

VIDs, PIDs, and Firmware: Design Decisions when Using TI USB Device Controllers

Connectivity Solutions

ABSTRACT

There are certain design decisions that must be made by any system designer using TI's USB controller devices, involving the vendor ID (VID) and product ID (PID) of the USB device and the location of firmware. If these are not configured properly, the device will not be enumerated correctly by the USB host.

This document discusses how these are handled within a system using one of TI's USB device controllers. These controllers include TUSB2136, TUSB3210, TUSB3410, TUSB5052, and TUSB6250. The document is intended as an overview of these subjects. Upon reading it, the system designer should possess enough knowledge to make these decisions and know where to obtain the information necessary to implement the chosen scheme.

Contents

1	Introduction	2
2	Firmware Location.....	2
2.1	Storage in an EEPROM	3
2.2	Storage on the PC	4
2.2.1	TUSB3410/5052 UART Driver.....	4
2.2.2	AppLoader Driver	4
2.2.3	Custom Driver with "Integrated" AppLoader.....	5
3	VIDs/PIDs and Serial Numbers: How Windows Associates USB Devices with Drivers.....	5
3.1	Where Are VID/PIDs stored?.....	6
3.2	What Happens When the Device is Inserted onto the Bus?	6
3.3	EEPROM Serialization.....	7
4	USB Descriptors	7
5	EEPROM Selection	8
5.1	I ² C EEPROM Types.....	8
5.2	Type II Device.....	9
5.3	Type III Device.....	9
6	Summary of Configuration Options	10
6.1	Configuration Table.....	10
6.2	System Diagrams.....	15
7	TI Website	17
	References.....	17

Figures

Figure 1.	Writing Three Bytes of Data Starting at Address (i)	9
Figure 2.	Reading Two Bytes of Data Starting at Address (i)	9
Figure 3.	Writing Two Bytes of Data Starting at Address (i).....	10
Figure 4.	Reading One Byte of Data at Address (i).....	10
Figure 5.	Class Drivers, with Firmware Stored in EEPROM.....	15
Figure 6.	Custom Driver without Firmware Downloaded	15
Figure 7.	Custom Driver with Firmware Download Capability.....	16
Figure 8.	Hub-Equipped Device (TUSB2136/TUSB5052) Used with Class Driver.....	16
Figure 9.	Hub-Equipped Device (TUSB2136/TUSB5052) Used with Custom Driver	17

Tables

Table 1.	Recommended Configurations	12
Table 2.	Recommended Configurations (continued).....	13
Table 3.	Recommended Configurations (continued).....	14

1 Introduction

This document is intended to provide a top-level overview of the way in which TI's USB controller devices report themselves to the USB host and equip themselves with application firmware. TI's USB controller devices include TUSB3210, TUSB3410, TUSB2136, TUSB5052, and TUSB6250. This group may also include any other devices under the TUSB3xxx and TUSB6xxx nomenclatures.

By reading this document, the reader should possess enough knowledge to choose a firmware/descriptor scheme and know where to obtain the information necessary to implement that scheme.

Note that the term *USB device* is used throughout this document. This refers to the piece of USB equipment being designed, which includes a TI USB controller as part of its design to handle the USB connection. The term does not refer to the TUSBxxxx device, which is instead referred to as the USB controller.

Also note that a differentiation is often made between *production-ready* systems, and *development* of systems or *evaluation* of the USB controller. This is because USB devices being produced for sale to multiple end users have certain requirements that may not apply to applications contained within the engineer's lab. One key requirement for production systems is that they include an EEPROM device, the reasons for which will be made clear later in this document.

2 Firmware Location

TI's USB controllers, all of which integrate an 8051/8052 microcontroller and use on-board RAM to store code during execution, provide a choice of where to store the microcontroller firmware prior to loading to this RAM: in on-board EEPROM, or on the PC.

Note that no matter where the application firmware is stored, the controller always begins running its ROM-based bootcode upon power-up. After performing some initialization tasks, it looks for a valid header on the EEPROM device. The *header* refers to a properly-structured set of descriptor blocks, as defined by each controller's data sheet. Each descriptor block contains information that defines the USB device. A descriptor block usually contains either USB descriptor information or executable firmware.

If a valid header is found, it begins parsing and processing the descriptor blocks in the header. If it finds that one of these blocks contains firmware, it loads that firmware and transfers execution to it. (In some cases, the bootcode may handle enumeration by the host before loading the firmware from the EEPROM.) If it does not find firmware in the header, the bootcode handles enumeration by the host and then waits for application firmware to be downloaded from the PC.

Generally speaking, there are no strong advantages or disadvantages to placing the firmware in EEPROM or on the PC. However, depending on the application, engineers may have a preference, and for this reason both methods are provided.

2.1 Storage in an EEPROM

The firmware can be stored in an EEPROM located on the USB controller's I²C port. As described in the introduction to this section, this is the first place the bootcode checks when looking for application firmware.

