



# Microchip's Accessory Framework for Android(tm)



Portions of this page are modifications based on work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions.

Microchip Technology Inc. Copyright (c) 2011. All rights reserved.

Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions.

## Table of Contents

	<b>1</b>
<b>Introduction</b>	<b>2</b>
<b>SW License Agreement</b>	<b>3</b>
<b>Release Notes</b>	<b>7</b>
<b>What's Changed</b>	<b>7</b>
v1.01.01	7
<b>Terms and Definitions</b>	<b>7</b>
<b>Supported Demo Boards</b>	<b>7</b>
<b>Requirements, Limitations, and Potential Issues</b>	<b>8</b>
<b>Using the Library</b>	<b>10</b>
<b>Library Architecture</b>	<b>10</b>
<b>How the Library Works</b>	<b>10</b>
Configuring the Library	10
Required USB callbacks	10
HardwareProfile.h	12
usb_config.h	12
usb_config.c	13
Initialization	14
Keeping the Stack Running	15
Detecting a Connection/Disconnection to an Android Device	15
Sending Data	16
Receiving Data	17
<b>Firmware API</b>	<b>19</b>
<b>API Functions</b>	<b>19</b>
AndroidApplsReadComplete Function	19
AndroidApplsWriteComplete Function	20
AndroidAppRead Function	21
AndroidAppStart Function	22
AndroidAppWrite Function	23
AndroidTasks Function	23

<b>Error Codes</b>	<b>24</b>
USB_ERROR_BUFFER_TOO_SMALL Macro	24
<b>Configuration Definitions</b>	<b>25</b>
NUM_ANDROID_DEVICES_SUPPORTED Macro	25
<b>Configuration Functions</b>	<b>25</b>
AndroidAppDataEventHandler Function	25
AndroidAppEventHandler Function	26
AndroidAppInitialize Function	27
<b>Events</b>	<b>28</b>
EVENT_ANDROID_ATTACH Macro	28
EVENT_ANDROID_DETACH Macro	28
<b>Type Definitions</b>	<b>29</b>
ANDROID_ACCESSORY_INFORMATION Structure	29
 <b>Running the Demos</b>	 <b>31</b>
<b>Creating the Setup</b>	<b>31</b>
New to Microchip	31
Getting the Tools	31
New to Android	31
Updating the Android OS	32
Nexus S	32
Updating the SDK	33
Eclipse IDE	33
Version v2.3.x	33
Version v3.x	34
<b>Basic Accessory Demo</b>	<b>35</b>
Getting the Android Application	35
From source	35
From Android Marketplace	35
Preparing the Hardware	36
Running the demo	36
 <b>Creating an Android Accessory Application using the Open Accessory Framework</b>	 <b>38</b>
<b>Creating the Project</b>	<b>38</b>
<b>Accessing the Accessory From the Application</b>	<b>40</b>
 <b>FAQs, Tips, and Troubleshooting</b>	 <b>41</b>

<b>My PIC32 project gets a run time exception. What could be wrong?</b>	<b>41</b>
<b>How do I debug without access to ADB?</b>	<b>41</b>
<b>What if I need design assistance creating my accessory?</b>	<b>42</b>
<b>The firmware stops working when I hit a breakpoint.</b>	<b>42</b>
<b>If I hit the "Home" or "Back" buttons while the accessory is attached, the demo no longer works.</b>	<b>43</b>
<b>Why don't all of the features of the demo work?</b>	<b>43</b>
<b>What if I need more support than what is here?</b>	<b>43</b>
<b>The Android App Crashes</b>	<b>44</b>
<b>My data doesn't appear on the Android device until I send a second packet.</b>	<b>44</b>

## **Index**

### **a**

# 1

## 2 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.

# 3 SW License Agreement

MICROCHIP IS WILLING TO LICENSE THE ACCOMPANYING SOFTWARE AND DOCUMENTATION TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE FOLLOWING TERMS. TO ACCEPT THE TERMS OF THIS LICENSE, CLICK "I ACCEPT" AND PROCEED WITH THE DOWNLOAD OR INSTALL. IF YOU DO NOT ACCEPT THESE LICENSE TERMS, CLICK "I DO NOT ACCEPT," AND DO NOT DOWNLOAD OR INSTALL THIS SOFTWARE.

## NON-EXCLUSIVE SOFTWARE LICENSE AGREEMENT

This Nonexclusive Software License Agreement ("Agreement") is a contract between you, your heirs, successors and assigns ("Licensee") and Microchip Technology Incorporated, a Delaware corporation, with a principal place of business at 2355 W. Chandler Blvd., Chandler, AZ 85224-6199, and its subsidiary, Microchip Technology (Barbados) II Incorporated (collectively, "Microchip") for the accompanying Microchip software including, but not limited to, Graphics Library Software, IrDA Stack Software, MCHPFSUSB Stack Software, Memory Disk Drive File System Software, mTouch(TM) Capacitive Library Software, Smart Card Library Software, TCP/IP Stack Software, MiWi(TM) DE Software, and/or any PC programs and any updates thereto (collectively, the "Software"), and accompanying documentation, including images and any other graphic resources provided by Microchip ("Documentation").

1. Definitions. As used in this Agreement, the following capitalized terms will have the meanings defined below:

- a. "Microchip Products" means Microchip microcontrollers and Microchip digital signal controllers.
- b. "Licensee Products" means Licensee products that use or incorporate Microchip Products.
- c. "Object Code" means the Software computer programming code that is in binary form (including related documentation, if any), and error corrections, improvements, modifications, and updates.
- d. "Source Code" means the Software computer programming code that may be printed out or displayed in human readable form (including related programmer comments and documentation, if any), and error corrections, improvements, modifications, and updates.
- e. "Third Party" means Licensee's agents, representatives, consultants, clients, customers, or contract manufacturers.
- f. "Third Party Products" means Third Party products that use or incorporate Microchip Products.

2. Software License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to:

- a. use the Software in connection with Licensee Products and/or Third Party Products;
- b. if Source Code is provided, modify the Software, provided that no Open Source Components (defined in Section 5 below) are incorporated into such Software in such a way that would affect Microchip's right to distribute the Software with the limitations set forth herein and provided that Licensee clearly notifies Third Parties regarding such modifications;
- c. distribute the Software to Third Parties for use in Third Party Products, so long as such Third Party agrees to be bound by this Agreement (in writing or by "click to accept") and this Agreement accompanies such distribution;
- d. sublicense to a Third Party to use the Software, so long as such Third Party agrees to be bound by this Agreement (in writing or by "click to accept");
- e. with respect to the TCP/IP Stack Software, Licensee may port the ENC28J60.c, ENC28J60.h, ENC24J600.c, and ENC24J600.h driver source files to a non-Microchip Product used in conjunction with a Microchip ethernet controller;
- f. with respect to the MiWi (TM) DE Software, Licensee may only exercise its rights when the Software is embedded on a Microchip Product and used with a Microchip radio frequency transceiver or UBEC UZ2400 radio frequency transceiver



which are integrated into Licensee Products or Third Party Products.

For purposes of clarity, Licensee may NOT embed the Software on a non-Microchip Product, except as described in this Section.

