Microchip's Accessory Framework for Android Help

Microchip Libraries for Applications (MLA)

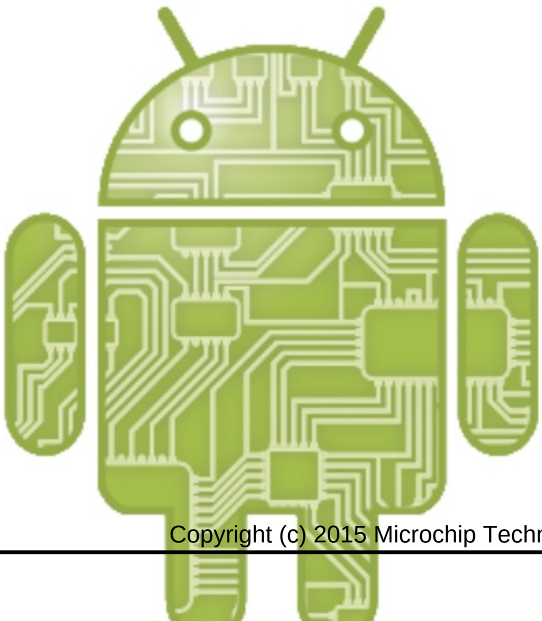


Table of Contents

Microchip's Accessory Framework for Android Help

1 Microchip's Accessory Framework for Android

Modules

Name	Description
Library Interface	This section describes the Application Programming Interface (API) functions of the Microchip's Accessory Framework for Android. Refer to each section for a detailed description.

Description

This section describes the Application Programming Interface (API) functions of the Microchip's Accessory Framework for Android.

Refer to each section for a detailed description.

1.1 Introduction

Microchip's Accessory Framework for Android

for

Microchip Microcontrollers

The Microchip's Accessory Framework for Android for Android provides a mechanism to transfer data to and from an Android application through the USB of the microcontroller.

Description

The Microchip's Accessory Framework for Android for Android provides a mechanism to transfer data to and from an Android application through the USB of the microcontroller.

1.2 Release Notes

General library release information.

Description

Microchip's Accessory Framework for Android Version 1.04.02, 2016-04-28

Requirements

The Microchip's Accessory Framework for Android requires Android versions v2.3.4 or v3.1 or later. The Open Accessory API is not available in OS versions earlier than this. If the target device is using an older version than this, the library will not be able to connect to that device.

Known Issues:

1. The read() function in the Android OS will not throw an IOException when the file stream under it is closed on file streams created from the USB Open Accessory API. This creates issues when applications or services close down and try to free resources. If a read is in progress, then this can result in the ParcelFileDescriptor object being locked until the accessory is detached from the Android device. This is present in version v2.3.4 and v3.1 of the Android OS.
   - Workaround: Since the Read() request never completes resulting in locked resources, a workaround can be implemented in the application layer. If the accessory and the application implement a command for the application to indicate to the accessory that the app is closing (or is being paused), then the accessory can respond back with an acknowledge packet. When the app receives this ACK packet, it knows not to start a new read() (since that read() request will not be able to terminated once started).
2. The available() function in the Open Accessory API in the Android OS always throws an IOException error. This function is not available for use.
3. This release only shows connecting to an Android device with the Android device as the USB device. Most phones and tablets operate in this mode. A few Android devices at the time of this release are capable of being a USB host as well. Examples for using this mode of operation are not provided in this release. Firmware to talk to these host capable Android devices can be found at www.microchip.com/usb or www.microchip.com/mal. Application example to access accessories running in device mode will follow shortly. The Open Accessory API allows connections in either direction.

1.2.1 Terms and Definitions

Describes terms used by this document.

Description

This section defines some of the terms used in this document.

Open Accessory API or **Open Accessory Framework** - this is the API/framework in the Android development environment that allows the Android applications to transmit data in and out of the available USB port. This is provided by Google through the Android SDK.

Microchip's Accessory Framework for Android - This defines the firmware library and Android application examples provided in this package by Microchip.

1.2.2 Supported Demo Boards

Describes which demo boards are supported by this software release.

Description

The following demo boards are supported in this release:

- Accessory Development Starter Kit for Android (PIC24F Version) (DM240415)
- Explorer 16 (DM240001) with USB PICtail+ Board (AC164131) with any of the following Processor Modules:
  - PIC24FJ256GB110 PIM (MA240014)
  - PIC24FJ64GB004 PIM (MA240019)
  - PIC24FJ256GB210 PIM (MA240021)
- PIC24F Starter Kit (DM240011)

Since each board has different hardware features, there may be limitations on some of the boards for each of the demos. For example, if the board does not have a potentiometer but the demo uses one, that feature of the demo will not work.

1.2.3 Revision History

Specific release revision information.

Description

Specific release revision information.

1.2.3.1 1.04.02

Updated for changes in the USB host include files. No changes to Android drivers or demos.

1.2.3.2 1.04.01

Updated the demos to use the new HID host API.

1.2.3.3 1.04

The Android development libraries have been ported to the MLA's new folder structure and re-integrated into the MLA. Data types and code style have been modified to conform to the new MLA standards.

1.2.3.4 1.02.01

Made changes to allow some tablets that do not follow the AOA protocol correctly to attach. Some devices appear to use the wrong VID/PID in AOA mode. Previously these devices would not work because the AOA driver was validating the VID/PID specified in the AOA protocol document.

1.2.3.5 1.02

Updated to support audio and HID controls released in Android Open Accessory Protocol version 2 (2012).

1.2.3.6 1.01.02

1. Fixed issue in Android OpenAccessory driver that could cause a memory leak.
    - Stack files affected: usb_host_android.c (USB stack files)

1.3 SW License Agreement

This software distribution is controlled by the Legal Information at www.microchip.com/mla_license.

Description

This software distribution is controlled by the Legal Information at www.microchip.com/mla_license.

1.4 Using the Library

This topic describes the basic architecture of the Microchip's Accessory Framework for Android and provides information and examples on how to use it.

Description

This topic describes the basic architecture of the Microchip's Accessory Framework for Android and provides information and examples on how to use it.

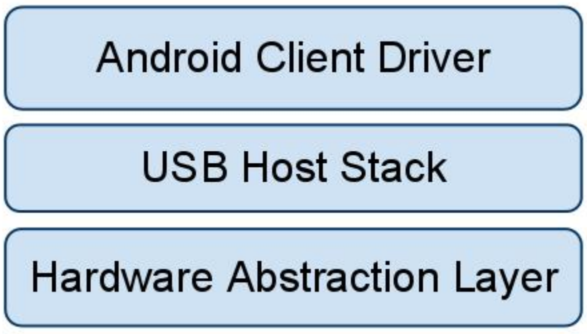
Interface Header File: usb_host_android.h

1.4.1 Library Architecture

Describes the organization of library modules.

