THIS SPEC IS OBSOLETE

Spec No:	001-67825
Spec Title:	AN5083 - IMPLEMENTATION OF MULTIPLE INTERFACE ISOCHRONOUS USB COMPOSITE PERIPHERAL USING EZ-HOST/EZ-OTG(TM)
Replaced by:	NONE

Implementation Of Multiple Interface Isochronous USB Composite Peripheral Using EZ-HOST/EZ-OTG™

Author: Narayana Murthy M

Associated Project: Yes

Associated Part Family: CY7C67300/CY7C67200

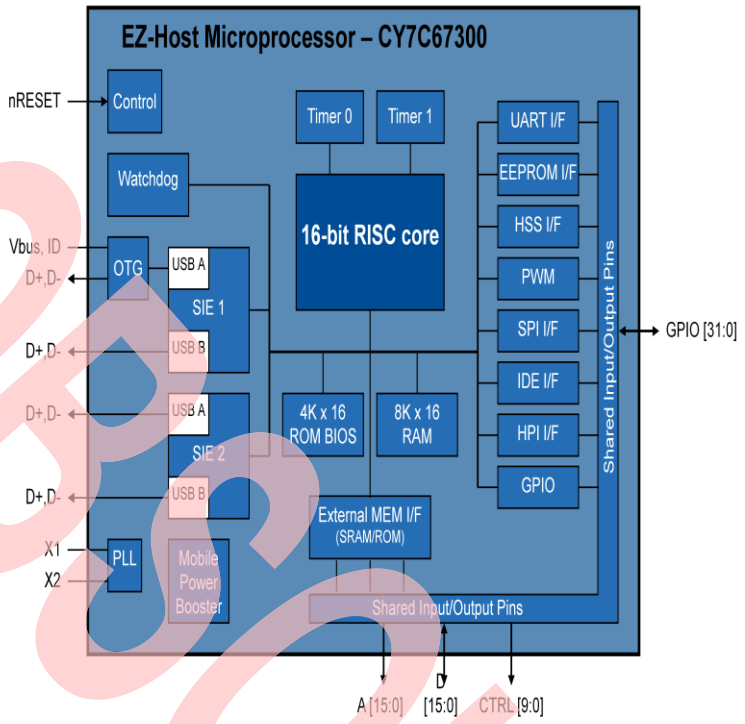
This application note provides details on how to implement a Multiple Interface-based USB peripheral using either EZ-HOST or EZ-OTG. It introduces basic blocks and operating modes of EZ-HOST/EZ-OTG chipsets. It also explains generic firmware code required to support any multi-interface USB peripheral on EZ-HOST/EZ-OTG. An example of a combination of Audio and HID class peripheral firmware is included to show multiple interface implementations. The note concludes with explanation on how to test this firmware using EZ-HOST/EZ-OTG development kit board and a Windows PC.

Contents

1 Introduction

The Cypress EZ-HOST and EZ-OTG, commonly termed as OTG-Host, are USB 2.0 Full/Low speed Host/Peripheral controllers. Both the chipsets are based on CY16: a 16-bit proprietary RISC processor. Two distinct Serial-interface-Engines (SIE1 and SIE2) on each of those chipsets provide flexibility to configure them in OTG, Host, or Peripheral mode. Figure 1 summarizes the functional blocks inside the EZ-HOST chip.

Figure 1. EZ-HOST Chip Block Diagram



EZ-HOST and EZ-OTG have many similarities and few differences.

Similarities:

- Internal CY16 RISC based processor
- Support USB Full or Low speed functionality
- 16K internal SRAM
- 8K BIOS
- Support OTG and UART debug functionality

Differences:

Table 1. EZ-HOST and EZ-OTG Features Difference

EZ-HOST	EZ-OTG
Supports up to 4 host ports	Supports up to 2 host ports
External memory bus(XRAM of up to 64KB support)	No external memory available
32 possible GPIO pins	25 possible GPIO pins
4 Pulse-width-Modulation(PWM) device support	No PWM support.

For full details on the EZ-HOST and EZ-OTG features refer to the respective datasheets.

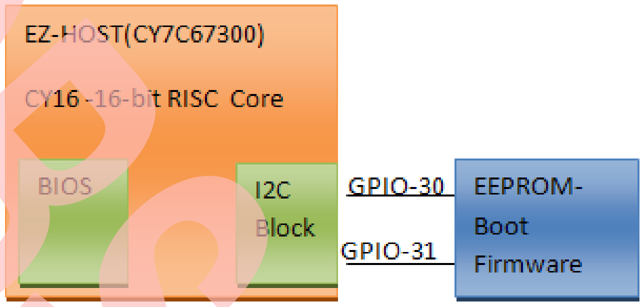
2 Operating Modes of OTG-Host

The OTG-Host operates in three functional modes. The functionality depends on how the firmware is designed to run on different types of hardware. Following is the detailed description of each of these modes.

2.1 Standalone Mode

In this mode the firmware is designed in such a way to work with OTG-HOST internal CY16 RISC core. Below is the snapshot (Figure 2) of Standalone mode operation.

Figure 2. EZ-Host in Standalone Mode Operation

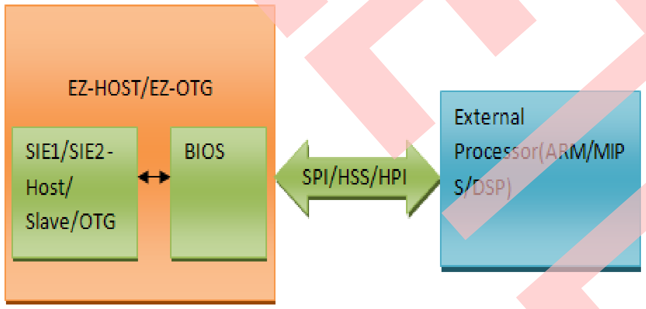


Initially after power-on cycle the OTG-Host BIOS checks boot-strap pins (GPIO[30:31]) logic to determine which of the four interfaces(SPI/HSS/HPI/I2C) the chip need to be configured. Once configured the chip will have capability to communicate with other system processors (via SPI/HSS/HPI) or communicate with I2C EEPROM to boot the stored firmware image .If GPIO[31:30]=11 the chip will configure itself for standalone mode. In standalone mode the firmware could support different USB Class Peripherals/Host like Audio; CDC class which contains multiple interfaces.

2.2 Co-processor Mode

In this mode an external system processor like ARM,MIPS or DSP can communicate with EZ-HOST/EZ_OTG Host/peripheral/OTG ports using internal BIOS of OTG-HOST. Below Figure 3 shows the Co-processor mode functional view.

Figure 3. OTG-HOST In Co-processor Mode



2.3 Standalone and Co-processor Mode Combination

In this mode the firmware of OTG-HOST is downloaded by external processor using one of the Co-processor interface(SPI/HSS/HPI).The firmware handles most of the tasks and free up external processor bandwidth for other tasks. Typical examples include OTG-HOST enumerating as Audio class peripheral .In this design the OTG-HOST accepts the audio data from PC and transfer it to DSP processor through SPI/HSS/HPI Co-processor interfaces in DMA mode.For more details on this mode refer to CY3663 development kit software -documentation and firmware examples included.

3 Multiple Interface and Configuration Requirements

As per USB 2.0 spec a Composite device is defined as “A device that has multiple interfaces controlled independently of each other”. The composite device contains a single USB device address.

There are several USB class devices with single interface as default. Some devices require multiple interfaces, multiple configurations or combination of both. In multiple configuration device only one configuration will be active at any instant and the host selects the initial configuration during enumeration.

In Multiple interfaces each interface contains different capabilities. Each interface performs a common function achieved by a single or collection of Endpoints. Some of the common usage examples include

- Mass storage Hard Disk + HID: In this the HID interface detects button press on the hard disk enclosure and performs functions such as data backup from PC Host through Mass storage interface.
- Keyboard + Mouse
- Video + USB Hard disk

There are several other examples which are combination of multiple USB classes and different types of hardware.

There can also be multiple alternate interfaces for each interface defined in a USB device. Each alternate interface can contain a slight variation in requirements and capabilities of the interface. Any device that wants to support isochronous transfers must support multiple alternate interfaces in order to pass USBCV testing. The USB 2.0 specification, section 5.6.3 states:

All device default interface settings must not include any isochronous endpoints with non-zero data payload sizes (specified via wMaxPacketSize in the endpoint descriptor).

Alternate interface settings may specify non-zerodata payload sizes for isochronous endpoints. If the isochronous endpoints have a large data payload size, it is recommended that additional alternate configurations or interface settings be used to specify a range of data payload sizes. This increases the chance that the device can be used successfully in combination with other USB devices.

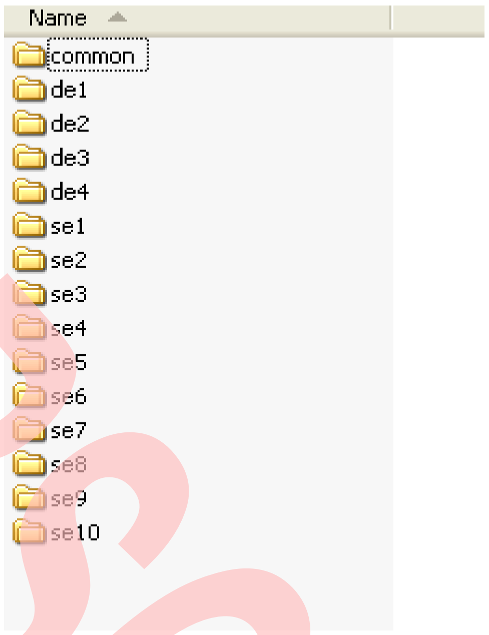
So, in order to support isochronous transfers and be in compliance with the USB specification, a device MUST support multiple alternate interfaces, and the more alternates it supports, the better chance it has to operate successfully under all conditions.

4 Details of OTG-Host Firmware and Frameworks

Before we discuss how to design a multiple-interface based OTG-Host firmware it is important to understand how any firmware example is structured in OTG-Host and its several other components related to make it a functional binary image.

The CY3663 development kit contains EZ-HOST and EZ-OTG development boards. Along with them the CY3663 kit software contains several sample firmware examples that are designed to run on these development boards. Following Figure 4 provides a snapshot overview of all the firmware examples of the kit. These examples are located at *C:\Cypress\USB\OTG-Host\Source\stand-alone* directory after CY3663 DVK Software installation.

Figure 4. CY3663 DVK Firmware Examples Directory Overview



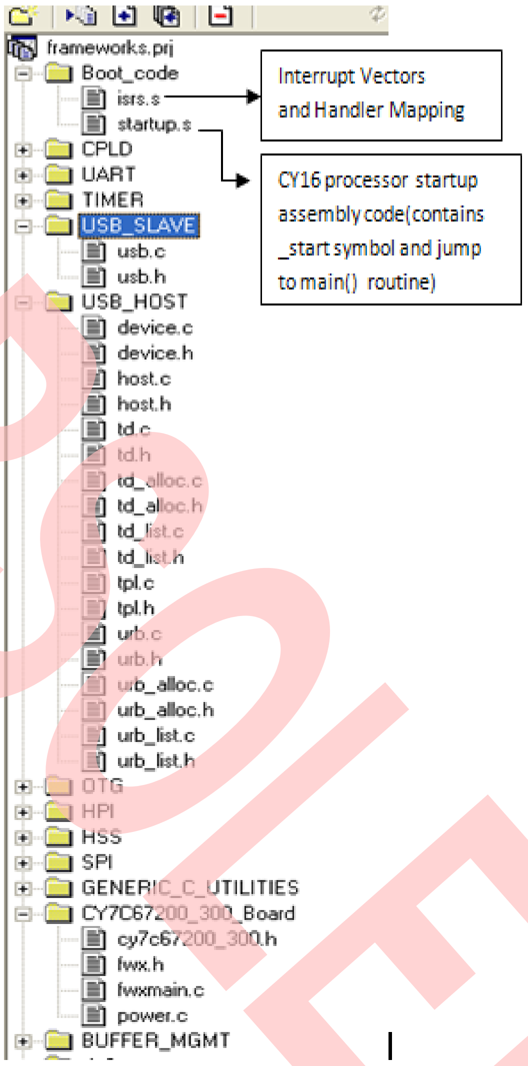
Name

common
de1
de2
de3
de4
se1
se2
se3
se4
se5
se6
se7
se8
se9
se10

Figure 4 shows three sets of folders

- **Simple Examples (se1-se10):** Each of these examples configure EZ-HOST or EZ-OTG development boards in Host/Peripheral modes .For more details refer to *USB Multi-Role Device Design By Example.pdf* by John Hyde. This document is part of CY3663 DVK kit documentation.
- **Design Examples (de1-de4):** These design examples are best described and are explained in detail when running the OTG-Host Navigator (accessed from Start>Programs>Cypress>USB>OTGHost>OTG-Host Navigator). The design examples demonstrate functionality as indicated:
  - Design example 1 (de1) – OTG
  - Design example 2 (de2) – Dual Slave
  - Design example 3 (de3) – Peripheral and Host Pass Through
  - Design example 4 (de4) – Dual Host
- **/common folder:** This folder is called OTG-Host Frameworks. Following Figure 5 provides overview of files in frameworks folders.

Figure 5. View of Frameworks folder in a Project



As shown in Figure 5, the frameworks folder source files can be categorized

- **startup.s:** Typical of any RISC processor CY16 also contains start up code in this assembly file. The BIOS after boot jumps to _**start** symbol defined in this file. At the end of initialization there is jump instruction to **main()** routine defined in fwxmain.c file in the frameworks folder itself.
- **isrs.s:** There are 48 hardware interrupt vectors defined for OTG-HOST. The default empty interrupt service routines (ISR) associated with each interrupt can be replaced with user defined routines. The vector mappings are copied from 0x000-0x00FF memory range of OTG-HOST. The BIOS or firmware uses these locations to jump to relevant ISR in case of a hardware interrupt event. Below is the sample code.

Listing 1: SIE1-USB Slave mode-Standard Request Interrupt Handler

```
ifdef FWX_INCLUDE_SUSB1_STANDARD_HANDLER
.ifdef FWX_INCLUDE_SLAVE_USB_SUPPORT

mov     [old_susb1_standard_int],
 [SUSB1_STANDARD_INT*2]
 mov     [SUSB1_STANDARD_INT*2], susb1_standard_int
.endif ; FWX_INCLUDE_SLAVE_USB_SUPPORT
```

```
.endif ; FWX_INCLUDE_SUSB1_STANDARD_HANDLER

.susb1_standard_int:
    push     [CPU_FLAGS_REG]
    int      PUSHALL_INT              ; save R0-R14 to stack.
    mov      r0, SIE1_DEV_REQ
   call     susb1_standard_handler  ; Call the C level ISR.
   mov      [new_susb1_standard_ret], r0
   int      POPALL_INT              ; restore R0-R14 from stack.
   pop      [CPU_FLAGS_REG]
    cmp      [new_susb1_standard_ret], 0
    je       0f
   ret
```

For more details on interrupts refer to *OTG-Host BIOS User Manual.pdf* from CY3663 DVK documentation.

Interface Source Files: The OTG-Host Frameworks folder also contains source files to access several interfaces defined in OTG-Host USB/UART/HSS/SPI/HPI... (Refer to Figure 1). At the top of each of these source files contains a frameworks macro definition.

Table 2. Frameworks Macro and Source Definitions

Frameworks Macro (defined in fwxcfg.h file)	Frameworks Source File	Function supported
FWX_INCLUDE_HOST_USB_SUPPORT	host.c ,host.h	USB Host mode
FWX_INCLUDE_SLAVE_USB_SUPPORT	usb.c , usb.h	USB Slave mode

As shown in the above Table 2 all the source files defined in frameworks folder are controlled by #define macros in *fwxcfg.h* .This single header file contains all the macros to enable/disable relevant interface functionality in frameworks files. By enabling the macro relevant to the interface the firmware examples (se1-se10 & de1-de4) can access that particular interface.

5 Supporting Multiple interfaces in OTG-Host

To support multiple interfaces the following factors are to be considered

- Interrupt vector Support for USB Peripheral Enumeration
- USB Multiple Interface Initialization and Enumeration

5.1 Interrupt Vector Support for USB Peripheral Enumeration

During enumeration process the PC host will send several requests like Standard, Class and Vendor command requests. Following Table 3 summarizes the hardware interrupts and ISR invoked when PC sends these requests.

Table 3. OTG-HOST USB Peripheral Interrupt Support

Host PC USB Requests	OTG-Host Interrupt Vectors SUSBx (x = 1 →SIE1, 2→SIE2)	Interrupt Service Routine – C handler /Pointers (susbx x = 1→SIE1 2→SIE2)
RESET	SUSBx_DELTA_CONFIG_INT	susbx_delta_config_handler()
STANDARD REQUESTS	SUSBx_STANDARD_INT,	susbx_standard_handler()
GET_DEVICE_DESCRIPTOR	SUSBx_DEVICE_DESCRIPTOR_VEC	Pointer to descriptor Buffer
GET_CONFIG_DESCRIPTOR	SUSBx_CONFIGURATION_DESCRIPTOR_VEC	Pointer to descriptor Buffer
GET_STRING_DESCRIPTOR	SUSBx_STRING_DESCRIPTOR_VEC	Pointer to descriptor Buffer

Host PC USB Requests	OTG-Host Interrupt Vectors SUSBx (x = 1 →SIE1, 2→SIE2)	Interrupt Service Routine – C handler /Pointers (susbx x = 1→SIE1 2→SIE2)
SET_CONFIGURATION/ SET_INTERFACE	SUSBx_STANDARD_INT + SUSBx_DELTA_CONFIG_INT	susbx_standard_handler()+ susbx_delta_config_handler()
VENDOR REQUESTS	SUSBx_VENDOR_INT	susbx_vendor_handler()
CLASS REQUESTS	SUSBx_CLASS_INT	susbx_class_handler()
STATUS PHASE(ACK,NAK etc)	SUSBx_FINISH_INT	Internal BIOS handle
ERROR CONDITIONS(STALL)	SUSBx_STALL_INT,	Internal BIOS handle

5.2 USB Multiple Interface Initialization and Enumeration

In the previous sections the concepts of composite device are explained. To implement a composite device which is a combination of multiple configuration and interfaces a (Audio + HID) peripheral device is used as an example reference in this note. This example source code is attached along with this note.

Note: It should be noted that this example is only a minimal implementation of these classes. It is not tested with a real audio and HID hardware. This example should serve as baseline code to further develop into a complete end design.

5.2.1 Initialization

As a prerequisite to multiple interface peripheral enumeration, a set of descriptors should be defined. Listing 2 shows an outline of all the descriptors necessary to implement a composite device with four interfaces.

- Interface 0 is an audio control interface,
- Interface 1 is an audio output device (with respect to the host),
- Interface 2 is an audio input device (with respect to the host)
- Interface 3 is a HID interface.

When enumerated, this device will appear to the host as a composite device, comprised of a USB audio device (with a line input and audio output) and a HID device.

Listing 2: Outline of Complete Descriptors for Four Interface Audio / HID Composite Device

```
Device Descriptor (device descriptor)
Configuration Descriptor (master
configuration descriptor, all-interfaces)
Configuration Descriptor
Interface 0 (audio control interface – no
endpoints)
Interface Descriptor
Class Specific Descriptor(s)
Interface 1 (audio output from host – EP2)
Interface Alt 0 Descriptor (no endpoint,
zero bandwidth)
Interface Alt 1 Descriptor
Class Specific Descriptor(s)
Endpoints Descriptor(s)
Interface Alt 2 Descriptor
Class Specific Descriptor(s)
Endpoints Descriptor(s)
. . .
Interface 2 (audio input to host – EP4)
Interface Alt 0 Descriptor (no endpoint,
zero bandwidth)
Interface Alt 1 Descriptor
```

```
Class Specific Descriptor(s)
Endpoints Descriptor(s)
Interface Alt 2 Descriptor
Class Specific Descriptor(s)
Endpoints Descriptor(s)
. . .
Interface 3 (HID IN interface – EP5)
Interface Descriptor
Class Specific Descriptor(s)
Endpoints Descriptor(s)
```

The SIE1 (Serial Interface Engine) of OTG-Host is used as composite USB peripheral in this example. Listing 2 shows the SIE1 initialization (defined in sie1.c of audiodem *(Audio + HID)* firmware folder attached to the note).

Listing 3: SIE1 Initialization-USB Peripheral Mode

```
void sie1_init(void)
   {
       ………………….
       …………………
/* Set device descriptor pointer to point at our device descriptor. */
   WRITE_REGISTER( SUSB1_DEV_DESC_VEC, (uint16)
                                                 &device_descriptor );

 /* Set config descriptor pointer. */

 WRITE_REGISTER( SUSB1_CONFIG_DESC_VEC,
(uint16) &sie1_descriptors.config_descriptor );

  /* Set string descriptor pointer.  */
   WRITE_REGISTER( SUSB1_STRING_DESC_VEC,
(uint16) &sie1_string_descriptor );
    ……….
}
```

Upon initialization, the above device descriptor and master configuration descriptors are installed in the data vector table (SUSBx_DEV_DESC_VEC and SUSBx_CONFIG_DESC_VEC) respectively.

5.2.2 Enumeration

The enumeration process of OTG-Host as a composite peripheral device is not a straightforward sequence. In the last section we showed how OTG-Host Composite peripheral-(Audio + HID) descriptors i.e. Device, Configuration and String descriptors are mapped to the corresponding hardware interrupt vectors. When the Host PC sends the USB Standard requests the OTG-Host BIOS parses these descriptors and uses the data obtained to automatically configure the endpoints.

To complete the remaining part of enumeration process for a composite device(Audio + HID) on OTG-Host the following factors needs to be explained.

- OTG-Host BIOS and Frameworks Limitation
- Implementation in Firmware to support a Composite device.

OTG-Host BIOS and Frameworks Limitation: The OTG-Host BIOS and frameworks do not inherently support multiple interfaces or configurations. Additional firmware code must be written in order to accomplish this. Following is the rule statement based on which the additional code is designed to support a composite device in the firmware.

"To enable multiple interface and configuration support on the OTG-Host, a configuration descriptor with all interfaces and endpoints must be created as in the single interface case. This full descriptor will be sent to the host during enumeration to allow the host to learn the capabilities of the device. In addition, another descriptor must be created for the BIOS to properly configure the endpoints. This descriptor must contain a single configuration descriptor, a single interface descriptor, and an endpoint descriptor for each endpoint in the device. The configuration and interface descriptors are mostly ignored by the BIOS parsing, but the endpoint descriptors must be set up with the desired endpoint parameters."

Implementation in Firmware to support a Composite device: After Host receives Device, Configuration and String descriptor it will send SET_CONFIGURATION and SET_INTERFACE requests. We will trap these requests and load a new configuration descriptor that describes the device as specified in the SET_CONFIGURATION and SET_INTERFACE requests. The BIOS will parse this new descriptor and set up the endpoints accordingly. This new configuration descriptor, which we'll call the BIOS endpoint descriptor, can be simplified to the structure shown in Listing 4 for our example (Audio + HID) with five endpoints.

Listing 4: Outline of BIOS Endpoint Descriptor

```
    Dummy Configuration Descriptor
    Dummy Interface Descriptor
    Endpoint 1 descriptor
    Endpoint 2 descriptor
    Endpoint 3 descriptor
    Endpoint 4 descriptor
    Endpoint 5 descriptor
```

The new configuration descriptor called the BIOS endpoint descriptor is declared in the audiodem firmware source as shown below Listing 5.

Listing 5: BIOS Endpoint declaration in (Audio + HID) Example

```
#define NUM_ENDPOINTS 5
typedef struct
{
USB_CONFIG_DESCRIPTORcfg_descATTR_PACKED;
USB_INTERFACE_DESCRIPTORif_descATTR_PACKED;
USB_ENDPOINT_DESCRIPTORep_desc[NUM_ENDPOINTS] ATTR_PACKED;
} BIOS_ENDPOINT_DESCRIPTORS;
BIOS_ENDPOINT_DESCRIPTORS bios_ep_desc ATTR_SIE1_DESCR_SECTION;
```

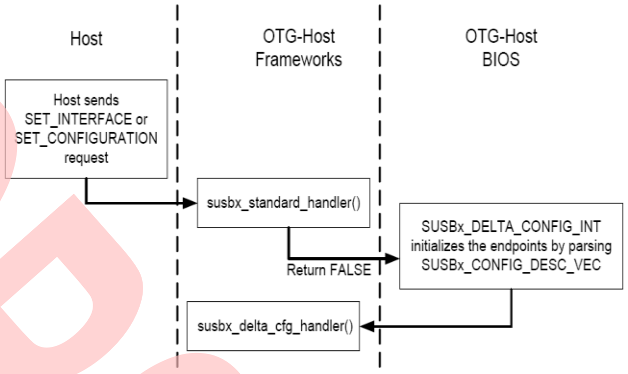
The configuration and interface descriptors have to be there, but the content of these descriptors in our BIOS endpoint descriptor doesn't really matter. It's only important that they be real descriptors (length and type), so for simplicity's sake, we'll just use configuration and interface descriptors that are already defined. We'll only need to change one field (*bNumInterfaces*) to keep the BIOS blissfully ignorant that we are implementing multiple interfaces. A dummy endpoint descriptor data structure is used as a placeholder to fill all the endpoint slots in the BIOS endpoint descriptor until we put some useful information there. Note that we also need to write the "real" descriptors into the vectors during sie1_init() in order to initialize properly. Listing-6 shows how Bios endpoint descriptor fields are populated during initialization phase of SIE1.

Listing 6: BIOS Endpoint Descriptor Initialization

```
void sie1_init()
{
Int i;
WRITE_REGISTER( SUSB1_DEV_DESC_VEC, &device_descriptor );

WRITE_REGISTER( SUSB1_CONFIG_DESC_VEC, &master_config_descriptor );
bios_ep_desc.cfg_desc = sie1_descriptors.config_descriptor;

bios_ep_desc.cfg_desc.bNumInterfaces = 1; /* Don't let the BIOS think we're doing*/
/* multiple interfaces */

bios_ep_desc.if_desc = sie1_descriptors.interface_0;
for (i=0;i<NUM_ENDPOINTS;i++)
{
/* dummy_ep is just an endpoint descriptor with everything equal to zero except */
/* the length and type */
bios_ep_desc.ep_desc[i] = dummy_ep;
…………………………….
……………………….. .
}
```

This BIOS endpoint descriptor needs to be filled with proper values when host PC send SET_CONFIGURATION and SET_INTERFACE requests. Figure shows how OTG-Host responds to these two requests during enumeration.

Figure 6. OTG-Host Response to SET_CONFIGURATION and SET_INTERFACE Requests


Host
OTG-Host Frameworks
OTG-Host BIOS
Host sends SET_INTERFACE or SET_CONFIGURATION request
susbx_standard_handler()
Return FALSE
SUSBx_DELTA_CONFIG_INT initializes the endpoints by parsing SUSBx_CONFIG_DESC_VEC
susbx_delta_cfg_handler()

As can be seen from the above figure when Host sends either of these requests the susbx_standard_handler() ISR is invoked. This is also shown previously in Table 3.Inside this handler routine we can parse the SET_UP packet fields to find whether the received request is SET_CONFIG or SET_INTERFACE.

SET_CONFIG : The SET_INTERFACE request will not be called for the interfaces that don't have multiple alternates. Therefore, the endpoints for these interfaces will have to be set up somewhere else. An appropriate place for this is in the SET_CONFIGURATION request in *susb1_standard_handler()*. Every device has at least one configuration, which is selected via a SET_CONFIGURATION request during enumeration. Since SET_CONFIGURATION is one of the requests that results in the BIOS parsing descriptors, we can set up our endpoint descriptor here the same way as in the SET_INTERFACE request. In our example, we have to configure interface 3 using this method, since it is the HID interface and does not have any alternates. Listing-7 shows the code in the SET_CONFIGURATION request used to enable this HID endpoint.

This code is found in the *susb1_standard_handler()* function of SIE1.C.

Listing 7: SET_CONFIGURATION Request in susb1_standard_handler()

```
if (pReq->wIndex == 1)
{
/* If interface 1 is called for, copy the endpoint descriptors for the appropriate
alternate */
if (pReq->wValue == 0)
{
/* Alternate 0 has no endpoint, so let's use the dummy ep for EP2 */
bios_ep_desc.ep_desc[EP2-1] = dummy_ep;
}
else
{
/* Copy from sie1_descriptors */
bep_desc.ep_desc[EP2-1].bEndpointAddress =
sie1_descriptors.interface_1[pReq->wValue 1].endpoint_descr.bEndpointAddress;
bios_ep_desc.ep_desc[EP2-1].bmAttributes =
sie1_descriptors.interface_1[pReq->wValue -1].endpoint_descr.bmAttributes;
. . .
bios_ep_desc.ep_desc[EP2-1].wMaxPacketSize = sie1_descriptors.interface_1[pReq->wValue
-1].endpoint_descr.wMaxPacketSize;
}
}
/* Keep the BIOS happy by making it believe there's only one interface and alternate
*/
pReq->wValue = 0;
pReq->wIndex = 0;
WRITE_REGISTER(SUSB1_CONFIG_DESC_VEC, (uint16) &bios_ep_desc);
return FALSE;
```

```
Excerpt of SET_INTERFACE request in susb1_standard_handler()
/* Load EP5 endpoint descriptor from main descriptor */
bios_ep_desc.ep_desc[EP5-1] = sie1_descriptors.hid_endpoint;
WRITE_REGISTER(SUSB1_CONFIG_DESC_VEC, (uint16) &bios_ep_desc);
return FALSE;
```

SET_INTERFACE: For each interface with multiple alternates, the host will send a SET_INTERFACE request to select the initial setting for the interface, and again whenever the interface needs to be changed. The interface will normally be changed frequently during operation with isochronous endpoints such as in Audio + HID firmware example attached along with the note, as the host attempts to add or release bandwidth as needed.

A case in susb1_standard_handler() in SIE1.C can trap this request and change the BIOS endpoint descriptor to the settings for the requested alternate interface. Listing-7 shows how this code looks specifically for interface 1, with the code at the end common to all SET_INTERFACE requests. We return FALSE to let the BIOS handle the command, since the BIOS handler will be responsible for parsing the descriptors.

- *pReq->wIndex* holds the interface number currently being changed.
- *pReq->wValue* holds the requested alternate number for the specified interface.

Note that we set *wValue* and *wIndex* to zero as shown in Listing-7 before returning control to the BIOS. This is needed because the BIOS will refuse to handle requests with multiple configurations or interfaces, knowing that it cannot handle them. We need to trick the BIOS into thinking this is a single configuration or interface, and that it's okay to handle it normally.

The process we use to initialize our HID interface would be the same process we would use to support multiple configurations. *pReq->wValue* contains the configuration value requested by the host, so this would be used to determine how to set up your endpoints per the host's request. Note that if you are implementing multiple configurations, you will need to set pReq->wValue to 1 before returning false in order to make the BIOS believe there is only one configuration.

We also need to reload the master configuration descriptor in the reset ISR so that our SIE initializes properly. This simple step is shown in Listing-8. After any bus reset occurs,we will enumerate again, going through all the previous steps to set up our endpoints with SET_INTERFACE and SET_CONFIGURATION.

Listing 8: Reloading the Master Configuration Descriptor in the Reset ISR found in SIE1.c

```
void sie1_slave_reset_isr(void)
{
    WRITE_REGISTER( SUSB1_CONFIG_DESC_VEC,
    &master_config_descriptor );
}
```

As shown in Figure 6, the BIOS *SUSBx_DELTA_CONFIG_INT* function is called whenever the BIOS is asked to handle a *SET_CONFIGURATION* or SET_INTERFACE (as well as after a *USB_RESET* and *SUSBx_INIT_INT*). In this interrupt function, the BIOS parses through the descriptor pointed to by SUSB1_CONFIG_DESC_VEC looking for endpoint descriptors.

It then automatically sets up the appropriate endpoint control register, enabling the endpoint, setting the isochronous bit if necessary, and configuring whether the endpoint is an IN or an OUT endpoint.

The BIOS delta config interrupt also resets both the transfer arming bit in the endpoint control register and the contents of the endpoint address register (endpoint buffer pointer). Since the frameworks *susbx_delta_cfg_handler()* is called after the BIOS delta config interrupt is finished, any recurring transfers need to be armed in the frameworks *susbx_delta_cfg_handler()* with a call to *susb_send()* (IN transfers)or *susb_receive()*(OUT transfers).

6 Testing the Composite Device (Audio + HID) Example

The firmware example takes care of everything the host needs to see in order to enumerate properly with the audio and HID drivers. After enumeration, the ISO IN and OUT are armed into buffers of length 192. The HID IN endpoint is enabled but not armed, therefore it will NAK all requests by default. In order to send HID data, a frame should be prepared and *susb_send()* should be called on EP5.

The firmware file that is associated to this application note includes two separate directories. The first directory, called *audiodem*, is the main application directory that includes the application specific C source files, the *makefile* and *fwxcfg.h* configuration file. The second directory is called common and is the main CY3663 frameworks directory. Place these files in a location that is convenient to navigate to in the BASH window. The image supplied with the project (*audiodem_scan.bin*) is scanwrapped and ready to program to an EEPROM using QTUI2C.

To re-compile the image for EEPROM, build with "[cy]$ *make wrap".* To re-compile for use with the GNU Debugger (GDB), the project will have to be rebuilt with "[cy]$ *make DEBUG=1*". Compiling the project with the option of clean ([cy]$ *make clean)* may be required if the message "*Nothing to be done for 'all'* " is displayed.

The example design will run on the EZ-OTG or EZ-HOST daughter boards from the CY3663 development kit. This design can also be implemented on the CY4640 mass storage reference design board with the population of a USB peripheral jack onto SIE1 (J22). Included in this application note collateral is a utility called qtuload. Qtuload can be used to load a scanwrapped image directly into RAM and is much faster than writing to the EEPROM. This makes it very convenient for experimentation.

The usage of qtuload is:

[cy]$ *qtuload fname_scan.bin base_addr call_addr*

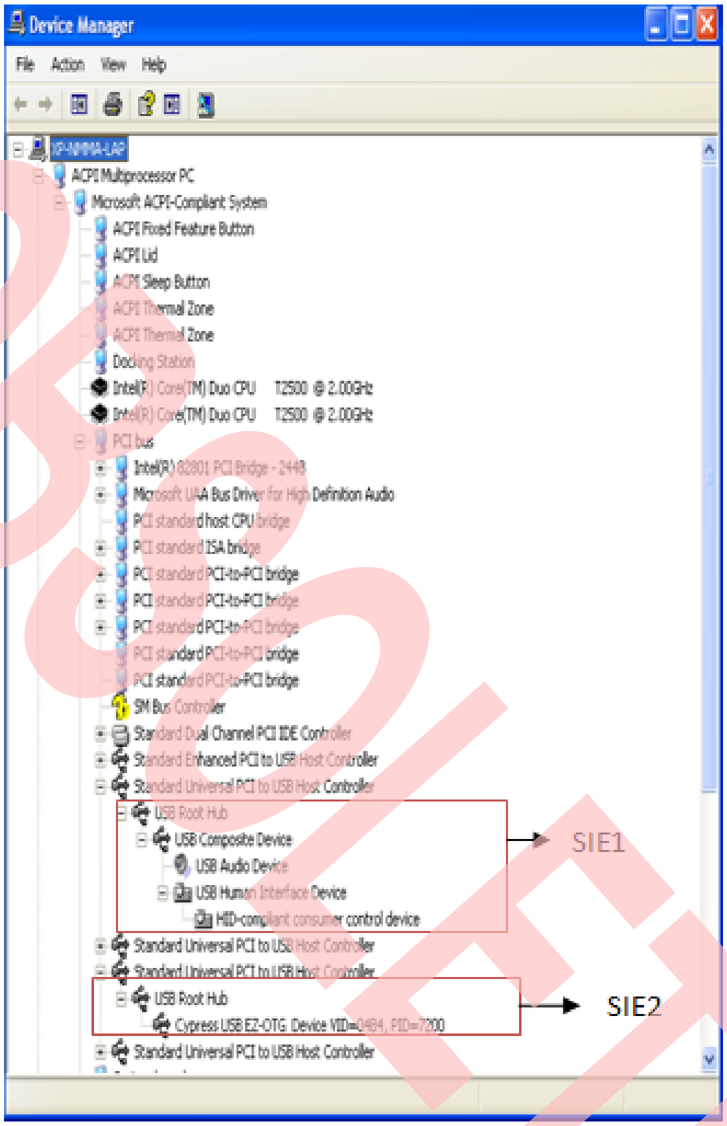
In order to load the image using this utility:

1. Power up your EZ-Host board with all of the switches on SW1 in the off position.
2. From a BASH window, navigate using the cd command to the directory where audiodem is placed.
3. At the BASH prompt type in "qtuload audiodem_scan.bin 0x1000 0x1000". In our example code the base_addr is defined in the *audiodem.ld* (linker) file and is 0x1000. This is also the call_addr. You should see the driver name and device displayed indicating that the driver was found followed by the statement informing you that the *audiodem_scan.bin* file is being loaded along with its size.

7 Evaluating the Design on a PC

The Device Manager with the design running and both SIEs connected to the host appears as shown in Figure 7. You can see the USB audio device and the HID device listed as a USB Composite device. This verifies that the composite device peripheral (SIE1) has enumerated properly. SIE2 has enumerated as the EZ-OTG debug peripheral, which is used for development.

Figure 7. OTG-Host SIE1 and SIE2 Device List



```
XP-NIMMA-LAP
  ACPI Multiprocessor PC
    Microsoft ACPI-Compliant System
      ACPI Fixed Feature Button
      ACPI Lid
      ACPI Sleep Button
      ACPI Thermal Zone
      ACPI Thermal Zone
      Docking Station
      Intel(R) Core(TM) Duo CPU     T2500  @ 2.00GHz
      Intel(R) Core(TM) Duo CPU     T2500  @ 2.00GHz
      PCI bus
        Intel(R) 82801 PCI Bridge - 2448
        Microsoft UAA Bus Driver for High Definition Audio
        PCI standard host CPU bridge
        PCI standard ISA bridge
        PCI standard PCI-to-PCI bridge
        PCI standard PCI-to-PCI bridge
        PCI standard PCI-to-PCI bridge
        PCI standard PCI-to-PCI bridge
        PCI standard PCI-to-PCI bridge
        SM Bus Controller
        Standard Dual Channel PCI IDE Controller
        Standard Enhanced PCI to USB Host Controller
        Standard Universal PCI to USB Host Controller
          USB Root Hub
            USB Composite Device
              USB Audio Device
              USB Human Interface Device
                HID-compliant consumer control device
        Standard Universal PCI to USB Host Controller
        Standard Universal PCI to USB Host Controller
          USB Root Hub
            Cypress USB EZ-OTG Device VID=04B4, PID=7200
        Standard Universal PCI to USB Host Controller
```

7.1 Audio OUT transfer test

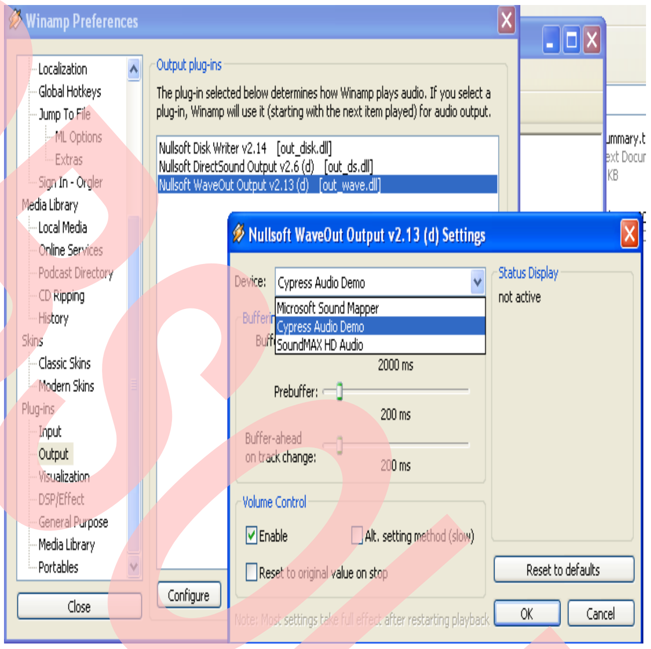
The OUT transfers can be tested with any audio output program (such as Winamp or Windows Media Player) by configuring the USB audio device as your output. In Winamp (shown in Figure 8), this is done by selecting “Preferences” from the “Options” menu. Select the “Output Plugin”, click on the “WaveOut output”, and then click the “Configure” button.

Select “Cypress Audio Demo” from the list of audio output devices listed in the drop-down menu.Once you select the USB audio device (in this case “Cypress Audio Demo”) as your output, the audio data streaming into the OTG-Host can be seen using the QTUDUMP debug utility and viewing the memory location for *sie1_isoch_buffer_fromhost*. Look at the LST file to find the memory address for this buffer. When the audio is running, the contents of the buffer will constantly change as new audio data arrives. If the audio pauses, you will see the contents in the buffer stop updating.

7.2 Audio IN Transfer Test

The IN transfers can be tested using the Sound Recorder program built into Microsoft Windows. Select “Audio Properties” from the Edit menu of the program and choose “Cypress Audio Demo” as the default device for sound recording. The example firmware contains some preloaded data that will produce a tone/buzz when Sound Recorder records from the USB input

Figure 8. Configuring Winamp Audio Output



Winamp Preferences

Output plug-ins

The plug-in selected below determines how Winamp plays audio. If you select a plug-in, Winamp will use it (starting with the next item played) for audio output.

Nullsoft Disk Writer v2.14 [out_disk.dll]
Nullsoft DirectSound Output v2.6 (d) [out_ds.dll]
Nullsoft WaveOut Output v2.13 (d) [out_wave.dll]

Nullsoft WaveOut Output v2.13 (d) Settings

Device: Cypress Audio Demo

Microsoft Sound Mapper
Cypress Audio Demo
SoundMAX HD Audio

2000 ms

Prebuffer: 200 ms

Buffer-ahead on track change: 200 ms

Status Display: not active

Volume Control: [✓ Enable] Alt. setting method (slow); Reset to original value on stop

Note: Most settings take full effect after restarting playback

8 Application Tips

Since the OTG-Host lacks analog interfaces and DSP functionality, a design involving isochronous audio endpoints will usually be implemented in a main processor (perhaps a DSP) with the OTG-Host acting as a coprocessor. Fortunately, the OTG-Host offers several coprocessor interfaces (SPI, HPI, and HSS) to make this easy. GPIO31 and GPIO30 must be set by external circuitry for the desired coprocessor interface, which will force the OTG-Host to boot into its BIOS and look for commands on the selected interface. The master processor can then download the code image into the OTG-Host and run it. This will run the firmware on the OTG-Host, but will also allow the external processor direct memory access (DMA) into the OTG-Host to read, write, execute interrupts, etc. Software on the master processor works with software and BIOS functions on the OTG-Host to make the design work. De2_app.c (in the “source/coprocessor/de_app” directory of CY3663) contains some routines that can be implemented on the master processor to download the code image to the OTG-Host, as well as to read or write to OTG-Host memory over any of the coprocessor interfaces.

9 Summary

The EZ-OTG and EZ-Host devices are versatile devices that can be used as a USB host or peripheral (or both at once). With the techniques and the example presented here, multiple configurations, multiple interfaces, and multiple interface alternates can be easily implemented in the OTG-Host. The availability of simple coprocessor modes allows the device to be used seamlessly with other processors in a design. This combination of features allows the device to be used in many useful designs, such as to provide an easy way to get isochronous audio data back and forth between a digital signal processor and a PC.

Document History

Document Title: AN5083 - Implementation of Multiple Interface Isochronous USB Composite Peripheral using EZ-HOST/EZ-OTG™

Document Number: 001-67825

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3183867	NMMA	03/03/2011	New Spec.
*A	4316558	MDDD	03/21/2014	Updated in new template. Completing Sunset Review.
*B	5693377	MDDD	04/12/2017	Updated template

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709