*Application Note*

# ***Implement HID and CDC USB Composite Device With TMS320F28P559SJ-Q1 in IVI***

*Joe Ji*

**ABSTRACT**

The SoC based intelligent cabin brings the smart and diversified driving experiences in electric vehicles. USB interface has been widely used in the personal electronic and industry for decades. Since the USB application is mature and flexible, the application starts to take the data exchange roles in the human interface of intelligent cabin, such as IVI and touch screen instead of a CAN network. This application note shows the design to implement the HID and CDC USB composite devices to realize the SCI, button, and touchscreen device enumeration and data report with the TMS320F28P559SJ-Q1 device. The demo code and test case can be found in the C2000Ware SDK v6.01 and this E2E thread.

## Table of Contents

## Trademarks

All trademarks are the property of their respective owners.

# 1 Introduction

## 1.1 Intelligent Cabin and IVI System

The intelligent cabin in electric vehicles (EVs) is equipped with advanced technologies to enhance the driving experience, comfort, and safety. This includes large displays, voice command, internet access, and over-the-air updates. In-vehicle infotainment (IVI) is the key human interface of intelligent cabin that provides entertainment, information, and connectivity to drivers and passengers, encompassing features such as navigation, audio and video playback, and smartphone integration.

## 1.2 USB Interface Application in IVI

The USB interface is increasingly used in IVI systems for various functions, including media playback, device charging, and data transfer due to the highly standardized scalability specifications. This not only supports users to connect personal electronic devices, but also supports the communication between SoC and other IVI peripherals such as touchscreen, button, camera, and speakers. The TI C2000 real time MCU TMS320F28P55x has a variety of communication interface integration, including PMBus, I2C interfaces, CAN-FD, USB 2.0 integrates MAC + PHY, SPI, SCI, and LIN to support the different communication bridges to the USB. One possible USB connection structure in the IVI is shown in Figure 1-1.

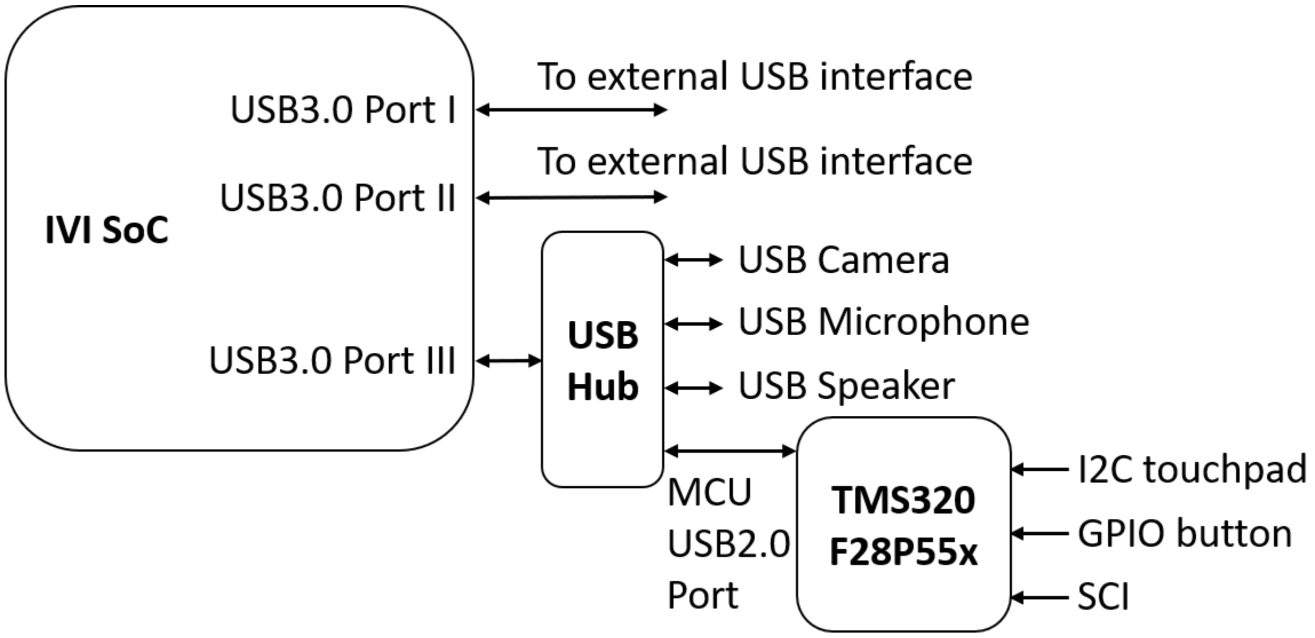


**Figure 1-1. USB Connection Structure in IVI**

## 1.3 TMS320F28P55x Introduction

The TMS320F28P55x (F28P55x) is a member of the C2000™ real-time microcontroller family of scalable, ultra-low latency devices designed for efficiency in automotive body electronics and lighting. The real-time control subsystem is based on TI's 32-bit C28x DSP core, which provides 150MHz of signal processing performance for floating- or fixed-point code running from either on-chip flash or SRAM. The C28x CPU is further boosted by the Floating-Point Unit (FPU), Trigonometric Math Unit (TMU), and VCRC (Cyclical Redundancy Check) extended instruction sets, speeding up common algorithms key to real-time control systems.

The CLA allows significant offloading of common tasks from the main C28x CPU. The CLA is an independent 32-bit floating-point math accelerator that executes in parallel with the CPU. Additionally, the CLA has dedicated memory resources and the CLA can directly access the key peripherals that are required in a typical control system. Support of a subset of ANSI C is standard, as are key features such as hardware breakpoints and hardware task-switching. The F28P55x supports up to 1088KB of flash memory divided into four 256KB banks plus one 64KB bank, which enables the programming of one bank and execution in another bank in parallel. Up to 133KB of on-chip SRAM is also available to supplement the flash memory.

The Live Firmware Update hardware enhancements on F28P55x allow fast context switching from the old firmware to the new firmware to minimize application downtime when updating the device firmware.

High-performance analog blocks are integrated on the F28P55x real-time microcontroller (MCU) and are closely coupled with the processing and PWM units to provide real-time signal chain performance. Twenty-four PWM channels, all supporting frequency-independent resolution modes, enable control of various power stages from a 3-phase inverter to power factor correction and advanced multilevel power topologies.

The inclusion of the Configurable Logic Block (CLB) allows the user to add custom logic and potentially integrate FPGA functions into the C2000 real-time MCU. Interfacing is supported through various industry-standard communication ports (such as SPI, SCI, I2C, PMBus, LIN, and CAN FD) and offers multiple pin-muxing options for signal placement. The functional block diagram of TMS320F28P55x is shown in Figure 1-2.

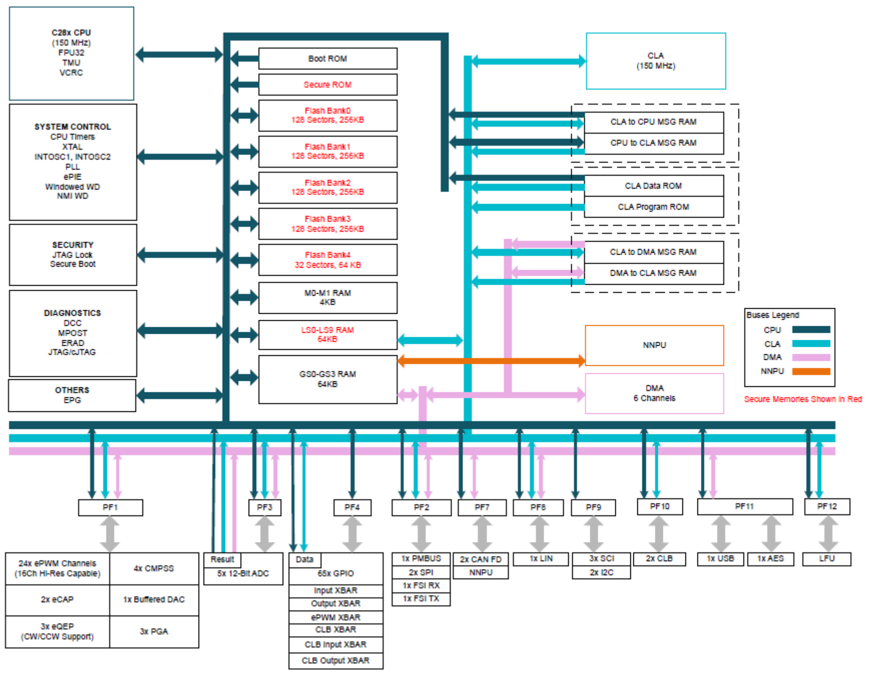


Figure 1-2. TMS320F28P55x Functional Block Diagram

# 2 USB Composite Device Enumeration and Data Report

This section introduces the USB composite device enumeration and data report format according to the standard USB protocol. This section also highlights the file directions to find definitions in the demo project and includes the software details realized in the demo. For more USB information, see the USB-IF official website.

## 2.1 USB Composite Device Enumeration

USB composite device enumeration depends on the USB descriptors, the single HID or CDC device descriptor must be verified separately in projects first before building the composite device descriptor.

### *2.1.1 Descriptor Structure*

The composite device descritptor structure is shown in Figure 2-1. The detailed functions of each descriptor are explained in the following sections.

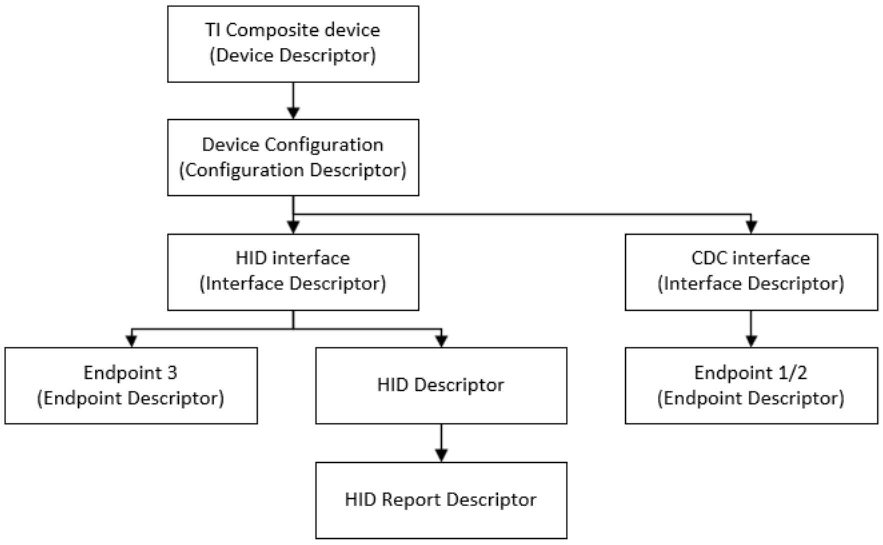


**Figure 2-1. Composite Device Descriptor Structure**

### 2.1.2 Descriptor Types

This section explains the descriptors details to realize the USB enumeration.

#### 2.1.2.1 Device Descriptor

The device descriptor is the first descriptor queried by the host during enumeration. This is to communicate to the host what specification of USB the device complies with and how many possible configurations are available on the device. Upon successful processing of the device descriptor, the host reads all the configuration descriptors.

In the TI USB composite device project, the device descriptor structure is defined in usblib.h. The instance of the structure is defined in tCompositeInstance structure in *usbdcomp.h* called by the USBDCompositeInit function. The variable values are initialized by the BuildCompositeDescriptor function in *usbcomp.c*. The key elements of the device descriptor are shown in Table 2-1.

```
typedef struct
{
 uint8_t bLength;
 uint8_t bDescriptorType;
 uint16_t bcdUSB;
 uint8_t bDeviceClass;
 uint8_t bDeviceSubClass;
 uint8_t bDeviceProtocol;
 uint8_t bMaxPacketSize0;
 uint16_t idVendor;
 uint16_t idProduct;
 uint16_t bcdDevice;
 uint8_t iManufacturer;
 uint8_t iProduct;
 uint8_t iSerialNumber;
 uint8_t bNumConfigurations;
}
PACKED tDeviceDescriptor;
```

**Table 2-1. Key Elements of a Device Descriptor**

| Key Elements | Description |
|---|---|
| bcdUSB | Informs the host of what version of USB the device supports |
| bDeviceClass | 00 - The device class is defined in the Interface Descriptor<br>FF - the device class is Vendor class<br>any other number is the specification for the class of this device |
| idVendor | 16-bit number assigned by USB.org to the manufacturer of the product |
| idProduct | 16-bit product model ID assigned by the vendor to this product |
| bNumConfigurations | How many different configurations are available for this device |

### 2.1.2.2 Configuration Descriptor

A device can have more than one configuration. Each device configuration is assigned a number. The configuration descriptor serves two purposes:

1. Informs the host interfaces quantity in the configuration. The composite device requires more than one interface in the configuration.
2. Device power consumption of each configuration: If the device is capable of controlling the power consumption, the device offers more than one configuration. Each configuration advertises how much power is consumed if the configuration is activated.

Thus, multiple configuration can be claimed. Only one configuration can be active at any time. When a configuration is active, all of the interfaces and endpoints are available to the host. Devices that have multiple interfaces are referred to as composite devices. One physical product with one available USB connector appears to the host as two separate devices.

In the TI USB composite device project, the configuration descriptor structure is defined in *usblib.h*. The instance of the structure is defined in *tCompositeInstance* structure in *usbdcomp.h* called by the *USBDCompositeInit* function. The variable values are initialized by *BuildCompositeDescriptor* in *usbcomp.c*. The key elements of a configuration descriptor are listed in Table 2-2.

```
typedef struct
{
 uint8_t bLength;
 uint8_t bDescriptorType;
 uint16_t wTotalLength;
 uint8_t bNumInterfaces;
 uint8_t bConfigurationValue;
 uint8_t iConfiguration;
 uint8_t bmAttributes;
 uint8_t bMaxPower;
}
PACKED tConfigDescriptor;
```

**Table 2-2. Key Elements of an Interface Descriptor**

| Key Elements | Description |
|---|---|
| bNuminterfaces | Number of Interface Descriptor tables available |
| MaxPower | Power load of this device if the host activates this configuration |

### 2.1.2.3 Interface Descriptor

An interface descriptor describes the details of the function of the product. For example, if the device is a keyboard, the specified device class is Human Interface Device (HID) and the number of endpoints is two. See the *USB Device Classes* page for details on USB Device Class codes in *usblib.h*. Specific definitions can be found in the appropriate device class header files.

In the TI USB composite device project, the interface descriptor structure is defined in *usblib.h*, the instance and values are initialized by *BuildCompositeDescriptor* in *usbcomp.c* called by *USBDCompositeInit* function, the key elements of an interface descriptor are listed in Table 2-3.

```
typedef struct
{
 uint8_t bLength;
 uint8_t bDescriptorType;
 uint8_t bInterfaceNumber;
 uint8_t bAlternateSetting;
 uint8_t bNumEndpoints;
 uint8_t bInterfaceClass;
 uint8_t bInterfaceSubClass;
 uint8_t bInterfaceProtocol;
 uint8_t iInterface;
}
PACKED tInterfaceDescriptor;
```

Table 2-3. Key Elements of an Interface Descriptor

| Key Elements | Description |
|---|---|
| bNumEndpoints | Number of endpoints in the interface |
| bInterfaceClass | USB device class used to set transfer types for the endpoints |

### 2.1.2.4 Endpoint Descriptor

Each endpoint used by device contains a descriptor. The descriptor provides the endpoint address, the size of the endpoint, and the data transfer type used to access the endpoint.

In the TI USB composite device project, the endpoint descriptor structure is defined in *usblib.h*. The instance and values are initialized by *BuildCompositeDescriptor* in *usbcomp.c* called by USBDCompositeInit function. The key elements of an interface descriptor are shown in Table 2-4.

```
typedef struct
{
 uint8_t bLength;
 uint8_t bDescriptorType;
 uint8_t bEndpointAddress;
 uint8_t bmAttributes;
 uint16_t wMaxPacketSize;
 uint8_t bInterval;
}
PACKED tEndpointDescriptor;
```

Table 2-4. Key Elements of an Interface Descriptor

| Key Elements | Description |
|---|---|
| bEndpointAddress | The address of the endpoint |
| wMaxPacketSize | Length of the endpoint |
| bInterval | How often in frames is this endpoint to be serviced by the host |

### 2.1.2.5 String Descriptor

String descriptors are optional. Human-readable strings that the host OS can display is not used in this project.

```
typedef struct
{
 uint8_t bLength;
 uint8_t bDescriptorType;
 uint8_t bString;
}
PACKED tStringDescriptor;
```

## 2.2 USB Composite Device HID Data Report

This section describes the HID data report definition after USB composite device enumeration is complete.. The CDC data format is simple. The data can be directly filled into or obtained from the USB endpoints with USB Endpoint APIs, so this section focuses on the USB HID data report.

### *2.2.1 Data Report Item*

The HID data reporting uses the *item* as the basic message element, and changes the *item* headers to identify the different HID devices report contents. The HID report descriptor must be specified once the interface descriptor defines a HID interface. The item format is shown in Figure 2-2. Bits 23-8 represent the data to be transmitted. Bits 7-0 are the item header. bSize specifies the data length of the item in bytes. bType specifies the item types. The definition details are shown in the following text. bTag specifies the item types which is described in Configuration Descriptor. The complete definition of bTag can be found on the USB-IF official website.

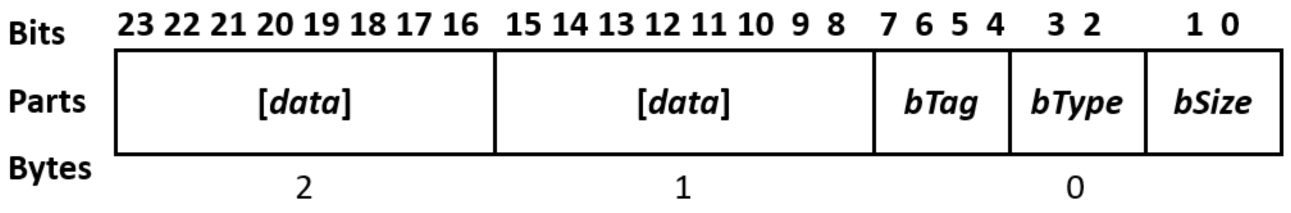


**Figure 2-2. Data Report Items Structure**

bSize = 0 – No available data

bSize = 1 – 1-byte available data

bSize = 2 – 2-bytes available data

bSize = 3 – 4-bytes available data

bType = 0 – Main item

bType = 1 – Global item

bType = 2 – Local item

### *2.2.2 HID Report Descriptor Structure*

The data report items are organized into the structure shown in Figure 2-3. The types of items used in the HID data report are described in the following sections.

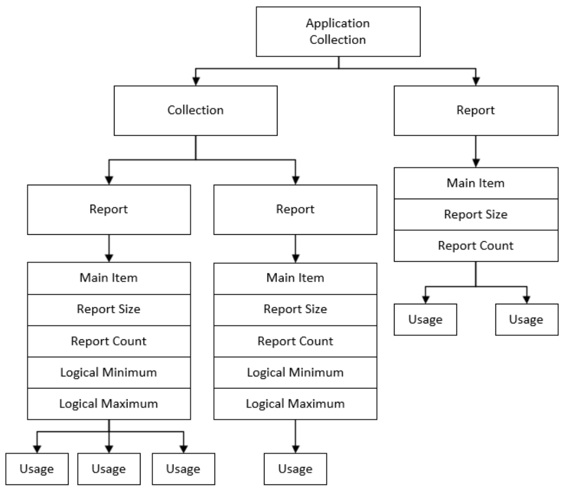


**Figure 2-3. HID Report Descriptor Structure**

#### 2.2.2.1 Main Item

Main item is used to define the data domain or a group of data domain. The data domain types are defined with bTag, which is listed in Table 2-5

**Table 2-5. Main Item Definition and bTag Value**

| Item Type | bTag Value | Description |
|---|---|---|
| Input | 0x8 | USB device reports data to USB host |
| Output | 0x9 | USB host send data to USB device |
| Feature | 0xB | Return configuration information |
| Collection | 0xA | Start of data domain |
| End Collection | 0xC | End of data domain |

With the input, output, and feature item types, the data bytes come after the bTag, describes the detailed functions of this main item. The complete definition can be found in USB-IF documents hid-6.2.2.7. The data bytes come after the collection. End collection item types are listed in Table 2-6.

**Table 2-6. bTag Details on Collection and End Collection Item Types**

| Item Type | bTag Value | Description |
|---|---|---|
| Collection | 0xA0 | Physical: defines the USB device physical structures and connection. For example, key and scroll wheel |
| | 0xA1 | Application: defines the input and corresponding output signal of the USB device. For example, touchpad finger position, volume control. |
| | 0xA2 | Logical: maps the physical and application. For example defines the scroll wheel (physical) as the volume control (application) |
| | 0xA3 | Start of data domain |
| End Collection | 0xC0 | Close an item collection |

#### 2.2.2.2 Global Item

Global item is used to define the data properties which are listed in Table 2-7. This takes effect for all the following items until another global item appears. The complete definition can be found in USB-IF documents hid-6.2.2.7.

**Table 2-7. Global item Definition and bTag Value**

| Item Type | bTag Value | Description |
|---|---|---|
| Usage Page | 0x0 | Classification of this item. For example, button page and keyboard page |
| Logical Minimum | 0x1 | The minimum data value of this item |
| Logical Maximum | 0x2 | The maximum data value of this item |
| Physical Minimum | 0x3 | The minimum data value of this item |
| Physical Maximum | 0x4 | The maximum data value of this item |
| Report Size | 0x7 | The length of the data domain (in bytes) |
| Report Count | 0x9 | The number of the data domain |
| ReportId | 0x8 | Report ID |

#### 2.2.2.3 Local Item

The local item is used to define the data detail properties after the global item. The definition of different local items are listed in Table 2-8. This takes effect for all the following items until another global item appears. The complete definition can be found in USB-IF documents hid-6.2.2.8.

**Table 2-8. Local Item Definition and bTag Value**

| Item Type | bTag Value | Description |
|---|---|---|
| Usage | 0x0 | Specification of the usage according to the usage page. |
| Usage Minimum | 0x1 | The minimum data value of this usage |
| Usage Maximum | 0x2 | The maximum data value of this usage |

## 2.3 Example of Building a HID Report Descriptor

The HID report descriptor instance is claimed in g_pui8CustomReportDescriptor[] in usbhidcustom.c. Using a point 1 report descriptor as an example, there are four usages divided into two usage pages in one collection. The format of each usage is specified with LogicalMinimum/ LogicalMaximum/ ReportSize/ ReportCount/ Input.

```
UsagePage(USB_HID_DIGITIZERS),
Usage(USB_HID_FINGER),
Collection(USB_HID_LOGICAL),
Usage(USB_HID_CONTACT_ID),
LogicalMinimum(0),
LogicalMaximum(9),
ReportSize(8),
ReportCount(1),
Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE | USB_HID_INPUT_ABS),
Usage(USB_HID_TOUCH_PRESS),
LogicalMinimum(0),
LogicalMaximum(1),
ReportSize(1),
ReportCount(1),
Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE | USB_HID_INPUT_ABS),
ReportSize(7), // Pad to 1byte
ReportCount(1),
Input(USB_HID_INPUT_CONSTANT | USB_HID_INPUT_ARRAY | USB_HID_INPUT_ABS),
UsagePage(USB_HID_GENERIC_DESKTOP),
Usage(USB_HID_X),
LogicalMinimum(0),
LogicalUi16Maximum(4095),]
ReportSize(16),
ReportCount(1),
Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE | USB_HID_INPUT_ABS),
Usage(USB_HID_Y),
LogicalMinimum(0),
LogicalUi16Maximum(4095),
ReportSize(16),
ReportCount(1),
Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE | USB_HID_INPUT_ABS),
EndCollection,
```

## 3 Software Realization

This section shows the USB APIs that are included in the C2000ware SDK USB driverlib and demo project. With the USB APIs, the USB composite device enumeration and data report is realized. The relationship between the USB APIs and USB descriptors is shown in USB Composite Device Enumeration and Data Report.

### 3.1 APIs for USB Composite Device Initialization

The APIs use the USB descriptors to initialize the USB peripherals. After those APIs are called, the USB device can be enumerated by the USB host. The data report format is also initialized in those APIs.

#### *3.1.1 USBStackModeSet*

This function is called in USBLib_init() in C2000Ware_libraries_init() to initialize the USB port in USB device mode. There is no return code with this function.

**Table 3-1. USBStackModeSet API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| ui32Index | Specifies the USB controller whose mode of operation is to be set. There is only one USB IP in F28P55 so this parameter must be set to 0. |
| iUSBMode | Input eUSBModeDevice to make USB operates in device mode. |
| pfnCallback | No callback function is required. Set the value to 0. |

#### *3.1.2 USBDCDCCompositeInit*

This function is to initialize the required parameters for the specific CDC device and assign the structure data entry point to the psCompEntry[] array to build the composite device descriptor in USBDCompositeInit function. There is no return code with this function.

**Table 3-2. USBDCDCCompositeInit API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| ui32Index | Specifies the USB controller whose mode of operation is to be set. There is only one USB IP in F28P55 so this parameter must be set to 0. |
| psCDCDevice | Points to a structure containing customizing operation parameters of the USB CDC device. |
| psCompEntry[] | Composite device entry to initialize when creating a composite device. |

#### *3.1.3 USBDHIDCustomCompositeInit*

This function is to initialize the necessary parameters for the specific HID device and assign the structure data entry point to the psCompEntry[] array to build the composite device descriptor in USBDCompositeInit function. The USBDHIDCustomCompositeInit() calls USBDHIDCompositeInit() to realize the underlay HID composite device descriptor configuration. There is no return code with this function.

**Table 3-3. USBDHIDCustomCompositeInit API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| ui32Index | Specifies the USB controller whose mode of operation is to be set. There is only one USB IP in F28P55 so this parameter must be set to 0. |
| psCustomDevice | Points to a structure containing parameters customizing the operation of the HID device. |
| psCompEntry[] | Composite device entry to initialize when creating a composite device. |

### 3.1.4 USBDCompositeInit

This function uses the structures that are initialized by USBDCDCCompositeInit and USBDHIDCompositeInit to initialize basic operation and prepare for the enumeration composite class device. There is no return code with this function.

**Table 3-4. USBDCompositeInit API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| ui32Index | Specifies the which mode of operation is to be set. There is only one USB IP in F28P55 so this parameter must be set to 0. |
| psDevice | Points to a structure containing parameters customizing the operation of the composite device. The parameters are specified by USBDCDCCompositeInit and USBDHIDCompositeInit in psCompEntry[]. |
| ui32Size | The size in bytes of the data pointed to by the pui8Data parameter. |
| pui8Data | The data area that the composite class can use to build up descriptors. |

## 3.2 APIs for USB Composite Device CDC Data Report

APIs use the USB buffers to transmit and receive the CDC data which is received and transmitted by the SCIA interface.

### 3.2.1 USBBufferSpaceAvailable

This function returns the number of free bytes in the buffer. This API is called in ReadSCIData() function in USBSCIRXIntHandler() to check the number of bytes that can be read from the USB controller before the buffer is full.

**Table 3-5. USBBufferSpaceAvailable API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| psBuffer | The pointer to the buffer instance which is to be queried. |

### 3.2.2 USBBufferWrite

This function writes the supplied data into the transmit buffer. This API is called in ReadSCIData() function in USBSCIRXIntHandler(). The transmit buffer data is packaged according to the constraints imposed by the lower layer in use and sent to the USB controller as soon as possible. Once a packet is transmitted and acknowledged, a USB_EVENT_TX_COMPLETE event is sent to the application callback indicating the number of bytes that are sent from the buffer. Attempts to send more data than the transmit buffer range results in data loss. The value returned by the function indicates the actual number of bytes written to the buffer.

**Table 3-6. USBBufferWrite API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| psBuffer | The pointer to the buffer instance that is to be queried. |
| pui8Data | The first byte of data that is to be written. |
| ui32Length | The number of bytes of data to write to the buffer. |

### 3.2.3 USBBufferRead

This function reads up to ui32Length bytes of data received from the USB host into the supplied application buffer. If the receive buffer contains fewer than ui32Length bytes of data, the data that is present is copied and the return code indicates the actual number of bytes copied to pui8Data.

**Table 3-7. USBBufferRead API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| psBuffer | The pointer to the buffer instance from which data is to be read. |
| pui8Data | The buffer into which the received data is written. |
| ui32Length | The size of the buffer pointed to by pui8Data. |

### 3.2.4 USBDCDCTxHandler

This function is called by the CDC driver to query the events related to operation of the transmit data channel (the IN channel carrying data to the USB host). Only a USB_EVENT_TX_COMPLETE event is acceptable for this function. The function is pending here if there are any other events happening when DEBUG is enabled. The return code of this function is 0.

**Table 3-8. USBDCDCTxHandler API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCBData | Customized callback pointer for this channel. |
| ui32Event | Identifies the event that is causing the notification. |
| ui32MsgValue | Event-specific value. |
| pvMsgData | Event-specific pointer. |

### 3.2.5 USBDCDCRxHandler

This function is called by the CDC driver to query the events related to operation of the receive data channel (the OUT channel carrying data from the USB host). The CDC driver fills the SCI FIFO once USB_EVENT_RX_AVAILABLE event occurs. The return code 1 indicates that SCI is in the transmission process. The return code 0 indicates that SCI is currently idle.

**Table 3-9. USBDCDCRxHandler API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCBData | Customized callback pointer for this channel. |
| ui32Event | Identifies the event that is causing the notifications. |
| ui32MsgValue | Event-specific value. |
| pvMsgData | Event-specific pointer. |

## 3.3 APIs for USB Composite Device HID Data Report

The APIs use the USB buffers to transmit the simulated HID data. This includes the ten points touchpad position and the keyboard characters. Before calling the HID Report API, g_eCustomState==MOUSE_STATE_IDLE must be checked. Otherwise the report is unreliable.

### *3.3.1 USBDHIDCustomTouchEvent*

This function is called by MultTouchSimHandler() to report touchpad state changes and touch position to the USB host. The return code indicates whether the transmission is successful or has an error.

**Table 3-10. USBDHIDCustomTouchEvent API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCustomDevice | Pointer to the HID device instance structure. |
| psEvent | Relative touch pointer movement event report. |

### *3.3.2 USBDHIDCustomReportKey*

This function is called by *KeySimHandler ()* to report a keyboard input to the USB host. There is no return code with this function.

**Table 3-11. USBDHIDCustomReportKey API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCustomDevice | Pointer to the hid device instance structure. |
| ui8Key | Key board input event report. |

## 3.4 APIs for USB Composite Device Simulation

The APIs call the underlay APIs described in APIs for USB Composite Device Initialization to APIs for USB Composite Device HID Data Report to realize the USB HID and CDC data report.

### *3.4.1 MultTouchSimHandler*

This function is to simulate the ten points touch event and report the touchpad position to the USB host.

### *3.4.2 KeySimHandler*

This function is to simulate the keyboard input event and report the keyboard characters to the USB host.

## 3.5 APIs for USB Device Operate States Query

The APIs use the callback function to indicate the USB device working state.

### *3.5.1 USBDCDCControlHandler*

This function is called by the CDC driver to perform control-related operations on behalf of the USB host. These functions include setting and querying the serial communication parameters, setting handshake line states, and sending break conditions.

**Table 3-12. USBDCDCControlHandler API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCBData | Client-supplied callback pointer for this channel. |
| ui32Event | Identifies the event the user is being notified about. |
| ui32MsgValue | Event-specific value. |
| pvMsgData | Event-specific pointer. |

### *3.5.2 CustomHandler*

This function is called by the customer HID driver to perform control-related operations and query device information on behalf of the USB host.

**Table 3-13. CustomHandler API Input Variables and Descriptions**

| Input Variables | Description |
|---|---|
| pvCBData | Client-supplied callback pointer for this channel. |
| ui32Event | Identifies the event that is being notified about. |
| ui32MsgValue | Event-specific value. |
| pvMsgData | Event-specific pointer. |

## 4 System Test

This section shows how to verify the functions with the demo project.

### 4.1 Test Setup

Setting up the test bench with TMDSCNCD28P55X controlCARD evaluation module is simple. The MCU USB port physical interface is already integrated in the broad with a USB-C socket. Users can connect the XDS110 debugger USB and the MCU USB to the PC with two USB cables. Once the hardware connection is connected, build and download the firmware to the board. Click *Resume* to run the example.

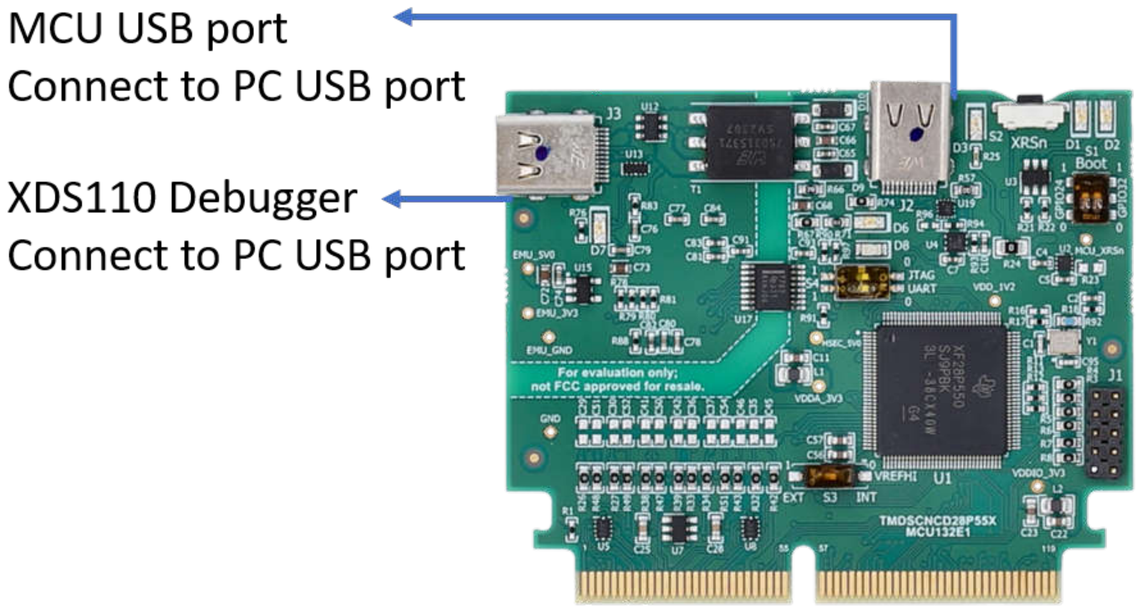
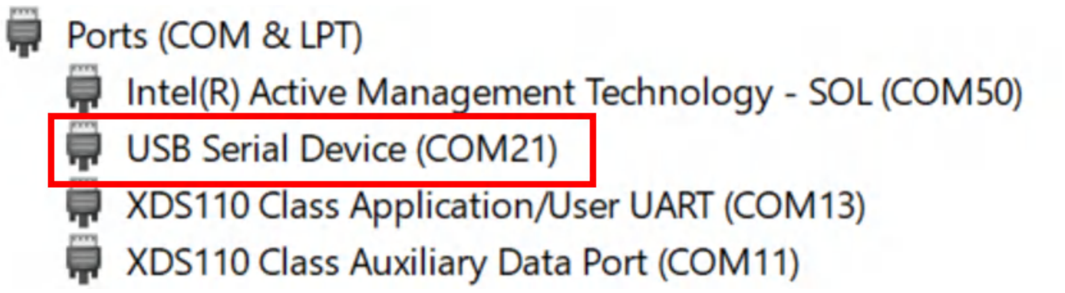


**Figure 4-1. Hardware Setup for Test**

### 4.2 USB SCI CDC Device Function Test

To verify the CDC device function, users can see a new USB Serial Device is added to Ports (COM&LPT) in window device manager, as shown in Figure 4-2. Users can open the SCI host such as TeraTerm or sscom in PC and connect to both XDS110 Class Application/User UART(COMx) and USB Serial Device (COMx) and set the baud rates to 115200 to do the loopback test.

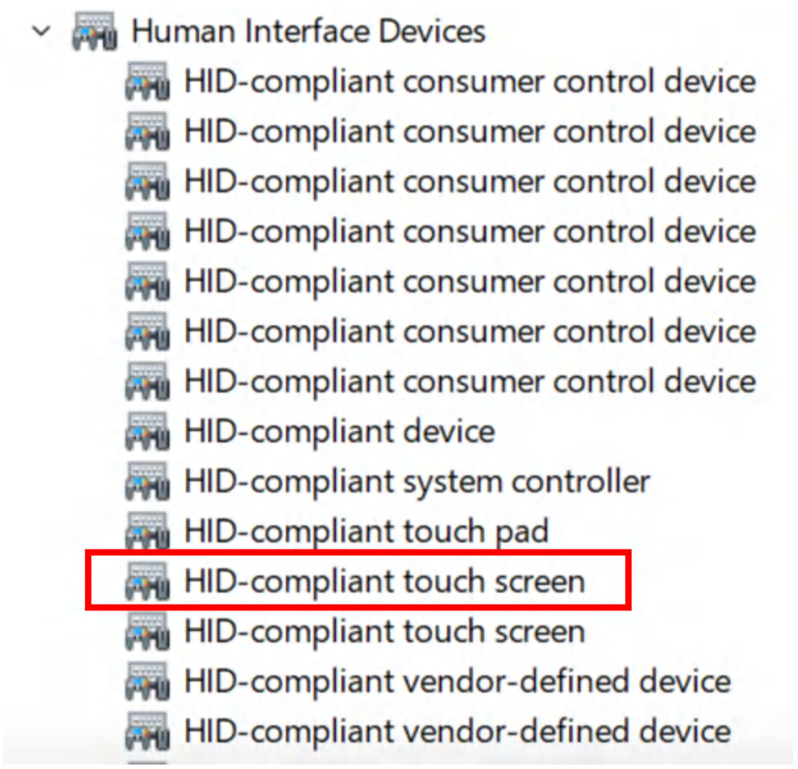


Ports (COM & LPT)
Intel(R) Active Management Technology - SOL (COM50)
USB Serial Device (COM21)
XDS110 Class Application/User UART (COM13)
XDS110 Class Auxiliary Data Port (COM11)

**Figure 4-2. PC Device Manager View of USB Ports**

### 4.3 USB Touch Screen HID Device Function Test

To verify the button HID device function, users can see a new HID-compliant touch screen is added to human interface devices in the Window Device Manager, as shown in Figure 4-3. Users can see the 10 points movement in the screen if the PC screen can support touch function.



Human Interface Devices
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant consumer control device
HID-compliant device
HID-compliant system controller
HID-compliant touch pad
HID-compliant touch screen
HID-compliant touch screen
HID-compliant vendor-defined device
HID-compliant vendor-defined device

**Figure 4-3. PC Device Manager View of USB HID Touch Screen Devices**

## 4.4 USB Button HID Device Function Test

To verify the Button HID device function, users can see a new HID Keyboard Device is added to Keyboards in Window Device Manager, as shown in Figure 4-4. Users can open the text editing tools such as Notepad, and see the numbers one through eight that are printed in the text file.

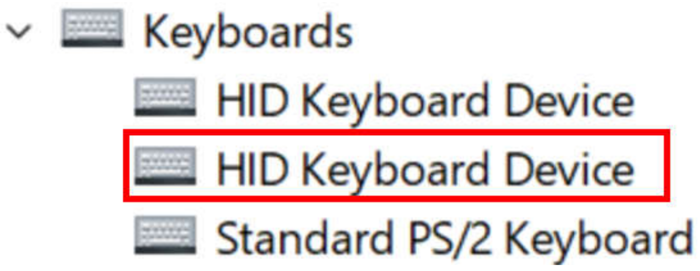


**Figure 4-4. PC Device Manager View of USB HID Keyboard Devices**

## 5 Summary

This application note shows the design to implement HID and CDC USB composite device to realize the SCI, button, and touchscreen device enumeration and data report with TMS320F28P559SJ-Q1. The demo code and test case are also shared in C2000ware SDK v6.01.

## 6 References

- Texas Instruments, *TMS320F28P55x Real-Time Microcontrollers*, data sheet.
- Texas Instruments, *TMS320F28P55x Real-Time Microcontrollers*, technical reference manual.
- Texas Instruments, *TMDSCNCD28P55X controlCARD Information Guide*, user's guide.
- USB, *USB Documentation Library*, documentation.
- USB-IF, *USB-IF HID Descriptor tool*, descriptor tool.
- USB, *Device Class Definition for Human Interface Devices (HID)*, firmware specification.
- USB, *HID Usage Tables*, usage tables.

# A Appendix

## USB Full Speed Transmission Limitation and Workaround

The USB full speeds only allows maximum 64 bytes data transmission in one package. If there are multiple package transmissions required such as with HID touch screen data, TxPktRdy cannot be cleared by the hardware in time. This causes data loss during multiple package transmissions.

### Related Code

```
usbdenum.c:USBDEP0StateTx()[2580]:
g_psDCDInst[0].iEP0State = eUSBStateTx;
if(ui32NumBytes > EP0_MAX_PACKET_SIZE)
 ui32NumBytes = EP0_MAX_PACKET_SIZE;
pui8Data = (uint8_t *)g_psDCDInst[0].pui8EP0Data;
g_psDCDInst[0].ui32EP0DataRemain -= ui32NumBytes;
g_psDCDInst[0].pui8EP0Data += ui32NumBytes;
USBEndpointDataPut(USB_BASE, USB_EP_0, pui8Data, ui32NumBytes);
usb.c:USBEndpointDataPut()[2806]:
if(HWREGB(ui32Base + USB_O_CSRL0 + ui32Endpoint) & ui8TxPktRdy)
 return(-1);
```

### Workaround

Add enough delay cycle to make sure the TxPktRdy is cleared by the hardware correctly.

### Related Code

```
if(HWREGB(ui32Base + USB_O_CSRL0 + ui32Endpoint) & ui8TxPktRdy)
{
 for (ui8Timeout = 0; ui8Timeout < 5; ui8Timeout++) {
 USBDelay(1);
 if(!(HWREGB(ui32Base + USB_O_CSRL0 + ui32Endpoint) & ui8TxPktRdy))
 {
 break;
 }
 }
 if(HWREGB(ui32Base + USB_O_CSRL0 + ui32Endpoint) & ui8TxPktRdy)
 {
 //Force clean rdy bit
 HWREGB(ui32Base + USB_O_CSRL0 + ui32Endpoint) &= ~ui8TxPktRdy;
 }
}
```