

- Port accessibility
  - If more than one connector is accessible at any point of time then they need to be able to work at the same time
- Checklists
  - Peripheral
  - Systems


MICROCHIP
Regional Training Centers
DRD Example Circuit
PIC24F or PIC32MX
USB device
V_{USB}
A/D
V_{BUS}
Input
D-
D+
5v
3.3v
PPTC
150µF
1µF
2KΩ
2KΩ
1µF
150KΩ
Full size A
Receptacle
V_{BUS}
D-
D+
GND
B, Micro B,
or Mini B
Receptacle
V_{BUS}
D-
D+
GND
© 2009 Microchip Technology Incorporated. All Rights Reserved.
COM3202
Slide 31


MICROCHIP
Regional Training Centers
Development Resources

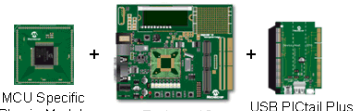

MICROCHIP
Regional Training Centers
Software Architecture
User Code
Function Drivers
Class Drivers
Device
OTG Controls
Host
Hardware
© 2009 Microchip Technology Incorporated. All Rights Reserved.
COM3202
Slide 33

Software Examples Available
- www.microchip.com/usb -

- Embedded Host
  - Data logging to a thumb drive
  - Bootloading from a thumb drive
  - MCHPUSB host – temperature, pot reader
  - Printer Host (PCL5 and PostScript)
  - HID Host – talking to a keyboard
  - CDC Host – hosting a serial to USB converter
- Dual Role Device
  - MSD Host + HID Device

Demo Tools Available

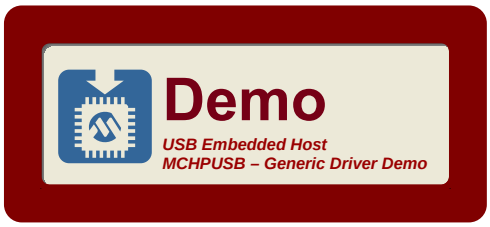
- Development Kit
  - PIC24F USB PIM (MA240014)
  - PIC32MX USB PIM (MA320002)
  - USB PICtail™ Plus Daughter Card (AC164131)
  - Explorer 16 (DM240001)



Demo Tools Available

- Starter Kits
  - PIC24F Starter Kit (DM240011)
  - PIC32MX USB Starter Board (DM320003)

MICROCHIP
Regional Training Centers



MICROCHIP
Regional Training Centers

Summary (Part 1)

- We covered:
  - What USB host options are available
  - How they are different
    - Mechanical
    - Electrical
  - Embedded Host & DRD Certification considerations
  - Available development resources
  - Built/Ran Embedded Host Demo
  - OTG protocol/design covered in a separate class

MICROCHIP
Regional Training Centers

References

1) `EH_MR_rev1.pdf` – "Requirements and Recommendations for USB Products with Embedded Hosts and/or Multiple Receptacles rev 1.0"
2) `EH_compliance_v1_0.pdf` – "USB Embedded Host Compliance Plan rev 1.0"
3) `compchksys080205.pdf` – "USB Compliance Checklist (Systems)"
4) `compchkperi080205.pdf` – "USB Compliance Checklist (Peripherals)"

Both available on class CDROM in `\Users Guides & Data Sheets\USB Standards`, or from www.usb.org

Objectives (Part 2)

When you finish this section you will be able to:

- Design an embedded host application using the Microchip USB Framework
- Implement a "generic" client driver for the Microchip USB Framework
- Demonstrate your embedded host application and driver

Agenda (Part 2)

- Review of Key USB Concepts
- USB Embedded Host Framework Basic Structure
- Application Design Using Existing Client Driver
  - Lab 1 – Implement Application using Microchip-Provided Generic Client Driver
- Client Driver Design
- Enumeration & Initialization
  - Lab 2 – Implement a Custom “Polled” Generic Client Driver
- Event Handling
  - Lab 3 – Implement a Custom “Event-Driven” Generic Client Driver
- VBus Monitoring & Stack Shutdown
- Summary/References

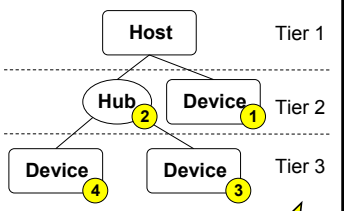
MICROCHIP

Regional Training Centers

Review of Key USB Concepts

Basics

- USB is a Tiered Star Network
- Host is com master
  - Polled bus
- Hubs expand network
- Devices are addressed & enabled by the host



Each Device is assigned an address from 1-127 by the host

Endpoints & Transfer Types

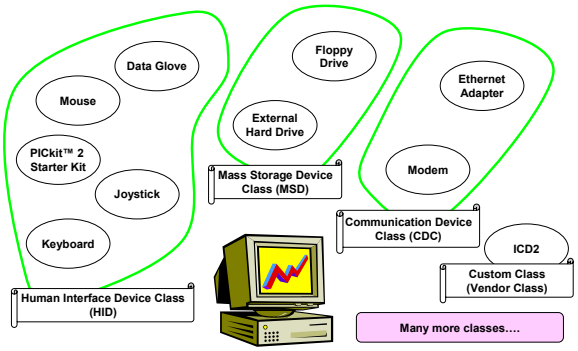
- Data Transfer to/from Endpoints
  - Up to 32 Endpoints (16 IN/16 OUT)
  - 1 Transfer Type Per Endpoint
- Data Transfer Types:
  - Control – ex. Enumeration Data
  - Interrupt – ex. Key-press Data
  - Isochronous – ex. Audio Data
  - Bulk – ex. Thumb Drive Data

Summary - Data Transfer Types

Transfer/ Endpoint Type	Polling Interval	% Reserved BW/Frame for all transfers of this type	Max. # Data Bytes/Frame/Endpoint (Max# transactions per frame @ Max Ep Size)*	Data Integrity
Interrupt	Fixed, Periodic	90	64 (1 x 64)	Yes
Isochronous	Fixed, Periodic	90	1023 (1 x 1023)	No
Bulk	Variable, Uses Free Bandwidth	0	1216 (19 x 64)	Yes
Control	Variable	10	832 (13 x 64)	Yes

*Assumes transfers use maximum packet sizes allowed per Ep type

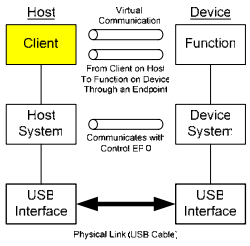
Device Classes



Programmers Model

USB Layers

- Client SW Talks to Function SW "Virtually"
- Control by Host System Layer
- Actual Data Transfer Occurs on Physical Layer



Functional Interfaces

- **Host Side "Client" API**
  - `MPUSBOpen(VID, PID, Endpoint, Direction)`
  - `MPUSBRead(Pointer, Size, Timeout)`
  - `MPUSBWrite(Pointer, Size, Timeout)`
  - `MPUSBClose(Handle)`
- **Device Side "Function" API**
  - `putrsUSBUSART(const rom char *data)`
  - `putsUSBUSART(char *data)`
  - `mUSBUSARTTxRom(rom byte *pData, byte len)`
  - `mUSBUSARTTxRam(byte *pData, byte len)`
  - `getsUSBUSART(char *buffer, byte len)`
  - `byte mCDCGetRxLength(void)`

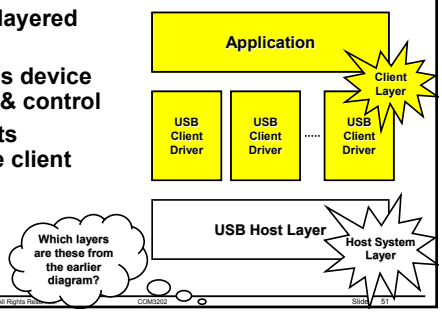
MICROCHIP

Regional Training Centers

USB Embedded Host Framework
- Basic Structure -

USB Embedded Host FW Stack
- Structure -

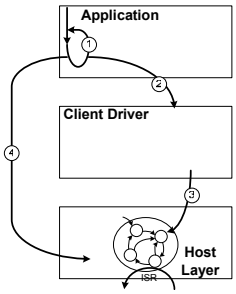
- Uses a layered design
- Provides device access & control
- Supports multiple client drivers



Driver Design

Flow of Calls

1. Main Loop
2. Driver API
3. Host CDI
4. USBTasks()



MICROCHIP

Regional Training Centers

Application Design Using Existing Client Driver

Application Design

Objective:

Design an embedded host application using the Microchip USB Framework

- What should the Application do?
- How do I access the device?
- How do I use the USB Framework?

Application Design

What should the application do?

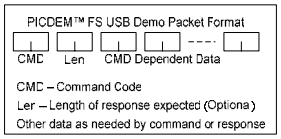
Up to you, Normally…

- **Demo the PICDEM™ FS USB Board (a custom/vendor-class USB peripheral)**
  - Read Device FW Revision Number
  - Read Potentiometer
  - Read Temperature
  - Display data read
  - Control 2 LEDs on the Device

Application Design

How do I access the device?

- **Command-Response Protocol Packets**



Example:

```
Command:   0x00, 0x02  ("Get Firmware Version")
Response:  0x00, 0x02, 0x01, 0x00 ("v1.0")
```

See Appendix B in the Lab Manual for the full command set

Application Design

Accessing the device (continued)…

- **Generic Function/Client Driver API**
  - `USBHostGenericRead(...)`
  - `USBHostGenericWrite(...)`
  - `USBHostGenericRxIsBusy(...)` –or- `USBHostGenericRxIsComplete(...)`
  - `USBHostGenericGetRxLength(...)`
  - `USBHostGenericTxIsBusy(...)` –or- `USBHostGenericTxIsComplete(...))`

Complete API documented in AN1143 and USB Embedded Host Library Help File

Application Design

How do I use the USB Framework?