Each USB controller's bootcode handles application firmware in the EEPROM somewhat differently. Generally speaking, there are two methods. The first is to load the code immediately after it is found and begin executing it, prior to enumeration by the USB host. In the second method, the bootcode only sets a flag when application firmware is found, handles enumeration directly, and then loads the firmware from the EEPROM.

TI provides a utility that generates a properly-formatted header file, called the *Header Generator* utility. The *Header Generator* is available on the TI website and uses a simple scripting scheme to instruct the utility in how to form the header.

For details of how a given USB controller handles firmware in the EEPROM, refer to the data manual for that controller and the readme documentation included with the *Header Generator* utility. The readme also helps explain the ways in which each controller's bootcode differ.

Theoretically, code download is somewhat faster via the EEPROM. However, this is not detectable by the end user.

When using TUSB3210, EEPROM storage is the only option. This is because a vendor ID (VID) and product ID (PID) should exist somewhere in the EEPROM for any system in production (see Section 3, *VIDs/PIDs: How Windows Associates USB Devices with Drivers*), and the only method for this supported by TUSB3210's bootcode is to do it programmatically in firmware. This firmware must reside in the EEPROM.

2.2 Storage on the PC

If stored on the PC, the firmware binary file is downloaded to the USB device after the enumeration process. As described previously, the bootcode handles enumeration. If the host driver's INF file is configured properly, the host will associate this device with a driver. The driver can then download the application firmware to the USB controller, to which execution is subsequently transferred.

An advantage to storing firmware on the PC rather than in an EEPROM device is that the device can be significantly smaller if it is only required to store a few USB descriptors. This can result in cost savings. However, it should be noted that an EEPROM should be used in all production-ready applications. This is because the manufacturer's vendor ID and product ID must be resident within the USB device hardware, so that it can report this information to the host at enumeration. This must be stored in the EEPROM device (see Section 4, *USB Descriptions*, for more information).

As discussed in Section 2.1, firmware for TUSB3210 must be stored in EEPROM for production-ready USB devices.

If firmware is to be downloaded from the PC, it is necessary that the driver have this special functionality. The following sections address the three driver situations that can be used for this purpose.

2.2.1 TUSB3410/5052 UART Driver

TI provides a TUSB3410/5052 UART driver/firmware solution for USB/RS232 bridge applications. This driver has the capability of downloading firmware to the USB device.

Note that when firmware is downloaded from the PC using the UART driver with the TUSB5052 device, the TUSB5052's bootcode performs a disconnect/reconnect before transitioning execution to the firmware. This results in the user receiving two audible/visual cues of driver load under some OSs (including Windows XP™), even though the device is being associated with only one driver. This can be avoided by storing the firmware in EEPROM. The TUSB3410 does not execute this disconnect/reconnect in the bootcode.

2.2.2 AppLoader Driver

TI's *AppLoader* (formerly known as "*Firmware Updater*") is a driver intended for evaluation and development purposes (not for distribution with finished products). Its sole purpose is to download firmware to the USB device. It is particularly useful for software development, since it eliminates the need to use an EEPROM programmer every time firmware is changed.

Since it provides no additional functionality, it is necessary for the USB device to be re-associated with a different driver after firmware is downloaded. To do this, the firmware performs a disconnect/reconnect on the bus; upon reconnect, it can report a new vendor ID and product ID (see Section 3, *VIDs/PIDs: How Windows Associates USB Devices with Drivers*), which causes the host to associate the device with a different driver. (Please refer to the "readme" document included with the *AppLoader* driver for instructions on using the driver.)

If the system designer wishes to store firmware on the PC, there are certain restrictions that must be considered. First, TI does not recommend using the two-driver solution (*AppLoader* plus a separate functional driver) for production applications. Issues have been known to arise relating to standby/hibernation. When power is removed from the USB controller (i.e., if it is bus-powered and the PC goes into hibernation), the controller will lose its application firmware. The PC is not aware of this, and a conflict results the next time it tries to access the USB device.

Another disadvantage with the two-driver solution involves the audible and visual cues that some OSs (i.e., Windows XP) provide to the end user when a connection occurs. Because a disconnect/reconnect is performed, these cues occur twice.

The first problem can be solved by integrating *AppLoader*'s firmware-download capability into the functional driver, similar to what TI's UART driver does. An integrated driver can check to be sure the USB device has application firmware following a suspend/resume event and download it if necessary. See Section 2.2.3 for more information.

2.2.3 Custom Driver with “Integrated” *AppLoader*

The system designer can integrate firmware-download functionality into a custom driver. For this purpose, TI makes the source code available for *AppLoader*. This provides a production-capable means of storing the USB controller firmware on the PC. An example of an “integrated” driver is TI's UART driver for use with TUSB3410/TUSB5052.