3. Documentation License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to use the Documentation in support of Licensee's authorized use of the Software

4. Third Party Requirements. Licensee acknowledges that it is Licensee's responsibility to comply with any third party license terms or requirements applicable to the use of such third party software, specifications, systems, or tools. Microchip is not responsible and will not be held responsible in any manner for Licensee's failure to comply with such applicable terms or requirements.

5. Open Source Components. Notwithstanding the license grant in Section 1 above, Licensee further acknowledges that certain components of the Software may be covered by so-called "open source" software licenses ("Open Source Components"). Open Source Components means any software licenses approved as open source licenses by the Open Source Initiative or any substantially similar licenses, including without limitation any license that, as a condition of distribution of the software licensed under such license, requires that the distributor make the software available in source code format. To the extent required by the licenses covering Open Source Components, the terms of such license will apply in lieu of the terms of this Agreement. To the extent the terms of the licenses applicable to Open Source Components prohibit any of the restrictions in this Agreement with respect to such Open Source Components, such restrictions will not apply to such Open Source Component.

6. Licensee Obligations. Licensee will not: (i) engage in unauthorized use, modification, disclosure or distribution of Software or Documentation, or its derivatives; (ii) use all or any portion of the Software, Documentation, or its derivatives except in conjunction with Microchip Products or Third Party Products; or (iii) reverse engineer (by disassembly, decompilation or otherwise) Software or any portion thereof. Licensee may not remove or alter any Microchip copyright or other proprietary rights notice posted in any portion of the Software or Documentation. Licensee will defend, indemnify and hold Microchip and its subsidiaries harmless from and against any and all claims, costs, damages, expenses (including reasonable attorney's fees), liabilities, and losses, including without limitation product liability claims, directly or indirectly arising from or related to the use, modification, disclosure or distribution of the Software, Documentation, or any intellectual property rights related thereto; (ii) the use, sale and distribution of Licensee Products or Third Party Products; and (iii) breach of this Agreement.

7. Confidentiality. Licensee agrees that the Software (including but not limited to the Source Code, Object Code and library files) and its derivatives, Documentation and underlying inventions, algorithms, know-how and ideas relating to the Software and the Documentation are proprietary information belonging to Microchip and its licensors ("Proprietary Information"). Except as expressly and unambiguously allowed herein, Licensee will hold in confidence and not use or disclose any Proprietary Information and will similarly bind its employees and Third Party(ies) in writing. Proprietary Information will not include information that: (i) is in or enters the public domain without breach of this Agreement and through no fault of the receiving party; (ii) the receiving party was legally in possession of prior to receiving it; (iii) the receiving party can demonstrate was developed by the receiving party independently and without use of or reference to the disclosing party's Proprietary Information; or (iv) the receiving party receives from a third party without restriction on disclosure. If Licensee is required to disclose Proprietary Information by law, court order, or government agency, Licensee will give Microchip prompt notice of such requirement in order to allow Microchip to object or limit such disclosure. Licensee agrees that the provisions of this Agreement regarding unauthorized use and nondisclosure of the Software, Documentation and related Proprietary Rights are necessary to protect the legitimate business interests of Microchip and its licensors and that monetary damage alone cannot adequately compensate Microchip or its licensors if such provisions are violated. Licensee, therefore, agrees that if Microchip alleges that Licensee or Third Party has breached or violated such provision then Microchip will have the right to injunctive relief, without the requirement for the posting of a bond, in addition to all other remedies at law or in equity.

8. Ownership of Proprietary Rights. Microchip and its licensors retain all right, title and interest in and to the Software and Documentation including, but not limited to all patent, copyright, trade secret and other intellectual property rights in the Software, Documentation, and underlying technology and all copies and derivative works thereof (by whomever produced). Licensee and Third Party use of such modifications and derivatives is limited to the license rights described in Sections this Agreement.

9. Termination of Agreement. Without prejudice to any other rights, this Agreement terminates immediately, without notice by Microchip, upon a failure by Licensee or Third Party to comply with any provision of this Agreement. Upon termination, Licensee and Third Party will immediately stop using the Software, Documentation, and derivatives thereof, and immediately destroy all such copies.

10. Warranty Disclaimers. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. MICROCHIP AND ITS LICENSORS ASSUME NO RESPONSIBILITY FOR THE ACCURACY, RELIABILITY OR APPLICATION OF THE SOFTWARE OR DOCUMENTATION. MICROCHIP AND ITS LICENSORS DO NOT WARRANT THAT THE SOFTWARE WILL MEET REQUIREMENTS OF LICENSEE OR THIRD PARTY, BE UNINTERRUPTED OR ERROR-FREE. MICROCHIP AND ITS LICENSORS HAVE NO OBLIGATION TO CORRECT ANY DEFECTS IN THE SOFTWARE.

11. Limited Liability. IN NO EVENT WILL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER ANY LEGAL OR EQUITABLE THEORY FOR ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS. The aggregate and cumulative liability of Microchip and its licensors for damages hereunder will in no event exceed \$1000 or the amount Licensee paid Microchip for the Software and Documentation, whichever is greater. Licensee acknowledges that the foregoing limitations are reasonable and an essential part of this Agreement.

12. General. THIS AGREEMENT WILL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF ARIZONA AND THE UNITED STATES WITHOUT REGARD TO CONFLICTS OF LAWS PROVISIONS. Licensee agrees that any disputes arising out of or related to this Agreement, Software or Documentation will be brought in the courts of the State of Arizona. This Agreement will constitute the entire agreement between the parties with respect to the subject matter hereof. It will not be modified except by a written agreement signed by an authorized representative of the Microchip. If any provision of this Agreement will be held by a court of competent jurisdiction to be illegal, invalid or unenforceable, that provision will be limited or eliminated to the minimum extent necessary so that this Agreement will otherwise remain in full force and effect and enforceable. No waiver of any breach of any provision of this Agreement will constitute a waiver of any prior, concurrent or subsequent breach of the same or any other provisions hereof, and no waiver will be effective unless made in writing and signed by an authorized representative of the waiving party. Licensee agrees to comply with all export laws and restrictions and regulations of the Department of Commerce or other United States or foreign agency or authority. The indemnities, obligations of confidentiality, and limitations on liability described herein, and any right of action for breach of this Agreement prior to termination, will survive any termination of this Agreement. Any prohibited assignment will be null and void. Use, duplication or disclosure by the United States Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer-Restricted Rights clause of FAR 52.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clauses in the NASA FAR Supplement. Contractor/manufacturer is Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85224-6199.

If Licensee has any questions about this Agreement, please write to Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85224-6199 USA. ATTN: Marketing.

Copyright (c) 2011 Microchip Technology Inc. All rights reserved.

License Rev. No. 04-091511

## 4 Release Notes

Microchip's Accessory Framework for Android Version 1.1.0, October 2011

### Peripherals

Type/Use	Specific/Configurable	Polled/Interrupt	Limitations (see page 8)
USB module in host mode	USB1 module	Interrupt	None

## 4.1 What's Changed

### 4.1.1 v1.01.01

- Fixed issue with PIC32 configuration bits being set incorrectly resulting in PIC32 demos not functioning correctly.
  - Stack files changed: none (demo main.c file only)
- Fixed issue with Android accessory manager Java class error that cause data to be sent twice for every transmission from the Android device to the accessory.
  - Stack files changed: none (USBAccessoryManager.java file changed for both the v2.3.x and v3.x demos)

## 4.2 Terms and Definitions

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.