Description

The Android Accessory driver when in host mode is just a client driver on top on top of the Microchip USB host stack as seen in the below diagram.



Users interface through the Android driver API described in this document to access the Android device.

1.4.2 How the Library Works

Describes how the library works.

Description

This section describes how the library works.

1.4.2.1 Configuring the Library

Describes how to configure the library.

Description

The source code for Microchip's Accessory Framework for Android is distributed with Microchip's Libraries for Applications. This software package can be obtained from www.microchip.com/mla.

Please select the instructions in the following sections that correspond to the version that you are using.

1.4.2.1.1 Required USB Callbacks

Describes callback function that the user must implement in application code.

Description

Microchip's Accessory Framework for Android is currently based off of Microchip's USB host stack. The USB host stack uses a couple of call back functions to allow the user to make key decisions about the stack operation at run time. These functions must be implemented for the library to compile correctly. The function names are configurable through the usb_config.h file (see the usb_config.h section for more information); the default function names in the demo code are `USB_ApplicationEventHandler()` and `USB_ApplicationDataEventHandler()`. For more detailed information about these functions or the USB library, please refer to www.microchip.com/mla. This download includes the USB host library code as well as more detailed documentation about that library.

The data events are consumed by the Android client driver. So the user application data event handler doesn't need to do anything. It needs to be present for the library to link successfully but it can just return FALSE.

```
BOOL USB_ApplicationDataEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size
)
{
    return FALSE;
}
```

The regular event handler has a little more work that needs to be done. This handler notifies the user of device attach and detach events. For Android devices this is covered in the Detecting a Connection/Disconnection to an Android Device section. This function also notifies the user about errors that might occur in the USB subsystem. These can be useful for debugging development issues or logging issue once released in the field. The last important duty that this function provides is determining if the power required by the attached device is available. This is done through the EVENT_VBUS_REQUEST_POWER event. Remember in this event that the amount of power requested through the USB bus is the power required/2, not the power required. Below is full implementation of the USB event handler that will work with a single attached Android device.

```
BOOL USB_ApplicationEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    switch( event )
    {
        case EVENT_VBUS_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 (((USB_VBUS_POWER_EVENT_DATA*)data)->current <= (MAX_ALLOWED_CURRENT / 2))
            {
                return TRUE;
            }
            else
            {
                DEBUG_ERROR( "Device requires too much current\r\n" );
            }
            break;

        case EVENT_VBUS_RELEASE_POWER:
        case EVENT_HUB_ATTACH:
        case EVENT_UNSUPPORTED_DEVICE:
        case EVENT_CANNOT_ENUMERATE:
        case EVENT_CLIENT_INIT_ERROR:
        case EVENT_OUT_OF_MEMORY:
        case EVENT_UNSPECIFIED_ERROR:
        case EVENT_DETACH:
             //Fall-through
        case EVENT_ANDROID_DETACH:
            device_attached = FALSE;
            return TRUE;
            break;

        // Android Specific events
        case EVENT_ANDROID_ATTACH:
            device_attached = TRUE;
            device_handle = data;
            return TRUE;

        default :
            break;
    }
    return FALSE;
}
```

1.4.2.1.2 usb_config.h

Describes header file information that must be specified to configure USB.

Description

usb_config.h is used to configure the build options of this library. This file provides several configuration options for customizing the USB stack. There are a few options that are required. The Microchip USBConfig utility can be used to create an appropriate usb_config.h configuration file.

The `USB_SUPPORT_HOST` option must be enabled.

The `USB_ENABLE_TRANSFER_EVENT` option must be enabled.

The `USB_HOST_APP_DATA_EVENT_HANDLER` option must be defined and the function that is referenced must be implemented.

The `USB_ENABLE_1MS_EVENT` option must be enabled.

The `AndroidTasks()` function should be added to the `USBTasks()` function call or it should be called periodically from the user application.

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

The `USB_SUPPORT_BULK_TRANSFERS` option should be defined.

If you modify the TPL in the usb_config.c file (see section usb_config.c for more details), then the `NUM_TPL_ENTRIES` and `NUM_CLIENT_DRIVER_ENTRIES` entries in the usb_config.h file should be updated to match.

Below is a complete example of a usb_config.h file for an Android accessory demo:

```
#define _USB_CONFIG_VERSION_MAJOR 0
#define _USB_CONFIG_VERSION_MINOR 0
#define _USB_CONFIG_VERSION_DOT   12
#define _USB_CONFIG_VERSION_BUILD 0

#define USB_SUPPORT_HOST

#define USB_PING_PONG_MODE  USB_PING_PONG__FULL_PING_PONG

#define NUM_TPL_ENTRIES 9
#define NUM_CLIENT_DRIVER_ENTRIES 3

#define USB_ENABLE_TRANSFER_EVENT

#define USB_HOST_APP_DATA_EVENT_HANDLER USB_ApplicationDataEventHandler
//#define USB_ENABLE_SOF_EVENT
#define USB_ENABLE_1MS_EVENT
```

```
#define USB_MAX_GENERIC_DEVICES 1
#define USB_NUM_CONTROL_NAKS 20
#define USB_SUPPORT_INTERRUPT_TRANSFERS
#define USB_SUPPORT_BULK_TRANSFERS
#define USB_SUPPORT_ISOCHRONOUS_TRANSFERS
#define USB_NUM_INTERRUPT_NAKS 3
#define USB_INITIAL_VBUS_CURRENT (100/2)
#define USB_INSERT_TIME (250+1)
#define USB_HOST_APP_EVENT_HANDLER USB_ApplicationEventHandler

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

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

For more information about the usb_config.h file, please refer to the MCHPFSUSB stack help file.

1.4.2.1.3 usb_config.c

Describes source file information that must be specified to configure USB.

Description

The usb_config.c file contains structures needed by the USB Library. There are two main sections to the usb_config.c file. The first is the Targeted Peripheral List (TPL). The TPL is defied by the USB OTG specification as the list of devices that are allowed to enumerate on the device. The TPL is just an array of USB_TPL objects that specify the devices that can be attached. There are two ways that these devices are entered into the table. They are either entered as Class/Subclass/Protocol pairs (CL/SC/P). The second method is by Vendor ID (VID) and Product ID (PID) pairs. This will allow a specific device, not device type, to enumerate.

As of Android Open Accessory Protocol v2.0, there are six entries that are needed for Android devices. When attempting to find an appropriate client driver for an Android device, the USB Library will attempt to match the device's descriptor values to entries in the TPL. Since each Android device may appear different to the host controller, there isn't an easy way to add support for a given Class/Subclass/Protocol pair since each Android device might expose different USB interface types. Likewise, since there isn't a list of all VID/PIDs for Android devices, and this implementation isn't future-proof, you can use the normal VID/PID entry either. For cases like these it may be useful to use the TPL_IGNORE_CLASS, TPL_IGNORE_SUBCLASS, and TPL_IGNORE_PROTOCOL flags to indicate that this driver should enumerate every device regardless of its interfaces or its actual class type pair.