The engineer should be aware that the end user may still receive two cues for USB device connect. This is because the bootcode in TUSB2136, TUSB3210, and TUSB5052 perform a disconnect before transferring execution to the application firmware (which must then reconnect). The bootcode in TUSB3410 and TUSB6250 does not perform this disconnect/reconnect and therefore will provide only the initial audible/visual cues to the end user when the USB device is enumerated.

Note that this option can be implemented only with custom drivers. USB class drivers, being part of the OS, cannot be altered.

3 VIDs/PIDs and Serial Numbers: How Windows Associates USB Devices with Drivers

Every USB device has two codes that help distinguish it from other USB devices that a host may encounter: the Vendor ID (VID) and Product ID (PID). A VID/PID unique to a particular USB device must be contained within the device hardware to comply with the USB specification.

The VID and PID are each two bytes long. Every equipment vendor must petition the USB Implementers Forum for a unique VID. The PID can be anything the vendor wishes, but it is a good idea to make the PID unique to a particular design.

Furthermore, USB devices of a given VID/PID combination can be serialized. This allows the operating system to track not only a particular model, but also a specific board of that model. If it is important that configuration settings be associated with a particular board, it is strongly recommended that the boards be serialized.

It is extremely helpful for the engineer to understand how the host and the USB device use VIDs/PIDs as they interact with each other.

3.1 Where Are VID/PIDs Stored?

In a system using a TI USB controller, there is usually more than one VID/PID pair to consider.

There is one VID/PID pair common to any such system: the defaults stored in the controller's bootcode. The default VID for any of these controllers is TI's (0x0451). The default PID is usually the last four digits of the controller's device number; for example, 0x3410 for TUSB3410. (Check individual datasheets for the specific values.)

Because the default VID in the bootcode is TI's, any production USB device should include an EEPROM to store a unique VID/PID. This is the second VID/PID present within most systems. They can be stored within a device or hub descriptor specified in the EEPROM header, or they may be set programmatically by firmware in the EEPROM. If stored in an EEPROM device or hub descriptor, the bootcode immediately uses them to replace the controller's defaults, and they will be reported to the USB host during enumeration. If they are located in EEPROM-based firmware, they won't take effect until firmware executes.

A third VID/PID pair is possible if the *AppLoader* driver is used to download firmware to the USB controller, rather than storing it in the EEPROM. In these cases, the firmware must re-write the active values with new ones. This is because of the necessity of causing the host to re-associate the device with a different driver. If the device reports the same VID/PID to the host after disconnect/reconnect, the host will once again associate the device with the *AppLoader*.

If the firmware is downloaded from the PC using an "integrated" driver (a functional driver with firmware-download capability included within), there is no need for the firmware to change (or otherwise contain) the VID/PID. The device is already properly associated with its final driver. This is true of TI's UART firmware downloaded via the TUSB3410/5052 UART driver; it does *not* modify the VID/PID. (However, it should be noted that older versions of the UART firmware (written before 4/15/03) did overwrite the active VID/PID.)

This abundance of options creates room for confusion. See Section 6, *Recommended Configurations*, for TI's recommended configurations.

3.2 What Happens When the Device is Inserted onto the Bus?

As an overview, here are all VID/PID-related activities, in the order they occur after powerup:

1. The USB controller's bootcode assigns the default VID/PID values as the active ones. If no values are later used to overwrite these defaults, they are the ones that will be reported to the USB host.
2. If a valid header is found in an EEPROM on the I²C bus, and if a device descriptor is found, the VID/PID found in the descriptor will be used to overwrite the active values in the controller.
3. Firmware has the ability to replace the active VID/PID values in the controller. If "autoexec" firmware (firmware that runs prior to USB enumeration) is found in the EEPROM on the I²C bus, it immediately loads and executes; and if it overwrites the active VID/PID values, these are the ones that are reported to the USB host. If a VID/PID exists both in a header-based device descriptor and in EEPROM firmware, the one in firmware overwrites the one in the header.

4. The USB device is enumerated by the host, and the active VID/PID values are reported. Using these values, the host associates this USB device with a driver.
5. If the driver associated with the USB device includes the ability to download firmware, it will do this following enumeration. The USB controller then begins executing this code. If the driver is the *AppLoader*, the firmware should overwrite the active PID and cause a disconnect/reconnect in order to prompt the host to re-associate the device with a different driver.

3.3 EEPROM Serialization

If it is important that configuration settings on the host be associated with a specific board, and not just a certain product model, TI highly recommends serializing the EEPROM device. For example, when using TUSB3410 or TUSB5052 with the UART driver/firmware solution, the designer may wish to ensure a particular device is always associated with the same virtual COM port. If the user has two of these devices, and neither is serialized, the operating system will not be able to dependably identify the boards. In Windows, a phenomenon called “COM port hopping” may occur, in which a new virtual COM port is assigned every time the device is attached.

The solution is to include a string descriptor in the EEPROM designated as the serial number. The device descriptor must include a valid index to this descriptor, and the serial number string descriptor in each EEPROM device should contain a unique value. Most EEPROM programmers are able to assign incremental values to particular locations within their mask files, and this mechanism can be used to serialize the EEPROMs.

