SDK6 Application Note: USB

Version 1.0

November 26, 2014

U.S.
3101 Jay Street
Ste.110
Santa Clara, CA 95054, USA
Phone: +1.408.734.8888
Fax: +1.408.734.0788

Taiwan
Suite C1, No. 1, Li-Hsin Road 1
Science-Based Industrial Park
Hsinchu 30078, Taiwan
Phone: +886.3.666.8828
Fax: +886.3.666.1282

Japan - Yokohama
Shin-Yokohama Business Center Bldg.
5th Floor
3-2-6 Shin-Yokohama, Kohoku-ku,
Yokohama, Kanagawa, 222-0033, Japan
Phone: +81.45.548.6150
Fax: +81.45.548.6151

Korea
6 Floor, Hanwon-Bldg.
Sunae-Dong, 6-1, Bundang-Gu
SeongNam-City, Kyunggi-Do
Republic of Korea 463-825
Phone: +031.717.2780
Fax: +031.717.2782

Hong Kong
Unit A&B, 18/F, Spectrum Tower
53 Hung To Road
Kwun Tong, Kowloon
Phone: +85.2.2806.8711
Fax: +85.2.2806.8722

China - Shenzhen
Unit E, 5th Floor
No. 2 Finance Base
8 Ke Fa Road
Shenzhen, 518057, China
Phone: +86.755.3301.0366
Fax: +86.755.3301.0966

China - Shanghai
9th Floor, Park Center
1088 Fangdian Road, Pudong New District
Shanghai 201204, China
Phone: +86.21.6088.0608
Fax: +86.21.6088.0366

I Contents

II Preface

This document provides technical details using conventions to help the user differentiate key concepts at a glance.

II.1 Typographical Conventions

Example	Description
Amage, Chameleon	Software names
Save, File > Save	GUI commands and command sequences
Return, Ctrl+Shift	Computer keys (+ simultaneous)
Power, Reset, Home	Hardware buttons
GPIO81, **CLK_AU**	Hardware external pins
VIL, VIH, VOL, VOH	Hardware pin parameters
amb_performance_t **amb_operating_mode_t** **amb_set_operating_mode()**	API details (e.g., functions, structures, and type definitions)

II.2 Command Prompts

The following is a list of the typographical conventions used in this document. Commands are preceded by prompts that indicate the environment where the command is to be typed.

Indicate the command to be typed in the Linux workstation that is used to compile A9/A12 Flexible Linux SDK.

```
build $
```

Indicate the command to be typed in the Linux machine that is connected to A9/A12 Flexible Linux platform.

```
host $
```

Indicate the command to be typed in the console window connected to the A9/A12 Flexible Linux platform.

```
a5s #
```

1 Introduction

This document describes the procedure to implement USB related functions and the solution for common issues.

1.1 Introduction: Scope of Document

This document includes USB flow, behavior, and issues in RTOS. These issues are non-applicable to the non-RTOS platform (ex. Linux).

2 USB Device

This chapter introduces the procedure on how to implement the USB device function in an application layer.

2.1 USB Device: Initialization

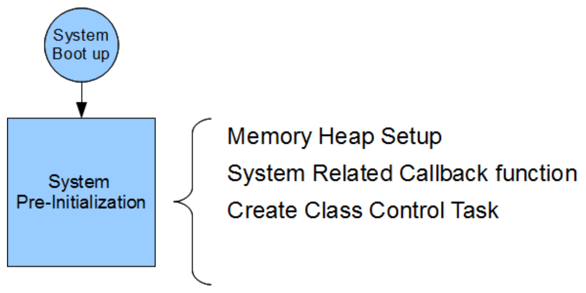
System Boot up
System Pre-Initialization
Memory Heap Setup
System Related Callback function
Create Class Control Task

Figure 2-1 Flow Chart of Initialization.

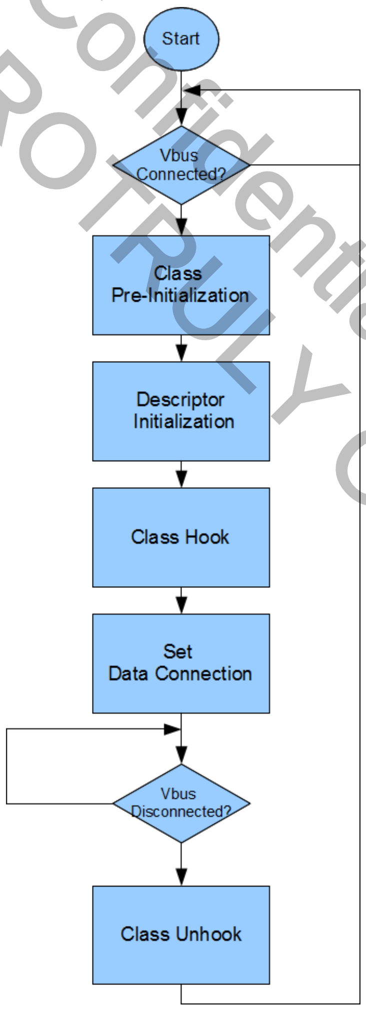

Figure 2-2 Flow Chart of Class Initialization and De-initialization.

2.1.1 .Initialization: System Pre-Initialization

```
AmbaUSB_System_SetMemoryPool(&AmbaBytePool_Cached, &AmbaBytePool_NonCached);

/* Register Vbus callback function */
AmbaUSB_System_RegisterVbusCallback(&AmbaUsbVbusCallBack);

/* Create Class Task and wait vbus present */
RetVal = AmbaKAL_TaskCreate(&AmbaUsbClassCtrlTask, "AmbaUSB_DeviceClassTask", 255,
                                    USB_DeviceClassCtrlTask, 0, AmbaUsbClassStack,
                                    sizeof(AmbaUsbClassStack), AMBA_KAL_AUTO_START);
if (RetVal != OK) {
    AmbaPrint("creat class start task fail 0x%2x\n", RetVal);
}
```

2.1.1.1 System Pre-Initialization: Memory Heap Setup

In this stage, the application provides the cached and non-cached memory pointer for the USB stack by calling "**AmbaUSB_System_SetMemoryPool**".

2.1.1.2 System Pre-Initialization: System Related Callback function

Then, the application hooks the system related callback functions to the kernel stack by calling the following:

"**AmbaUSBD_System_RegisterVbusCallback**" or "**AmbaUSB_System_RegisterVbusCallback**".

The structure of the callback function is given below:

```
typedef struct _USB_DEV_VBUS_CB_s_ {
    void        (*VbusConnectCB)(void);
    void        (*VbusDisconnectCB)(void);
    void        (*SystemStart)(void);
    void        (*SystemRelease)(void);
    void        (*SystemConfigured)(UINT16 index);
    void        (*SystemSuspended)(void);
    void        (*SystemResumed)(void);
    void        (*SystemReset)(void);
} USB_DEV_VBUS_CB_s;
```

VbusConnectCB/**VbusDisconnectCB** is called when the USB Vbus hits connected/disconnected status, which is detected by the task at the USB kernel level.

SystemStart is called after the system is informed that the Vbus is connected. It is suggested to leave this field empty and let the USB driver use the default **SystemStart** function to initialize the USB stack and hardware controller.

SystemRelease is called when the application calls the "**AmbaUSB_System_ClassUnHook**" to unhook the USB class (See Section 2.2.5).

SystemConfigured/ SystemSuspended/ **SystemResumed**/ **SystemReset** are called when the USB host sends Set Configuration/ Suspend/ Resume/ Reset events to the device.

Note that the application should not allocate too many local static resources in these callback functions, otherwise stack overflow of the kernel task could occur.

2.1.1.3 System Pre-Initialization: Create Class Control Task

The application needs to create a task to handle the class hook/unhook according to the Vbus status. This task will be mentioned in detail later.

2.2 USB Device: Class Initialization and De-Initialization

When the Vbus is connected/ disconnected, the application is informed about the status through the hooked callback functions. As mentioned above, the application creates a class control task in the pre-initialization stage, which should handle class initialization matters, such as class pre-initialization, USB descriptor initialization, class hook, set data connection, etc.

```
static void USB_DeviceClassCtrlTask(UINT32 UnUsed)
{
    USB_CLASS_INIT_s ClassConfig = {UDC_CLASS_NONE, 0, 0};
    static UINT32 preVbusConnect = 0;
    VOID *driver_context = NULL;

    while (1) {
        if ((VbusConnect) && (preVbusConnect == 0)) {

            AmbaUsbDeviceClass = AmbaUserSetting.UsbClass;
            ClassConfig.classID = AmbaUSB_ClassCtrl[AmbaUsbDeviceClass].ClassID;

            if (AmbaUSB_ClassCtrl[AmbaUsbDeviceClass].InitFunc) {
                AmbaUSB_ClassCtrl[AmbaUsbDeviceClass].InitFunc();
                ClassConfig.ClassTaskPriority =
                        AmbaUSB_ClassCtrl[AmbaUsbDeviceClass].TaskPriority;
                ClassConfig.ClassTaskStackSize =
                        AmbaUSB_ClassCtrl[AmbaUsbDeviceClass].TaskStackSize;
            }
            AmbaUSB_Custom_SetDevInfo(AmbaUsbDeviceClass);
            AmbaUSB_System_ClassHook(&ClassConfig);

            if(ClassConfig.classID == UDC_CLASS_STREAM) {
                AmbaKAL_TaskSleep(100);
            } else {
```

```
                        AmbaKAL_TaskSleep(AMBA_USB_DATA_CONN_DELAY);
                    }
                    AmbaUSB_System_SetDeviceDataConn(1);
                    preVbusConnect = 1;
                } else if ((VbusConnect == 0) && (preVbusConnect == 1)) {
                    preVbusConnect = 0;
                    /* UnHook USB Class */
                    AmbaUSB_System_ClassUnHook(&ClassConfig);
                }
                AmbaKAL_TaskSleep(100);
        }
}
```

2.2.1 Class Initialization and De-Initialization: Class Pre-Initialization

Before the USB class starts running, the application should prepare class-specified job if any, such as mounting an appointed slot in the mass storage class and loading objects in the MTP class. Make sure all callback functions in each class-specified structure are hooked.

2.2.2 Class Initialization and De-Initialization: Descriptor Initialization

The application should import the USB descriptor in this stage by calling the "**AmbaUSBD_Descriptor_Init**" or "**AmbaUSB_Descriptor_Init**". The USB kernel would use the default descriptor to run the specific class if this stage is bypassed.

2.2.3 Class Initialization and De-Initialization: Class Hook

The application calls the "**AmbaUSBD_System_ClassHook**" or "**AmbaUSB_System_ClassHook**" to hook specified class to USB stack and provides the resource information of the kernel class task.

2.2.4 Class Initialization and De-Initialization: Set Data Connection

Before Vbus is connected, data connection is disabled until everything is ready. The application can decide when to enable the data connection by calling "**AmbaUSBD_System_SetDataConn**" or "**AmbaUSB_System_SetDeviceDataConn**". The delay between the class hook and data connection is suggested as 100ms.

2.2.5 Class Initialization and De-Initialization: Class Unhook

When Vbus is disconnected, the application should call "**AmbaUSB_System_ClassUnHook**" to release the USB class and allocated resource.

3 USB Host

This chapter introduces how to implement the USB host function in the RTOS application layer. If the system has dual OS (RTOS and Linux), the application **MUST NOT** use RTOS USB APIs to control other OS's USB host.

A9 has only one USB PHY which can switch the function between the device controller and the host controller exclusively. In dual OS, if the system starts as an USB device and the owner is RTOS, the application should call Ambalink APIs to switch the owner to the Linux host. The application also needs to call the Ambalink APIs to switch back to the RTOS device. However, it is strongly recommended to soft-reset the whole system to ensure clean release of the USB resource.

A12 has two USB PHY. PHY0 is switchable and PHY1 is fixed to host controller. PHY1 is suggested for Linux only and PHY0 can switch between the RTOS device and host. **Since there is only one host controller, the host cannot run on RTOS and Linux at the same time even though A12 has two USB PHY**.

The block diagram of A12 USB is provided below:

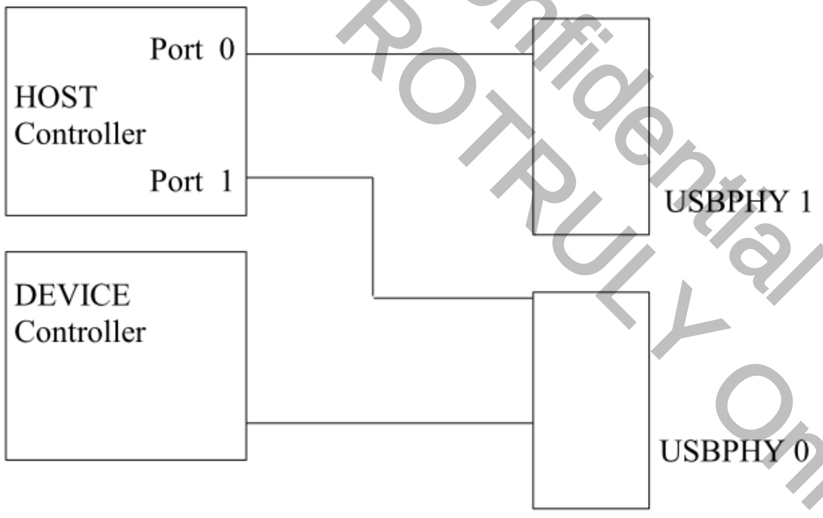

Figure 3 A12 USB.

3.1 USB Host: Initialization

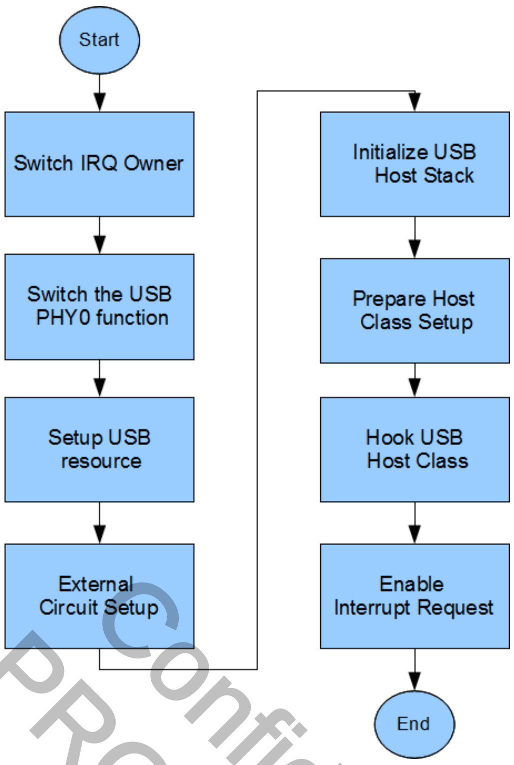

Figure 3-4 Flow Chart of USB Host Initialization.

3.1.1 Initialization: Switch the IRQ Owner

In the case of the dual OS, the IRQ owner can be switched between RTOS and Linux. The application can call "**AmbaUSBH_System_SetUsbOwner**" or "**AmbaUSB_System_Host_SetUsbOwner**" to take the IRQ back from Linux.

```
void AmbaUSB_System_Host_SetUsbOwner(USB_HOST_IRQ_OWNER_e owner, int update)
```

Also, note that parameter "update" is suggested to be set as zero to disable the IRQ at the beginning.

3.1.2 Initialization: Switch the USB PHY0 Function

The application can decide which USB function (host or device) by calling "**AmbaUSBH_System_SetPhy0Owner**" or "**AmbaUSB_Host_Init_SwitchUsbOwner**".

3.1.3 Initialization: Setup USB Resource

The application must provide the resource information for the USB stack. Call “**AmbaUSB_System_SetMemoryPool**” to give the memory pool of cached and non-cached memory. Call “**AmbaUSB_System_HostSystemSetup**” to give the memory information for the USB stack.

3.1.4 Initialization: External Circuit Setup

The application should setup the USB related GPIO hardware function in this stage. For A9, GPIO 9 and GPIO7 should be configured as follows:

AmbaGPIO_ConfigAltFunc(GPIO_PIN_9_USB_EHCI_PWR);

AmbaGPIO_ConfigAltFunc(GPIO_PIN_7_USB_OC);

GPIO_9 is configured as output to indicate EHCI power state. GPIO_7 is configured as input for external circuit to inform host controller over-current is happened.

Besides, the application should inform the USB controller of the trigger state of over-current circuit by calling “**AmbaUSB_Host_System_SetEhciOCPolarity**”. The default value is 1, which means that it is low-triggered.

For example, the USB over-current of A12 EVK is high-triggered, and hence, the application should set: AmbaUSB_Host_System_SetEhciOCPolarity(0);

3.1.5 Initialization: Initialize USB Host Stack

After above stage is complete, the application calls “**AmbaUSB_Host_System_Init**” to initialize the USB host stack and the power on the host controller. The priority and stack size of enumeration and host control dispatcher are decided here.

3.1.6 Initialization: Prepare Host Class Setup

In this stage, information for the specified class is given before the USB host runs. For example, the application can give the slot name of the file system for the storage class:

```
if (UsbHostClass == AMBA_USB_HOST_CLASS_STORAGE) {
    AmbaUSB_Host_Class_Storage_SetSlotInfo(SCM_SLOT_USB);
}
```

3.1.7 Hook USB Host Class

The application implements class initialization here if there is any and decides the priority and stack size of the class-specified thread in the kernel. Finally, the application calls “**AmbaUSB_Host_Class_Hook**” or “**AmbaUSBH_System_ClassHook**” to start the USB host stack.

3.1.8 Initialization: Enable Interrupt Request

The application enables IRQ here by calling “**AmbaUSB_Host_Init_EnableISR**” or “**AmbaUSBH_System_EnableISR**”.

3.2 USB Host: USB Host Termination

Current USB host stack does not support termination function. Please inform USB service team and provide the detailed specification per customer requirements.

3.2.1 USB Host: USB Host Class Overview

Currently, USB host supports only storage class. Please inform the USB service team and provide the detailed specification per customer requirements.

All USB class should support the USB hub since the hub class is installed inside the USB kernel.

3.2.2 USB Host Class Overview: Storage Class

Storage class supports “bulk-only” mass storage class protocol and one logic unit. Media with composite storage device is forbidden (ex. card reader with multiple cards). The maximum transfer data payload is 512K bytes. Performance varies depending on the storage media.

3.2.3 Storage Class: Interaction with File System

Storage class provides four APIs for file system interaction. “**AmbaUSBH_Storage_SetSlotInfo**” or “**AmbaUSB_Host_Class_Storage_SetSlotInfo**” APIs are used for the application to decide the slot that the file system prepares for storage media.

“**AmbaUSBH_Storage_GetStatus**” or “**AmbaUSB_Host_Class_Storage_GetStatus**” APIs are used for the file system to get the current status of the storage media.

“**AmbaUSBH_Storage_Read**” or “**AmbaUSB_Host_Class_Storage_Read**” APIs are used for the file system to get the content from the storage media. The unit is the sector number.

“**AmbaUSBH_Storage_Write**” or “**AmbaUSB_Host_Class_Storage_Write**” APIs are used for the file system to send the content to the storage media. The unit is the sector number.

Except for “**AmbaUSBH_Storage_SetSlotInfo**”, other interactions with the file system could be hooked by the system kernel (middle ware). The application can just ignore them.

3.2.3.1 Storage Class: Test Function

Storage class provides several test functions for application. The easiest way for testing is to check the slot in the console. For example, type “cd i:” to check the content in the storage media. Move the file to/from the storage media.

If the file system does not provide “mv” or “copy” commands, the application can implement the test command by calling the “**AmbaUSBH_Storage_FileCopy**” or “**AmbaUSB_Host_Class_Storage_FileCopy**” APIs:

```
static INT32 USB_Host_FileCopy(UINT32 SourceAddr, UINT32 DestinAddr)
{
    char *pFilePathSource = NULL, *pFilePathDestin = NULL;

    pFilePathSource = (char*)SourceAddr;
    pFilePathDestin = (char*)DestinAddr;
    AmbaUSB_Host_Class_Storage_FileCopy(pFilePathSource, pFilePathDestin);
    return OK;
}
```

Storage class also provides throughput test function “**AmbaUSBH_Storage_Thrughput**” or “**AmbaUSB_Host_Class_Storage_Thrughput**”:

```
static void USB_Host_ThruputTest(void)
{
#define USB_HOST_MAX_ARG_NUM      5
    AMBA_SCM_STATUS_s status;
    int slot = 'i' - 'a';
    UINT32 bs_multi[USB_HOST_MAX_ARG_NUM];
    AmbaSCM_GetSlotStatus(slot, &status);
    if (status.CardPresent == 0) {
        return;
    }
    memset(bs_multi, 0, sizeof(bs_multi));
    AmbaUSB_Host_Class_Storage_Thrughput(slot, bs_multi, 0);
    return;
}
```

Currently, **bs_multi** has no effect and does not need to be modified.

4 Questions and Answers

4.1 Questions and Answers: Host and Device Switch.

4.1.1 Host and Device Switch: In dual OS system, how to switch between RTOS device and Linux host?

Before switching to Linux host, the application should make sure that the Class Unhook is finished (See 2.2.5) and the USB cable is disconnected by detecting the Vbus status. Then, the application uses **AmbaLink API** to get the Linux loading USB host module.

Similarly, the application uses **AmbaLink API** to unload the host module in Linux and calls "**AmbaUSBD_System_SetUsbOwner**" or "**AmbaUSB_System_SetUsbOwner**" to assign the IRQ to RTOS. Then, the application must call "**AmbaUSBH_System_SetPhy0Owner**" or "**AmbaUSB_Host_Init_SwitchUsbOwner**" to set the USB PHY as the device mode. Besides, the application should configure the external circuit correctly; e.g. a Vbus detection circuit.

AmbaLink APIs are not in the scope of this document. Please check related API documents for more information.

4.1.2 Host and Device Switch: In RTOS, how to switch between device and host?

The current SDK does not support run time switch between the device and the host in RTOS.

4.2 Questions and Answers: Host Function

4.2.1 Host Function: Can application disable USB host in run time?

The current SDK does not support the USB Host stack release. However, applications can call "**AmbaUSBH_System_DisableISR**" or "**AmbaUSB_Host_Init_DisableISR**" to disable the host interrupt.

4.2.2 Host Function: Does USB host support EHCI and OHCI?

The current USB host supports only EHCI.

4.3 Questions and Answers: Device Function

4.3.1 Device Function: Does USB device controller support low speed transfer?

The USB device controller does not support low speed transfer.

4.3.2 Device Function: How to use specific GPIO as Vbus pin?

The application can use “**AmbaUSB_System_SetVbusPin**” to change the Vbus pin. Otherwise, the USB driver would use default pin for Vbus detection.

4.3.3 Device Function: How to know if device is configured by USB host?

There are two methods to determine if device is configured by the USB host.

First, use the callback function **SystemConfigured** (see Section 2.1.1.2) when the host sends set configuration events. The application can also get the configuration number given by the USB host.

Second, call “**AmbaUSB_System_IsConfigured**” or “**AmbaUSBD_System_IsConfigured**”. The return value is Boolean.

4.3.4 Device Function: How to know the enumeration speed?

The application can call “**AmbaUSBD_System_GetConnectSpeed**” or “**AmbaUSB_System_GetConnectSpeed**” to get the enumeration speed. Return value of zero means full speed. Otherwise, it means high speed.