```
// *****************************************************************************
// USB Embedded Host Targeted Peripheral List (TPL)
// *****************************************************************************
USB_TPL usbTPL[NUM_TPL_ENTRIES] =
{
      /*[1] Device identification information
        [2] Initial USB configuration to use
        [3] Client driver table entry
        [4] Flags (HNP supported, client driver entry, SetConfiguration() commands allowed)
    ---------------------------------------------------------------------
                [1]                      [2][3] [4]
    ---------------------------------------------------------------------*/
    { INIT_CL_SC_P( 0x01ul, 0x01ul, 0x00ul ),   0, 2, {TPL_CLASS_DRV | TPL_IGNORE_PROTOCOL}
}, // Audio class
    { INIT_CL_SC_P( 0x01ul, 0x02ul, 0x00ul ),   0, 2, {TPL_CLASS_DRV | TPL_IGNORE_PROTOCOL}
}, // Audio class
    { INIT_CL_SC_P( 0xFFul, 0xFFul, 0xFFul ),   0, 0, {TPL_CLASS_DRV | TPL_IGNORE_CLASS |
TPL_IGNORE_SUBCLASS | TPL_IGNORE_PROTOCOL} }, // Android accessory

    { INIT_VID_PID( 0x18D1ul, 0x2D00ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
```

```
    { INIT_VID_PID( 0x18D1ul, 0x2D01ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
    { INIT_VID_PID( 0x18D1ul, 0x2D02ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
    { INIT_VID_PID( 0x18D1ul, 0x2D03ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
    { INIT_VID_PID( 0x18D1ul, 0x2D04ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
    { INIT_VID_PID( 0x18D1ul, 0x2D05ul ), 0, 1, {TPL_EP0_ONLY_CUSTOM_DRIVER} }, //
Enumerates everything
};
```

All of the entries that correspond to the Android accessory device should point to the entry in the Client Driver table the corresponds to the Android drivers. In the above example the two CL/SC/P entries for audio devices point to client driver entry 2, which contains audio handling functions. The six VID/PID entries that correspond to Android devices point to client driver entry 1, which provides handling functions for the app and specifies the ANDROID_INIT_FLAG_BYPASS_PROTOCOL flag, which indicates that the device is already know to be in accessory mode. The CL/SC/P entry that ignores class, subclass, and protocol points to entry 0, which contains the same app handling functions but does not bypass initialization. The client driver table needs register the functions used by each driver. The functions that need to be registered are the Initialization function, the event handler, the data event handler (if implemented), and the initialization flags value (see example below).

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

CLIENT_DRIVER_TABLE usbClientDrvTable[NUM_CLIENT_DRIVER_ENTRIES] =
{
    {
        AndroidAppInitialize,
        AndroidAppEventHandler,
        AndroidAppDataEventHandler,
        0
    },
    {
        AndroidAppInitialize,
        AndroidAppEventHandler,
        AndroidAppDataEventHandler,
        ANDROID_INIT_FLAG_BYPASS_PROTOCOL
    },
    {
        USBHostAudioV1Initialize,
        USBHostAudioV1EventHandler,
        USBHostAudioV1DataEventHandler,
        0
    }
};
```

For more information about the usb_config.c file, please refer to the MCHPFSUSB documentation available in the installation found at www.microchip.com/mla.

1.4.2.2 Initialization

Describes how to initialize the library.

Description

There are two main steps to initializing the firmware. The first is to initialize the USB host stack. This is done via the `USBInitialize()` function as seen below:

```
USBInitialize(0);
```

The second step that is required for the initialization is for the application to describe the accessory to the Android client driver so that it can pass this information to the Android device when attached. This is done through the `AndroidAppStart()` function. This function takes in a pointer an `ANDROID_ACCESSORY_INFORMATION` structure which contains all of the information about the accessory. An example is seen below:

```
static char manufacturer[] = "Microchip Technology Inc.";
static char model[] = "Basic Accessory Demo";
static char description[] = DEMO_BOARD_NAME_STRING;
static char version[] = "2.0";
static char uri[] =
"https://play.google.com/store/apps/details?id=com.microchip.android.BasicAccessoryDemo_API1
2&hl=en";
static char serial[] = "N/A";

ANDROID_ACCESSORY_INFORMATION myDeviceInfo =
{
    manufacturer,
    sizeof(manufacturer),
    model,
    sizeof(model),
    description,
    sizeof(description),
    version,
    sizeof(version),
    uri,
    sizeof(uri),
    serial,
    sizeof(serial)
};

int main(void)
{
    //Initialize the USB host stack
    USBInitialize(0);

    //Send my accessory information to the Android client driver.
    AndroidAppStart(&myDeviceInfo);

    //Go on with my application here...
```

Note that the `AndroidAppStarter()` function should be called before the Android device is attached. It is recommended to call this function after initializing the USB Android client driver but before calling the `USBTasks()` function.

1.4.2.3 Keeping the Stack Running

Describes how to create an application framework to allow the library to run correctly.

Description

The Microchip USB host stack receives and logs events via an interrupt handler, but processes them as the USBTasks() (or USBHostTasks()) function is called. This limits the amount of time spent in an interrupt context and helps limit context related issues. This means that in order to keep the USB host stack running, the USBTasks() function needs to be called periodically in order to keep processing these events.

```
int main(void)
{
    //Initialize the USB stack
    USBInitialize(0);

    //Pass my accessory information to the Android client driver
    AndroidAppStart(&myDeviceInfo);

    while(1)
    {
        //Keep the USB stack running
        USBTasks();

        //Do my application specific stuff here
        //...
    }
}
```

The rate at which USBTasks() is called will contribute to determining the throughput that the stack is able to get, the timeliness of the data reception, and the accuracy and latency of the events thrown from the stack.

1.4.2.4 Detecting a Connection/Disconnection

Describes how to detect a a connection/disconnection event.

Description

The USB Host stack notifies users of attachment and detachment events through an event handler call back function. The name of this function is configurable in source code projects. In pre-compiled projects, this function is named `USB_ApplicationEventHandler()`.

The Android client driver uses this same event handler function to notify the user of the attachment or detachment of Android devices. The Android client driver adds the `EVENT_ANDROID_ATTACH` and `EVENT_ANDROID_DETACH` events. These two events are key to interfacing to the attached Android device. The `data` field of the attach event provides the handle to the Android device. This handle must be passed to all of the read/write functions to it is important to save this information when it is received. Similarly the detach event specifies the handle of the device that detached so that the application knows which device detached (if multiple devices are attached).

```
void* device_handle = NULL;
static BOOL device_attached = FALSE;

int main(void)
{
    //Initialize the USB stack
    USBInitialize(0);

    //Send the accessory information to the Android client driver
    AndroidAppStart(&myDeviceInfo);

    while(1)
    {
        //Keep the USB stack running
        USBTasks();

        //If the device isn't attached yet,
        if(device_attached == FALSE)
        {
            //Continue to the top of the while loop to start the check over again.
            continue;
        }

        //If the accessory is ready, then this is where we run all of the demo code

        if(readInProgress == FALSE)
        {
            //This example shows how the handle is required for the transfer functions
            errorCode = AndroidAppRead(device_handle,
                                       (BYTE*)&command_packet,
                                       (DWORD)sizeof(ACCESSORY_APP_PACKET));
            //If the device is attached, then lets wait for a command from the application
            if( errorCode != USB_SUCCESS)
            {
                //Error
                DEBUG_ERROR("Error trying to start read");
            }
            else
            {
                readInProgress = TRUE;
            }
        }
    }
}

BOOL USB_ApplicationEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    switch( event )
    {
        //Android device has been removed.
        case EVENT_ANDROID_DETACH:
            device_attached = FALSE;
            device_handle = NULL;
            return TRUE;
            break;

        //Android device has been added.  Must record the device handle
        case EVENT_ANDROID_ATTACH:
            device_attached = TRUE;
            device_handle = data;
            return TRUE;

        //Handle other events here that are required...
        //...
```

1.4.2.5 Sending Data

Describes how to send data to an Android device.

Description

There are two functions that are associated with sending data from the Accessory to the device: `AndroidAppIsWriteComplete()` and `AndroidAppWrite()`. The AndroidAppWrite() function is used to send data from the Accessory to the Android device. The AndroidAppIsWriteComplete() function is used to determine if a previous transfer has completed. Remember from the Keeping the Stack Running section that the `USBTasks()` function needs to be called in order to keep the stack running. This means that you shouldn't loop on the `AndroidAppIsWriteComplete()` function. Instead use either a state machine or booleans to indicate what you need to do.

```
    while(1)
    {
        USBTasks();

        //Do some extra stuff here to see if the buttons have updated

        if( writeInProgress == TRUE )
        {
            if(AndroidAppIsWriteComplete(device_handle, &errorCode, &size) == TRUE)
            {
                writeInProgress = FALSE;

                if(errorCode != USB_SUCCESS)
                {
                    //Error
                    DEBUG_ERROR("Error trying to complete write");
                }
            }
        }

        if((buttonsNeedUpdate == TRUE) && (writeInProgress == FALSE))
        {
            response_packet.command = COMMAND_UPDATE_PUSHBUTTONS;
            response_packet.data = pushButtonValues;

            errorCode = AndroidAppWrite(device_handle,(BYTE*)&response_packet, 2);
            if( errorCode != USB_SUCCESS )
            {
                DEBUG_ERROR("Error trying to send button update");
            }
            else
            {
                buttonsNeedUpdate = FALSE;
                writeInProgress = TRUE;
            }
        }
    }
```

1.4.2.6 Receiving Data

Describes how to receive data from an Android device.

Description

Receiving data from the Android device is very similar to sending data. There are two functions that are used: `AndroidAppIsReadComplete()` and `AndroidAppRead()`.

The `AndroidAppRead()` function is used to start a read request from the Android device. In the situation where the Android device is the USB peripheral, this will initiate an IN request on the bus. If the Android device doesn't have any information it will respond with NAKs. One key thing to know about the read function is that the buffer passed to the read function must always be able to receive at least one packets worth of USB data. For full-speed USB devices this is 64 bytes.

The `AndroidAppIsReadComplete()` function is used to determine if a read request was completed. The read request will terminate if a couple of conditions occur. The first is if the total number of bytes requested has been read. The second is if a packet with less than the maximum packet length is received. This typically indicates that fewer bytes than requested are available and that no more packets are immediately pending. While this is true for most cases, it may not be true for every case. If the target application is one of those exceptions, keep in mind that you may have to call the read function multiple times in order to receive a complete transmission from the applications perspective. Remember from the Keeping the Stack Running section that the `USBTasks()` function needs to be called in order to keep the stack running. This means that you shouldn't loop on the `AndroidAppIsReadComplete()` function. Instead use either a state machine or booleans to indicate what you need to do.

```
    while(1)
    {
        //Keep the stack running
        USBTasks();

        //Do some extra stuff here

        if(readInProgress == FALSE)
        {
            errorCode = AndroidAppRead(device_handle, (uint8_t*)&read_buffer,
(uint32_t)sizeof(read_buffer))
            //If the device is attached, then lets wait for a command from the application
            if( errorCode != USB_SUCCESS)
            {
                //Error
                DEBUG_ERROR("Error trying to start read");
            }
            else
            {
                readInProgress = TRUE;
            }
        }

        if(AndroidAppIsReadComplete(device_handle, &errorCode, &size) == TRUE)
        {
            readInProgress = FALSE;

            //We've received a command over the USB from the Android device.
            if(errorCode == USB_SUCCESS)
            {
                //We've received data, process it here (or elsewhere if desired)
                switch(read_buffer[0])
                {
                    case COMMAND_SET_LEDS:
                        SetLEDs(read_buffer[1]);
                        break;
                    default:
                        //Error, unknown command
                        DEBUG_ERROR("Error: unknown command received");
                        break;
                }
            }
            else
            {
                //Error
                DEBUG_ERROR("Error trying to complete read request");
            }

        }
    }
```

1.5 Library Interface

This section describes the Application Programming Interface (API) functions of the Microchip's Accessory Framework for Android.

Refer to each section for a detailed description.

1.5.1 API Functions

API functions defined by the library.

Functions

	Name	Description
	AndroidTasks	Tasks function that keeps the Android client driver moving
	AndroidAppStart	Sets the accessory information and initializes the client driver information after the initial power cycles.
	AndroidAppRead	Attempts to read information from the specified Android device
	AndroidAppIsReadComplete	Check to see if the last read to the Android device was completed
	AndroidAppWrite	Sends data to the Android device specified by the passed in handle.
	AndroidAppIsWriteComplete	Check to see if the last write to the Android device was completed
	AndroidAppHIDRegister	Registers a HID report with the Android device
	AndroidAppHIDSendEvent	Sends a HID report to the associated Android device

Module

Library Interface

Description

API functions defined by the library.

1.5.1.1 AndroidTasks Function

Tasks function that keeps the Android client driver moving

File

usb_host_android.h

Syntax

```
void AndroidTasks();
```

Description

Tasks function that keeps the Android client driver moving. Keeps the driver processing requests and handling events. This function should be called periodically (the same frequency as USBHostTasks() would be helpful).

Remarks

This function should be called periodically to keep the Android driver moving.

Preconditions

AndroidAppStart() function has been called before the first calling of this function

Function

void AndroidTasks(void)

1.5.1.2 AndroidAppStart Function

Sets the accessory information and initializes the client driver information after the initial power cycles.

File

usb_host_android.h

Syntax

```
void AndroidAppStart(ANDROID_ACCESSORY_INFORMATION* accessoryInfo);
```

Description

Sets the accessory information and initializes the client driver information after the initial power cycles. Since this resets all device information this function should be used only after a compete system reset. This should not be called while the USB is active or while connected to a device.

Remarks

None

Preconditions

USB module should not be in operation

Function

void AndroidAppStart(ANDROID_ACCESSORY_INFORMATION *info)

1.5.1.3 AndroidAppRead Function

Attempts to read information from the specified Android device

File

usb_host_android.h

Syntax

```
uint8_t AndroidAppRead(void* handle, uint8_t* data, uint32_t size);
```

Description

Attempts to read information from the specified Android device. This function does not block. Data availability is checked via the AndroidAppIsReadComplete() function.

Remarks

None

Preconditions

A read request is not already in progress and an Android device is attached.

Return Values

Return Values	Description
USB_SUCCESS	Read started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlRead to read from a control endpoint.

USB_ENDPOINT_ILLEGAL_DIRECTION	Must read from an IN endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Read is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.
USB_ERROR_BUFFER_TOO_SMALL	The buffer passed to the read function was smaller than the endpoint size being used (buffer must be larger than or equal to the endpoint size).

Function

uint8_t AndroidAppRead(void* handle, uint8_t* data, uint32_t size)

1.5.1.4 AndroidAppIsReadComplete Function

Check to see if the last read to the Android device was completed

File

usb_host_android.h

Syntax

```
bool AndroidAppIsReadComplete(void* handle, uint8_t* errorCode, uint32_t* size);
```

Description

Check to see if the last read to the Android device was completed. If complete, returns the amount of data that was sent and the corresponding error code for the transmission.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful
- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

Transfer has previously been requested from an Android device.

Return Values

Return Values	Description
true	Transfer is complete.
false	Transfer is not complete.

Function

bool AndroidAppIsReadComplete(void* handle, uint8_t* errorCode, uint32_t* size)

1.5.1.5 AndroidAppWrite Function

Sends data to the Android device specified by the passed in handle.

File

usb_host_android.h

Syntax

```
uint8_t AndroidAppWrite(void* handle, uint8_t* data, uint32_t size);
```

Description

Sends data to the Android device specified by the passed in handle.

Remarks

None

Preconditions

Transfer is not already in progress. USB module is initialized and Android device has attached.

Return Values

Return Values	Description
USB_SUCCESS	Write started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlWrite to write to a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must write to an OUT endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Write is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

uint8_t AndroidAppWrite(void* handle, uint8_t* data, uint32_t size)

1.5.1.6 AndroidAppIsWriteComplete Function

Check to see if the last write to the Android device was completed

File

usb_host_android.h

Syntax

```
bool AndroidAppIsWriteComplete(void* handle, uint8_t* errorCode, uint32_t* size);
```

Description

Check to see if the last write to the Android device was completed. If complete, returns the amount of data that was sent and the corresponding error code for the transmission.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful
- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

Transfer has previously been sent to Android device.

Return Values

Return Values	Description
true	Transfer is complete.
false	Transfer is not complete.

Function

bool AndroidAppIsWriteComplete(void* handle, uint8_t* errorCode, uint32_t* size)

1.5.1.7 AndroidAppHIDRegister Function

Registers a HID report with the Android device

File

usb_host_android.h

Syntax

```
bool AndroidAppHIDRegister(uint8_t address, uint8_t id, uint8_t* descriptor, uint8_t
length);
```

Description

Registers a HID report with the Android device

Remarks

None

Preconditions

HID device already attached

Function

bool AndroidAppHIDRegister(uint8_t address, uint8_t id, uint8_t* descriptor, uint8_t length);

1.5.1.8 AndroidAppHIDSendEvent Function

Sends a HID report to the associated Android device

File

usb_host_android.h

Syntax

```
uint8_t AndroidAppHIDSendEvent(uint8_t address, uint8_t id, uint8_t* report, uint8_t
length);
```

Description

Sends a HID report to the associated Android device

Remarks

None

Preconditions

HID device should have already been registers with the AndroidAppHIDRegister() function

Function

uint8_t AndroidAppHIDSendEvent(uint8_t address, uint8_t id, uint8_t* report, uint8_t length);

1.5.2 Error Codes

Error codes defined by the library.

Macros

Name	Description
USB_ERROR_BUFFER_TOO_SMALL	Error code indicating that the buffer passed to the read function was too small. Since the USB host can't control how much data it will receive in a single packet, the user must provide a buffer that is at least the size of the endpoint of the attached device. If a buffer is passed in that is too small, the read will not start and this error is returned to the user.

Module

Library Interface

Description

Error codes defined by the library.