If serialization is not used, the operating system recognizes this by the lack of a valid index to a string descriptor within the device descriptor. At this point, it attempts its own method of providing unique identification. Windows 9x operating systems assign unique values based on its location within the USB tree, and therefore a board is subject to re-assignment anytime the tree changes. Windows 2k/XP operating systems assign values based on incidental data, such as the USB port into which the device is plugged. This means that a board moved from one port to another port can be identified as a different device. Both methods provide some measure of unique identification, but neither is correct on a consistent basis. The only way to consistently provide unique identification is EEPROM serialization.

String descriptors can be implemented in the EEPROM header of TUSB3410 or TUSB6250, or can be implemented programmatically in the firmware of the other devices. In the latter case, this of course requires that firmware be kept in EEPROM. Example configuration files for TUSB3410 are included with the Header Generator utility that demonstrate creating a serial number string descriptor.

4 USB Descriptors

There are many USB descriptors that can be included in the EEPROM, as defined by the USB specification, Sections 9.5 and 9.6. A few of them can be stored separately in the header, depending on the USB controller being used; the rest can be handled programmatically in firmware. (The *Header Generator* readme indicates which types are available for a given controller device.) At a minimum, there are two descriptor types that should be included in the EEPROM for any production USB device.

The first is the device descriptor. This provides basic information about the USB device to the host. Most importantly, it includes the VID and PID. This is essential because it differentiates the device from any other USB device the host may encounter.

The second is the string descriptor. Strings encapsulated by the string descriptors are displayed to the end user the first time the device is enumerated by Windows. Without these descriptors, the default strings will be reported to the host when the device enumerates. As a result, Windows presents the USB device to the end user as something TI-specific; for example, “TI TUSB3410 Boot Device”. Unless this is deemed acceptable, it is necessary to include a unique string descriptor in the EEPROM. (Note that after the first enumeration, Windows pulls the display strings from the driver associated with this device.) A string descriptor is also used in EEPROM serialization, as discussed in section 3.3.

For TUSB2136 and TUSB5052, there is a descriptor block type called HUB_INFO_BASIC. This is not a USB descriptor per se, but rather consolidates pertinent information for several USB descriptors, including VID and PID.

5 EEPROM Selection

There are certain restrictions to the types of EEPROM devices that can be used. I²C EEPROM devices available on the market can be grouped by the way they are addressed on the I²C bus. It is necessary to consider this when selecting a device for use with a TI USB controller, since some types are not supported with some controllers, and in some cases the device address may need to be configured differently depending on the type.

5.1 I²C EEPROM Types

I²C EEPROM devices can be divided into three categories, based on data memory size. Type I devices have small memory, typically in the 16- to 128-byte range. Unlike Type II and Type III devices, it does not have a device address. Therefore, only one master and one slave can be on the bus. The USB controllers' bootcode does not support this type of device.

A Type II device has a memory size equal or larger than that of a Type I device. This type of device normally has three address pins. Therefore, up to 8 devices can be on the same I²C bus. Some devices with larger memory size might only have one or two address pins. Therefore, the rest of the address bits in the device address byte become part of the data address. Normally, a Type II device has a one-byte data address in the protocol. There could be from 8 to 11 addressing bits, depending on the vendor's implementation. Usually, a Type II device has no more than 2K bytes of data memory.

A Type III device has a larger memory size. This type of device normally also has three address pins. Therefore, up to 8 devices can be on the same I²C bus. Some devices with larger memory sizes might only have one or two address pins. Therefore, the rest of the address bits in the device address byte become part of the data address. Normally, a Type III device has a two-byte data address in the protocol. There could be from 16 to 19 addressing bits, depend on vendor's implementation.

The USB controllers' bootcode supports Type II and Type III devices.

5.2 Type II Device

Fig. 1 shows the SDA line during access of a Type II device. Three bytes of data are written to addresses i , $i+1$, and $i+2$, respectively. P0, P1, and P2 represent device address pins A2, A1, and A0 on the chip. A value of 1010b in the high nibble of the device address byte is used for the memory device. This is defined in the I²C specification. One byte of data address, A7 to A0, is transmitted before the data bytes.

Figure 2 shows an example of a sequential read.