- Use `USBConfig.exe`

  Configuration Code Generator w/GUI
  - Choose Microchip client drivers
  - Select configuration settings
  - Define the host's Targeted Peripheral List

Application Design

Main Tab

- Select the target device
- Select the device type
- Select the ping-pong mode used by the USB interface*

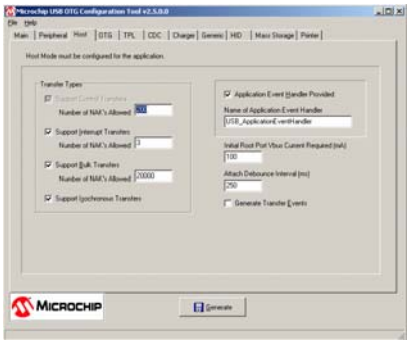
*PIC32 supports "Ping-Pong on All Endpoints", PIC24 supports all modes.

Application Design

Host Tab

- Choose Transfer Types & Settings
- Choose Power Settings
- Define Application Event Handler
- Enable "Transfer Complete" Event Signaling

More on Events, later…



Application Design

TPL Tab

- Description
- VID/PID or Class, Subclass, & Protocol
- Microchip Client Driver
- Initial Configuration & Flags

Application Design

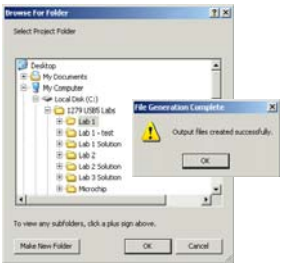
Driver Tab

- Generic, HID, Mass Storage
- Choose host mode
- Adjust any parameters

Application Design

Generates:

- `usb_config.c`
  - TPL Table
  - Driver Table
- `usb_config.h`
  - Defines for Configuration Options
  - `USBInitialize()`
  - `USBTasks()`



Application Design

Example `usb_config.h`

```
#define USB_SUPPORT_HOST
#define USB_PING_PONG_MODE                  USB_PING_PONG__FULL_PING_PONG
#define NUM_TPL_ENTRIES                     1
#define USB_MAX_GENERIC_DEVICES             1
#define USB_NUM_CONTROL_NAKS                20
#define USB_SUPPORT_INTERRUPT_TRANSFERS
#define USB_NUM_INTERRUPT_NAKS              3
#define USB_INITIAL_VBUS_CURRENT            (100/2)
#define USB_INSERT_TIME                     (250+1)
#define USBTasks()                  \
    {                               \
        USBHostTasks();             \
    }
#define USBInitialize(x)            \
    {                               \
        USBHostInit(x);             \
    }
```


Application Design

Example of `usb_config.c`

```
USB_TPL usbTPL[] =
{
    { INIT_VID_PID( 0x04D8ul, 0x000Cul ), 1, 0, {0|SET_CONFIG} }, // PICDEM FS
      USB
};

CLIENT_DRIVER_TABLE usbClientDrvTable[] =
{
    {
        USBHostGenericInit,
        USBHostGenericEventHandler,
        0
    }
};
```


Application Design

Example Application:

```
#include “USB/usb.h”
#include “USB/usb_host_generic.h”

int main ( void )
{
    USBInitialize(0);

    while (1)
    {
        USBTasks();

        // Call “Generic” driver API routines as needed
        //   ** No blocking Code **
    }

    return 0;
}
```


Application Design Using Existing Client Driver

Summary:

- Implement the application
- Use Microchip Client Driver for device access
- Configure the USB Host Framework with `USBConfig.exe`
- Include `usb_config.c` in your project
- Include `usb.h` & client driver header (eg. `usb_host_generic.h`)
  (`usb_config.h` gets included by `usb.h`)
- Include the USB Host Framework Files
- Call API routines as appropriate

MICROCHIP

Regional Training Centers

Lab 1: Implement Application Using Microchip-Provided Generic Client Driver

Lab 1 Objectives

- **Assemble Hardware**
  - PIC24 or PIC32 PIM
  - Explorer 16
  - USB PICtail™ Plus
  - PICDEM FSUSB programmed with default factory application
- **Use MPLAB® IDE to build application skeleton**
- **Use USBConfig.exe to configure the USB**
- **Use REAL ICE™ emulator to program the application into the Explorer 16 board**
- **Add missing "To Do" items**
- **Advanced: Add Unimplemented Features**

Lab 1 Summary

- The application initializes the USB Framework, and waits for a device
- The application state machine switches non-blocking “tasks”
- The application sends command packets, receives response packets, and displays the data received
- The application utilizes the generic client driver “polled” API functions to send/receive data
- The generic client driver transfers the packets to and from the device

MICROCHIP

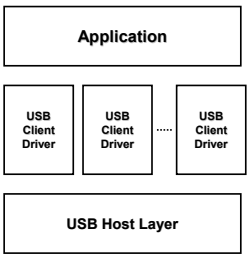
Regional Training Centers

Client Driver Design

Client Driver Design

USB Embedded Host Framework

- Uses a layered design
- Provides device access & control
- Supports multiple client drivers



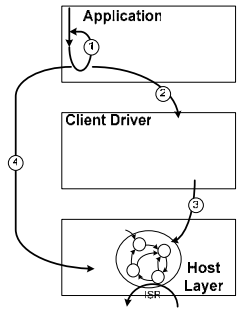
Client Driver Design

- **Why Split out the "Client Driver" from the Application?**
  - **So the application can be written without worrying about USB details**
  - **So drivers can be easily replaced or mixed & matched as needed for different applications**
- **Why support multiple Client Drivers?**
  - **A single host may support more than 1 device**
    (Even if it only has 1 USB port)
  - **Hubs may add tiers to the "network"**
    (Hubs are not yet supported)
  - **Because a device may be "composite"**

Client Driver Design

Flow of Calls

1. **Main Loop**
2. **Driver API**
3. **Host CDI**
4. **USBTasks()**



Client Driver Design

Client Driver Interface (CDI) Routines:

Write Example:

```
flags.txBusy = 1;
if (USBHostWrite(address, GENERIC_OUT_EP, &buffer, length) != USB_SUCCESS)
    flags.txBusy = 0;    // Clear flag to allow re-try
...
```

Read Example:

```
flags.rxBusy = 1;
rxLength = 0;
if (USBHostRead(address, GENERIC_IN_EP, &buffer, length) != USB_SUCCESS)
    gc_DevData.flags.rxBusy = 0;    // Clear flag to allow re-try
...
```

Client Driver Design

CDI Routines (continued):

```
if (flags.rxBusy)
{
    if (USBHostTransferIsComplete(address, GENERIC_IN_EP,… &byteCount ))
    {
        flags.rxBusy = 0;
        rxLength     = byteCount;
    }
}

if (flags.txBusy)
{
    if (USBHostTransferIsComplete(address, GENERIC_OUT_EP,… &byteCount ))
    {
        flags.txBusy = 0;
    }
}
```

Where would this logic go?

Client Driver Design

A polled driver may have a "Tasks" routine:

- Update driver state
- Called along with host "Tasks" routine
- Handled by "`USBTasks()`"
- Called from the application's main loop
- "`USBTasks()`" is a macro

```
#define USBTasks()                  \
    {                               \
        USBHostTasks();             \
        USBHostGenericTasks();      \
    }
```

Does `USBConfig.exe` handle this?

Yes! If needed.

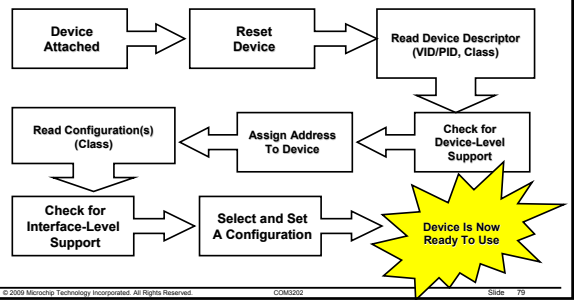
MICROCHIP

Regional Training Centers

Enumeration and Initialization

Review of Key Concepts

The Enumeration Process



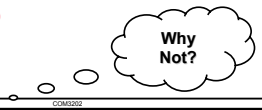
Enumeration & Initialization

Microchip USB Framework Initialization

- The App must call "`USBInitialize()`" before calling any other USB routine
- Implemented as a macro

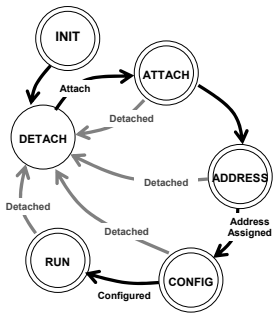
```
#define USBInitialize(x)            \
    {                               \
        USBHostInit(x);             \
        USBHostGenericInit(x)       \
    }
```

- Can be used to initialize other layers
  Not recommended
  (Unless necessary)



Enumeration & Initialization

- Host Layer State Machine
- Enumerates Device
- Initializes Client Driver



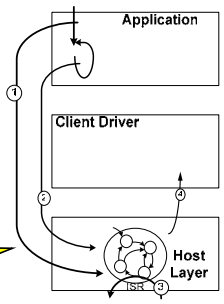
Enumeration & Initialization

Client Driver Interface "Call Back" Routines

- `USBHostGenericInit(…)`
  - Initializes the generic driver
  - Called by the host layer (up the stack)
  - Called after the device has been enumerated (not on system boot)
- `USBHostGenericEventHandler(…)`
  - Receives bus events (like device detach)
  - We will look at events closer later…

Enumeration & Initialization

- `USBInitialize()`
  (Called before main loop)
- `USBTasks()`
  (Called in main loop)
- Device Attached
- `USBHostGenericInit()`



What if I support 5, 10 or 20 clients?

We need a table-driven method

Enumeration & Initialization

Client Driver Table

```
p->Initialize(…)
p->EventHandler(…)
p->Initialize(…)
p->EventHandler(…)
```

Client Driver

```
USBHostGenericInit(…)
{
  rx_size = 0;
  ...
  flags = INITIALIZED;
  return TRUE;
}

USBHostGenericEventHandler(…)
{
  switch(event)
  {
  case EVENT_DETACH:
      ...
  }
  ...
}
```

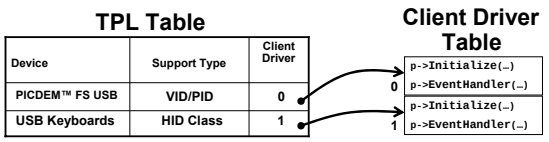
But, how does the host layer know which driver to call?

Enumeration & Initialization

The TPL Table

- Identifies the device
- Identifies the support type
- Provides an index into the client driver table

TPL Table + Client Driver Table = Call to Right Driver



MICROCHIP

Regional Training Centers

Lab 2: Implement a Custom "Polled" Generic Client Driver

Lab 2 Objectives

- Implement a generic "polled" client Driver
- Define a client driver table for it
- Define a TPL table for our device
- Test it using the application from Lab 1

Lab 2 Summary

- The host layer enumerates the device
- Then, the host layer looks up the appropriate driver and calls its `Initialize()` routine
- The application accesses the device via the driver's polled API functions
- Using table-driven methods, the host layer can manage multiple client drivers

MICROCHIP

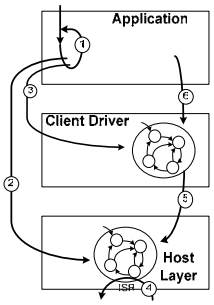
Regional Training Centers

Event Handling

Event Handling

Review Polling

1. `USBTasks()`
2. `USBHostTasks()`
3. `USBHostGenericTasks()`
4. Event causes ISR to fire
5. Client Driver polls Host Layer to discover event
6. If needed, the App polls the Client driver to discover the event

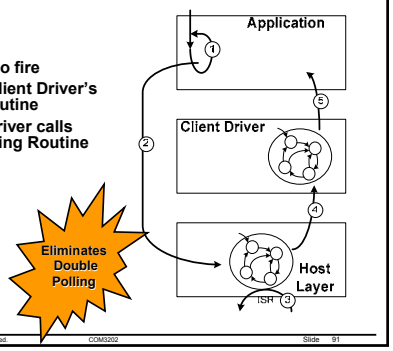


This seems like a lot of time wasted "double" polling for status changes.

Event Handling

Event driven

1. `USBTasks()`
2. `USBHostTasks()`
3. Event causes ISR to fire
4. Host Layer Calls Client Driver's Event Handling Routine
5. If needed, Client Driver calls App's Event Handling Routine



Event Handling

A Minimal Client Driver Event Handler (EVENT_DETACH)

```
BOOL USBHostGenericEventHandler ( BYTE address, USB_EVENT event,
                                  void *data,   DWORD     size )
{
    switch (event)
    {
    case EVENT_DETACH:
        // Notify that application that the device has been detached.
        USB_HOST_APP_EVENT_HANDLER(address, EVENT_GENERIC_DETACH,
                                   data, sizeof(BYTE) );
        flags.val = 0;
        address   = 0;
        return TRUE;

    default:
        return FALSE;
    }
}
```


Event Handling

Client Driver Transfer Event Handling

```
case EVENT_TRANSFER:
    // Notifiy the application of the completion of an Rx or Tx transfer.
    if ( pData->bEndpointAddress == (GENERIC_IN_EP) )
    {
        flags.rxBusy = 0;
        rxLength     = pData->dataCount;
        USB_HOST_APP_EVENT_HANDLER(address, EVENT_GENERIC_RX_DONE,
                                   pData->pUserData, pData->dataCount );
    }
    else if ( pData->bEndpointAddress == (GENERIC_OUT_EP) )
    {
        flags.txBusy = 0;
        USB_HOST_APP_EVENT_HANDLER(address, EVENT_GENERIC_TX_DONE,
                                   pData->pUserData, pData->d
    }
    break;
```

So, how many different events are there?

Event Handling

Standard Events

- USB_EVENT enum
- Some sent to client, then optionally to application
- Some are used by the device stack
- Some indicate errors – directly sent to the application's event handler
- Documented in "Help" file
- Can be extended by drivers

```
typedef enum
{
    EVENT_NONE = 0,
    EVENT_TRANSFER,
    EVENT_SOF,
    EVENT_RESUME,
    EVENT_SUSPEND,
    EVENT_RESET,
    EVENT_ATTACH,
    EVENT_DETACH,
    EVENT_HUB_ATTACH,
    EVENT_STALL,
    EVENT_SETUP,
    EVENT_VBUS_SES_END,
    EVENT_VBUS_SES_REQUEST,
    EVENT_VBUS_SES_VALID,
    EVENT_VBUS_OVERCURRENT,
    EVENT_REQUEST_POWER,
    EVENT_RELEASE_POWER,
    EVENT_UNSUPPORTED_DEVICE,
    EVENT_CANNOT_ENUMERATE,
    EVENT_CLIENT_INIT_ERROR,
    EVENT_OUT_OF_MEMORY,
    EVENT_UNSPECIFIED_ERROR,
    EVENT_USER_BASE     = 10000,
    EVENT_BUS_ERROR     = UINT_MAX
} USB_EVENT;
```

How?

Event Handling

Extending USB_EVENT type for driver-specific events

- Start with EVENT_USER_BASE
- Increment by one for each event
- Add an offset for the driver.

```
#ifndef EVENT_GENERIC_OFFSET
    #define EVENT_GENERIC_OFFSET 0
#endif
#define EVENT_GENERIC_ATTACH  ( EVENT_USER_BASE + EVENT_GENERIC_OFFSET + 0)
#define EVENT_GENERIC_DETACH  ( EVENT_USER_BASE + EVENT_GENERIC_OFFSET + 1)
#define EVENT_GENERIC_TX_DONE ( EVENT_USER_BASE + EVENT_GENERIC_OFFSET + 2)
#define EVENT_GENERIC_RX_DONE ( EVENT_USER_BASE + EVENT_GENERIC_OFFSET + 3)
```

Why add a driver offset?

Application Event Handling

Application event handler

```
BOOL Demo_App_EventHandler ( BYTE address, USB_EVENT event,
                             void *data,   DWORD size )
{
    switch (event)
    {
    case EVENT_GENERIC_ATTACH:
        // Handle the attach event
        break;
    case EVENT_GENERIC_DETACH:
        // Handle the detach event
        break;
    case EVENT_GENERIC_TX_DONE:
        // Handle the data transmit done
        break;
    ...
    }
    return TRUE;
}
```

In usb_config.h

```
#define USB_HOST_APP_EVENT_HANDLER Demo_App_EventHandler
```

Application Event Handling
- USB Certification -

- Requirement
  - No Silent Failures – must provide indication of
    - Hub error message
    - Device not supported message
    - Over current notification
    - Reset-able over current protection
    - Drop voltage
    - Etc.
- Must use LCD, UART, or LEDs to provide a recognizable, independent notification of the error.
- Application Event Handler function must be defined to catch/display these events to the end-user

Application Event Handler
- USB Certification -

Application event handler (See Appendix E in handout for full details)

```
BOOL Demo_App_EventHandler ( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    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;
      ...
```


MICROCHIP

Regional Training Centers

Lab 3: Implement a Custom "Event-Driven" Generic Client Driver

Lab 3 Objectives

- Implement an event driven generic client Driver
- Define driver-specific events
- Add Event Handler to Application
- Understand different application flow to make efficient use of events

Lab 3 Summary

- An event-driven implementation calls up the stack
- Event-driven method eliminates the need to "double poll"
- Event driven method scales better

  (Easier to support multiple clients)
- Some events are sent directly to the application's event handler - will need to be processed if certification is desired

MICROCHIP

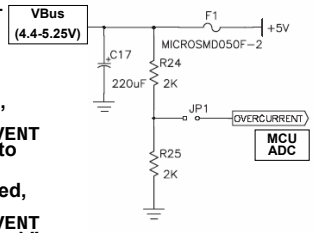
Regional Training Centers

VBus Monitoring and Stack Shutdown

VBus Overcurrent Monitoring

- VBus monitoring is the application's responsibility.
  - Technique is application-specific.
- Mass Storage projects implement function `MonitorVBUS( )`
- When detect overcurrent, call host layer function "`USBHostVbusEvent(EVENT_VBUS_OVERCURRENT)`" to shutdown USB
- When overcurrent restored, call "`USBHostVbusEvent(EVENT_VBUS_POWER_AVAILABLE)`" to turn the USB peripheral back on

Explorer 16 + USB PICTail Plus



Refer to project "USB Host - Mass Storage - Thumb Drive Data Logger"

USB Host Stack Shutdown

- `USBHostShutdown(void)`
  - Callable by the Application layer
    - Shuts down all USB activity, detaching all devices
  - Turns off the USB module, frees memory and resets the host layer state machine
  - `EVENT_DETACH` sent to client event handler
  - `EVENT_VBUS_RELEASE_POWER` sent to application event handler
- Restart:
  - Start calling `USBTasks()`
    - USB module re-initialized in one of the states

Summary (Part 2)

- Easy-to-use configuration tool available for Microchip drivers
- The Microchip USB Framework handles most of the USB Details
- Layered architecture provides flexibility & scalability
- Polled and Event-Driven implementations are supported
- An Application Event Handler is required for Embedded Host Certification

Questions?

(Part 2) App Note References
- www.microchip.com/usb -

- AN1140
  - USB Embedded Host Stack
- AN1141
  - USB Embedded Host Stack Programmers Guide.
- AN1143
  - USB Generic Client on an Embedded Host.

MICROCHIP

Regional Training Centers

Part 3

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

Objectives (Part 3)

- Understand the requirements for mass storage devices
- Understand the structure and configuration of the Mass Storage Client Driver
- Explain the functionality and configuration options of Microchip's Memory Disk Drive (MDD) File System Library
- Create and manipulate files on a USB thumb drive using PIC® USB microcontrollers

Agenda (Part 3)

- Introduction to Mass Storage Devices
- Configuring The Mass Storage Client Driver and Media Interface Layer
- The File System Layer (Microchip MDD File System Library)
  - File Creation
    - Lab 4: Creating and Deleting Files
  - File I/O
    - Lab 5: Reading and Writing
- Common Failure Modes
- Summary

MICROCHIP

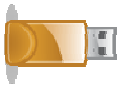
Regional Training Centers

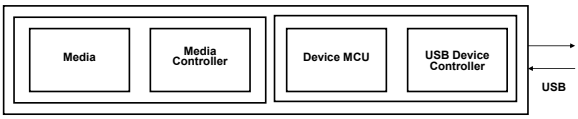
Introduction to USB Mass-Storage Devices

The Mass-Storage Class

- Includes devices that transfer files
  - Hard Drives
  - CD/DVD Drives
  - Flash-memory Drives
  - Cameras
- In PCs, these devices appear as drives
  - Users can easily copy, move or delete these files

USB Mass-Storage Devices
- Hardware Requirements -





USB Mass-Storage Devices
- Functional Requirements -

- The hardware/firmware must perform the following functions:
  - Detect and respond to generic USB requests and other events on the bus
  - Detect and respond to class-specific USB mass-storage requests for information or other action
  - Detect and respond to SCSI commands received in USB transfers
    - Read/Write blocks of data to/from the media using Logical Block Addressing
    - Request status information
    - Control device operation

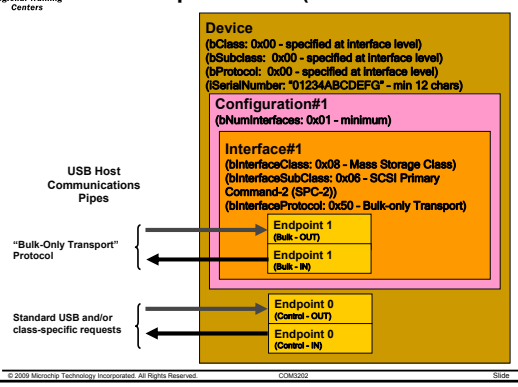
Transport Protocols

- USB Mass Storage devices use media command sets from several existing protocols
  - SCSI Primary Commands – 2 (SPC-2)
  - Multi-Media Command Set (MMC-5)
- These commands are used to perform various functions, such as reading, writing, checking media status etc.
- The command blocks of these command sets are placed in a USB wrapper which follows a USB transport protocol
  - Most common protocol is called “Bulk-Only Transport”

Bulk-Only Transport Protocol

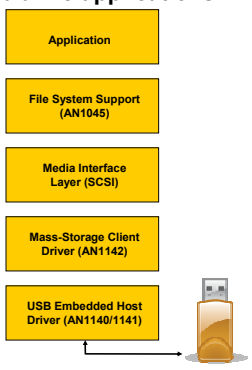
- A data transfer has 2-3 stages:
  - "Command Block Wrapper" (CBW)
    - 31 byte packet (see AN1142)
    - Sent to the OUT EP
  - Data (optional – depends on command)
  - "Command Status Wrapper" (CSW)
    - 13 byte packet (see AN1142)
    - Read from the IN EP
- CBW contains a 128-bit SCSI command block for the device to execute
  - Generated by the mass storage client driver

The Mass-Storage Class
- Descriptor Fields (Generic SCSI Media) -



Embedded Host Stack
- Support for thumb drive applications -

- File System Support Layer
  - Provides an interface to "file" data structures, used in all major PC operating systems
- Media Interface Layer
  - Converts file system commands to SCSI commands
- Mass-Storage Client Driver
  - Wraps SCSI commands in a USB wrapper
- Host Layer
  - Handles enumeration and USB bus details



MICROCHIP
Regional Training Centers

Demo

USB Embedded Host
Datalogging to a Thumbdrive

MICROCHIP

Regional Training Centers

Configuring The Mass Storage Client Driver and Media Interface Layer

MICROCHIP
Regional Training Centers

USB Configuration (Main)

Main Tab

- Select the target device
- Select the device type
- Select the ping-pong mode used by the USB interface*

*PIC32 supports "Ping-Pong on All Endpoints", PIC24 supports all modes.

USB Configuration - Host

Host Tab

- Enable support for Bulk and Control transfers
- Allow a large number of NAKs
- Increase the attach-debounce interval
- Define the application event handler
- Transfer Events *not* required for MSD client driver

USB Configuration - TPL

TPL Tab

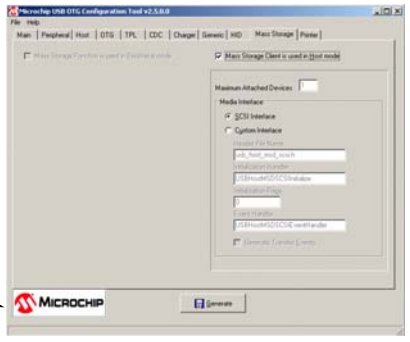
- Enter Description
- Enter Class, Subclass, Protocol ID values
- Select Mass Storage Client Driver
- Click “Add to TPL”

USB Configuration – Mass Storage

Mass Storage Tab

- Check “Mass Storage Client is used in Host Mode”
- Make sure SCSI interface is selected

Click Generate!



USB Configuration

Example `usb_config.h`

```
#define NUM_TPL_ENTRIES 1
#define USB_NUM_CONTROL_NAKS 200
#define USB_SUPPORT_BULK_TRANSFERS
#define USB_NUM_BULK_NAKS 20000
#define USB_INITIAL_VBUS_CURRENT (100/2)
#define USB_INSERT_TIME (250+1)
#define USB_HOST_APP_EVENT_HANDLER USB_ApplicationEventHandler

// Host Mass Storage Client Driver Configuration

#define USB_MAX_MASS_STORAGE_DEVICES 1

// Helpful Macros

#define USBTasks()                  \
    {                               \
        USBHostTasks();             \
        USBHostMSDTasks();          \
    }

#define USBInitialize(x)            \
    (                               \
        USBHostInit(x)              \
    )

#endif
```

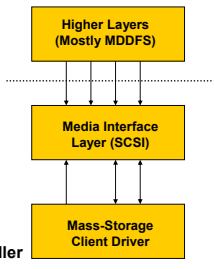

USB Configuration

Example of `usb_config.c`

```
// ******************************************************************************
// Media Interface Function Pointer Table for the Mass Storage client driver
// ******************************************************************************

CLIENT_DRIVER_TABLE usbMediaInterfaceTable =
{
    USBHostMSDSCSIInitialize,
    USBHostMSDSCSIEventHandler,
    0
};

// ******************************************************************************
// Client Driver Function Pointer Table for the USB Embedded Host foundation
// ******************************************************************************

CLIENT_DRIVER_TABLE usbClientDrvTable[] =
{
    {
        USBHostMSDInitialize,
        USBHostMSDEventHandler,
        0
    }
};

// ******************************************************************************
// USB Embedded Host Targeted Peripheral List (TPL)
// ******************************************************************************

USB_TPL usbTPL[] =
{
    { INIT_CL_SC_P( 0x08ul, 0x06ul, 0x50ul ), 0, 0, {TPL_CLASS_DRV} } // SCSI Thumb Drives
};
```


Media Interface Layer APIs
- `usb_host_msd_scsi.c` -

- **Detect a valid attached drive**
  - Called by application layer
  - `USBHostMSDSCSIMediaDetect()`
- **Initialization**
  - Called by file system layer
  - `USBHostMSDSCSIMediaInitialize()`
- **Blocking Read/Write API**
  - Called by the file system layer
  - `USBHostMSDSCSISectorWrite()`
  - `USBHostMSDSCSISectorRead()`
- **Event-Handler**
  - Call-back by the client driver event handler
  - `USBHostMSDSCSIEventHandler()`



Media Interface Layer
- Blocking Read/Write Operations -

```
BYTE USBHostMSDSCSISectorRead( DWORD sectorAddress, BYTE *dataBuffer )
{
    DWORD   byteCount;
    BYTE    commandBlock[10];
    BYTE    errorCode;

    // Fill in the command block with the READ10 parameters.
    commandBlock[0] = 0x28;     // Operation code
    commandBlock[1] = RDPROTECT_NORMAL | FUA_ALLOW_CACHE;
    commandBlock[2] = (BYTE) (sectorAddress >> 24);     // Big endian!
    commandBlock[3] = (BYTE) (sectorAddress >> 16);
    commandBlock[4] = (BYTE) (sectorAddress >> 8);
    commandBlock[5] = (BYTE) (sectorAddress);
    commandBlock[6] = 0x00;     // Group Number
    commandBlock[7] = 0x00;     // Number of blocks - Big endian!
    commandBlock[8] = 0x01;
    commandBlock[9] = 0x00;     // Control

    errorCode = USBHostMSDRead( deviceAddress, 0, commandBlock, 10,
            dataBuffer, mediaInformation.sectorSize );

    if (!errorCode)
    {
        while (!USBHostMSDTransferIsComplete( deviceAddress, &errorCode,
                &byteCount ))
        {
            USBTasks();
        }
    }

}
```


Event Generation

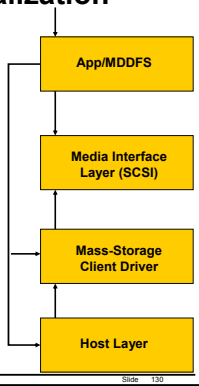
- The client driver can be configured to receive transfer events from the host layer
  - (via the Host Tab setting)
- The media interface layer can also be configured to receive these transfer events from the client driver
  - (via the Mass Storage Tab setting)
- Did we enable them with `USBConfig.exe` ?
  - No.
- Our file system functions block, so there is *no* need for the media layer to be event-driven
  - Media layer polls the MSD client driver
  - MSD client driver polls host layer

Mass Storage Client Driver Routines

- Polled API
  - `USBHostMSDRead(...)`
  - `USBHostMSDWrite(...)`
  - `USBHostMSDTransferIsComplete(...)`

Flow of Calls – Mass Storage Enumeration & Initialization

- Before the main loop
  - `USBInitialize()`
- In the main loop
  - `USBTasks()`
    - `USBHostTasks()`
    - `USBHostMSDTasks()`
  - `USBHostMSDSCSIMediaDetect()`
    - Called to detect attachment
- Device Attached
- Host layer enumerates drive and calls `USBHostMSDInitialize()`
  - Function updates a data structure
- Client driver tasks routine polls data structure to discover event
  - Calls `USBHostMSDSCSIInitialize()`
    - Saves USB address
- `USBHostMSDSCSIMediaDetect()` now returns TRUE
  - Calls file system initialization function `FSInit()`



Basic Thumb Drive Usage

```
#include "USB/usb.h"
#include "USB/usb_host_msd.h"
#include "USB/usb_host_msd_scsi.h"
#include "MDD File System/FSIO.h"
int main(void)
{
  USBInitialize(0);                          <- Initialize USB
  while(1) {
    USBTasks();                              <- Drive State Machines
    if(USBHostMSDSCSIMediaDetect()) {        <- Check for Device
      deviceAttached = TRUE;
      if(FSInit()) {
        myFile = FSfopen("test.txt","w");
        FSfwrite("This is a test",1,12,myFile);
        FSfclose(myFile);
        while(deviceAttached = TRUE){
          USBTasks();
          ...do other tasks
        }
      }
    }
  }
}
```

Once Device is Detected
1) Initialize the MDD FS
2) Open a file
3) Write to the file
4) And close the File
5) Do other tasks

MICROCHIP

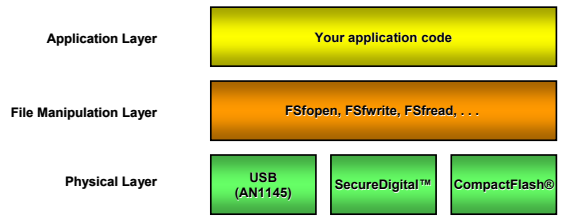
Regional Training Centers

The File System Layer (Microchip MDD File System Library)

Microchip MDD File System Library
- Overview -

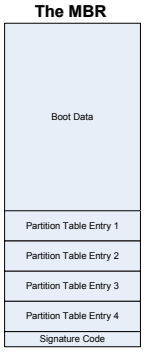
- Provides an interface to file systems compatible with ISO/IEC specification 9293 (commonly referred to as FAT12 and FAT16, used on earlier DOS operating systems by Microsoft® Corporation)
- Also supports the FAT32 file system (long filenames are not supported)
- Supports USB, SD/MMC and CF card physical interfaces.

MDD File System Library Layout



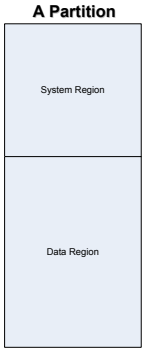
Device Organization

- Flash memory is organized into “sectors”
- The first sector of the device is usually the Master Boot Record (MBR)
- The MBR contains information about partitions



Partition Organization

- Several sectors are reserved for special use (the system region)
- Data sectors are grouped into “clusters”
- Each file and directory uses at least one cluster



MDDFS Operation

- Stores file information in `FSFILE` objects
- Reads/writes in one-sector blocks (typically 512 bytes)
- Caches a sector of the FAT and a sector of data
- FAT and data reads and writes are only performed when necessary
- MDD Function calls are <u>blocking</u>
  - Media interface layer (USB) must arrange to call `USBTasks()` to drive the USB host state machine to complete reads/writes.

Max. Partition Size, File Sizes

- FAT12:
  - Max. Partition Size: 32 MB
  - Max. File Size: 32 MB
- FAT16:
  - Max. Partition Size: 2 GB
  - Max. File Size: 2 GB
- FAT32:
  - Max. Partition Size: 2 TB (Windows: 32 GB)
  - Max. File Size: 4 GB

Licensing

- Before using MDD Library for any supported media in your designs, please review the MDD License Agremment `c:\Microchip Solutions\Microchip\MDD File System\License Agreement.pdf`
- You must investigate potential licensing requirements
  - http://www.microsoft.com/iplicensing/

The FSFILE Structure
- (Defined in FSIO.h)-

- Contains file parameters:
  - The file name and extension
  - Information about the directory that contains the file's entry
  - The current position in the file
  - The starting location of the file on the device
  - The size of the file (in bytes)

Date/Time Sources
- (Defined in FSconfig.h)-

- None
  - Date and time will be written to a static value and incremented when updated
  - Will not put accurate time stamp information in file entries
- Real-time Clock & Calendar
  - Date and time will be updated based on the value in the RTCC module registers
  - User must configure RTCC before operation

- User Defined Clock
  - User updates times manually with SetClockVars function
  - Should be called before creating a file or directory and before closing a file

The FSInit() Function

- Initializes data structures
- Loads device information from the MBR and Boot Sector
- Initializes Media
- Prototype:

```
int FSInit (void);
```

- Returns:
  - TRUE if initialization is successful
  - FALSE otherwise

The FSfopen() Function

- Loads file information or creates a new file
- Prototype:

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

- Arguments
  - fileName: The name of the file to open
  - Mode: READ, WRITE, or APPEND
- Returns:
  - A pointer to the initialized file object on success
  - NULL on failure

Example

```
FSFILE * pointer;
pointer = FSfopen ("FILE.TXT", "w");
if (pointer == NULL)
      // Error
```

The FSfclose() Function

- Updates information in the root and FAT
- Frees the memory used by the FSFILE object
- Prototype:
  `int FSfclose (FSFILE * fo);`
- Argument: A pointer to the file to close
- Returns:
  - 0 if the file was closed successfully
  - EOF (-1) otherwise

```
FSFILE * pointer;
pointer = FSfopen ("FILE.TXT", "w");
if ( FSfclose (pointer) != 0)
      // Error
```


The FSremove() Function

- Deletes the FAT entries for a file
- Marks the root directory entry as deleted
- Prototype:
  `int FSremove (const char * fileName);`
- Argument: The name of the file to delete
- Returns:
  - 0 on success
  - EOF (-1) otherwise

Example

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


The FSrename() Function

- Allows the user to rename files
- Prototype:
  `int FSrename (const char * fileName, FSFILE * fo);`
- Arguments:
  - `fileName`: The new name of the file
  - `fo`: Pointer to the file being renamed
- Returns:
  - 0 if file is renamed successfully
  - EOF (-1) otherwise
- Can also rename directories

Example

```
FSFILE * pointer;
pointer = FSfopen ("FILE.TXT", "w");
if (FSrename ("FILE2.TXT", pointer) != 0)
      // Error
```


MICROCHIP

Regional Training Centers

Lab 4: Creating and Deleting Files

MICROCHIP Regional Training Centers

Lab 4 Objectives

- Complete the "`MKFILE`" command, which will create an empty file
- Complete the "`REN`" command, which will rename an existing file
- Complete the "`DEL`" command, which will delete a file
- Be able to explain the functionality and parameters of:
  - `FSfopen()`
  - `FSfclose()`
  - `FSrename()`
  - `FSremove()`

MICROCHIP Regional Training Centers

Lab 4 Summary

- Files can be created using the `FSfopen()` function
- It's necessary to call `FSfclose()`to update file information when you're done using your file
- The `FSrename()`function can be used to rename an existing file
- The `FSremove()`function can be used to delete an existing file

The FSfread() Function

- Reads 'n' objects of 'size' bytes from an open file and store them in the buffer specified by 'ptr'
- Prototype:

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

- Pre-condition:
  - File is open in READ mode
- Arguments:
  - `ptr`: A pointer to the buffer to store the data
  - `size`: The size of the objects to read (in bytes)
  - `n`: The number of objects to read
  - `stream`: The file to be read from
- Returns: The number of objects (not bytes) read

The FSfeof() Function

- Determines if the user has read up to the end of the file
- Prototype:

```
int FSfeof (FSFILE * stream);
```

- Pre-condition:
  - File is open in READ mode
- Argument: A pointer to the file to check
- Returns:
  - Non-zero if EOF is reached
  - 0 otherwise

The FSfwrite() Function

- This function will write 'n' items of 'size' bytes from the structure pointed to by 'ptr' to the file pointed to by 'stream'
- Prototype:

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

- Pre-condition:
  - File is opened in WRITE or APPEND mode
- Arguments:
  - `ptr`: A pointer to the data to be written
  - `size`: The size of the objects to write
  - `n`: The number of objects to write
  - `stream`: The file the data will be written to
- Returns: The number of objects written

The FSfprintf() Function

- This function will write specially formatted strings to a file
- Prototype:

```
int FSfprintf (FSFILE * fptr, const rom char * fmt, …);
```

- Arguments:
  - `fptr`: The file the data will be written to
  - `fmt`: The string to write
  - …: Format specifiers
- Returns:
  - Count of characters written on success
  - EOF (-1) on failure

FSfprintf() Format Specifiers (See AN1045 for Details)

- Flag characters (-, +, 0, ' ', #)
  - Selects a Prefix
- Field Width
- Field Precision
- Size Specification
- Conversion Specifier

Example

```
unsigned long test = 0x1234ABCD;
FSfprintf (fPtr, "Example %#30.20lX", test);
Output
Example          0X00000000000001234ABCD
```

Flag Characters

- `-`
  The result of the format conversion will be left justified
- `+`
  A '+' sign will be prefixed to a signed result if it is positive (negative results are automatically prefixed by '-')
- `0`
  The result will be prefixed with leading zeros until the field width is filled. Specifying precision or '-' flag will cause this flag to be ignored.
- `{space}`
  Prefixes a space char to the beginning of a positive result. If '+' is also specified, this flag will be ignored.
- `#`
  Present alternate forms of conversion
  Octal (o) results will be increased in precision and preceded by a 0
  Hex (x) conversions will be prefixed by "0x"
  Hex (X) conversions will be prefixed by "0X"
  Binary (b) conversions will be prefixed by "0b"
  Binary (B) conversions will be prefixed by "0B"

Field Width

The field width specifier follows the flags. If the result is shorter than the width, it is padded with leading spaces (or zeros if flagged) until it fills the field width. If the result is left justified, it will be followed by trailing spaces. If the field width is specified by an '*' character a 16-bit argument will be read from the list of format specifiers to specify the width. In this case, if the value is negative, it will be as if the '-' flag is specified, followed by a positive field width.

Field Precision

The field precision specifies the number of digits present in the converted value for an integer conversion or the maximum number of chars in the converted value for a string conversion. It is indicated by a period (.) followed by an integer value or an asterisk. If the precision isn't specified, the default (1) will be used. If it is specified by '*' a 16-bit argument will be read from the list of format specifiers to specify precision.

Size Specifiers

- Applies to any integer or pointer conversion specifier
- Determines what type of argument is read from the format specifier list

Argument	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	-

Conversion Specifiers

- c: The int argument is converted to an unsigned char value and the character represented by that value will be written
- d,i: The int argument is converted to a signed decimal
- o: The unsigned int argument is 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 unsigned binary
- x,X: The unsigned int argument will be converted to an unsigned hexadecimal using the letters 'a'-'f' (x) or 'A'-'F' (X) for 10-15

Conversion Specifiers

- s,S: Characters from a data (s) or program (S) memory array of char arguments are written until either a terminating zero character is seen or the number of chars is equal to the precision.
- p,P: The pointer to a data or program memory is converted to an equivalent sized unsigned int type and processed with the x (p) or X (P) conversion specifier. In C18 the pointer will be a 24-bit pointer if the 'H' size specifier is present; otherwise it will be a 16-bit pointer.
- %: A percent sign will be written.
- n: The number of characters written so far will be stored by this 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 (16-bit if no specifier is present).

Sample FSfprintf() Strings

Function Call	Output
`FSfprintf (p, "Test %10X", 0x12ef);`	`Test       12EF`
`FSfprintf (p, "Test %#20B", 0x12ef);`	`Test   0B0001001011101111`
`FSfprintf (p, "Test %#12.10x", 0x12ef);`	`Test   0x000012ef`
`FSfprintf (p, "Long %lu", 0x12032);`	`Long 73778`
`FSfprintf (p, "String %.5S", "String2");`	`String Strin`
`unsigned char n = 34;` `FSfprintf (p, "Result = %hhu%%", n);`	`Result = 34%`
`unsigned char n = 43;` `FSfprintf (p, "Result = %hhc%%", n);`	`Result = +%`

`FSfprintf (p, "Pos = %+d%c%cNeg = %d%c%cSpc = % d", 0x01, 0x0D, 0x0A, 0xFF, 0x0D, 0x0A, 0x01);`	`Pos = +1`
	`Neg = -1`
	`Spc =  1`

MICROCHIP

Regional Training Centers

Lab 5: Reading and Writing Files

Lab 5 Objectives

- Complete the "LOG" command, to log data from a peripheral
- Complete the "TYPE" command, to display file contents
- Complete the "COPY" command, to copy files or create a file from the terminal
- Be able to explain the functionality and parameters of:
  - `FSfread()`
  - `FSfwrite()`
  - `FSfprintf()`

Lab 5 Summary

- The `FSfwrite()` function can be used to write information to a file
- The `FSfread()` function can be used to read information from a file
- The `FSfprintf()` function will write specially formatted information to a file

FSconfig.h

- Firmware configuration options are located in `FSconfig.h`
  - Max concurrent open files
  - Media sector size
  - Timestamp clock mode
  - Feature disable options
  - Static/dynamic `FSFILE` allocation
  - Function pointers

HardwareProfile.h

- Hardware configuration options are located in `HardwareProfile.h`
  - System clock
  - Physical Interface type
  - I/O selection

MICROCHIP

Regional Training Centers

Common Failure Modes

Common Reported Failure Modes

- Unsupported Sector Size (MDDFS)
  - 512 bytes is most common
  - Editable in `FSconfig.h`
- Unsupported Media Interface (MSD Client)
  - Most use SCSI
  - Some use SFF-8070i
- Unsupported File System (MDDFS)
  - We support FAT
- If your thumb drive is not working and you *don't* see these errors, let us know!

Detecting/Responding to Errors

- Unsupported Sector Size (MDDFS)
  - Call `FSerror()` after failed `FSinit()`
  - Returns `CE_UNSUPPORTED_SECTOR_SIZE`
- Unsupported Media Interface (MSD Client)
  - Failure detected during enumeration
  - `EVENT_UNSUPPORTED_DEVICE` passed to the application event handler
- Unsupported File System (MDDFS)
  - Call `FSerror()` after failed `FSinit()`
  - Returns `CE_UNSUPPORTED_FS`

Summary (Part 3)

- We covered:
  - Basic USB Thumb Drive Application Architecture
    - USB Mass Storage Device Basics
    - Configuration of the Mass Storage Client Driver
    - MDD File System Library functions
    - Configuring the MDD File System Library

Additional Resources (Part 3)

- Application Note 1045, “Implementing File I/O Functions Using Microchip’s Memory Disk Drive File System Library”,
- Application Note 1142, “USB Mass Storage Class on an Embedded Host”,
- Application Note 1145, “Using a USB Flash Drive with an Embedded Host”,
- All available at Microchip Technology’s USB Development page at http://www.microchip.com/usb/

Additional Resources (Part 3)

- ISO®/IEC Specification 9293, "Information technology – Volume and file structure of disk cartridges for information interchange," available from www.iso.org
- "FAT32 File System Specification," available from Microsoft
- The USB Implementers Forum, Inc. at www.usb.org
- USB Mass Storage, by Jan Axelson (ISBN: 9781931448048)
- USB Complete 4/E, by Jan Axelson (ISBN: 9781931448086)

MICROCHIP

Regional Training Centers

Part 4

Using the USB Thumb Drive Boot Loader Application

Objectives (Part 4)

- Learn how to implement the USB Thumb Drive Boot Loader and application

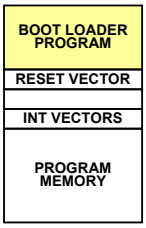
Agenda (Part 4)

- Definition
- USB Bootloader Architecture
- Application Configuration & Use
  - Lab 6: Update Firmware Using the Thumb Drive Boot Loader
- Summary and Questions

Boot Loader

Definition

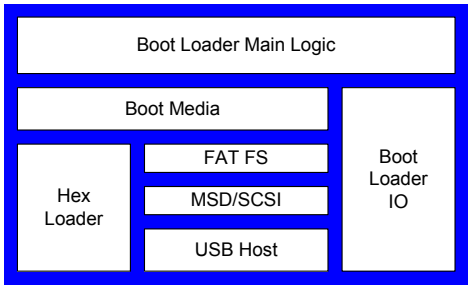
A *BOOT LOADER* is firmware located in a microprocessor's program memory which allows users to change the application code using an external interface such as RS-232, I2C, SPI or USB instead of using a special programming interface (e.g. ICSP or JTAG)



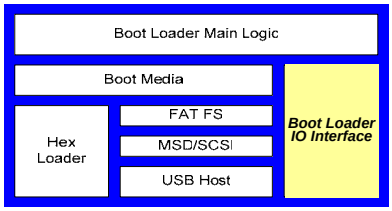
CONCEPTUAL VIEW OF PROGRAM FLASH

- FLASH must be "self-writing"
- Program memory "shifted"
- Linker scripts:
  - Boot loader code
  - Application code
- Boot loader dictates "programming" media

USB Host Boot Loader Architecture

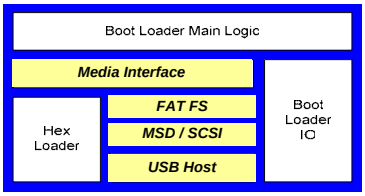


USB Host Boot Loader Architecture



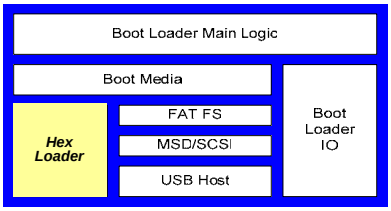
- Invoke or Cancel boot loader mode
- Communicate boot loader status
- APIs and build parameters in `boot_io.h` file

USB Host Boot Loader Architecture



- Defines medium used to access application
- Default wrappers for:
  - MDD FAT FS
  - USB MSD stack
  - Hex Loader
- APIs and build parameters in `boot_media.h` file

USB Host Boot Loader Architecture



- Used to decode and program image
- Default format is Intel Hex record
  - Replace or modify to change format
- APIs and build parameters in `boot_load.h` file

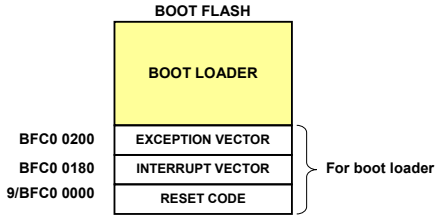
USB Host Boot Loader Basic Operation

```
Execute boot loader startup code
If (bootstrap condition met)
   {
   Initialize boot loader USB host stack
   Find application firmware file
   Parse file
   Program to flash
   }
Jump to application startup code
```

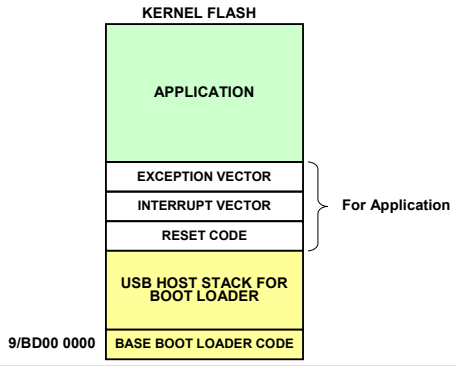

USB Host Boot Loader Restrictions

- FS Format: MDD FAT 16/32
- File Format: Intel Hex record
- Boot File Name: Change in `boot_config.h`
  - Default is "`image.hex`"
- USB Support: boot loader has host stack
- Configuration bits must be defined in boot loader code
- Program Flash must be shared with boot loader code
- Special linker scripts must be used for the application

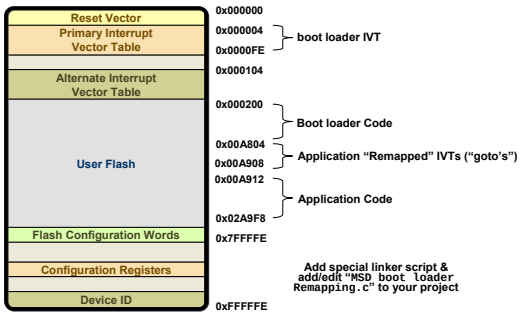
PIC32MX460 USB Host Boot Loader Memory Layout - Flash



PIC32MX460 USB Host Boot Loader Memory Layout – Kernel Flash



PIC24GB110 USB Host Boot Loader Default Memory Layout - Flash



USB Host Boot Loader `boot_config.h`

- **Define application file name**

```
#define BOOT_FILE_NAME "image.hex"
```

- **Define size of read buffer**

```
#define BL_READ_BUFFER_SIZE 512
```

- **Define application addressing**

```
// Address of main application's Startup code
#define APPLICATION_ADDRESS          0x9D020000

// These macros define the maximum size of a Flash
   block. Change procdefs.ld if these change.
#define PROGRAM_FLASH_BASE           0x1D020000
   // Physical address
#define PROGRAM_FLASH_LENGTH         0x00060000
   // Length in bytes
#define FLASH_BLOCK_SIZE             (1024 * 4)
   // Size in bytes
```

Lab 6 Objectives

- Build/run an existing application
  - USB "Generic Device" Demo
- Prepare the application for use with the thumb drive boot loader
- Program the thumb drive boot loader into the device
- Load/run the modified application from a thumb drive.

Summary (Part 4)

- We covered:
  - USB Thumb Drive BootloaderArchitecture
  - Configuring an Application for use with the boot loader

Additional Resources (Part 4)

- PIC32 USB Thumb Drive Boot loader Documentation
  - See Appendix D in Lab Manual

Class Summary

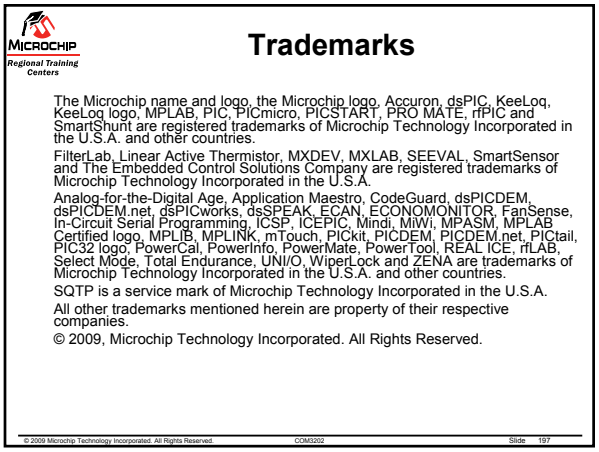
- Covered what USB embedded host enabled options are available, how they are different and where/when they should be selected.
- Covered how to use the Microchip USB Embedded Host framework to create a custom USB peripheral application on a PIC24/PIC32-based USB embedded host.
- Covered how to use the Microchip USB Embedded Host framework to add USB thumb drive capability to your application

Summary: App Note References

- www.microchip.com/usb -

- AN1045
  - Implementing File I/O Operations Using Microchip's MDD FS Library
- AN1140
  - USB Embedded Host Stack
- AN1141
  - USB Embedded Host Stack Programmers Guide.
- AN1142
  - USB Mass Storage Class on an Embedded Host.
- AN1143
  - USB Generic Client on an Embedded Host.
- AN1145
  - Using a USB Flash Drive on an Embedded Host


MICROCHIP
Regional Training Centers
Thank You.


MICROCHIP
Regional Training Centers
Trademarks
The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeeLoq, KeeLoq logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.
Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC32 logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.
All other trademarks mentioned herein are property of their respective companies.
© 2009, Microchip Technology Incorporated. All Rights Reserved.
© 2009 Microchip Technology Incorporated. All Rights Reserved.
COM3202
Slide 197


MICROCHIP
Regional Training Centers
Appendix
- OTG Slides From COM3202 v0.95-

Agenda

- Overview
- Mechanical
- Protocol
- Electrical
- Certification Considerations
- Resources (Examples, Classes, Software, etc.)

New 5th pin

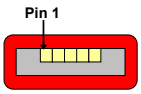
- Old connectors had 4 pins on the receptacle that were used: VBUS, GND, D+, and D-
- OTG connectors have 5 pins on the receptacle that are used: VBUS, GND, D+, D-, and ID
  - ID pin is used to determine which side of the cable is the A (host) side
  - ID should be pulled high through a resistor
    - Built into PIC24F and PIC32MX device with USB OTG devices

Mechanical

- OTG Plugs and Receptacles
  - Micro-B plug and receptacle 
  - Micro-A/B receptacle 
    - Only allowed on OTG products
  - Micro-A plug 
    - Indicates who is initially the hos

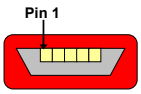
Mechanical

- Micro A/B Receptacle
  - Pin 1: VBus
  - Pin 2: D-
  - Pin 3: D+
  - Pin 4: ID
  - Pin 5: GND



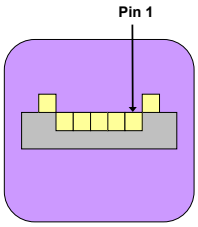
Mechanical

- Micro B Receptacle
  - Pin 1: VBus
  - Pin 2: D-
  - Pin 3: D+
  - Pin 4: ID
  - Pin 5: GND



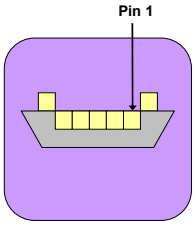
Mechanical

- Micro A Plug
  - Pin 1: VBus
  - Pin 2: D-
  - Pin 3: D+
  - Pin 4: GND (ID)
  - Pin 5: GND

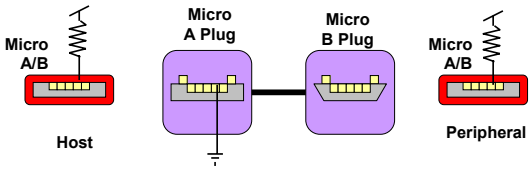


Mechanical

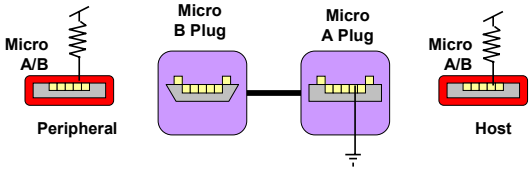
- Micro B Plug
  - Pin 1: VBus
  - Pin 2: D-
  - Pin 3: D+
  - Pin 4: Floating (ID)
  - Pin 5: GND



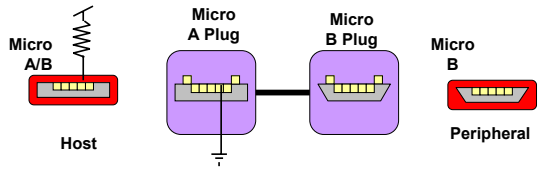
OTG Cable Example



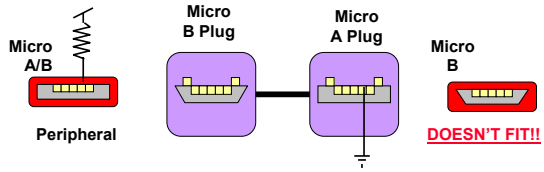
OTG Cable Example



OTG Cable Example



OTG Cable Example



Mechanical

- Cables
  - Allowable Types
    - Micro-A plug to Micro-B plug
    - Micro-A plug to Standard-A receptacle
      - Connect Std USB Thumb drive to OTG Host
    - Micro-B plug to Standard-A plug
      - Connect OTG B-device to PC Host
    - Captive cable with Micro-A plug
  - Length
    - 2 meters or less (different from USB-v2.0 limit of 5 meters)

Agenda

- Overview
- Mechanical
- Protocol
- Electrical
- Certification Considerations
- Resources (Examples, Classes, Software, etc.)

Agenda (Protocol)

- OTG Descriptor
- Set Feature Requests
- Targeted Peripheral List
- Session Request Protocol (SRP)
- Host Negotiation Protocol (HNP)

OTG Descriptor

- Returned in the GetDescriptor(Configuration) request
  - Required only if B-Device supports either SRP or HNP

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor (always 3)
1	bDescriptorType	1	Constant	OTG type = 9
2	bmAttributes	1	Bitmap	Attribute Fields D7-D2: reserved D1: HNP supported D0: SRP supported

Set Feature Requests

- `a_hnp_support`
  - Lets the B-device know that the A-Device supports HNP
  - Only allowed to set on B-devices that support HNP
  - Must be set before the device configuration is set
- `a_alt_hnp_support`
  - Lets the B-device know that it is connected to a port that does not support HNP but the A-Device has a port available that does.
- `b_hnp_enable`
  - Lets the B-device know that it is allowed to perform HNP

Set Feature Requests

- Can be set in the default, address, or configured states
- Only cleared at the end of a session or on a bus reset
  - Clear feature does not work on these features
- If HNP is not supported on the B-Device then it should STALL on any of these Set Feature requests

Targeted Peripheral List (TPL)

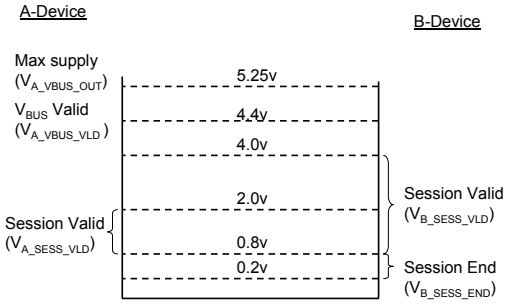
- List of supported devices for that embedded host and OTG
  - Devices not on that list will not be able to enumerate
  - Not able to list classes for OTG, allowable on embedded host
- Manufacturer, Model, and Description are minimally required

Session Request Protocol (SRP)

- Saves power on A-Device
  - B-Device needs way to request VBUS from A-Device
- Session
  - The time between when VBus rises above the valid threshold until it drops below the valid threshold again

2

Session Request Protocol (SRP)



Session Request Protocol (SRP)

- SRP support
  - DRDs required to be able to respond to and initiate SRP
  - A-Devices allowed to respond to SRP
  - B-Devices allowed to initiate SRP

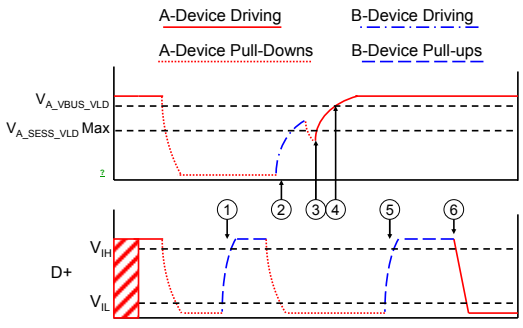
Session Request Protocol (SRP)

- Requesting V_{BUS}
  - V_{BUS} pulsing and/or D+ pulsing
    - B-Device required to be able to initiate both V_{BUS} and D+ pulsing
    - A-Device only required to recognize one of the two

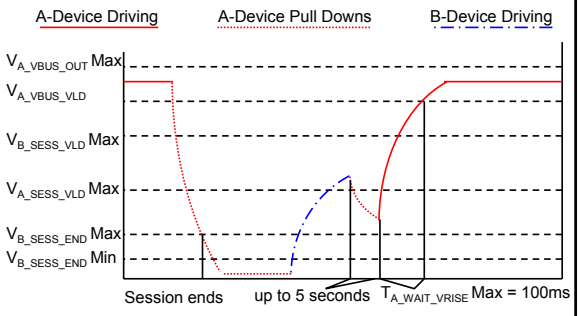
Session Request Protocol (SRP)

- B-Device
  - Before attempting to start new session must first determine the previous session has ended
    - Time the decay of the previous session end
    - Pull V_{BUS} down to speed up end of session

Session Request Protocol (SRP) D+ Pulsing and V_{BUS} Pulsing

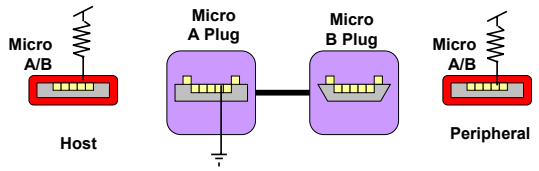


Session Request Protocol (SRP) V_{BUS} Pulsing



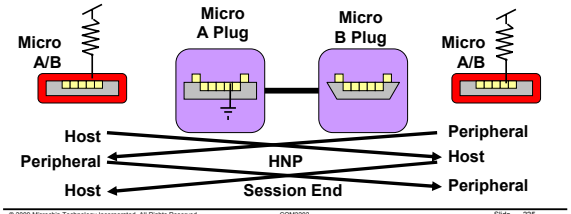
Host Negotiation Protocol (HNP)

- Cable determines which device is the host (A-Device) and the peripheral (B-Device)
  - Whichever device that has the Micro A plug plugged into its Micro A/B receptacle is the default host/A-Device



Host Negotiation Protocol (HNP)

- HNP allows devices to switch roles without having to switch cable
  - The B-Device will become the host until the session ends
  - A-Device continues to source the V_{BUS} power

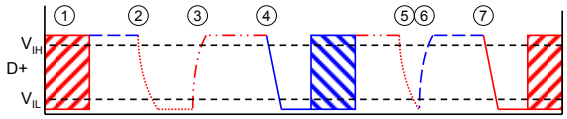


Host Negotiation Protocol

1) A-Device uses SetFeature(HNP)
2) During suspend the B-Device turns off D+ pull-up
3) A-Device turns D+ pull-up on.
4) B-Device detects D+ pull-up and asserts a bus reset
5) When B-Device is done, stops all bus activity and enables its D+ pull-up after idle state is reached
6) A-Device detects idle and disables its pull-up
7) A-Device either asserts reset or turns off V_{BUS}

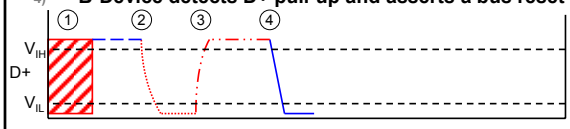
Host Negotiation Protocol (HNP)

A Host Bus Traffic
B Host Bus Traffic
A-Device Driving
A-Device Pull-Downs
A-Device Pull-ups
B-Device Driving
B-Device Driving
B-Device Pull-ups



Host Negotiation Protocol (HNP)

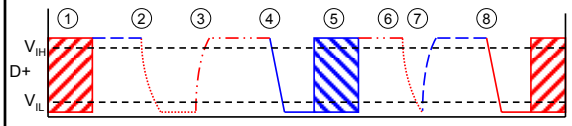
1) A-Device uses SetFeature(HNP)
2) During suspend the B-Device turns off D+ pull-up
3) A-Device turns D+ pull-up on
4) B-Device detects D+ pull-up and asserts a bus reset



A Host Bus Traffic
B Host Bus Traffic
A-Device Driving
A-Device Pull-Downs
A-Device Pull-ups
B-Device Driving
B-Device Driving
B-Device Pull-ups

Host Negotiation Protocol (HNP)

5) B is now the host and controls the bus
6) When B-Device is done, stops all bus activity
7) On the Idle condition the B-Device enables its D+ pull-up and the A-device disables its pull-up
8) A-Device either asserts reset or turns off V_{BUS}



A Host Bus Traffic
B Host Bus Traffic

A-Device Driving
A-Device Pull-Downs
A-Device Pull-ups

B-Device Driving
B-Device Driving
B-Device Pull-ups

Quiz!

1) True or False: I can make a compliant OTG device that supports all thumb drives (memory sticks).
2) I have an OTG cable.
   1) What plugs are on the cable?
   2) How do I know which is the default peripheral and host?
3) What is SRP used for?
4) What is HNP used for?

Agenda

- Overview
- Mechanical
- Protocol
- Electrical
- Certification Considerations
- Resources (Examples, Classes, Software, etc.)

Agenda (Electrical)

- VBus
  - Currents
  - Capacitance and Resistance limits
  - Rise/Fall times
- ID resistances
- Signal Propagation Times

OTG Current Sourcing Requirements

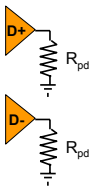
- A-Devices supporting loads <= 100mA
  - $I_{A\_VBUS\_OUT}$ min = 8mA
  - 4.4v <= $V_{A\_VBUS\_OUT}$ <= 5.25v
  - Must error if $V_{A\_VBUS\_OUT} < V_{A\_VBUS\_VLD}$
- A-Devices supporting loads > 100mA
  - 4.75v <= $V_{A\_VBUS\_OUT}$ <= 5.25v

OTG Current Draw Limits

- Dual Role Device
  - Unconfigured: 150uA average over 1ms
- Peripheral Only
  - Unconfigured: 8mA average over 1ms

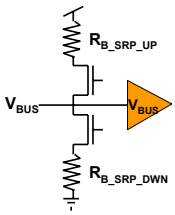
Pull-ups and Pull-downs

- A-Device
  - When Idle or acting as host, D- and D+ pull downs enabled
  - When acting as a peripheral, D+ pull down is disabled
  - Allowed to disable the pull downs during the interval of packet transmission when either a host or a peripheral
  - 14.25KΩ < Rpd < 24.8KΩ

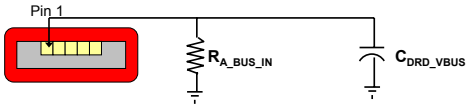


Pull-ups and Pull-Downs

- B-Device
  - $R_{B\_SRP\_UP}$ > 281Ω
  - $R_{B\_SRP\_DWN}$ > 656Ω
  - D+ pull-up same as USB 2.0 devices

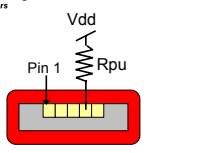


VBus (OTG)
- Allowable Resistance & Capacitance -



- When A-Device is powered but not supplying V_{Bus}, $R_{A\_BUS\_IN}$ max <= 100KΩ
- If A-Device supports V_{Bus} pulsing for SRP, $R_{A\_BUS\_IN}$ min >= 40KΩ, otherwise it can be lower
- 1.0µF < C_{DRD_VBUS} < 6.5µF
  - As compared to C_{HST_VBUS} > 120µF

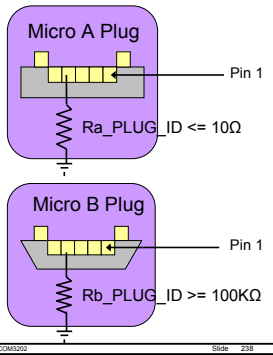
ID Resistances



$$\frac{Vdd * Ra\_PLUG\_ID}{(Rpu + Ra\_PLUG\_ID)} < V_{IL\text{-}MAX}$$

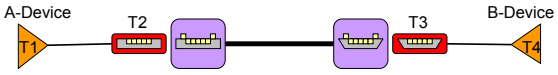
$$\frac{Vdd * Rb\_PLUG\_ID}{(Rpu + Rb\_PLUG\_ID)} > V_{IH\text{-}MIN}$$

For Vdd = 3.3v,
56.67Ω < Rpu < 25KΩ



Propagation Times (max)

- **T1 to T2 – From pin of controller in A-Device to pin of the USB connector**
  - **OTG A-Device = 1ns**
  - **Embedded Host or Full host = 3ns**
- **T2 to T3 – From pin of connector on A-Device to pin of connector on B-Device**
  - **OTG Cables = 10ns**
  - **Standard Cables = 26ns**
  - **Micro-A to Standard-A adaptor = 1ns**
- **T3 to T4 – From pin of connector to pin of controller on B-Device**
  - **1ns**



Quiz!

1) **True or False: If I plug in any 100mA normal USB device into an OTG device, everything should always be fine.**
2) **True or False: There is no electrical difference between an OTG host and an Embedded host.**

OTG vs. Embedded Host

	OTG	Embedded Host
SRP	Required	Optional
HNP	Required	Disallowed
Targeted Peripheral list	Limited to specific Manufacturer/ Model/ Description entries	Allowed to support generic classes (i.e.- any HID mouse)
Mechanical	Micro A/B	A
Electrical	1.0µF < C_{DRD_VBUS} < 6.5µF	C_{HST_VBUS} > 120µF

Agenda

- Overview
- Mechanical
- Protocol
- Electrical
- Certification Considerations
- Resources (Examples, Classes, Software, etc.)

Certification Considerations Embedded Host

- Checklists
  - Systems
- No Silent Failures
  - Hub error message
  - Device not supported message
- Power
  - Over current notification
  - Resettable over current protection
  - Drop voltage
- TPL

Certification Considerations OTG

- Checklists
  - OTG
  - Peripheral
  - Systems
- SRP
- HNP
- TPL
- Power restrictions
  - Un-configured power

Certification Considerations DRD

- Port accessibility
  - If more than one connector is accessible at any point of time then they need to be able to work at the same time
- Checklists
  - Peripheral
  - Systems