1.5.2.1 USB_ERROR_BUFFER_TOO_SMALL Macro

File

usb_host_android.h

Syntax

```
#define USB_ERROR_BUFFER_TOO_SMALL USB_ERROR_CLASS_DEFINED + 0
```

Description

Error code indicating that the buffer passed to the read function was too small. Since the USB host can't control how much data it will receive in a single packet, the user must provide a buffer that is at least the size of the endpoint of the attached device. If a buffer is passed in that is too small, the read will not start and this error is returned to the user.

1.5.3 Configuration Definitions

Configuration definitions used by the library.

Macros

Name	Description
NUM_ANDROID_DEVICES_SUPPORTED	Defines the number of concurrent Android devices this implementation is allowed to talk to. This definition is only used for implementations where the accessory is the host and the Android device is the slave. This is also most often defined to be 1. If this is not defined by the user, a default of 1 is used. This option is only used when compiling the source version of the library. This value is set to 1 for pre-compiled versions of the library.

Module

Library Interface

Description

Configuration definitions used by the library.

1.5.3.1 NUM_ANDROID_DEVICES_SUPPORTED Macro

File

usb_host_android.h

Syntax

```
#define NUM_ANDROID_DEVICES_SUPPORTED 1
```

Description

Defines the number of concurrent Android devices this implementation is allowed to talk to. This definition is only used for implementations where the accessory is the host and the Android device is the slave. This is also most often defined to be 1. If this is not defined by the user, a default of 1 is used.

This option is only used when compiling the source version of the library. This value is set to 1 for pre-compiled versions of the library.

1.5.4 Configuration Functions

Configuration functions defined by the library.

Functions

	Name	Description
	AndroidAppInitialize	Per instance client driver for Android device. Called by USB host stack from the client driver table.
	AndroidAppEventHandler	Handles events from the host stack
	AndroidAppDataEventHandler	Handles data events from the host stack

Module

Library Interface

Description

Configuration functions defined by the library.

1.5.4.1 AndroidAppInitialize Function

Per instance client driver for Android device. Called by USB host stack from the client driver table.

File

usb_host_android.h

Syntax

```
bool AndroidAppInitialize(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

Per instance client driver for Android device. Called by USB host stack from the client driver table.

Remarks

This is a internal API only. This should not be called by anything other than the USB host stack via the client driver table

Preconditions

None

Return Values

Return Values	Description
true	initialized successfully
false	does not support this device

Function

bool AndroidAppInitialize(uint8_t address, uint32_t flags, uint8_t clientDriverID)

1.5.4.2 AndroidAppEventHandler Function

Handles events from the host stack

File

usb_host_android.h

Syntax

```
bool AndroidAppEventHandler(uint8_t address, USB_EVENT event, void * data, uint32_t size);
```

Description

Handles events from the host stack

Remarks

This is a internal API only. This should not be called by anything other than the USB host stack via the client driver table

Preconditions

None

Return Values

Return Values	Description
true	the event was handled
false	the event was not handled

Function

bool AndroidAppEventHandler(uint8_t address, USB_EVENT event, void *data, uint32_t size)

1.5.4.3 AndroidAppDataEventHandler Function

Handles data events from the host stack

File

usb_host_android.h

Syntax

```
bool AndroidAppDataEventHandler(uint8_t address, USB_EVENT event, void * data, uint32_t
size);
```

Description

Handles data events from the host stack

Remarks

This is a internal API only. This should not be called by anything other than the USB host stack via the client driver table

Preconditions

None

Return Values

Return Values	Description
true	the event was handled
false	the event was not handled

Function

bool AndroidAppDataEventHandler(uint8_t address, USB_EVENT event, void *data, uint32_t size)

1.5.5 Events

Events generated by the library.

Macros

Name	Description
EVENT_ANDROID_ATTACH	This event is thrown when an Android device is attached and successfully entered into accessory mode already. The data portion of this event is the handle that is required to communicate to the device and should be saved so that it can be passed to all of the transfer functions. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.
EVENT_ANDROID_DETACH	This event is thrown when an Android device is removed. The data portion of the event is the handle of the device that has been removed. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.
EVENT_ANDROID_HID_REGISTRATION_COMPLETE	This event is thrown after a HID report is successfully registered. That report is now available for use by the application
EVENT_ANDROID_HID_SEND_EVENT_COMPLETE	The requested report has been sent to the requested device.

Module

Library Interface

Description

Events generated by the library.

1.5.5.1 EVENT_ANDROID_ATTACH Macro

File

usb_host_android.h

Syntax

```
#define EVENT_ANDROID_ATTACH ANDROID_EVENT_BASE + 0
```

Description

This event is thrown when an Android device is attached and successfully entered into accessory mode already. The data portion of this event is the handle that is required to communicate to the device and should be saved so that it can be passed to all of the transfer functions. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.

1.5.5.2 EVENT_ANDROID_DETACH Macro

File

usb_host_android.h

Syntax

```
#define EVENT_ANDROID_DETACH ANDROID_EVENT_BASE + 1
```

Description

This event is thrown when an Android device is removed. The data portion of the event is the handle of the device that has been removed. Always use this definition in the code and never put a static value as the value of this event may change based on various build options.

1.5.5.3 EVENT_ANDROID_HID_REGISTRATION_COMPLETE Macro

File

usb_host_android.h

Syntax

```
#define EVENT_ANDROID_HID_REGISTRATION_COMPLETE ANDROID_EVENT_BASE + 2
```

Description

This event is thrown after a HID report is successfully registered. That report is now available for use by the application

1.5.5.4 EVENT_ANDROID_HID_SEND_EVENT_COMPLETE Macro

File

usb_host_android.h

Syntax

```
#define EVENT_ANDROID_HID_SEND_EVENT_COMPLETE ANDROID_EVENT_BASE + 3
```

Description

The requested report has been sent to the requested device.

1.5.6 Type Definitions

Types defined by the library.

Enumerations

Name	Description
ANDROID_AUDIO_MODE	Defines the available audio modes

Module

Library Interface

Structures

Name	Description
ANDROID_ACCESSORY_INFORMATION	This structure contains the informatin that is required to successfully create a link between the Android device and the accessory. This information must match the information entered in the accessory filter in the Android application in order for the Android application to access the device. An instance of this structure should be passed into the AndroidAppStart() at initialization.

Description

Types defined by the library.

1.5.6.1 ANDROID_ACCESSORY_INFORMATION Structure

File

usb_host_android.h

Syntax

```
typedef struct {
  char* manufacturer;
  uint8_t manufacturer_size;
  char* model;
  uint8_t model_size;
  char* description;
  uint8_t description_size;
  char* version;
  uint8_t version_size;
  char* URI;
  uint8_t URI_size;
  char* serial;
  uint8_t serial_size;
  ANDROID_AUDIO_MODE audio_mode;
} ANDROID_ACCESSORY_INFORMATION;
```

Members

Members	Description
char* manufacturer;	String: manufacturer name
uint8_t manufacturer_size;	length of manufacturer string

char* model;	String: model name
uint8_t model_size;	length of model name string
char* description;	String: description of the accessory
uint8_t description_size;	length of the description string
char* version;	String: version number
uint8_t version_size;	length of the version number string
char* URI;	String: URI for the accessory (most commonly a URL)
uint8_t URI_size;	length of the URI string
char* serial;	String: serial number of the device
uint8_t serial_size;	length of the serial number string

Description

This structure contains the informatin that is required to successfully create a link between the Android device and the accessory. This information must match the information entered in the accessory filter in the Android application in order for the Android application to access the device. An instance of this structure should be passed into the AndroidAppStart() at initialization.

1.5.6.2 ANDROID_AUDIO_MODE Enumeration

File

usb_host_android.h

Syntax

```
typedef enum {
  ANDROID_AUDIO_MODE__NONE = 0,
  ANDROID_AUDIO_MODE__44K_16B_PCM = 1
} ANDROID_AUDIO_MODE;
```

Members

Members	Description
ANDROID_AUDIO_MODE__NONE = 0	No audio support enabled
ANDROID_AUDIO_MODE__44K_16B_PCM = 1	44K 16B PCM audio mode enabled

Description

Defines the available audio modes

1.6 Running the Demos

This section describes the Android demos available with the MLA and how to run them.

Description

This section describes the Android demos available with the MLA and how to run them.

1.6.1 New to Android

This section points users to resources and tools for those that are new to developing on the Android platform.

Description

If you are new to developing under Android, there is extensive information available from the Android developer's website: http://developer.android.com/index.html.

For instructions on where to find the development tools and how to install them, please refer to the following links:

- http://developer.android.com/sdk/index.html
- http://developer.android.com/sdk/installing.html

Once the tools are installed, we recommend that you follow through a few of the example tutorials provided below as well as read through some of the below web pages for more information about Android development before moving forward to your Open Accessory project:

- http://developer.android.com/guide/index.html
- http://developer.android.com/guide/topics/fundamentals.html
- http://developer.android.com/resources/tutorials/hello-world.html

1.6.2 Basic Accessory Demo

This is the basic accessory demo that shows simple bi-directional communication from the Android device to the attached accessory.

Description

This is the basic accessory demo that shows simple bi-directional communication from the Android device to the attached accessory.

1.6.2.1 Getting the Android Application

Describes how to get the Android Application used for this demo.

Description

This section describes how to get the Android Application used for this demo.

1.6.2.1.1 From Source

Describes how to obtain the app from source code.

Description

The source code for the example Android application is included in this installation. You should be able to compile and use the IDE of your choice to directly load the example application as you would any other Android example program. Once the application is loaded on the Android device you can remove the USB connection to the IDE and connect it to target accessory to run the demo.

1.6.2.1.2 From Android Marketplace

Describes how to obtain the app from the Android Marketplace.

Description

The example application can be downloaded from the Android Marketplace. You should be able to find the demo application by searching for "Microchip" and looking for the "Basic Accessory Demo" application.

You can also download the file by:

1) Go the the following link in the browser:

https://market.android.com/details?id=com.microchip.android.BasicAccessoryDemo&feature=search_result

2) Click on the below link from an Android device capable of running the demo:

market://details?id=com.microchip.android.BasicAccessoryDemo

1.6.2.2 Preparing the Hardware

Describes how to prepare the hardware to run the demo.

Description

Before attempting to run the demo application, insure that the correct firmware for the demo application has been loaded into the target firmware.

The firmware for this example can be found in the "Basic Accessory Demo/Firmware" folder of this distribution. Open the correct MPLAB.X project folder and change the configuration in the configuration drop down box to match the hardware platform you are using. Compile and program the firmware into the device.

1.6.2.3 Running the Demo

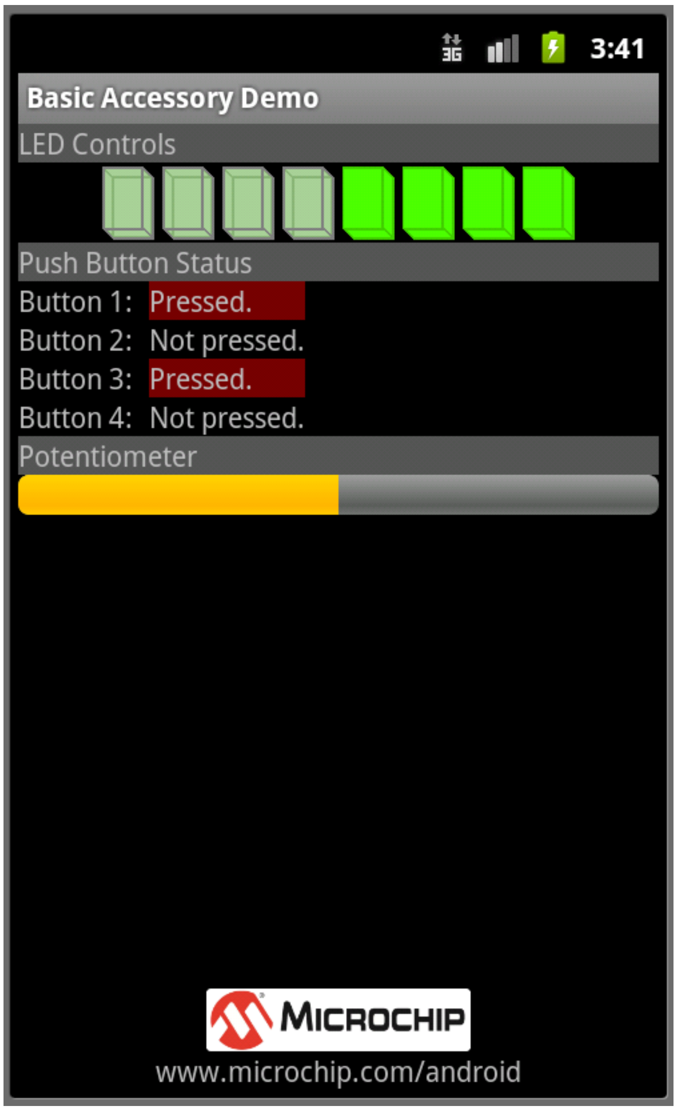
Describes how to run the basic accessory demo.

Description

To run the Basic Accessory Demo:

1. Attach the Accessory Development Starter Board to the Android device using the connector provided by the Android device’s manufacturer. Please make sure that the accessory is attached to the Android device before launching the application.

2. Once the demonstration application has been loaded, go to the Application folder on the Android device. Open the “Basic Accessory Demo” application.



When the application launches, there are three general sections to the application:

• LED Controls – Pressing any of the 8 buttons on the Android screen sends a command from the Android device to the accessory, indicating that the LED status has changed and provides the new LED settings. The image on the screen toggles to show the state of the LEDs. Note that pin multiplexing on your demo boards may prevent all switches or all buttons from functioning, depending on the PIC and board selected.

• Push Button Status – This section indicates the status of the push buttons on the accessory board. When a button is pressed, the text will change to Pressed.

• Potentiometer – This indicates the percentage of the potentiometer on the accessory board.

1.6.3 Basic Accessory Plus Keyboard Demo

This is the Basic Accessory Demo with integrated USB keyboard functionality.

Description

The Basic Accessory Plus Keyboard Demo integrates the functionality of an Android accessory together with the functionality of a USB keyboard. When connected to an Android device, the demo will function in the same was as the Basic Accessory Demo. When connected to a USB keyboard, the demo will display any typed characters on its LCD/LED screen.

1.6.4 Audio with Controls Demo

This demo shows how to create a simple audio device with HID controls for next, previous, play/pause, and volume control.

Description

This demo shows how to create a simple audio device with HID controls for next, previous, play/pause, and volume control.

This demo requires no app associated with it. A user can use any audio source on the target tablet/phone to generate the audio.

1.6.4.1 Running the Demo

Describes how to run the Audio with Controls demo.

Description

1. Attach the Accessory Development Starter Board to the Android device using the connector provided by the Android device’s manufacturer. Please make sure that the accessory is attached to the Android device before launching the application.

2. Once attached, the core audio should now be pumped to the accessory. Please note that the audio data is received by the application code, but is not used as there is not currently a board supported that has the capability to generate audio.

3. The pushbuttons on the board control the next, previous, and play/pause feature. Please press and hold each button for a second to enable the feature. The potentiometer on the board will control the audio.

1.7 Creating an Android Accessory Application

Offers references for creating an Android Accessory Applications.

Description

Offers references for creating an Android Accessory Applications.

1.7.1 Accessing an Accessory From the Application

Describes how to access the accessory from an Android application.

Description

There are several steps that are required in order to gain access to the accessory from the application. These topics are covered in detail at the following link: http://developer.android.com/guide/topics/usb/index.html. This site covers all of the steps required to access the device in either of the modes. Please also refer to the demo applications provided in this distribution.

1.8 FAQs, Tips, and Troubleshooting

1.8.1 How do I debug without access to the ADB?

Though the USB is connected to the accessory now instead of the IDE for debugging, you can still access the ADB interface through a network. Please see http://developer.android.com/guide/topics/usb/index.html for more information about how to set this up.

You might also consider using a USB analyzer to determine what is actually happening on the USB bus.

1.8.2 What if I need design assistance creating my accessory?

If you have questions about the library, our parts, or any of our reference codes/boards, please feel free to contact Microchip for support (What if I need more support than what is here?).

If you need someone to assist you in creating a portion of your design, Microchip has design partners that can assist in the portion of your design that you need help with. You can find a list of design partners at the following address: http://microchip.newanglemedia.com/partner_matrix. At the moment there isn't an option to filter for Android specialists. The best option to filter by right now is USB.

1.8.3 The firmware stops working when I hit a breakpoint.

The USB protocol has periodic packets sent out that keep the attached device active. Without this packet, the bus goes into an idle state. Normally when a breakpoint is hit in the code both the CPU and all peripherals halt at that instruction. This causes the USB module to stop running resulting in the attached peripheral to go into the idle state. The firmware still thinks that the peripheral is active. This results in a break in communication.

There is a way to tell the microcontroller to leave the peripheral enabled when a breakpoint is hit. This will allow the USB module to continue to run and sent out the Start-of-Frame(SOF) packets required to keep the bus alive. This is done via the following methods:

Go under "File->Project Properties". In the project configuration window, select the project configuration that you are using. Under that configuration, select the debugger that is in use. In the resulting debugger menu, select the "Freeze Peripherals" option in the drop down box. Uncheck the "USB" or "U1CNFG" options if you see them. If you don't see these options, uncheck the "All other peripherals" option.

1.8.4 If I hit the "Home" or "Back" buttons while the accessory is attached, the demo no longer works.

If you hit the "Home" or "Back" buttons while the accessory is attached and the demo no longer runs, the code likely tried to close the FileInputStream to release control of the accessory. v2.3.4 and v3.1 of the Android OS have an issue where closing the ParcelFileDescriptor or FileInputStream will not cause an IOException in a read() call on the FileInputStream. This results in the ParcelFileDescriptor being locked until either the accessory detaches or until the read function returns for some other reason. Please see the Release Notes section for other known issues or limitations.

- Workaround: Since the Read() request never completes resulting in locked resources, a workaround can be implemented in the application layer. If the accessory and the application implement a command for the application to indicate to the accessory that the app is closing (or is being paused), then the accessory can respond back with an acknowledge packet. When the app receives this ACK packet, it knows not to start a new read() (since that read() request will not be able to terminated once started).

1.8.5 Why don't all of the features of the demo work?

The demo application on the Android device was written with the assumption that there were 8 LEDs, 4 push buttons, and a potentiometer available on the accessory. Not all of these features are available on the supported hardware platforms. Where these features are not available, these functions do not work.

The Explorer 16 board is an even more complex situation. Even though the base board does include these features, some processor modules don't have all of these features routed to the processor. Also on some processor modules the features are routed to the same pin as other features so both can't be used easily in the same demo.

1.8.6 What if I need more support than what is here?

There are several options that you can use to get various kinds of help.

- You can try our forums at forum.microchip.com. The answers provided here will be by fellow developers and typically not from Microchip employees. The forum often provides a way to get answers very quickly to questions that might take longer for other support routes to answer.
- You can contact your local sales office for support. You can find the local office from www.microchip.com/sales. The local sales team should be able to direct you to a local support team that can help address some issues.
- You can submit support requests to our support system at support.microchip.com or search through the existing hot topics.
- You can also contact androidsupport@microchip.com for support.

Index