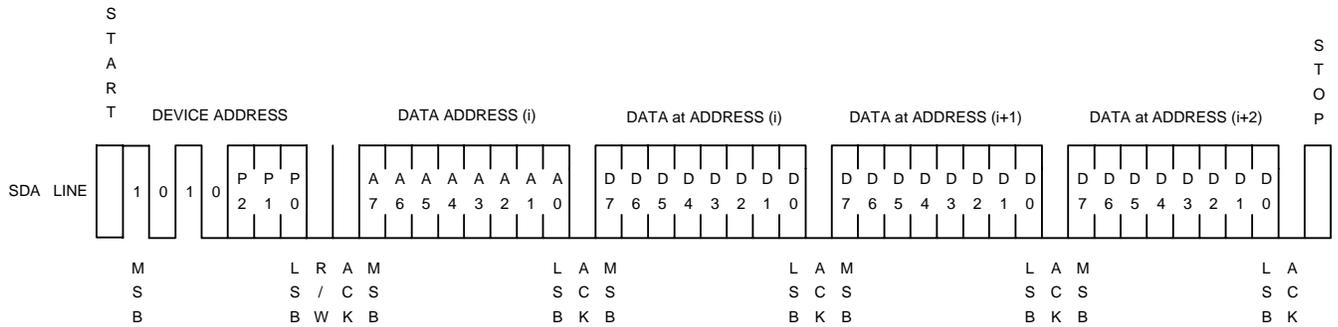


Figure 1. Writing Three Bytes of Data Starting at Address (i)

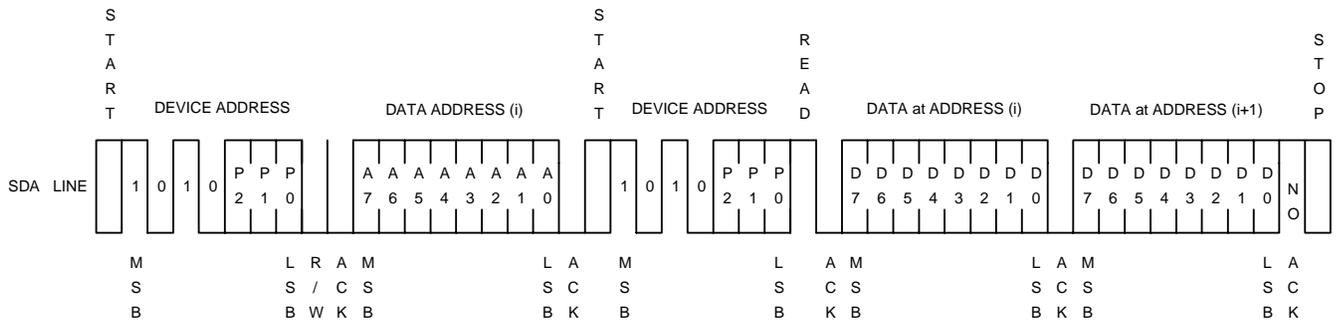


Figure 2. Reading Two Bytes of Data Starting at Address (i)

5.3 Type III Device

Fig. 3 shows the SDA line during access of a Type III device. Two bytes of data are written to addresses i and $i+1$. Two bytes of data address, A15 to A0, are sent out before the data bytes.

Figure 4 shows an example of a sequential read.

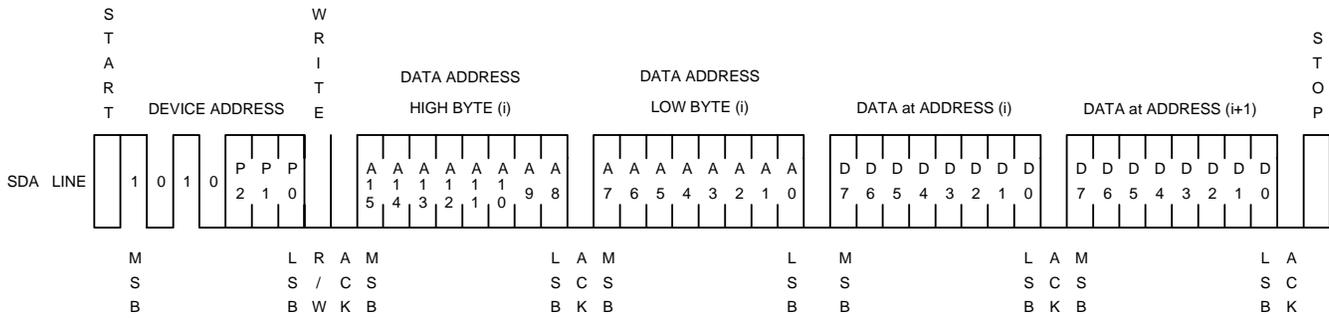


Figure 3. Writing Two Bytes of Data Starting at Address (i)