4.3.5 Device Function: How to simulate disconnection even if Vbus is connected?

Application can call “**AmbaUSBD_System_SetSoftwareConnec**t” to simulate Vbus disconnection.

```
void AmbaUSBD_System_SetSoftwareConnect(UINT32 enable, UINT32 connect)
```

When “enable” is 1, Vbus detection enters simulation mode. Application can set “connect = 0” to simulate disconnection.

4.3.6 Device Function: What does the error code of an USB API mean?

The application can call “**AmbaUSB_System_GetUxErrorString**” to convert the error code to string type.

4.3.7 Device Function: Does USB support hardware charger detection?

All the latest chips after A9 support hardware charger detection. There are two ways to decide if the current host is a wall charger.

First, call “**AmbaUSBD_System_DataContactDetection**” or “**AmbaUSB_System_DataContactDetection**” to detect the data contact after Vbus is detected.

```
UINT32 AmbaUSB_System_DataContactDetection(UINT32 t1, UINT32 t2)
```

t1 is the time(ms) for detection setup taking effect. t2 is the detection period(ms). Driver would return [data contact = true] if data contact lasts for 10ms in t2 period. For internal test we suggest [t1=10, t2=1000].

Second, call “**AmbaUSBD_System_ChargeDetection**” or “**AmbaUSB_System_ChargeDetection**” to check whether the host is a charger after the Vbus is detected. However, this API still returns 0 when connecting to some wall charger (ex. Apple wall charger).

Application should use method 1 as early as possible after Vbus is connected. Otherwise, host (PC) may consider it as an unrecognized device and stop the data contact.

5 Deprecated APIs

5.1 Deprecated APIs: AmbaUSB_System_SetConnect

Please use “**AmbaUSBD_System_SetSoftwareConnect**” (See Section 4.3.5).

5.2 Deprecated APIs: AmbaUSB_System_SetDeviceDataConnWithVbus

Deprecated. (This API is no longer used)

5.3 Deprecated APIs: AmbaUSB_Descriptor_SetCustomize

Please use “**AmbaUSB_Descriptor_Init**” (See Section 2.2.2).

6 Reference

1. **Ambarella SDK6 API USB V1.0**

Appendix 1 Related Resources

Please contact an Ambarella representative for other documents of potential interest.

Appendix 2 Important Notice

All Ambarella design specifications, datasheets, drawings, files, and other documents (together and separately, “materials”) are provided on an “as is” basis, and Ambarella makes no warranties, expressed, implied, statutory, or otherwise with respect to the materials, and expressly disclaims all implied warranties of noninfringement, merchantability, and fitness for a particular purpose. The information contained herein is believed to be accurate and reliable. However, Ambarella assumes no responsibility for the consequences of use of such information.

Ambarella Incorporated reserves the right to correct, modify, enhance, improve, and otherwise change its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

All products are sold subject to Ambarella’s terms and conditions of sale supplied at the time of order acknowledgment. Ambarella warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with its standard warranty. Testing and other quality control techniques are used to the extent Ambarella deems necessary to support this warranty.

Ambarella assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using Ambarella components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

Ambarella does not warrant or represent that any license, either expressed or implied, is granted under any Ambarella patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which Ambarella products or services are used. Information published by Ambarella regarding third-party products or services does not constitute a license from Ambarella to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from Ambarella under the patents or other intellectual property of Ambarella.

Reproduction of information from Ambarella documents is not permissible without prior approval from Ambarella.

Ambarella products are not authorized for use in safety-critical applications (such as life support) where a failure of the product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Customers acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of Ambarella products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by Ambarella. Further, Customers must fully indemnify Ambarella and its representatives against any damages arising out of the use of Ambarella products in such safety-critical applications.

Ambarella products are neither designed nor intended for use in automotive and military/aerospace applications or environments. Customers acknowledge and agree that any such use of Ambarella products is solely at the Customer’s risk, and they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

Appendix 3 Revision History

Revision	Date	Description
1.0	November 26 2014	Preliminary Release