## 4.3 Supported Demo Boards

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](#))
  - PIC32MX460F512L PIM ([MA320002](#))
  - PIC32MX795F512L PIM ([MA320003](#))
  - dsPIC33EP512MU810 ([MA330025-1](#))
- PIC24F Starter Kit 1 ([DM240011](#))
- PIC32 USB Starter Kit I or II ([DM320003-2](#))
- PIC32 Ethernet Starter Kit ([DM320004](#))
- dsPIC33E Starter Kit ([DM330012](#))
- PIC24E Starter Kit ([DM240012](#))

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.

## 4.4 Requirements, Limitations, and Potential Issues

This section describes the limitations and requirements for using the Microchip's Accessory Framework for Android.

### **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.

Please see the Creating the Setup (see page 31) section for details about how to get the correct tool versions. Please refer to the Creating the Android Application (see page 38) section for more information about how to select the right tool set when creating a new Android application.

### **Limitations and Potential 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 available in the USB stack release from the Microchip Application

Libraries. Firmware and example application source code to talk to these host capable Android devices can be found at [www.microchip.com/usb](http://www.microchip.com/usb) or [www.microchip.com/mal](http://www.microchip.com/mal).

## 5 Using the Library

### 5.1 Library Architecture

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.

### 5.2 How the Library Works

#### 5.2.1 Configuring the Library

There are two distributions of the Microchip's Accessory Framework for Android. One includes pre-compiled library files and the second includes the source code files.

The pre-compiled version can be found at [www.microchip.com/android](http://www.microchip.com/android).

The source code version can be found at [www.microchip.com/mal](http://www.microchip.com/mal).

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

##### 5.2.1.1 Required USB callbacks

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. These two functions are

USB\_ApplicationDataEventHandler() and USB\_ApplicationEventHandler() for the pre-compiled example. For the source example the function names are configurable through the usb\_config.h (see page 12) file (see the usb\_config.h (see page 12) section for more information). For more detailed information about these functions or the USB library, please refer to [www.microchip.com/mal](http://www.microchip.com/mal). 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 (see page 15) 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;
}

```



## 5.2.1.2 HardwareProfile.h

HardwareProfile.h provides configuration information to the source version of the library. This tells the library and demo code information about the hardware that it needs to know for configuration, such as the system clock speed, which pins are being used for certain stack or demo features, etc.

When moving this library to your own hardware platform, you will need to create your own HardwareProfile.h file that specifies the requirements of your board.

## 5.2.1.3 usb\_config.h

usb\_config.h is used to configure the build options of the source version of this library. At the moment this is also required in the pre-compiled format as well.

**When using with the pre-compiled format, please do not modify this file as it must match exactly how the library was built.**

For users developing with the source version of the library, this file provides several configuration options for customizing the USB stack. There are a few options that are required.

The USB\_SUPPORT\_HOST option must be enabled.

The USB\_ENABLE\_TRANSFER\_EVENT option must be enabled.

The USB\_HOST\_APP\_DATA\_EVENT\_HANDLER must be defined and the function that is referenced must be implemented.

The USB\_ENABLE\_1MS\_EVENT must be enabled.

The AndroidTasks (see page 23)() 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 should be defined.

For use with PIC32, the USB\_PING\_PONG\_MODE option must be set to USB\_PING\_PONG\_\_FULL\_PING\_PONG. This is also recommended for PIC24F but not required.

```
#define USB_PING_PONG_MODE    USB_PING_PONG__FULL_PING_PONG
```

If you modify the TPL in the usb\_config.c (see page 13) file (see section usb\_config.c (see page 13) 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_SUPPORT_HOST

#define USB_PING_PONG_MODE    USB_PING_PONG__FULL_PING_PONG

#define NUM_TPL_ENTRIES 2
#define NUM_CLIENT_DRIVER_ENTRIES 1

#define USB_ENABLE_TRANSFER_EVENT

#define USB_HOST_APP_DATA_EVENT_HANDLER USB_ApplicationDataEventHandler
#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_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.

### 5.2.1.4 usb\_config.c

The usb\_config.c file is only required for those working with the source code implementation of the library. This file is not used in the pre-compiled version of the library.

There are two main sections to the usb\_config.c file. The first is the Targeted Peripheral List (TPL). The TPL is defined 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.

There are two entries that are needed for Android devices. 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 Microchip has implemented the VID/PID combination of 0xFFFF/0xFFFF to indicate that this driver should enumerate every device regardless of its interfaces or its actual VID/PID pair.

The other entry in the table is the entry for the Android device once it has actually entered accessory mode. This can be either entered using the CL/SC/P of the Accessory interface or the magic VID/PID combinations specified by Google Inc. The magic VID/PID combination method is probably the preferred method (seen below):

```

// *****
// USB Embedded Host Targeted Peripheral List (TPL)
// *****
USB_TPL usbTPL[] =
{
    /*[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_VID_PID( 0x18D1u1, 0x2D00u1 ), 0, 0, {0} }, // Android accessory
    { INIT_VID_PID( 0x18D1u1, 0x2D01u1 ), 0, 0, {0} }, // Android accessory
    { INIT_VID_PID( 0xFFFFu1, 0xFFFFu1 ), 0, 0, {0} }, // 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 all three entries point to the client driver entry 0 (as noted by entry [3] set to 0). 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[] =
{
    {
        AndroidAppInitialize,
        AndroidAppEventHandler,
        AndroidAppDataEventHandler,
        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/mal](http://www.microchip.com/mal).

## 5.2.2 Initialization

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 (see page 22)() function. This function takes in a pointer an ANDROID\_ACCESSORY\_INFORMATION (see page 29) structure which contains all of the information about the accessory. An example is seen below:

```
//Define all of my string information here
char manufacturer[] = "Microchip Technology Inc.";
char model[] = "Basic Accessory Demo";
char description[] = "ADK - Accessory Development Starter Kit for Android (PIC24F)";
char version[] = "1.0";
char uri[] = "http://www.microchip.com/android";
char serial[] = "N/A";

//Pack all of the strings and their lengths into an
// ANDROID_ACCESSORY_INFORMATION structure
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.

## 5.2.3 Keeping the Stack Running

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.

## 5.2.4 Detecting a Connection/Disconnection to an Android Device

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 (see page 28) and EVENT\_ANDROID\_DETACH (see page 28) 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...
        //...
    }
}

```

## 5.2.5 Sending Data

There are two functions that are associated with sending data from the Accessory to the device: `AndroidAppIsWriteComplete` (see page 20)() and `AndroidAppWrite` (see page 23)(). The `AndroidAppWrite` (see page 23)() function is used to send data from the Accessory to the Android device. The `AndroidAppIsWriteComplete` (see page 20)() function is used to determine if a previous transfer has completed. Remember from the Keeping the Stack Running (see page 15) 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` (see page 20)() 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;
        }
    }
}

```

## 5.2.6 Receiving Data

Receiving data from the Android device is very similar to sending data. There are two functions that are used: `AndroidAppIsReadComplete` (see page 19)() and `AndroidAppRead` (see page 21)().

The `AndroidAppRead` (see page 21)() 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` (see page 19)() 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 (see page 15) 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` (see page 19)() 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,
                                   (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;
        }
    }

    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(command_packet.command)
            {
                case COMMAND_SET_LEDS:
                    SetLEDs(command_packet.data);
                    break;
                default:
                    //Error, unknown command
                    DEBUG_ERROR("Error: unknown command received");
                    break;
            }
        }
        else
        {
            //Error
            DEBUG_ERROR("Error trying to complete read request");
        }
    }
}
```

# 6 Firmware API

This section covers the API routines available in this distribution. These descriptions cover more of the interface of these functions. For example usages and more details about how to use these functions in conjunction with each other and in your system, please refer to the Using the Library (see page 10) section of the document.

## 6.1 API Functions

### Functions

	Name	Description
⇒	AndroidApplsReadComplete (see page 19)	Check to see if the last read to the Android device was completed
⇒	AndroidApplsWriteComplete (see page 20)	Check to see if the last write to the Android device was completed
⇒	AndroidAppRead (see page 21)	Attempts to read information from the specified Android device
⇒	AndroidAppStart (see page 22)	Sets the accessory information and initializes the client driver information after the initial power cycles.
⇒	AndroidAppWrite (see page 23)	Sends data to the Android device specified by the passed in handle.
⇒	AndroidTasks (see page 23)	Tasks function that keeps the Android client driver moving

### 6.1.1 AndroidApplsReadComplete Function

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

#### File

usb\_host\_android.h

#### C

```

BOOL AndroidAppIsReadComplete(
    void* handle,
    BYTE* errorCode,
    DWORD* 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.

**Parameters**

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 28) event
BYTE* errorCode	a pointer to the location where the resulting error code should be written
DWORD* size	a pointer to the location where the resulting size information should be written

**Return Values**

Return Values	Description
TRUE	Transfer is complete.
FALSE	Transfer is not complete.

**Function**

BOOL AndroidAppIsReadComplete(void\* handle, BYTE\* errorCode, DWORD\* size)

## 6.1.2 AndroidAppIsWriteComplete Function

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

**File**

usb\_host\_android.h

**C**

```
BOOL AndroidAppIsWriteComplete(  
    void* handle,  
    BYTE* errorCode,  
    DWORD* 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.

**Parameters**

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 28) event
BYTE* errorCode	a pointer to the location where the resulting error code should be written
DWORD* size	a pointer to the location where the resulting size information should be written

**Return Values**

Return Values	Description
TRUE	Transfer is complete.
FALSE	Transfer is not complete.

**Function**

BOOL AndroidApplsWriteComplete(void\* handle, BYTE\* errorCode, DWORD\* size)

---

## 6.1.3 AndroidAppRead Function

Attempts to read information from the specified Android device

**File**

usb\_host\_android.h

**C**

```
BYTE AndroidAppRead(  
    void* handle,  
    BYTE* data,  
    DWORD size  
);
```

**Description**

Attempts to read information from the specified Android device. This function does not block. Data availability is checked via the AndroidApplsReadComplete (see page 19)() function.

**Remarks**

None

**Preconditions**

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

**Parameters**

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 28) event
BYTE* data	a pointer to the location of where the data should be stored. This location should be accessible by the USB module
DWORD size	the amount of data to read.

**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 (see page 24)	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**

BYTE AndroidAppRead(void\* handle, BYTE\* data, DWORD size)

## 6.1.4 AndroidAppStart Function

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

**File**

usb\_host\_android.h

**C**

```
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 complete 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

**Parameters**

Parameters	Description
ANDROID_ACCESSORY_INFORMATION* info	the information about the Android accessory

Function

void AndroidAppStart( ANDROID\_ACCESSORY\_INFORMATION (see page 29) \*info)

6.1.5 AndroidAppWrite Function

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

File

usb\_host\_android.h

C

```
BYTE AndroidAppWrite(  
    void* handle,  
    BYTE* data,  
    DWORD 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.

Parameters

Parameters	Description
void* handle	the handle passed to the device in the EVENT_ANDROID_ATTACH (see page 28) event
BYTE* data	the data to send to the Android device
DWORD size	the size of the data that needs to be sent

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

BYTE AndroidAppWrite(void\* handle, BYTE\* data, DWORD size)

6.1.6 AndroidTasks Function

Tasks function that keeps the Android client driver moving

File

usb\_host\_android.h

C

```
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 (see page 22)() function has been called before the first calling of this function

Function

```
void AndroidTasks(void)
```

## 6.2 Error Codes

Macros

	Name	Description
	USB_ERROR_BUFFER_TOO_SMALL (see page 24)	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.

### 6.2.1 USB\_ERROR\_BUFFER\_TOO\_SMALL Macro

File

usb\_host\_android.h

C



```
#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.

## 6.3 Configuration Definitions

### Macros

	Name	Description
	NUM_ANDROID_DEVICES_SUPPORTED (  see page 25)	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.

### 6.3.1 NUM\_ANDROID\_DEVICES\_SUPPORTED Macro

#### File

usb\_host\_android.h

#### C

```
#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.

## 6.4 Configuration Functions

### Functions

	Name	Description
	AndroidAppDataEventHandler (  see page 25)	Handles data events from the host stack
	AndroidAppEventHandler (  see page 26)	Handles events from the host stack
	AndroidAppInitialize (  see page 27)	Per instance client driver for Android device. Called by USB host stack from the client driver table.

### 6.4.1 AndroidAppDataEventHandler Function

Handles data events from the host stack

File

usb\_host\_android.h

C

```
BOOL AndroidAppDataEventHandler(  
    BYTE address,  
    USB_EVENT event,  
    void * data,  
    DWORD 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

Parameters

Parameters	Description
BYTE address	the address of the device that caused the event
USB_EVENT event	the event that occurred
void* data	the data for the event
DWORD size	the size of the data in bytes

Return Values

Return Values	Description
TRUE	the event was handled
FALSE	the event was not handled

Function

BOOL AndroidAppDataEventHandler( BYTE address, USB\_EVENT event, void \*data, DWORD size )

## 6.4.2 AndroidAppEventHandler Function

Handles events from the host stack

File

usb\_host\_android.h

C

```
BOOL AndroidAppEventHandler(  
    BYTE address,  
    USB_EVENT event,  
    void * data,  
    DWORD 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

**Parameters**

Parameters	Description
BYTE address	the address of the device that caused the event
USB_EVENT event	the event that occurred
void* data	the data for the event
DWORD size	the size of the data in bytes

**Return Values**

Return Values	Description
TRUE	the event was handled
FALSE	the event was not handled

**Function**

BOOL AndroidAppEventHandler( BYTE address, USB\_EVENT event, void \*data, DWORD size )

---

## 6.4.3 AndroidAppInitialize Function

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

**File**

usb\_host\_android.h

**C**

```
BOOL AndroidAppInitialize(  
    BYTE address,  
    DWORD flags,  
    BYTE clientDriverID  
) ;
```

**Description**

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

**Remarks**

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

**Preconditions**

None

**Parameters**

Parameters	Description
BYTE address	the address of the device that is being initialized
DWORD flags	the initialization flags for the device
BYTE clientDriverID	the clientDriverID for the device

**Return Values**

Return Values	Description
TRUE	initialized successfully
FALSE	does not support this device







Function

BOOL AndroidApplInitialize( BYTE address, DWORD flags, BYTE clientDriverID )

6.5 Events

Macros

	Name	Description
	EVENT_ANDROID_ATTACH (  see page 28)	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 (  see page 28)	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.

6.5.1 EVENT\_ANDROID\_ATTACH Macro

File

usb\_host\_android.h

C

```
#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.

6.5.2 EVENT\_ANDROID\_DETACH Macro

File

usb\_host\_android.h

C


```
#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.

# 6.6 Type Definitions

## Structures

	Name	Description
	ANDROID_ACCESSORY_INFORMATION ( <a href="#">see page 29</a> )	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 ( <a href="#">see page 22</a> )() at initialization.

## 6.6.1 ANDROID\_ACCESSORY\_INFORMATION Structure

### File

usb\_host\_android.h

### C

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

### Members

Members	Description
char* manufacturer;	String: manufacturer name
BYTE manufacturer_size;	length of manufacturer string
char* model;	String: model name
BYTE model_size;	length of model name string
char* description;	String: description of the accessory
BYTE description_size;	length of the description string
char* version;	String: version number
BYTE version_size;	length of the version number string
char* URI;	String: URI for the accessory (most commonly a URL)
BYTE URI_size;	length of the URI string
char* serial;	String: serial number of the device
BYTE 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` (see page 22) at initialization.

# 7 Running the Demos

## 7.1 Creating the Setup

### 7.1.1 New to Microchip

This section covers where to find Microchip tools and how to set those tools up for those that are new to Microchip.

#### 7.1.1.1 Getting the Tools

If you are new to Microchip, then we welcome you to development on our line of processors. There are a few tools that you will need in order to get started developing.

You will need our IDE, MPLAB. There are two versions of the IDE available. The current released version of the IDE is available for Windows based computers and can be downloaded from [www.microchip.com/mplab](http://www.microchip.com/mplab). For Macintosh or Linux users, you are welcome to use the beta version of MPLAB X, our upcoming IDE version, from [http://ww1.microchip.com/downloads/mplab/X\\_Beta/index.html](http://ww1.microchip.com/downloads/mplab/X_Beta/index.html).

You will need a compiler. Windows users can download the appropriate compilers from [www.microchip.com/c30](http://www.microchip.com/c30) for the 16-bit based processors and [www.microchip.com/c32](http://www.microchip.com/c32) for the 32-based processors. Linux or Macintosh users can download versions of the compilers at [http://ww1.microchip.com/downloads/mplab/X\\_Beta/index.html](http://ww1.microchip.com/downloads/mplab/X_Beta/index.html).

Finally you will need a programmer or debugger. If you order the Accessory Development Starter Kit for Android, you will have received our PICkit 3 programmer/debugger with that board. This is one of our lower end programmer/debuggers. Other programmer/debuggers are available from the following links:

- [www.microchip.com/icd3](http://www.microchip.com/icd3)
- [www.microchip.com/realice](http://www.microchip.com/realice)

### 7.1.2 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>

---

## 7.1.3 Updating the Android OS

### Description

This section describes how to update the Android device to the versions required for use with the Open Accessory framework.

### 7.1.3.1 Nexus S

This section covers how to update the Nexus S to Android OS version 2.3.4.

#### Description

1. Look at the phones current version number by going to "Settings->About Phone->Android Version". If the version number is 2.3, go to step 2. If the version number is 2.3.1, go to step 4, if the version number is 2.3.2, go to step 6, if the version number is 2.3.3, go to step 8, if the version number is 2.3.4, then skip all of these instructions and just go to the firmware/APK file installation instructions.
2. Download the following file to your computer:  
[http://android.clients.google.com/packages/ota/google\\_crespo/a71a2082d553.signed-soju-GRH78-from-GRH55.a71a2082.zip](http://android.clients.google.com/packages/ota/google_crespo/a71a2082d553.signed-soju-GRH78-from-GRH55.a71a2082.zip)
3. Rename the file `update_v2_3_1.zip`
4. Download the following file to your computer:  
[http://android.clients.google.com/packages/ota/google\\_crespo/353e267378cd.signed-soju-GRH78C-from-GRH78.353e2673.zip](http://android.clients.google.com/packages/ota/google_crespo/353e267378cd.signed-soju-GRH78C-from-GRH78.353e2673.zip)
5. Rename the file `update_v2_3_2.zip`
6. Download the following file to your computer:  
[http://android.clients.google.com/packages/ota/google\\_crespo/98f3836cef9e.signed-soju-GRI40-from-GRH78C.98f3836c.zip](http://android.clients.google.com/packages/ota/google_crespo/98f3836cef9e.signed-soju-GRI40-from-GRH78C.98f3836c.zip)
7. Rename the file `update_v2_3_3.zip`
8. Download the following file to your computer:  
[http://android.clients.google.com/packages/ota/google\\_crespo/a14a2dd09749.signed-soju-GRJ22-from-GRI40.a14a2dd0.zip](http://android.clients.google.com/packages/ota/google_crespo/a14a2dd09749.signed-soju-GRJ22-from-GRI40.a14a2dd0.zip)
9. Rename the file `update_v2_3_4.zip`
10. Now connect the Nexus S phone to your computer. Enable the USB Mass Storage drive support on your phone (if it doesn't pop-up with the request, then slide down the notification bar from the top to enable it).
11. Copy all of the .zip files to the root directory of the drive created by the phone.
12. Make sure that the phone has at least ¼ of a charge left. Disconnect the phone from the computer.
13. Turn the phone off by holding the power button and selecting the Power-off option.
14. Hold down the up volume button on the phone.
15. While still holding the volume up button, press and hold the power button. The phone should boot into a boot loader menu. You may now let go of the buttons.
16. Use the volume down button to go to the "Recovery" option of the boot loader. Press the power button. This will make it appear as if the phone is rebooting. It will stop at a screen that has a triangle with a "!" mark inside of it.

17. Press and hold the power button. While still holding the power button, press the up volume button.
18. You will now see a system recovery menu. From this menu use the volume down button to select "apply update from /sdcard". Press the power button to select this option.
19. Use the volume down button to navigate to the lowest version number of files that you loaded onto the phone (so if you are running v2.3.1, you will need to point to the update\_v2\_3\_2.zip file, etc). Press the power button to install that file.
20. Repeat steps 18-19 until you have applied all of the updates for all of the versions successfully.
21. Once complete, select the reboot system option from the menu. Verify that the phone version is now v2.3.4.

---

## 7.1.4 Updating the SDK

### Description

This section discusses how to update to the correct API version to call the Open Accessory framework API.

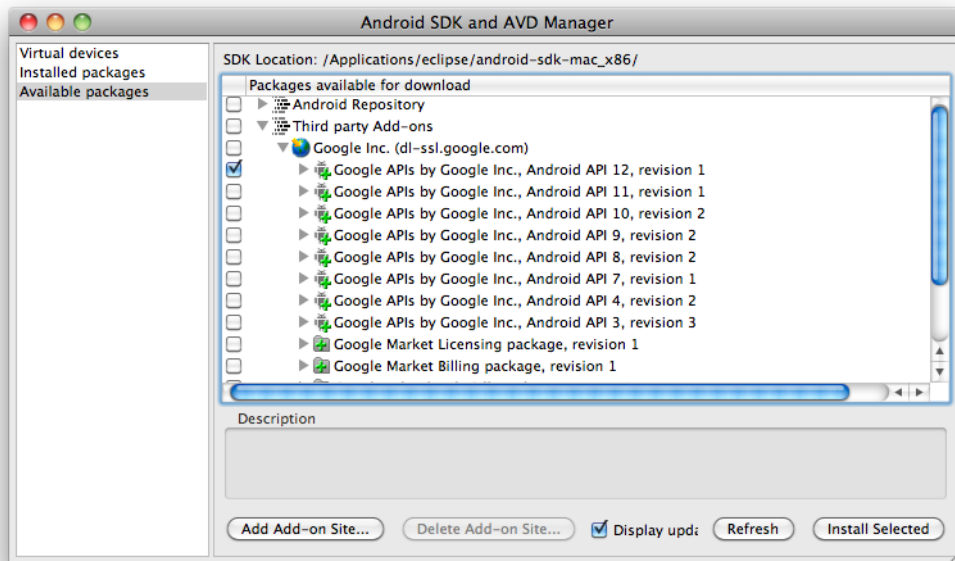
### 7.1.4.1 Eclipse IDE

The Open Accessory API is available in API level 12. There are two different ways to get API level 12 based on which OS version you are developing for.

#### 7.1.4.1.1 Version v2.3.x

To enable development for the Gingerbread OS line, versions v2.3.4 and later, you will need to get the API level 12 add-on for the Eclipse IDE.

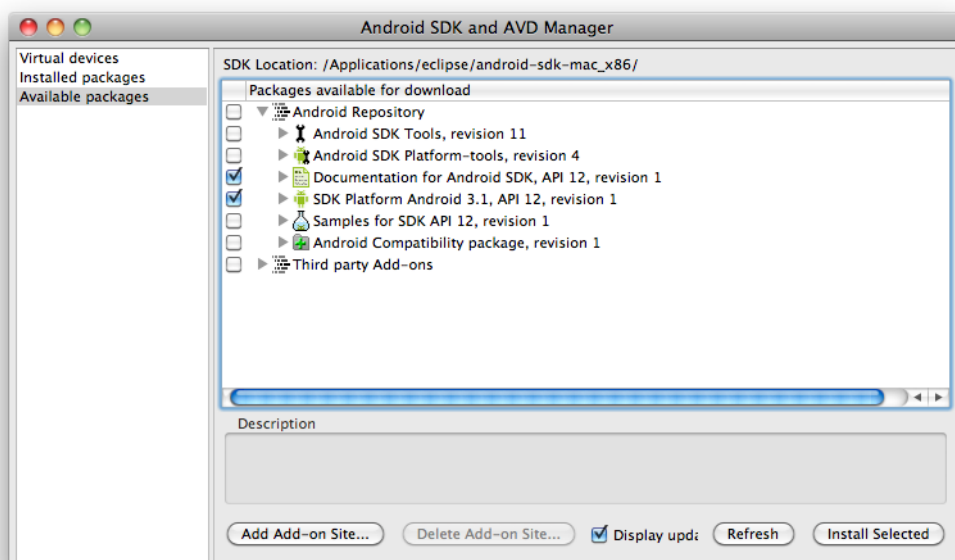
1. Launch the "Android SDK and AVD Manager" either through the Eclipse IDE or through the command line.
2. In the manager window's leftmost panel, select "Available Packages".
3. Expand the "Third Party Add-ons" option
4. Expand the "Google Inc. (dl-ssl.google.com)" option
5. Check the box next to the "Google APIs by Google Inc., Android API 12" option. You may also select other software packages or APIs that you wish to download.
6. Click the "Install Selected" button.



### 7.1.4.1.2 Version v3.x

To enable development for the Honeycomb OS line, versions v3.x and later, you will need to get the API level 12 add-on for the Eclipse IDE.

1. Launch the "Android SDK and AVD Manager" either through the Eclipse IDE or through the command line.
2. In the manager window's leftmost panel, select "Available Packages".
3. Expand the "Android Repository" option
4. Check the box next to the "SDK Platform Android 3.1, API 12" option and the "Documentation for Android SDK, API 12" options. You may also select other software packages or APIs that you wish to download.
5. Click the "Install Selected" button.



---

## 7.2 Basic Accessory Demo

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

---

### 7.2.1 Getting the Android Application

There are several methods for getting the example Android application running on the target Android device. Before attempting any of these methods, please insure that the Android device is running the appropriate version of the Android OS (Updating the Android OS (see page 32)).

Once the proper OS version is installed, please use one of the following methods to get the application into the Android device.

#### 7.2.1.1 From source

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.

#### 7.2.1.2 From Android Marketplace

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 to the following link in the browser:

[https://market.android.com/details?id=com.microchip.android.BasicAccessoryDemo&feature=search\\_result](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>

3) Use a bar code scanner to scan the following QR code:





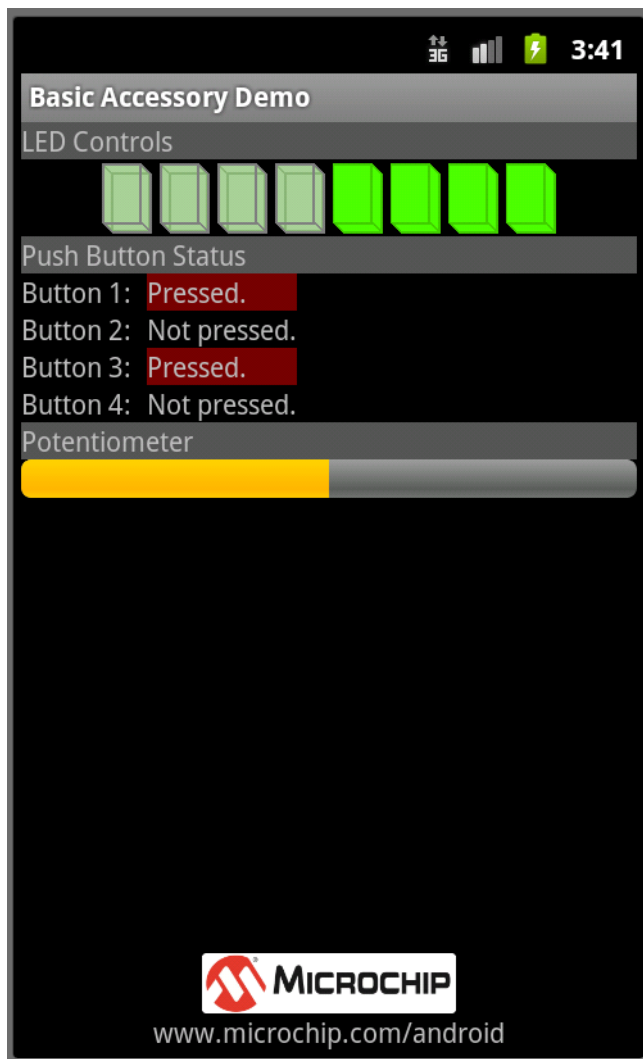
## 7.2.2 Preparing the Hardware

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 project file for your hardware platform for MPLAB 8. If you are using MPLAB X, open the MPLAB.X project folder and change the configuration in the configuration drop down box. Compile and program the firmware into the device.

## 7.2.3 Running the 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.
- 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.

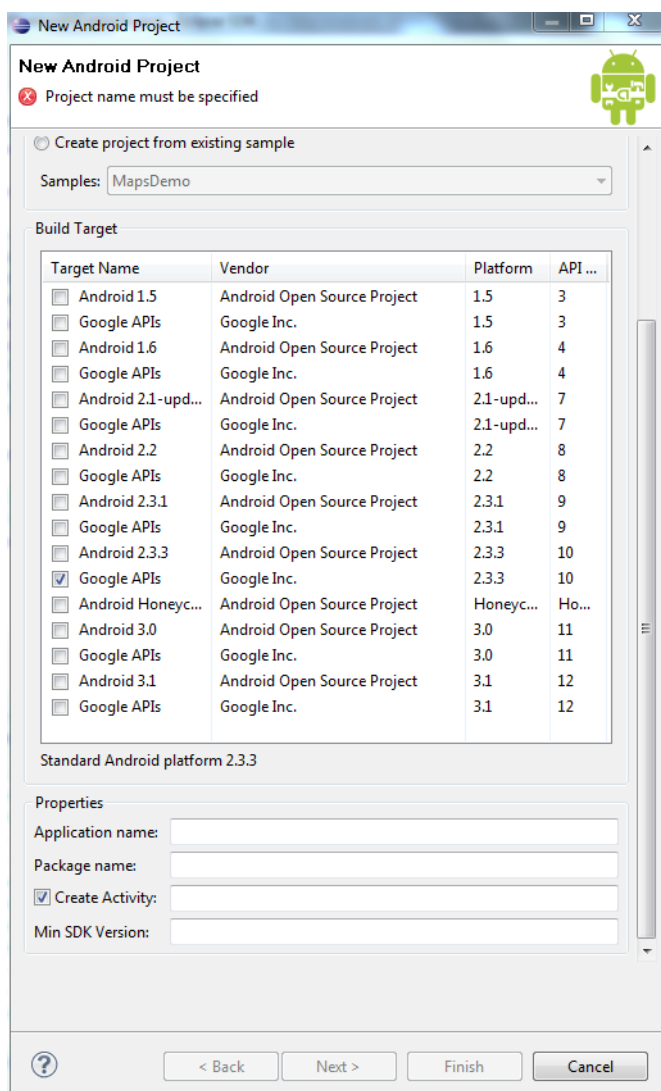
# 8 Creating an Android Accessory Application using the Open Accessory Framework

---

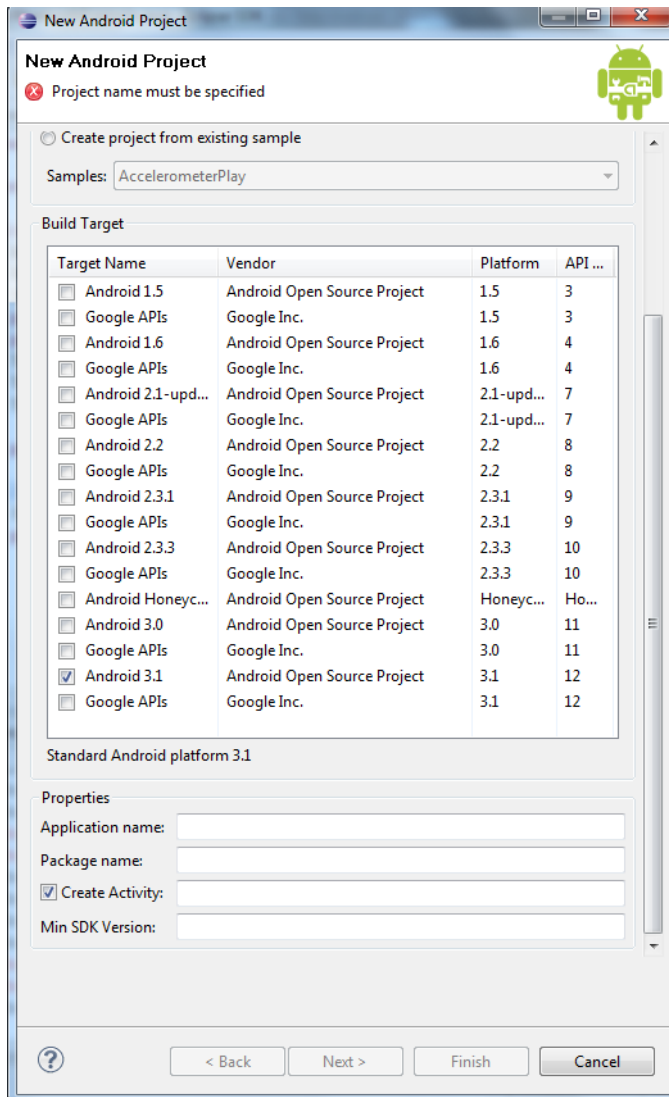
## 8.1 Creating the Project

When creating a new Android application that is going to be using the Open Accessory framework, it is important to select the correct API settings in order to be able to build the project successfully. Please make sure that your SDK is up to date and has the correct components by following the instructions in the Updating the SDK (see page 33). Also insure that the device that you will working with as the correct OS version that enables these features (Updating the Android OS (see page 32) section).

When creating the Android Application project that will use the Open Accessory Framework, you need to make sure select the correct Target OS version. For Gingerbread devices (v2.3.4 or later) you need to select API level 10 with the additional Google APIs (seen below).



For Honeycomb devices (v3.1 or later), select any of the API 12 versions (seen below):



This is the only special requirement for developing an application for Android accessories.

## 8.2 Accessing the Accessory From the Application

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.

## 9 FAQs, Tips, and Troubleshooting

### 9.1 My PIC32 project gets a run time exception. What could be wrong?

There are several issues that could be causing the runtime exceptions in a PIC32 project. Here are some things to check that might help you find the source of the issue.

1. Check to make sure that you have a heap size defined. The USB host stack uses dynamic memory allocation, thus needs a heap size defined. To add a heap to the project, go to the linker settings. This can be found in the "Project->Build Options->Project->MPLAB PIC32 Linker" menu in MPLAB 8. This can be found in the "File->Project Properties->(current build configuration)->C32->pic32-ld" in MPLAB X.

2. If a heap size is defined you can use the general exception vector to trap the error and determine what address caused the exception. This can be done using something similar to the following code:

```
#if defined(__C32__)
void _general_exception_handler(unsigned cause, unsigned status)
{
    unsigned long address = _CP0_GET_EPC();

    DEBUG_ERROR("exception");
    while(1){}
}
#endif
```

This will catch the exception and lock the code in a while(1). From this point you can halt the code and look at the address variable to see what address caused the exception. Use the disassembly listing to determine the corresponding line of C code.

### 9.2 How do I debug without access to 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 can also try following the following instructions for the Nexus S. These instructions will vary based on the device that you are trying to access.

1. Steps to unlock the bootloader:

1. Make sure to have ADB installed and configured correctly. Information about the ADB can be found here: <http://developer.android.com/guide/developing/tools/adb.html>
2. Reboot the device into bootloader mode by opening the command prompt, cd to the platform tools directory (C:\Program Files\Android\android-sdk\platform-tools) and entering the following command: adb reboot bootloader
3. In order to communicate with the phone while it's in bootloader mode, you have to install the proper driver, which is

found at the PDANet website: <http://www.junefabrics.com/android/download.php>

4. After installing the proper driver, cd to the platform tools directory (C:\Program Files\Android\android-sdk\platform-tools). Enter the following command into the command prompt: fastboot oem unlock
5. This should bring up a prompt that asks whether you want to unlock the bootloader. Select yes, and restart your phone. There should now be an unlocked icon at the bottom during bootup. NOTE: This may void your warranty, so continue at your own risk.
2. Steps to root the phone:
  1. Follow the steps listed at this website: <http://www.tech-exclusive.com/root-nexus-s-on-android-2-3-4/>
3. Steps to download the "ADB wireless" app:
  1. Make sure to have WiFi enabled.
  2. Open the marketplace app, and search for "adb"
  3. Download the first app "adbWireless"
  4. After the download finishes, open the app. If the app says that the phone is not rooted, then revisit the previous two steps.
  5. Press the big red button in the app to turn wifi debugging on, and enter the command below the button into the command prompt.

---

## 9.3 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? (see page 43)).

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](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.

---

## 9.4 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:

### MPLAB 8

Under the debugger->settings menu option, select the "Freeze on Halt" tab. Uncheck the "USB" or "U1CNFG" setting in the list. If neither of these items are in the list of peripherals, uncheck the "All other peripherals" option at the bottom of the list.

### MPLAB X

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.

---

## 9.5 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 Requirements, Limitations, and Potential Issues (see page 8) 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).

---

## 9.6 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.

---

## 9.7 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](http://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](http://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](https://support.microchip.com) or search through the existing hot topics.
- You can also contact [androidsupport@microchip.com](mailto:androidsupport@microchip.com) for support.

---

## 9.8 The Android App Crashes

If the Android application crashes or becomes non-responsive in any way, Microchip wants to know about it.

If this does happen, please following steps:

1. run the demo until you are able to replicate the app crash.
2. after the app has crashed, connect the Android device to the computer that you have the IDE/debugger installed on to.
3. navigate to the folder that has the ADB debugger installed (adb.exe for Windows based machines). This will be in your android SDK folder.
4. run the following: "adb bugreport >> report.txt". This will dump information from your Android device into the report.txt file. This will include the installed libraries, apps, and the LogCat.
5. e-mail the report.txt file and a description of what you were doing to cause the crash to [androidsupport@microchip.com](mailto:androidsupport@microchip.com). Please send as much information as you can about the crash and the system as possible including but not limited to the following:
  1. The Android device you are working with
  2. The version of the OS on the Android device you are using (Settings->About Phone/Tablet->Android Version)
  3. The hardware you are using for the accessory
  4. The firmware version of the Android OpenAccessory Framework from Microchip you are using. If you are unable to find the version information, please include the source files in your email.
  5. What you were doing at the time of the crash.

---

## 9.9 My data doesn't appear on the Android device until I send a second packet.

If you are sending data from the accessory to the Android device, and you don't see the data appear, it is likely because you haven't framed the USB data. USB transfers data in packets. Packets are lumped into larger entities called transfers. Transfers are only a logical entity and don't actually have any indicator on the bus. The way that many drivers on USB indicate the end of a transfer is by sending a packet that doesn't the MAX\_PACKET\_SIZE amount of data. This means that it is the end of a transfer. If the transfer ends exactly on a MAX\_PACKET\_SIZE multiple amount of bytes, a zero length packet is sent to indicate that it is the end of the transfer. The Android OS, like many other drivers and OSes, holds the USB data until it receives a complete transfer. Thus if you send a 1024 payload of data from the accessory to the Android device, the Android device may hold that transfer until it receives the next transfer (since it wasn't able to tell that the transfer was complete; 1024 is a multiple of the MAX\_PACKET\_SIZE of 64 bytes). If you wish to make sure that the data arrives immediately and you know that you are sending a payload that is exactly a multiple of 64-bytes, then after the firmware indicates the last transmission is complete, send a zero length packet to for the Android OS to pass the data up to the application.

# Index

## A

Accessing the Accessory From the Application 40  
ANDROID\_ACCESSORY\_INFORMATION structure 29  
AndroidAppDataEventHandler function 25  
AndroidAppEventHandler function 26  
AndroidAppInitialize function 27  
AndroidAppIsReadComplete function 19  
AndroidAppIsWriteComplete function 20  
AndroidAppRead function 21  
AndroidAppStart function 22  
AndroidAppWrite function 23  
AndroidTasks function 23  
API Functions 19

## B

Basic Accessory Demo 35

## C

Configuration Definitions 25  
Configuration Functions 25  
Configuring the Library 10  
Creating an Android Accessory Application using the Open Accessory Framework 38  
Creating the Project 38  
Creating the Setup 31

## D

Detecting a Connection/Disconnection to an Android Device 15

## E

Eclipse IDE 33  
Error Codes 24  
EVENT\_ANDROID\_ATTACH macro 28  
EVENT\_ANDROID\_DETACH macro 28  
Events 28

## F

FAQs, Tips, and Troubleshooting 41

Firmware API 19  
From Android Marketplace 35  
From source 35

## G

Getting the Android Application 35  
Getting the Tools 31

## H

HardwareProfile.h 12  
How do I debug without access to ADB? 41  
How the Library Works 10

## I

If I hit the "Home" or "Back" buttons while the accessory is attached, the demo no longer works. 43  
Initialization 14  
Introduction 2

## K

Keeping the Stack Running 15

## L

Library Architecture 10

## M

My data doesn't appear on the Android device until I send a second packet. 44  
My PIC32 project gets a run time exception. What could be wrong? 41

## N

New to Android 31  
New to Microchip 31  
Nexus S 32  
NUM\_ANDROID\_DEVICES\_SUPPORTED macro 25

## P

Preparing the Hardware 36

## R

- Receiving Data 17
- Release Notes 7
- Required USB callbacks 10
- Requirements, Limitations, and Potential Issues 8
- Running the demo 36
- Running the Demos 31

## S

- Sending Data 16
- Supported Demo Boards 7
- SW License Agreement 3

## T

- Terms and Definitions 7
- The Android App Crashes 44
- The firmware stops working when I hit a breakpoint. 42
- Type Definitions 29

## U

- Updating the Android OS 32
- Updating the SDK 33
- usb\_config.c 13
- usb\_config.h 12
- USB\_ERROR\_BUFFER\_TOO\_SMALL macro 24
- Using the Library 10

## V

- v1.01.01 7
- Version v2.3.x 33
- Version v3.x 34

## W

- What if I need design assistance creating my accessory? 42
- What if I need more support than what is here? 43
- What's Changed 7
- Why don't all of the features of the demo work? 43