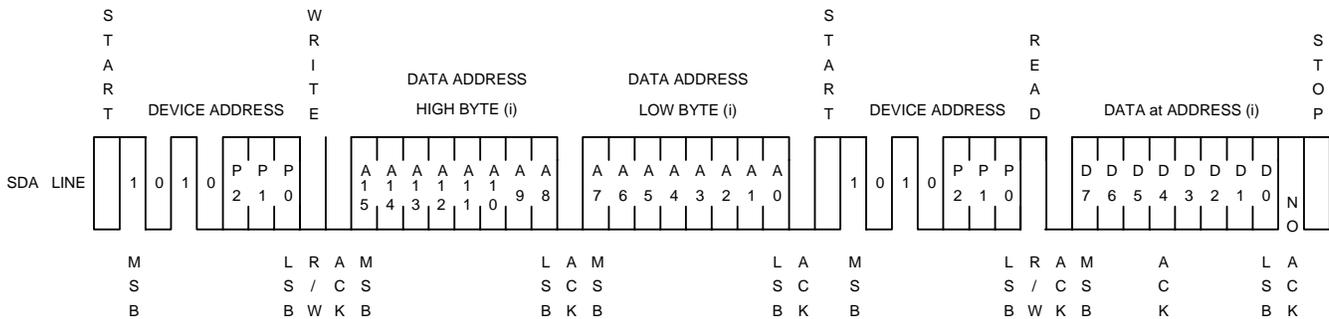


Figure 4. Reading One Byte of Data at Address (i)

6 Summary of Configuration Options

Because there are numerous possible combinations of VID/PID locations, descriptors, and firmware locations, TI recommends certain configurations. This section summarizes the information given previously in this document.

6.1 Configuration Table

Table 1 matches TI’s available USB controllers with five different combinations of driver classes and firmware storage options. “UART driver” refers to the TUSB3410/5052 UART driver for USB/UART bridge applications. “Class driver” refers to any class driver type defined by the USB specification and included with the OS. Note that the table addresses production-ready implementations only, and therefore AppLoader is not listed as a potential driver configuration.

In each case, a system diagram is identified that reflects this configuration. These diagrams are displayed in Section 6.2, *System Diagrams*.

Also in each case, a template configuration file for the *Header Generator* utility is recommended. Template files for many types of configurations are provided with the utility. The readme file included with the utility provides additional instruction in usage of these templates.

In all cases, ensure that the VID/PID in the driver's INF match the active VID/PID transmitted from the USB device. (Refer to Section 3, *VIDs/PIDs: How Windows Associates USB Devices with Drivers*, to determine which is the active VID/PID for any particular configuration.) Also ensure that, if firmware is downloaded from the PC, the binary file referenced in the INF is probably located and named.

Table 1. Recommended Configurations

	UART Driver – firmware in EEPROM	UART Driver – firmware on PC	Class Driver – firmware in EEPROM	Custom Driver – firmware in EEPROM	Custom Driver – firmware on PC
TUSB2136	Not applicable	Not applicable	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set hub, device, and string descriptors programmatically in firmware</p> <p>Device Desc. Contents: Unique VID/PID; class info in bDeviceClass, bDeviceSubclass, bDeviceProtocol</p> <p>Hub descriptor causes Windows to</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set programmatically in firmware (hub, device, string)</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>Hub descriptor causes Windows to load hub class driver. VID/PID in device descriptor causes Windows to associa</p>	Not recommended, because it isn't possible to set string descriptors in EEPROM header of TUSB2136. Therefore, without setting string descriptors programmatically in firmware-based EEPROM, Windows will report the device to the use with a TI-specific strin
			<p>Diagram: Fig. 8 Header Gen: #1</p>	<p>Diagram: Fig. 9 Header Gen: #1</p>	
TUSB3210	Not applicable	Not applicable	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors programmatically in firmware</p> <p>Device Desc. Contents: Unique VID/PID; class info in bDeviceClass, bDeviceSubclass, bDeviceProtocol</p> <p>Class info in device descriptor causes W</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors programmatically in firmware (not possible to store in header)</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>VID/PID in device descriptor causes Windows to associate device wit</p>	Not applicable
			<p>Diagram: Fig. 5 Header Gen: #2</p>	<p>Diagram: Fig. 6 Header Gen: #2</p>	

Table 2. Recommended Configurations (continued)

	UART Driver -- firmware in EEPROM	UART Driver -- firmware on PC	Class Driver -- firmware in EEPROM	Custom Driver -- firmware in EEPROM	Custom Driver -- firmware on PC
TUSB3410	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>VID/PID in device descriptor causes Windows to associate device with UART driver</p>	<p>Firmware Location: PC</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>VID/PID in device descriptor causes Windows to associate device with UART driver. Driver downloads firmware to device</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID; class info in bDeviceClass, bDeviceSubclass, bDeviceProtocol</p> <p>Class info in device descriptor causes Windows to as</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>VID/PID in device descriptor causes Windows to associate device with custom driver.</p>	<p>Firmware Location: PC</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>Must add firmware-download functionality to custom driver. VID/PID in device descriptor causes Windows to associate d</p>
	<p>Diagram: Fig. 6 Header Gen: #3</p>	<p>Diagram: Fig. 7 Header Gen: #4</p>	<p>Diagram: Fig. 5 Header Gen: #3 or #5</p>	<p>Diagram: Fig. 6 Header Gen: #3 or #5</p>	<p>Diagram: Fig. 7 Header Gen: #4</p>
TUSB5052	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set device and string descriptors in EEPROM header</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>Hub descriptor causes Windows to load hub class driver. VID/PID in device descriptor causes Windows to associate</p>	<p>Not recommended, because the TUSB5052 bootcode always performs a disconnect/reconnect when handing control to application firmware. This can cause double-notification to the user.</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set hub, device, and string descriptors programmatically in firmware</p> <p>Device Desc. Contents: Unique VID/PID; class info in bDeviceClass, bDeviceSubclass, bDeviceProtocol</p> <p>Hub descriptor causes Windows to l</p>	<p>Firmware Location: EEPROM</p> <p>EEPROM Descriptors: Set hub, device, and string descriptors programmatically in firmware</p> <p>Device Desc. Contents: Unique VID/PID</p> <p>Hub descriptor causes Windows to load hub class driver. VID/PID in device descriptor causes Win</p>	<p>Not recommended, because it isn't possible to set string descriptors in EEPROM header of TUSB5052. Therefore, without setting string descriptors programmatically in firmware-based EEPROM, Windows will report the device to the use with a TI-specific strin</p>
	<p>Diagram: Fig. 9 Header Gen: #6</p>		<p>Diagram: Fig. 8 Header Gen: #6</p>	<p>Diagram: Fig. 9 Header Gen: #6</p>	

Table 3. Recommended Configurations (cont.)

	UART Driver -- firmware in EEPROM	UART Driver -- firmware on PC	Class Driver -- firmware in EEPROM	Custom Driver -- firmware in EEPROM	Custom Driver -- firmware on PC
TUSB6250	Not applicable	Not applicable	Firmware Location: EEPROM EEPROM Descriptors: Set device and string descriptors in EEPROM header Device Desc. Contents: Unique VID/PID; class info in bDeviceClass, bDeviceSubclass, bDeviceProtocol Class info in device descriptor causes Windows to as	Not necessary	Not necessary
			Diagram: Fig. 5		

6.2 System Diagrams

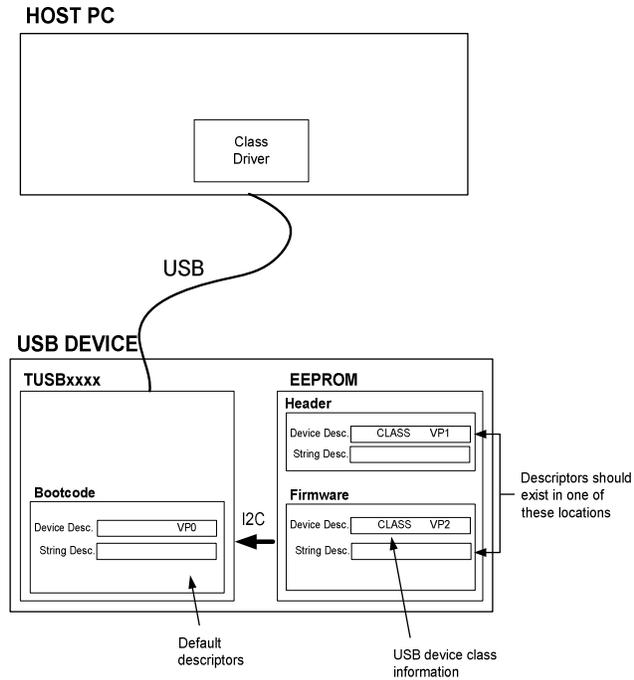


Figure 5. Class Drivers, with Firmware Stored in EEPROM

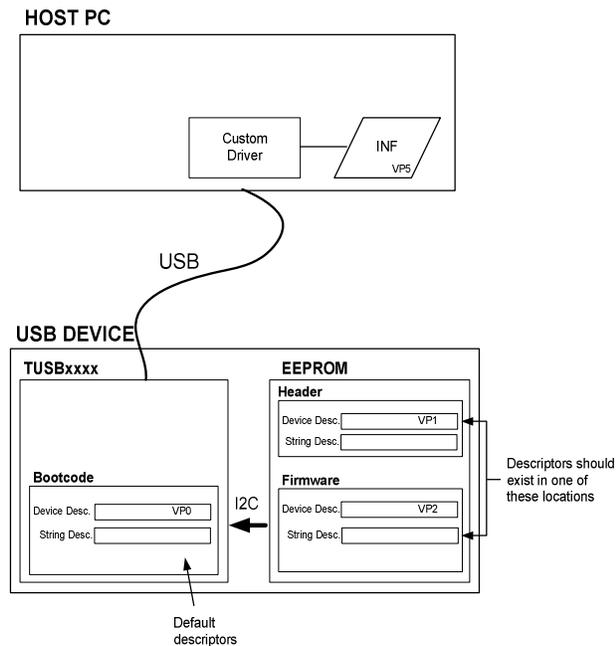


Figure 6. Custom Driver without Firmware Download

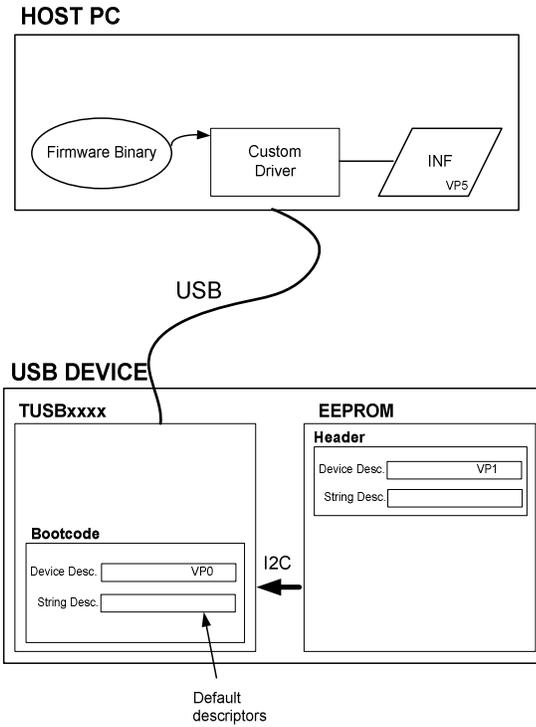


Figure 7. Custom Driver with Firmware Download Capability

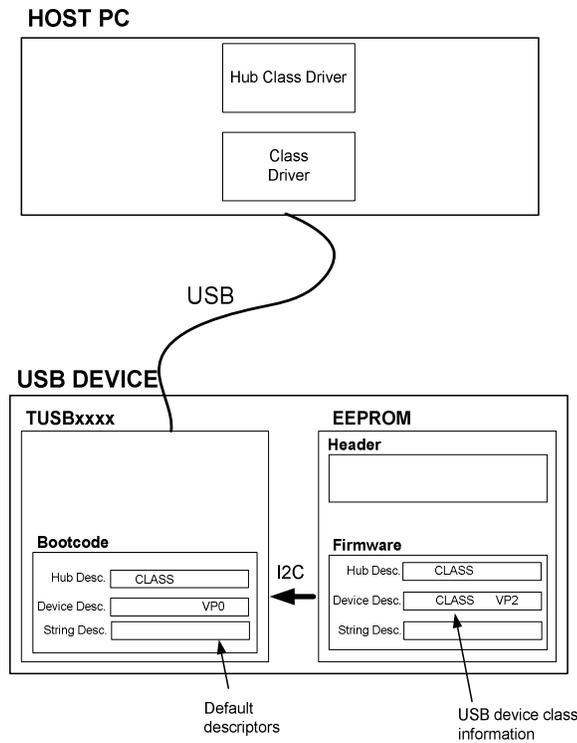


Figure 8. Hub-Equipped Device (TUSB2136/TUSB5052) Used with Class Driver

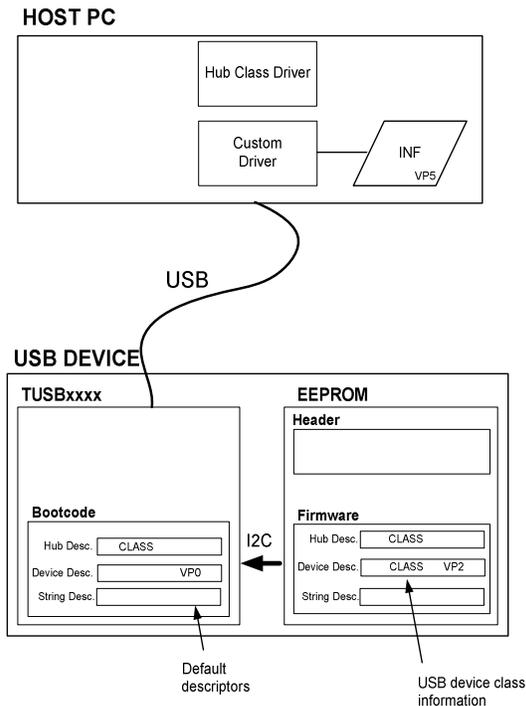


Figure 9. Hub-Equipped Device (TUSB2136/TUSB5052) Used with Custom Driver

7 TI Website

For more information, please visit TI's website for Connectivity Products at:

<http://www.ti.com/connectivity>

Here you will find information about TI's USB controllers. For information about specific devices, as well as access to application notes, EVMs, and software referenced in this document, use the part number search feature to find the product page for the device in question.

References

1. *Universal Serial Bus Specification, Rev. 2.0*
2. *TUSB2136 Data Manual (SLLS442)*
3. *TUSB3210 Data Manual (SLLS466)*
4. *TUSB3410 Data Manual (SLLS519)*
5. *TUSB5052 Data Manual (SLLS454)*
6. *TUSB6250 Data Manual (SLLS535)*
7. *Header Generator utility readme*
8. *TUSB2136/TUSB3210/TUSB5052/TUSB5152 USB Firmware Programming Flow 8052 Embedded (SLLU020)*

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to 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 TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI 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 TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated