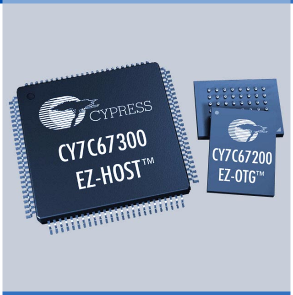
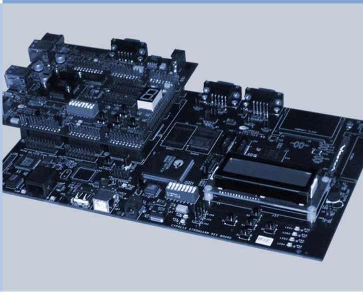
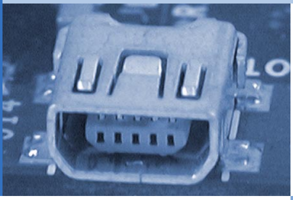


USB Multi-Role Device Design By Example





John Hyde

USB Multi-Role Device Design By Example

John Hyde

(Commissioned by Cypress Semiconductor)

USB Multi-Role Device Design By Example

John Hyde

Foreword by Brian Booker

We were pleased to get John Hyde to write this book on Cypress Semiconductor's behalf. It should be considered a supplement to his "USB Design By Example" in the same way that the OTG Supplement complements the USB 2.0 Specification. Only the new elements of multi-role device design are covered since these build on the established base defined by the USB specification.

Cypress Semiconductor supports all facets of USB design and has products covering the breadth and depth of possible USB solutions. The two products described in this book, the CY7C67200 EZ-OTG Host/Peripheral Controller and the CY7C67300 EZ-Host Host/Peripheral Controller will enable new designs particularly in the portable applications arena. This book, with the support of other Cypress documentation, should get your USB product idea rapidly into development and then into the prototype stages. We at Cypress Semiconductor are dedicated to supporting your design efforts through production and 24 hour support is available at www.cypress.com.

Acknowledgements:

I would like to thank Cypress Semiconductor for providing me with an opportunity to write about these two remarkable components. The EZ-Host and EZ-OTG include features that enabled me to solve a much wider range of customer projects. I particularly like the small size of the EZ-OTG since it enables USB to be used in many more solutions.

I could not have completed this book without the expert help of Cypress's Systems Engineering Team in Boise. I must particularly thank Rick Pennington, DeVerl Stokes, Ray Asbury, James Cahoon and Steven Connelly for their time, teaching and impressive development platform including three boards, Frameworks code and several design examples. Additionally, Simon Nguyen provided a wealth of information about the internal operation of the components, and his in-depth answers to all of my questions enabled better descriptions in the text. Several of Cypress's applications engineers, especially Glenn Roberts, Matt Leptich and Troy Gentry, provided valuable feedback during the review process and this also helped improve the quality and accuracy of the book.

I had a lot of fun writing this book and I trust that you will enjoy reading it. You will also have fun using these two new components from Cypress Semiconductor to create new and exciting USB solutions.

Table of Contents

List of Figures

Chapter 1: Expanding the USB Applications Range

USB has come a long way since its inception as a desktop PC expansion bus in 1995. The USB Specification defines a master-slave communications system and details two distinct roles – a host that is in control of all communications and a function that provides services to the host. Initial implementations partitioned these roles into individual silicon components. The recent “On-The-Go” (OTG) Supplement extends the original usage model for USB by adding the capability to build a dual-role device. Cypress Semiconductor has gone one step further with their EZ-Host and EZ-OTG components that integrate up to four hosts and two functions into a single component. These multi-role devices are fully programmable and enable a wider range of USB-solutions to be developed.

It is time to revisit the original USB specification, the additions made with USB 2.0 and the OTG Supplement, to fully understand the requirements of a “host” and a “function”. This introduction distills this information into the essential elements that you need to understand in order to make rapid progress on multi-role device designs. Even if you consider yourself as “USB-savvy” I would recommend reading this introduction since it contains key definitions that will make the remainder of the book easier to follow.

Original USB Design Intent

USB was originally conceived as a desktop PC expansion bus. The external IO connectivity of the mid-1990’s PC was based on serial ports and parallel ports – these interfaces used different connectors (which were physically large) and only offered point-to-point connections to peripherals. Various schemes were proposed so that devices could be daisy-chained on parallel ports but these did not receive wide adoption. There were also new peripherals, such as the telephone, that needed to be attached to the PC and needed additional capabilities that the serial and parallel ports could not provide.

USB was introduced in 1996 as a single-connector, protocol-based, serial bus to address the requirements of PC peripheral expansion. It supported two speeds, low at 1.5Mb/s and full at 12Mb/sec, suitable for many desktop PC expansion peripherals. Capabilities such as hot-insertion, support of isochronous (time-dependant information such as digital audio) data and a unified operating system driver model were also included. Also, for the first time, power was defined at the connector so that a peripheral device could officially consume current from the PC’s power supply. Figure 1-1 shows the major elements defined by the first USB specification.

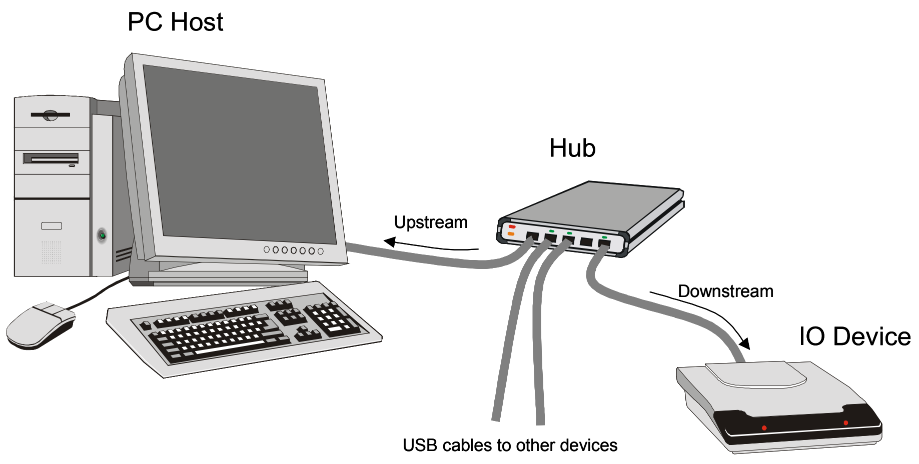


Figure 1-1. Standard USB terminology

It is important to emphasize that USB is a Master/Slave bus; it is defined to have one master and many slaves. The motivation behind this decision was lower system cost; this decision puts all of the complexity in the PC host since this is only implemented once. This enables the devices to be simple, therefore low cost. This implementation defines the two roles that exist in a USB environment: the *host* role that controls the communications and the *function* role that provides services to the host. A function may be a hub that is responsible for propagating USB signaling and power, or a device that is the target for USB data communications. In this context, a multi-role device can operate as one or more hosts and one or more devices — either independently or simultaneously. A multi-role device will typically not have the signal propagation properties of a hub, but later chapters will show that this crisp distinction is also becoming hazy.

In April 2000 a third speed of 480 Mb/sec was added in the USB 2.0 specification. The "PC expansion bus" usage model did not change, and the higher speed enabled high bandwidth peripherals such as mass storage devices and high frame rate video cameras.

The "OTG Supplement" to the USB 2.0 specification was approved in December 2001, and it extended the "PC expansion bus" usage model to essentially allow "peripheral devices to talk to each other without a PC." The USB protocol requires a host to communicate to a device; therefore one of the peripherals must assume the role of a host in order to enable this communication. Until this supplement was approved, a host had always been a PC – with this premise come many assumptions that need to be understood if we want a battery-powered, non-programmable, embedded system such as a digital still camera to become a USB host.

So lets look deep inside the USB specification to understand the responsibilities of a USB host. Our goal is to discover the essential requirements of a "minimal-host" – something that can effectively communicate and exchange data with a standard USB peripheral.

Host Role Responsibilities

The most important task for a USB host (and, for that matter, a USB device) is to manage and keep system power to a minimum. Data communication is a secondary task, and both must be done with the over-riding philosophy being ease-of-use. Figure 1-2 shows a simplified diagram of a USB device about to be connected to a USB host. The diagram shows a low/full speed connection – a high-speed connection also starts this way but the biasing resistors are later disabled during the speed negotiation phase.

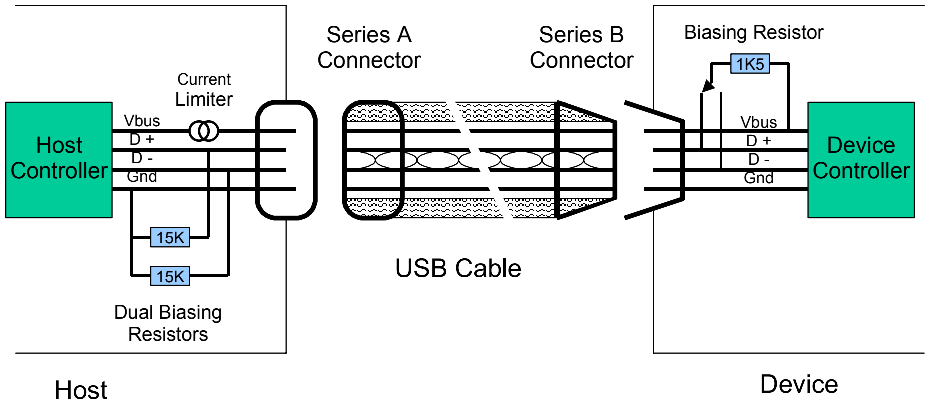


Figure 1-2. Basic host to device connection

A host is required to make 100mA available for a device during attachment to the bus. The device is required to signal its presence to the host, by attaching it's biasing resistor to D+ or D-, within 100msec of first consuming power from the bus. Note that if the device does not signal within 100msec then the host is not obligated to continue to supply power, and it may limit or disable current flow. Currently no PC hosts enforce this specification rule so using USB as 5volt, 100mA power supply has spawned products such as the USB desklamp and USB fan as shown in Figure 1-3. One day these non-compliant, USB products will only work for 100 msec…

Figure 1-3. Products that misuse USB as a power supply

Devices use up to 100mA during enumeration and may then request up to 500mA for operation. Providing 0.5 watts or 2.5 watts of power is not a problem for a PC host that is plugged into a main power source, but these values are large for a laptop PC and VERY large for a camera that may want to be a host. Lets study what is REALLY required.

Look again at Figure 1-2. The device is not required to consume up to 100mA from the host. If the device had its own power source (in USB specification terms, it is self-powered), either its own main power connection or batteries, then it could enumerate without consuming any power from the host. The device would use the presence of Vbus to know that it was attached to a host and would therefore signal its presence by attaching its biasing resistor.

The idea that a battery-powered host can enumerate a battery-powered device is looking promising. A host does not need to power the other device. Since the target application range of multi-role devices is portable equipment, then it looks as if we have the makings of a solution. More engineering is required to ensure reliable operation and interoperability with all USB devices. The details of Vbus detection and sequencing are discussed later in this chapter.

The host is also responsible for initiating all communications. USB uses pre-formatted packets to exchange data between a host and a device. The host is required to broadcast Start-Of-Frame (SOF) packets every millisecond (accurate to 500ppm). The host then sends addressed packets to targeted devices to read and write data. For a detailed description of this process refer to my “USB Design By

Example". The data transfer operation of a dual-role device follows the USB specification exactly.

The host is responsible for detecting that a new device has been attached to USB and, as part of the enumeration process, it must assign a unique address to this new device. The host then reads the device descriptors to discover the identity of the newly attached device, and it uses this information to assign and activate a device driver. The device driver completes the enumeration by enabling the new device with a Set_Configuration command.

A PC host will have a vast collection of device drivers available to support whatever USB device is attached. An embedded host is likely to support only a few devices - typically only a few devices are relevant to a particular embedded host. The OTG Supplement calls this a Targeted Peripheral List and enables an embedded host to support a known collection of devices without needing a hard drive full of device drivers.

So, the USB specification does not require that you use a PC as a host. We can define a "limited capability" host that has features suitable for an embedded system. The next section looks at the dual-role device as defined by the OTG Supplement and discusses the features added on top of the base requirements.

Ease of Use

Any extension or supplement to an existing standard must be done in a compatible and user-friendly way such that the inherent goodness of the standard is not compromised in any way. There were many opportunities for the OTG Supplement to diverge from the base USB Specification, so I must applaud the specification engineers for pre-solving every detail and enhancing the overall goodness of USB with OTG.

Much of the "goodness" of USB stems from its ease-of-use. The user's view of USB is effortless plug-and-play. But ease-of-use is a double-edged sword – it requires MORE effort and diligence on the product designer's part. We must design-in ease-of-use, and we shall see that the OTG Supplement requires it.

The system setup shown in Figure 1-4 is used as a vehicle to describe the details of a dual-role device. A digital still camera and a printer are connected via two USB ports to a PC. In Figure 1-5 the PC has been removed and the camera is connected directly to the printer. How is the design of a dual-role camera different from a device-only camera?

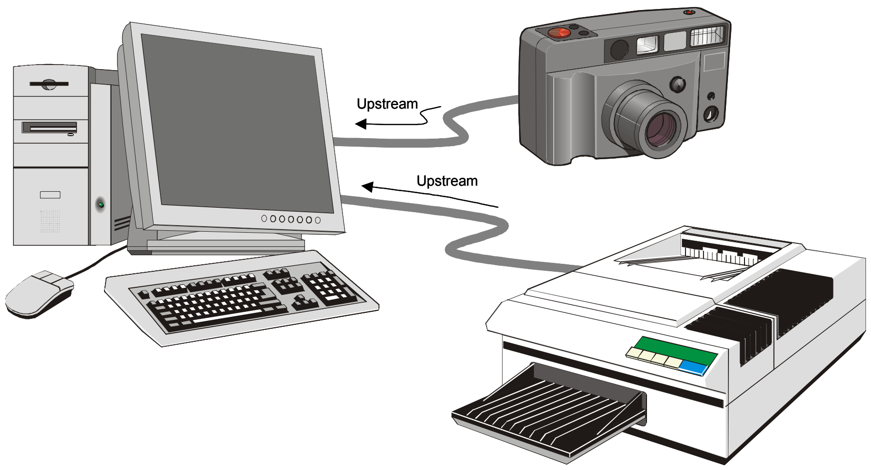


Figure 1-4. PC host with two devices

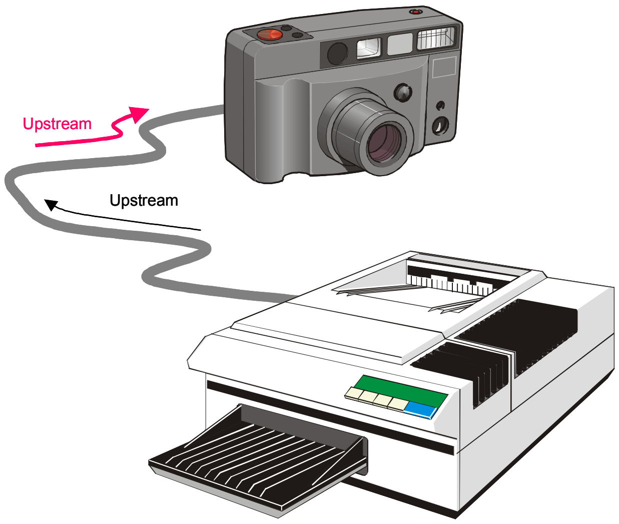


Figure 1-5. Embedded host and one device

Dual Role Device Implementation

First and foremost, a dual-role device is required to operate as a standard USB device. In this device role, the camera is attached via a USB cable to the PC; this cable has its upstream, or A, connector plugged into the PC host and its downstream, or B, connector plugged into the camera. A camera will likely use a

Mini-B connector rather than a standard B connector due to its much smaller size. The Mini-B connector was added to the USB specification in October 2000. Let us assume, for the moment, that the printer uses a standard B connector – we will consider the case where it uses a mini connector later. A dual-role device will use a Mini-AB connector as shown in Figure 1-6.

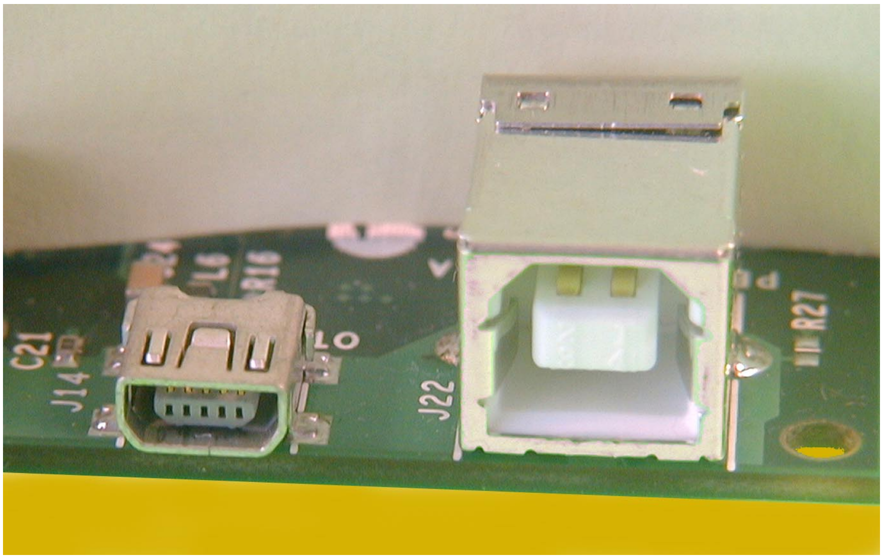


Figure 1-6. Mini-AB and standard-B connectors.

There is an extra pin in the Mini-AB connector, called ID, but there is not an extra wire in the cable. A Mini-AB connector can accept a Mini-A plug or a Mini-B plug shown in Figure 1-7. The ID pin of a Mini-A plug is connected to ground while the ID pin of a Mini-B plug is left floating. A dual-role device will implement a pull-up resistor on the ID pin so that it can detect the voltage level and therefore determine if a Mini-A or a Mini-B plug is installed.

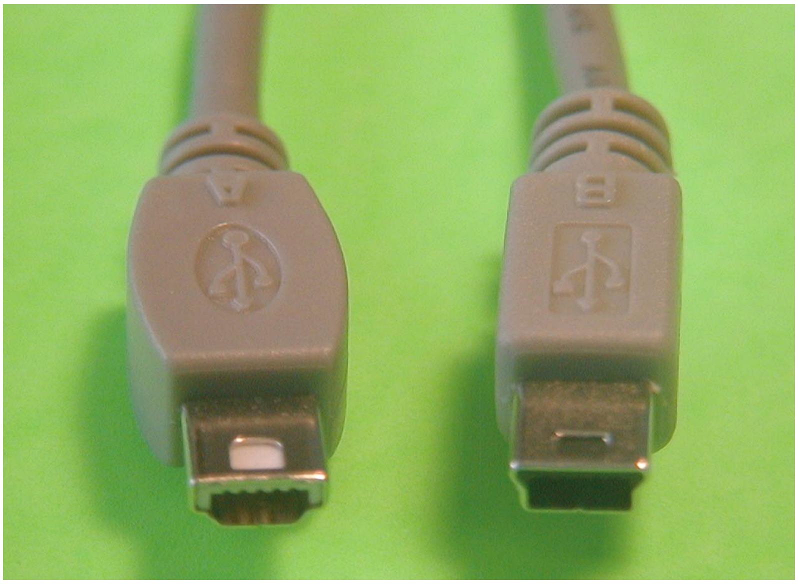


Figure 1-7. Mini-A plug and Mini-B plug insert into a Mini-AB connector

Removing the PC from Figure 1-4 left two A-connectors exposed. To connect the camera to the printer, as shown in Figure 1-5, you would use a Standard A-to-Mini-A adaptor on the printer cable or replace both cables with a Standard-B-to-Mini-A cable. In either case, a Mini-A connector plugged into the camera alerts it to transform itself from a role as a device into a role as a host.

During the cable swap the printer will have detected loss of Vbus and has therefore moved into an “unattached” state and is waiting for “the host to turn on again”, i.e. waiting for a valid voltage on Vbus. Let us also assume that the printer is a self-powered device (I haven’t come across one that isn’t) and therefore will not require any significant power from Vbus.

Transforming into a host

The act of changing the cables interconnecting the peripherals alerted the camera of its role change from device to host. Since the camera has the A-connector it is also called the A-device. The device with a B-connector is called the B-device. Realizing that many dual-role devices would be battery powered, the camera does not assume its host role immediately – some user action is required to start the process. Why have the connection powered up and actively consuming precious battery power if no communication is taking place?

When the camera is ready for action it drives a valid voltage on Vbus, and this is recognized by the B-device (printer). The camera enumerates the printer to ensure it can support it, transfers the picture that needs to be printed and then suspends the bus and removes Vbus.

But what if the B-device wanted to request some service from the host – how can it signal the A-device/host with no Vbus? The OTG Supplement authors added the concept of a session. When the connection is not being used it is in a power saving or dormant state. A Session Request Protocol (SRP) is defined in a way that allows the B-device to request the A-device/host to turn on Vbus to initiate a session. The OTG supplement defines two methods that the B-device can use to signal the A-device; one pulses a data line while the other pulses the Vbus line. An A-device must be designed to accept at least one of these methods and B-devices should be designed to use both methods. An example of the SRP is detailed in Chapter 6.

A host is required to be able to supply 8mA. A battery-powered host can supply 8mA far more easily than supplying 100mA. You may design a dual-role device that supports greater than the minimum of 8mA as required by the specification – this will allow the dual-role device to support low-power, bus powered devices such as some mice and keyboards.

Well, the easy case of a single dual-role device and a USB peripheral was pretty straight forward. Figure 1-8 shows a more complex situation where the printer has been replaced by another camera. You want to exchange some photos between your camera and a friend's camera. Both cameras are dual-role devices and will therefore have Mini-AB connectors.

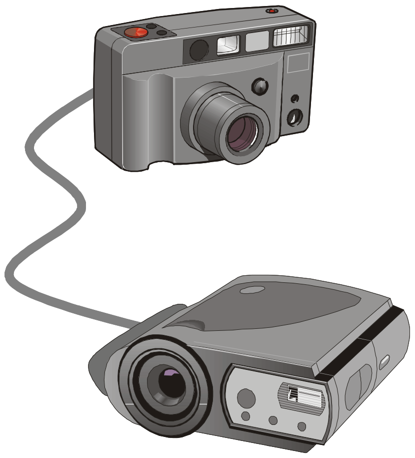


Figure 1-8. Interconnection of two dual-role device cameras

Since we are USB design engineers we will know to plug the A-end of the interconnecting cable into our camera and the B-end into our friend's camera. But we cannot expect a consumer to know this. The A-end and B-end look different (shown in Figure 1-7), but this subtlety will be missed by most consumers. The cable does fit the opposite way around so, on average, we can assume that 50% of users will plug the B end into "our" camera and the A-end into "our friend's" camera. This makes our camera the device and our friend's camera the host.

But we need our camera to be the host since we don't want to be pushing unknown buttons on our friend's camera to make the transfer. We could flash some error light and/or buzzer and force the consumer to study Chapter 6, Subsection 19 Paragraph 14 of the camera user manual to discover that they should reverse the cable, OR we could electronically switch the cable for them. I would recommend the second option. The OTG Supplement authors included a Host Negotiation Protocol (HNP) which allows the two dual-role devices to interchange roles.

The A-end of the cable establishes the default-host, and the B-end of the cable establishes the default-device. The HNP, like the SRP, is delightfully simple and only involves the switching on and off the biasing resistors. At the start of the sequence the A-host will be driving Vbus and will have a pull-down resistor on the data lines as shown in Figure 1-9.

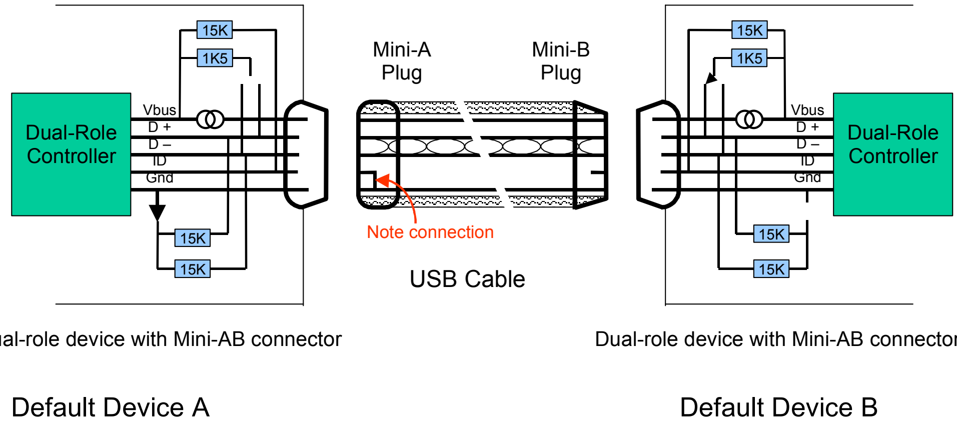


Figure 1-9. Default Bias resistor connections

The default-host (our friend's camera in this example) will have enumerated the default device (our camera) and discovered that it was a dual-role device. Since the default-host doesn't need to use the USB connection it signals to the default-device that it can use the bus if it needs to, and then it suspends the bus. This mechanism uses a standard USB descriptor and a standard USB command. A new OTG descriptor, shown in Figure 1-10, is defined, and the host uses a Set_Feature command to enable HNP operation. This is the same mechanism as Set_Feature (WakeUpEnable) so no new theory need be learned here.

Offset	Field	Value	Description
0	bLength	Number	Descriptor size = 3
1	bDescriptorType	Constant	OTG Type = 9
2	bmAttributes	Bitmap	OTG Device Characteristics b7..b2: reserved (0) b1: HNP supported [1] b0: SRP supported [2]

(1) True if device supports HNP. If true, SRP must also be true.
(2) Not used by A-Device during normal operation; used during compliance testing to automatically detect B-Device capabilities

Figure 1-10. New OTG Descriptor

The default-device detects the idle state on the bus and removes its pull-up resistor. The default-host detects this SE0 state on the bus and turns on its pull-up resistor to signal its ability to be a device. The default-device detects this pull-up and assumes the role of host by driving a reset onto the bus. This sequence is described in more detail in Chapter 6.

We have electronically “swapped the cable” and our camera is now the host. This was done in less than 100msec and this operation is unknown, as it should be, to the consumer. They inserted the cable “backwards” so we swapped it for them. It was increased design complexity, but this resulted in vastly improved ease of use – well worth the effort.

Notice that in all of the system hardware diagrams that included dual-role devices, a hub was not included. This was deliberate. A hub is not required to propagate the changing dc voltage levels defined by SRP and HNP. If a dual-role device detects that a hub is connected downstream of the Mini-AB connector, then it is not permitted to use SRP or HNP signaling. It is assumed that dual-role devices will be directly connected to other USB devices or other dual-role devices.

Chapter Summary

Careful study of the USB specification reveals the ability to build a “limited capability” USB host. This host need not provide 500mA or even 100mA of power to a device which is self powered. It is required to supply 8mA. This host need not support every available USB device but should provide a Targeted Peripheral List of the devices that it does support. The functionality of a limited host can be combined with a standard USB device to create a dual-role device.

The OTG Supplement defines the responsibilities of a dual-role device and specifies two protocols, the Host Negotiation Protocol and the Session Request Protocol, which propagate USB's Ease-Of-Use image. The OTG Supplement adds one additional descriptor that is processed using standard USB methods.

The OTG Supplement is elegant in its simplicity. The next chapter will describe silicon that implements this supplement in two capable components, the EZ-Host and the EZ-OTG.

Chapter 2: Getting to know EZ-Host and EZ-OTG

This chapter introduces the hardware and firmware aspects of the EZ-Host (CY7C67300) and the EZ-OTG (CY7C67200) components. You will discover that both parts have ample features that can address all of the issues raised in Chapter 1. Almost all of the hardware and firmware capabilities are configurable to suit many different product implementations, and you are not expected to use every feature in a single application. The firmware development environment uses a fully-configurable GNU tool set hosted on a Windows platform; most developers will only use these tools. The development environment is covered in the next chapter.

Hardware features

Figure 2-1 shows a block diagram of the EZ-Host and EZ-OTG components. The two parts are designed to be software compatible and are supplied in a 100-pin TQFP package and a 48-pin FBGA package respectively. The EZ-Host includes the capability to have external memory added while the EZ-OTG can only use its internal memory. Both devices have two Serial Interface Engines (SIEs) that may be independently configured as a USB host or as a USB device. Additionally, one SIE has hardware support for an OTG-style, dual-role device. The EZ-Host has two ports on each SIE while the EZ-OTG has one port on each SIE. Both parts include a BIOS in ROM that provides default management of the on-chip resources.

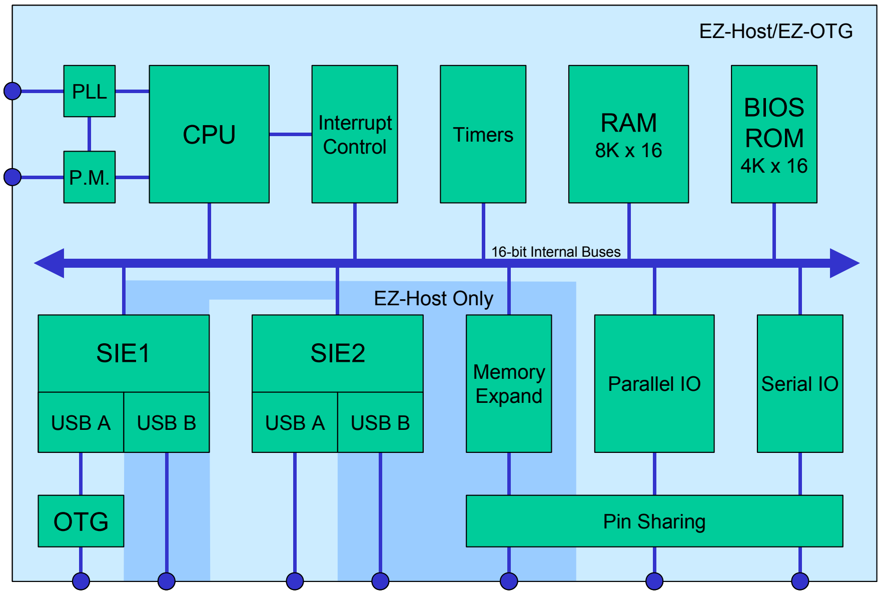


Figure 2-1. Block diagram of EZ-Host and EZ-OTG components

Central Processing Unit

The CPU is a 16-bit RISC implementation with 16 general-purpose, 16-bit registers mapped into memory via a regbank register (which is itself mapped into memory) as shown in Figure 2-2. This mapping technique allows the use of multiple register banks, if required, to process high frequency interrupts. Following a power-on cycle, regbank defaults to 0100H. All arithmetic and logical operations set FLAGS in accordance with 16-bit computations; the FLAGS register is also mapped into memory as shown in Figure 2-2. The internal architecture is optimized for this local memory-to-memory implementation.

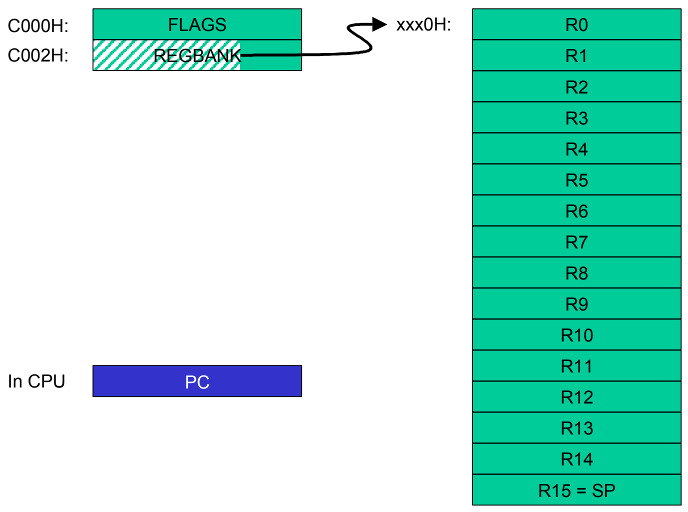


Figure 2-2. All registers, except PC, are memory-mapped

Registers R0 through R7 are used for general-purpose data manipulations and registers R8 through R15 are used as general-purpose pointers or for data manipulation. The instruction set uses R15 as a stack pointer. The instruction set includes a full set of orthogonal addressing modes that are well suited for a modern compiler.

Memory is byte addressable and the instruction set operates on byte variables if required (but note that the FLAGS are set assuming 16-bit operands). The 64KB memory space is used for code, data, IO locations, and register banks; some memory addresses are pre-assigned by the hardware as shown in the memory map in Figure 2-3. The BIOS makes additional address assignments – this is described in the BIOS section.

There are two programmable, hardware breakpoint registers that may be used by a debugger to trap CPU execution events at full speed.

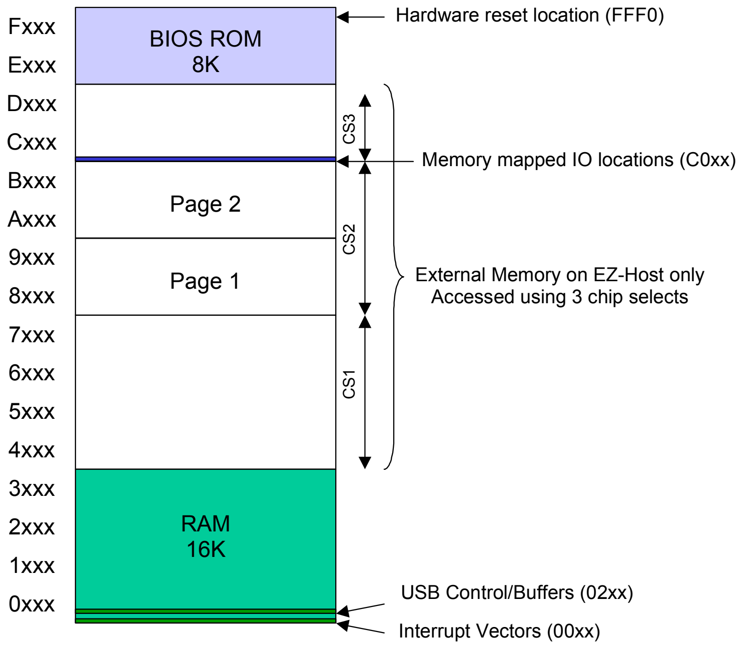

Fxxx
Exxx
Dxxx
Cxxx
Bxxx
Axxx
9xxx
8xxx
7xxx
6xxx
5xxx
4xxx
3xxx
2xxx
1xxx
0xxx
BIOS ROM
8K
Page 2
Page 1
RAM
16K
CS3
CS2
CS1
Hardware reset location (FFF0)
Memory mapped IO locations (C0xx)
External Memory on EZ-Host only
Accessed using 3 chip selects
USB Control/Buffers (02xx)
Interrupt Vectors (00xx)

Figure 2-3. Hardware assigned memory map

Memory Expansion Capability

Only the EZ-Host, in its 100-pin package, supports the addition of directly addressable external memory. This memory can be static RAM or ROM. BIOS makes some assumptions about where each type of memory is located, but the hardware just sees it as “external” memory. Three pre-decoded chip selects are available, one for each region, and each region can be populated with 8-bit or 16-bit memory with different access times.

Additional EZ-Host Capability

The additional pins of the EZ-Host package enable it to support an additional IO capability of four PWM channels and an IDE interface.

Integrated Timers

Both the EZ-Host and the EZ-OTG have three timers – two general purpose timers and one watchdog timer that can be enabled to reset the CPU.

Power Management

The CPU and IO devices are designed to operate at 48 MHz. An internal Phase Lock Loop generates this operating frequency from an external clock or crystal running at 12 MHz. The CPU can be slowed using a clock divider (2 through 16) or halted to conserve power. The CPU can also suspend itself, after enabling a variety of wakeup sources, to reduce power-consumption to a minimum.

The CPU core requires 3.0 volts to operate and this may be difficult when it is operating on battery power. Both the EZ-Host and EZ-OTG include an integrated power booster than can generate the required 3.0V from a battery that has dropped to as low as 2.7 volts. A few external components are required to implement this feature (shown in Figure 2-4), and these are only installed if a stable 3.3V voltage source is not available.

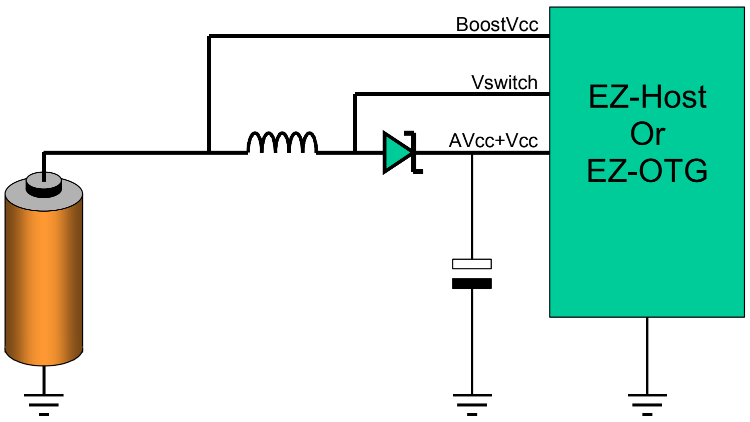


Figure 2-4. An integrated power booster is used for battery-powered applications

When operating as an OTG host the EZ-Host/EZ-OTG needs to generate a VBUS of 5.0 volts. A charge pump is integrated into each part that enables them to generate 5.0 volts from their 3.3V supply if required. A few external components, shown in Figure 2-5, are required for this charge pump. The EZ-Host/EZ-OTG can supply up to 10mA with this circuit, which is sufficient to support low-power, bus-powered devices such as some keyboards and mice. Note that if 5.0 volts is available in your design then the charge pump circuitry is not used and the external components are not required. The OTG VBUS connection should still be made since the EZ-Host/EZ-OTG requires this for detection and signaling.

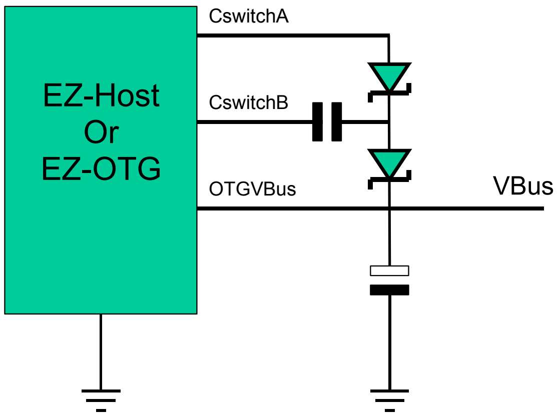


Figure 2-5. An integrated charge pump is used for battery-powered applications

USB Capabilities

Much of the functionality of the EZ-Host and the EZ-OTG is centered around their USB capabilities. Both components have two independent Serial Interface Engines (SIE) that enable connections to two independent USB segments. The EZ-Host supports two USB ports on each SIE while the EZ-OTG supports a single port on each SIE. The “A” connection on SIE 1 additionally supports all the features required by the On-The-Go supplement to the USB 2.0 Specification. It can be configured as a host port or a peripheral port and may be programmatically switched as required by an application.

You will discover in later chapters that the EZ-Host/EZ-OTG components come with a *Frameworks* firmware development code-base that implements full support for the on-chip resources including the logic of the Host Negotiation and Session Request Protocols (HNP and SRP). Figure 2-6 shows a summary of the possible USB configurations available with the EZ-Host and EZ-OTG components.

OTG Port	Host Port	Peripheral Port
1	1 or 2*	0
1	0	1
0	2, 3* or 4*	0
0	1 or 2*	1
0	0	2

** = EZ-Host only*

Figure 2-6. EZ-Host/EZ-OTG components support several USB configurations

Parallel IO

There are up to 25 independent input or output signals on the EZ-OTG component and 32 on the EZ-Host component. Each IO line can source and sink 4mA. This parallel IO may be configured as a Host Port Interface (HPI) that provides slave control and status ports for an external processor. Figure 2-7 shows the connection of an EZ-Host or EZ-OTG in this mode. The external processor is running some operating system that needs to support USB as a host or as a device. This processor will send commands and data to the EZ-Host/EZ-OTG using a full Link Control Protocol (LCP) implemented in the BIOS. In this mode the EZ-Host/EZ-OTG operates as a (very) smart peripheral and handles all of the low-level USB communications on behalf of the external processor. An example using LCP with a Linux host is presented in Chapter 7.

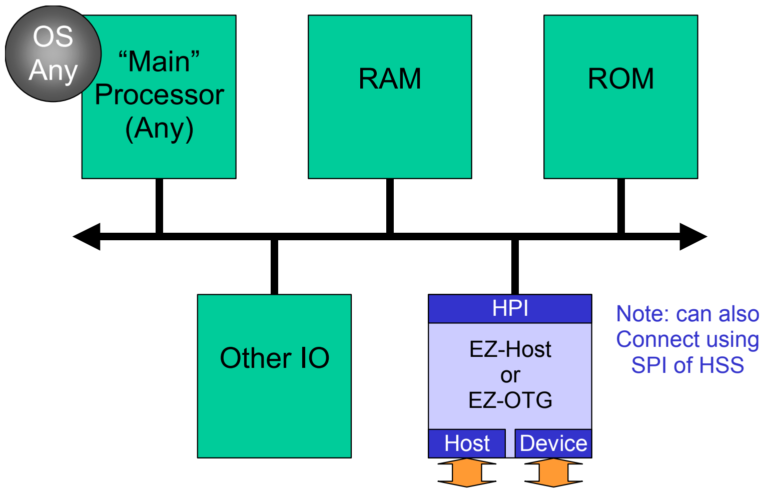


Figure 2-7. EZ-Host/EZ-OTG support a coprocessor mode

Serial IO

Both the EZ-Host and EZ-OTG additionally support a variety of serial protocols: High Speed Serial (HSS), industry-standard Serial Peripheral Interface (SPI), a UART designed as a serial debug port, and I2C. The I2C interface to serial EEPROM is always available and may be used to add code or data to the default BIOS configuration (this is described, in detail, in the scan section). A UART serial debug port is always available with the EZ-Host but it is only available when GPIO is selected for the parallel IO of the EZ-OTG. I found that, in general, I did not use this serial debug port since better functionality is available via a USB port.

BIOS also supports a Link Control Protocol (LCP) on the SPI port and the HSS port. Since these protocols do not have direct access to internal memory (the HPI port does) then additional data transfer steps must be used. If your USB bandwidth requirements for a co-processor are low then these serial interfaces enable a viable alternative to the HPI parallel connection. A common protocol allows different configurations for different applications while preserving the software investment. LCP is described in Chapter 7 with a co-processor example.

IO Summary

Figure 2-8 summarizes the IO configuration of the EZ-Host and the EZ-OTG components in each of the modes.

GPIO	HPI	IDE[1]	PWM	HSS	SPI	UART
GPIO31	SCL	SCL				
GPIO30	SDA	SDA				
GPIO29	OTGID	OTGID				
GPIO24	INT	nACK				
GPIO23	nRD	nRD				
GPIO22	nWR	nWR				
GPIO21	nCS	-				
GPIO20	A1	DIR				
GPIO19	A0	nRQT				
GPIO15	D15	D15		CTS[2]		
GPIO14	D14	D14		RTS[2]		
GPIO13	D13	D13		RxD[2]		
GPIO12	D12	D12		TxD[2]		
GPIO11	D11	D11			MOSI	
GPIO10	D10	D10			SCK	
GPIO9	D9	D9			SSI	
GPIO8	D8	D8			MISO	
GPIO7	D7	D7				TX[2]
GPIO6	D6	D6				RX[2]
GPIO5	D5	D5				
GPIO4	D4	D4				
GPIO3	D3	D3				
GPIO2	D2	D2				
GPIO1	D1	D1				
GPIO0	D0	D0				

Additional IO pins of EZ-Host

GPIO28						TX
GPIO27						RX
GPIO26			PWM3	CTS		
GPIO25						
GPIO18		A2	PWM2	RTS		
GPIO17		A1	PWM1	RxD		
GPIO16		A0	PWM0	TxD		

Memory Expansion of EZ-Host

D15				CTS		
D14				RTS		
D13				RxD		
D12				TxD		
D11					MOSI	
D10					SCK	
D9					SSI	
D8					MISO	
D0:7						
A0:18						
Control						

Note 1: EZ-Host only, Note 2: EZ-OTG only

Figure 2-8. EZ-Host and EZ-OTG IO functionality.

Firmware Features

The EZ-Host and EZ-OTG components are designed to be software compatible and they use the same BIOS. If BIOS accesses non-existent features on the EZ-OTG device then no errors or side-effects are created. EZ-Host has an external memory bus, and BIOS will use this to check for the presence of RAM and/or ROM – the EZ-OTG device will always return false to these tests.

BIOS contains 8KB of very dense CY16 assembler code and this creates a default configuration. The BIOS is designed to be over-rideable in whole (EZ-Host only since it supports external memory) or in parts. It is table-driven and makes extensive use of interrupt vectors. The BIOS is organized as a set of subroutines that are accessed via an interrupt vector table in low memory. This vector table has 48 hardware vector entries and 80 software vector entries as outlined in Figure 2-9.

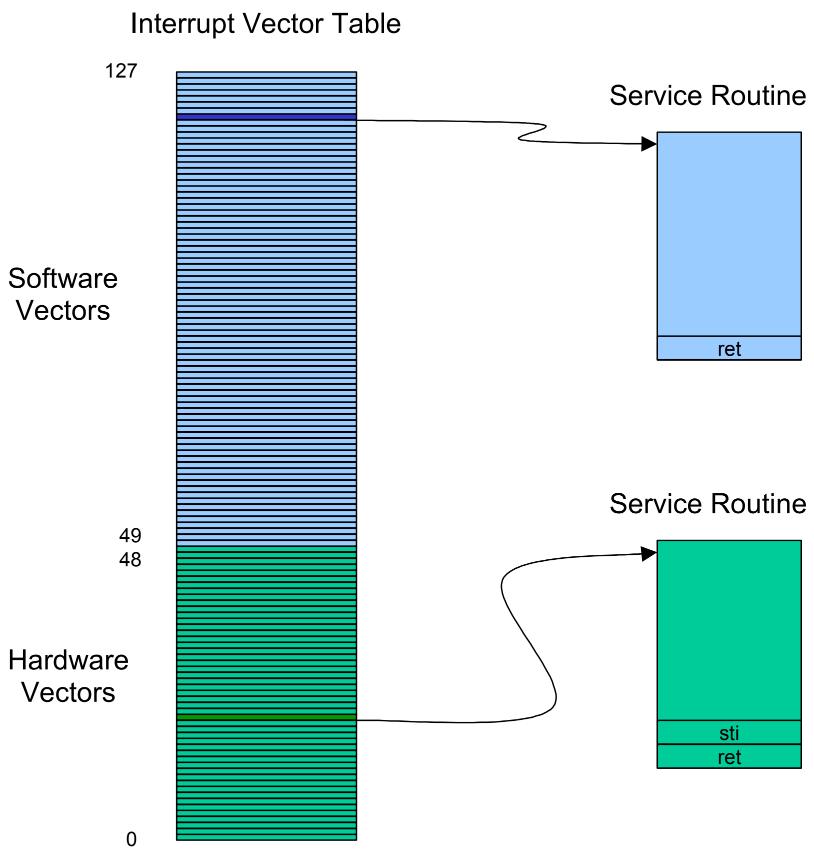


Figure 2-9. Interrupt Service Routines are accessed via a table in RAM

The operation of BIOS may be changed by replacing an existing interrupt service routine (ISR) or by chaining additional code to an existing interrupt service routine (how this is done is described next). ISR replacement involves replacing the routine address in the interrupt vector table with a new address. Chaining requires the existing routine address be saved before being replaced by the new routine address. The new routine, when it has finished its task, jumps to the previous routine using this saved address. If you wanted your new routine to execute AFTER the previous routine then you would CALL the previous routine and, once it returned, you would execute your new code. All three examples, pre-processing, post-processing and replacement, are shown in Figure 2-10. Chaining is an important concept and is used throughout the BIOS.

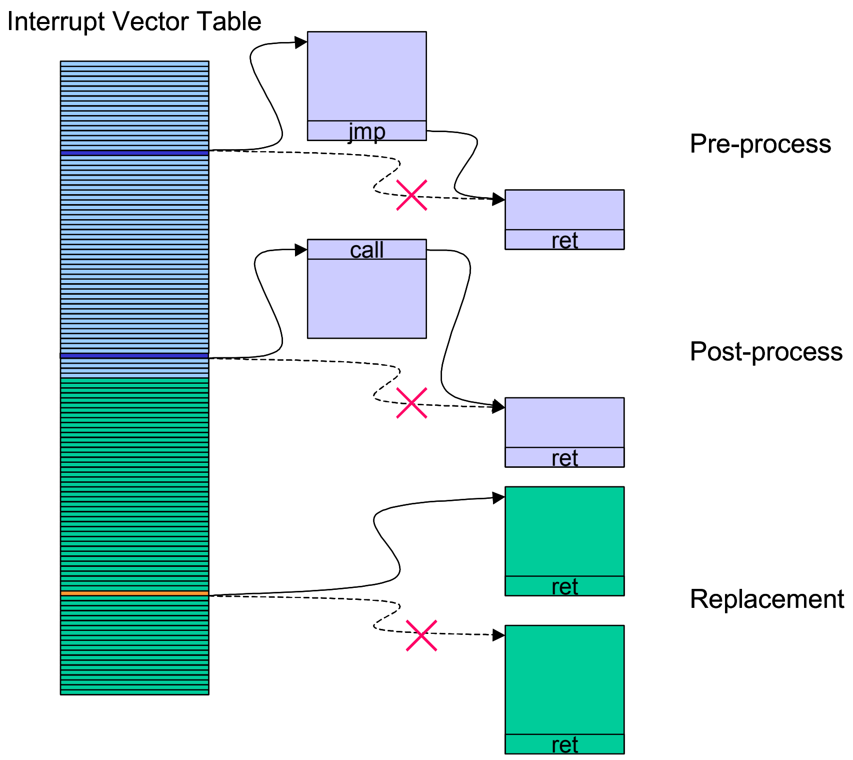


Figure 2-10. Changing or enhancing BIOS operation

BIOS Operation

The BIOS documentation from Cypress describes, in detail, the default operation of all of the assigned interrupt vectors. There are several unassigned vectors where you can add additional capability and you can change existing vectors to change how BIOS operates. The BIOS documentation defines the facts of operation, but three concepts deserve more explanation: Memory Management, Idle_Loop, and SCAN.

BIOS Memory Management

If you allow the BIOS initialization to complete (you don't have to, SCAN will describe how you over-ride this), BIOS will own all of the un-used RAM, and it provides a mechanism to allocate and free memory buffers to running programs. Figure 2-11 shows how memory is allocated by BIOS – free memory extends from the top of BIOS-use memory to 3FFFH (or 7FFFH if external memory is added to an EZ-Host part).

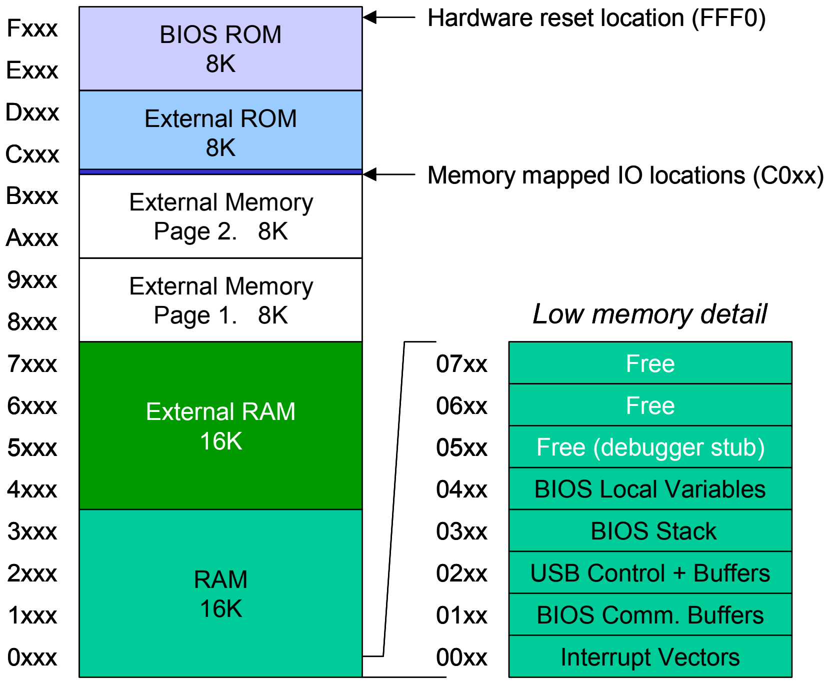


Figure 2-11. BIOS assigned memory map

BIOS uses four software vectors for memory allocation:

INT 69	Pointer to first memory allocation block
INT 68	Allocate x bytes of memory
INT 75	Return previously allocated memory
INT 76	Recalculate free memory (error recovery)

BIOS uses memory allocation blocks to describe available and used memory. Following initialization there will be two memory allocation blocks; INT 69 will point to the first one and this links to the second one. A memory allocation block contains two words – the first word describes how many bytes are present in this block and the second word is a busy (= 8000H) or available (= 0) flag. Figure 2-12 shows the memory allocation blocks at initialization and also after three memory blocks (0x1360, 0x100 and 0x2000 bytes) have been allocated and one block (0x100 bytes) returned.

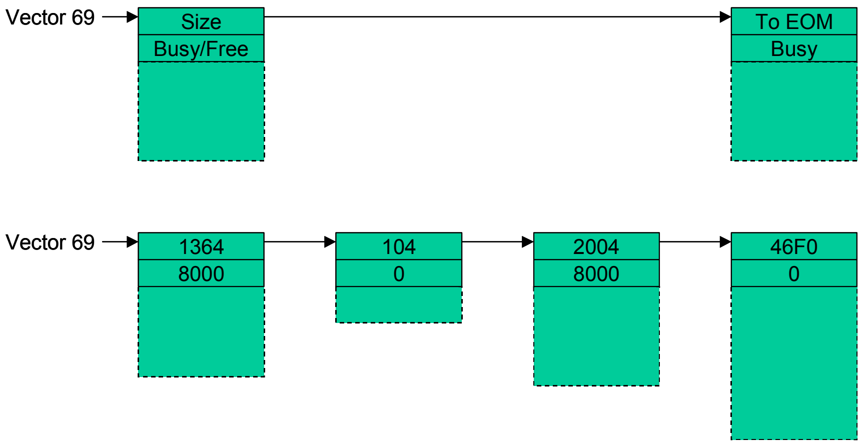


Figure 2-12. Typical Memory Allocation Blocks in use

Before I go into more detail I am going to recommend that you do not use this mechanism. It is a good scheme that is useful in allocating data buffers, but the problems come later when you want to load code into an allocated memory block (this is SCAN described later in this section). The major issue is that EZ-Host/EZ-OTG code is not inherently relocatable – any CALL or long JUMP will use an absolute address that needs to be "fixed up" prior to execution. Computed jumps or calls are very difficult to fix-up. SCAN does provide a fix-up mechanism but it assumes that code was written in assembler and all of the fix-up addresses are readily identifiable. Almost all of the examples that will be presented in this book are written in C, and no automated tool exists to enable a binary image of this code to be relocated. The examples will therefore use statically allocated code and data.

So why did I spend two pages describing a feature that I don't want you to use? Well, many routines within BIOS use the mechanisms to allocate memory, so we need to be aware that BIOS may think that our program space is "free to be allocated." The examples that we will create will need to give BIOS some memory that it can use. A sophisticated user may use this capability, but I do not recommend it for general use.

BIOS Idle Task

If we allow the BIOS initialization to complete it will set up an Idle_Task that executes a series of routines chained together. Remember that a chain is as strong as its weakest link – the mechanism is good provided we handle it correctly. BIOS uses three software vectors for Idle_Task management:

INT70	Start of Idle Chain
INT71	BIOS's Idle Task
INT72	Insert task into Idle Chain

The BIOS Idle_Task is a continuous loop as shown in Figure 2-13. If there is no work to do, then the CPU HALTs. It is brought out of halt by any enabled interrupt, which it first services. The CPU then executes the tasks in the Idle Chain to check for additional work to do. When completed it halts again waiting for any enabled interrupt.

```
Idle:                ; Pointed to by INT71
   addi r15, 2       ; Removed return address from stack
   halt              ; Wait for an enabled interrupt
   INT 70            ; Check the Idle Chain
   INT 71            ; Call myself
```

Figure 2-13. The BIOS Idle Loop

Adding a task to the Idle_Chain is implemented via Int72. You pass a pointer to the start of the new task in R0 and a pointer to the next task in the chain is returned in R0 – you should jump to this once your task is completed as shown in Figure 2-14.

```
Init_Another_Idle_Task:
   mov r0, Another _Idle_Task ; Provide pointer to my task
   INT 72                     ; Insert task into chain
   mov [Next_Task], r0        ; Save pointer to next task
   ret                        ; Initialization complete
Another _Idle_Task:
;
; insert your code here, it completes with…
;
   jmp [Next_Task]
```

Figure 2-14. Adding a new task to the Idle Chain

The default operation for BIOS is to initialize its numerous subsystems (UART, USB etc) and wait for work to do. You can over-ride this operation by adding code and data using the scan task.

BIOS Scan Operation

SCAN is the operation that is used to modify the default operation of BIOS. Early in the BIOS initialization it starts looking for “scan signatures” – it looks in many places but the easiest to explain is accesses to EEPROM connected to the I2C bus. Think of the EEPROM as a provider of a byte stream. BIOS will start reading from I2C address 0 and will continue to read sequentially until it runs out of “scan records.” A scan record consists of a header and some data as shown in Figure 2-15.

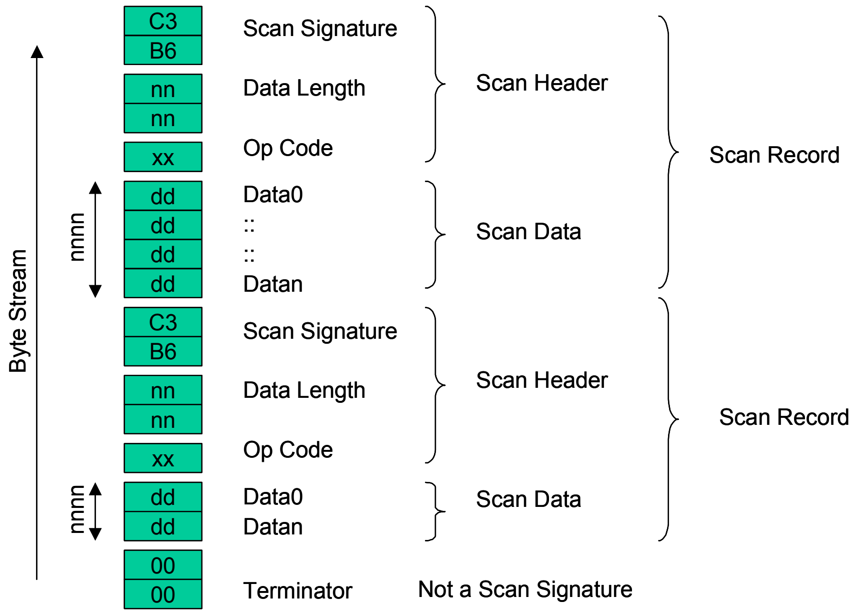


Figure 2-15. Detail of scan records in EEPROM

Scan checks the first word of a record for the special signature value of 0xC3B6, and if a match is not found then SCAN gives up on this byte stream and looks elsewhere. If a match is found then SCAN reads the next word as a DATA_LENGTH and the next byte as an OPCODE. There are currently 10 opcodes as shown in Figure 2-16.

Scan continues to look for scan signatures during run time via the USB and UART Idle_Tasks. This will allow program code and data to be downloaded at any time – the debugger uses this feature extensively.

Op Code	Interpret bytes as:	Effect
0	N ADDR D D	Copy N-2 bytes to memory starting at ADDR
1	N Vec D D	Copy N-1 bytes to memory pointed to by VEC
2	N Vec D D	Request N-1 byte buffer from BIOS, set VEC to point to this buffer. Copy N-1 bytes into this buffer
3	N Vec ADDR ADDR	Add the real address pointed to by VEC to the following list of ADDR's. This is a relocation fixup.
4	2 ADDR	Jump to ADDR. BIOS just gave the CPU away
5	2 ADDR	Call to ADDR, return control to BIOS
6	1 Vec	Execute VEC
7	N Vec ADDR ADDR	Copy N-2 bytes from memory at ADDR1 to I2C at ADDR2 VEC should be 64 or 65 ie I2C_Write
8	N Vec ADDR D D	Request a buffer from BIOS, copy N-? Bytes into buffer, Then copy buffer to I2C at ADDR, then free buffer VEC should be 64 or 65 ie I2C_Write
9	N Off Value Off Value	Write VALUE into 0xC000 + OFFSET, ie update configuration

Note: shaded field indicates a repeated field

Figure 2-16. SCAN has 10 defined Op Codes

Take a moment to study the effects of a SCAN operation, shown in Figure 2-16. This is a very powerful mechanism that can be used to download new code and data into RAM.

I described the EEPROM as a byte stream – SCAN will also accept a byte stream from the UART or from USB. Additionally the Link Control Protocol supported by the HPI, SPI and HSS interfaces offers a similar set of capabilities while in co-processor mode. These mechanisms make it very easy to load application programs into the EZ-Host and the EZ-OTG, and we shall run through some examples in the next few chapters.

BIOS has control of the CPU at power-up. It uses the CPU to execute the BIOS code including the SCAN function. It can load a new program using a variety of scan codes. Now focus on scancode (4). BIOS can pass ownership of the CPU to a loaded program. The effect of this is that BIOS will not complete its initialization and will not set up its Idle Task. The loaded program now has the responsibility of owning the Idle_Task. It could set up its own idle task, and this *must* call the BIOS Idle Chain (INT 70) to keep low level functions, such as SCAN, operating, or it could pass control back to BIOS via a restart of the Idle_Task.

Other BIOS functions

BIOS has a wide array of functions that deal with the low-level details of USB communications when operating as a host or as a device. These are best described with the aid of an example, but this means we need to be familiar with the EZ-Host/EZ-OTG development environment and example Frameworks. This will be the focus of the next chapter.

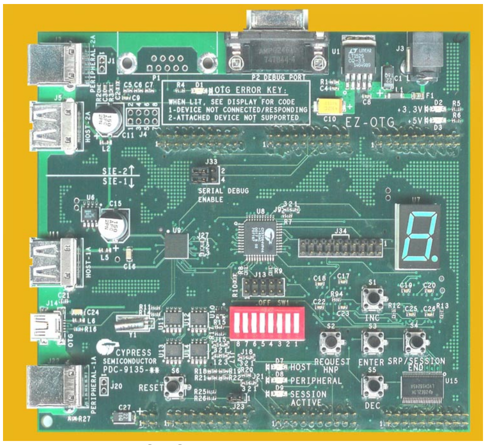
Chapter Summary

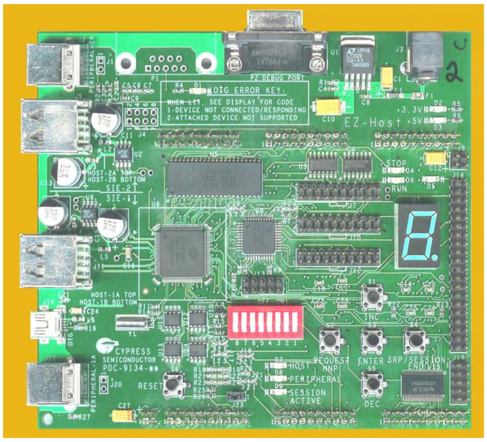
The EZ-Host and EZ-OTG components have a great deal of capability that we can use to implement a wide variety of USB based products. Two independent Serial Interface Engines (one with dual-role capability) are supported by a 16-bit RISC CPU, 16K of RAM, and parallel and serial IO functions. An integrated BIOS provides a default configuration with interrupt service routines to support essential low-level operation of all of the hardware features. The BIOS is over-rideable in whole (EZ–Host only) or in parts using a SCAN mechanism.

The next chapter will show how easy the EZ-Host/EZ-OTG components are to use.

Chapter 3: EZ-Host/EZ-OTG Development Environment

Firmware development for the EZ-Host/EZ-OTG employs Windows-based tools, and this chapter assumes that the Cypress CY3663 DVK is used as a target for the example code. The DVK includes a StrongArm-based single board computer (SBC) and two mezzanine boards, one EZ-Host based and the other EZ-OTG based as shown in Figure 3-1. In this chapter we will use the EZ-Host board in standalone mode as a vehicle to learn the toolset and debug process. The tools should be installed according to the Cypress documentation.


EZ-OTG Development Board


EZ-Host Development Board

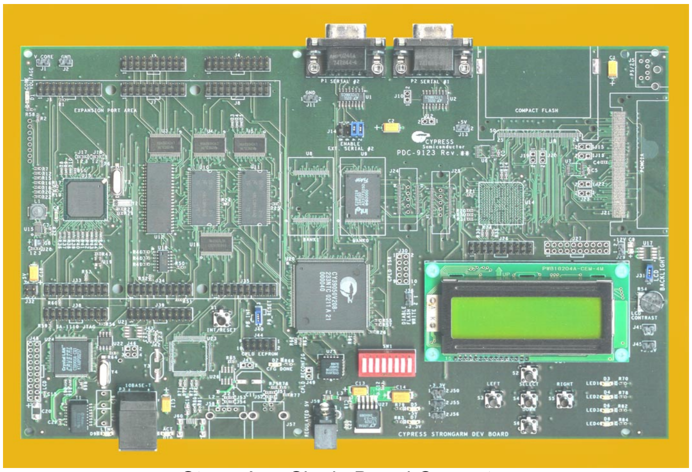

StrongArm Single Board Computer

Figure 3-1. Development kit includes an SBC and 2 mezzanine boards

Firmware is developed using the GNU Toolset hosted on a Windows-based platform via the Cygnus UNIX emulation engine, Cygwin. The full complement of GNU Tools (compiler, assembler, make, linker, debugger and utilities) are provided with the CY3663 installation, and these are integrated into a Windows environment, and are ready-to-use, as shown in Figure 3-2.

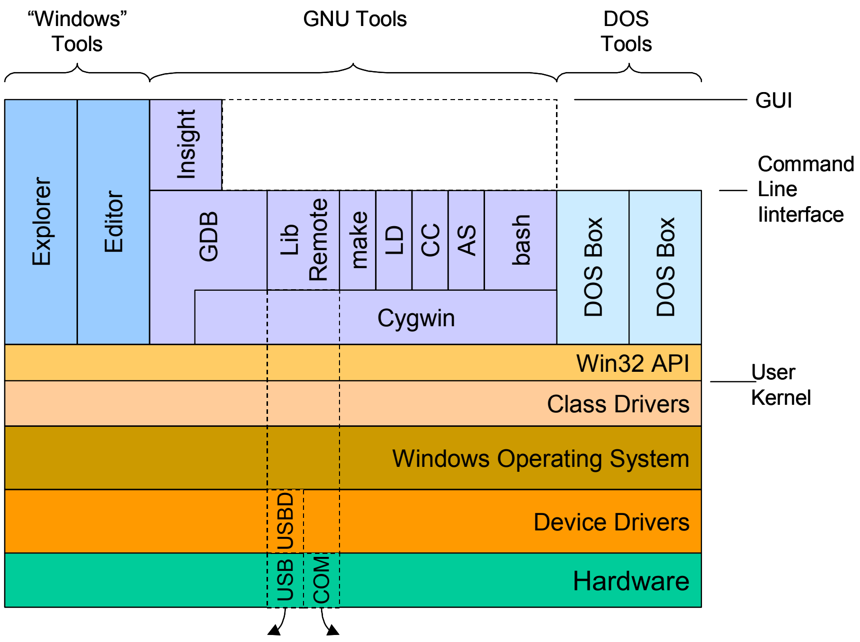


Figure 3-2. Simplified view of CY16 Development Environment

I included DOS tools in Figure 3-2 since they help to describe the GNU tools. Windows supports multiple "DOS boxes" within its windowed GUI environment. A DOS command is typically text input via a command line, and it creates text output. A series of DOS commands can be combined in a BATCH file (*.bat), and program output can be redirected to a text file for later viewing or processing.

All of the GNU tools are also command line based and, like DOS commands, may be run interactively or via a command file. The GNU tools call the command scripts, and these are much more sophisticated than their DOS counterparts. The CY3663 DVK includes all of the scripts that you will need for successful development – whether this is rebuilding one of the examples or creating your own. If you don't want to delve into the "how" the tools work then you can just use them and ignore the fact that they are GNU-based.

Another huge benefit of using the GNU tools is that the same process is used to generate and debug code for the CY16 components (EZ-Host and EZ-OTG) or the StrongArm component on the SBC. [Or, for that matter, ANY processor supported by the GNU code generation backend]. We specify the target processor in a makefile.

The GNU tools include a windowed debugger, called Insight, that sits on top of the standard GNU debugger, GDB. We will use this as part of our development process. Developing CY16 firmware is a straight forward process: we create source files and makefile scripts using a good multi-file text editor; we build our object code then download it to a target and debug it. I tell everyone that the software is the easy part of an embedded microcontroller project since it is only 1's and 0's. The hard part is getting them in the right order

CY16 Firmware Architecture

We saw, in Chapter 2, that the EZ-Host and EZ-OTG components have a wide range of hardware capabilities that can be employed to implement a wide variety of USB solutions. Harnessing these capabilities into an easy-to-use framework was quite a challenge but, by following the methodology outlined in this chapter, you will be able to quickly build custom application programs to meet your requirements.

I chose to increment into a full design example using a series of simple examples that explore individual features of the CY16 architecture. Each simple example builds on the previous one to create a multi-purpose design example. I will cover several capable design examples in later chapters.

The BIOS sets up an event-driven environment with chained idle tasks. Layered on top of BIOS is a development Frameworks, and layered on top of this framework is our application code (simple example or design example). Each of these layers consists of three distinct Tasks as shown in Figure 3-3.

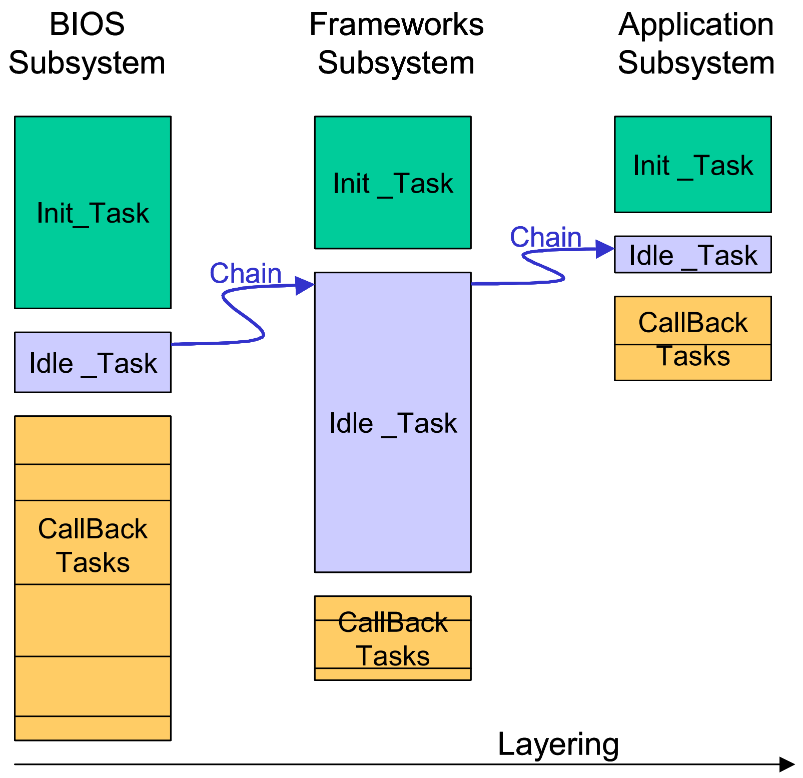


Figure 3-3. CY16 firmware is layered and is task based

The Init_Task is responsible for setting up the Idle_Task and any CallBack_Tasks. The Init_Task is only run once, so any memory it uses may be returned after it has executed. After execution, the Init task returns control of the processor back to the caller.

The Idle_Task must honor an Idle_Chain. All of the Idle_Tasks are dynamically linked together and each is responsible for maintaining this chain. There is no central kernel that is scheduling tasks. The primary scheduling mechanism is hardware interrupts. When not serving interrupts, the Idle_Task of each software subsystem runs and looks for work to do. Each subsystem calls the next using the chaining mechanism. The chaining mechanism is simple and involves a single call as will be shown in the examples.

CallBack_Tasks are procedures that are run as a result of some event occurring. This event could be a hardware interrupt, a software interrupt or some processing in an Idle_Task that created some work to be done.

All firmware that we develop for the EZ-Host/EZ-OTG will follow this format of Init_Task, Idle_Task and CallBacks. Each of the tasks is drawn as a single block in Figure 3-3, but this should not infer that each block is a single, monolithic procedure. The Init_Task and Idle_Task have a single entry point but will typically contain multiple procedures.

Frameworks Subsystem

The Frameworks subsystem, drawn as a single layer in Figure 3-3, contains a wide range of subsystems implemented at the application subsystem layer – our application is just one of the subsystems that it manages. Basically, if the EZ-Host/EZ-OTG has a hardware capability then Frameworks has a software subsystem to support it. Frameworks also contains software subsystems to manage the hardware on the mezzanine boards – buttons, dipswitches, LEDs, and seven segment display.

The Frameworks subsystem is built from the source files in the COMMON directory. Take a quick look into this directory and note that the common module list is very complete. Each module is designed around a CY16 capability, and individual features are contained within conditional compilation directives. You will not need to edit any files in this directory, but Frameworks is supplied in source format for those users who want to. A single file in each application directory, called fwxcfg.h, is used to select which features of Frameworks should be included to support the application program. My first set of “simple examples” will use pre-configured fwxcfg.h files, and we shall learn how to edit fwxcfg.h in a later chapter.

Simple Example #1 - Hello World

The first program we always write is “Hello World” where “Hello World” is displayed on a console. Our target system, the EZ-Host mezzanine board, does not have a console, but it does have a seven-segment display and some buttons. Our “Hello World” program will first display the letter “H” on the seven-segment display then advance through the other letters of hello as we press any button. This uses next-to-no features of the EZ-Host (hardware or BIOS) allowing us to focus upon the process of creating and debugging a program in this environment.

All of the source files for se1 (simple example #1) are contained within the se1 directory. If you have installed the Cypress CD-ROM in its default location, all of the simple example directories will be located in C:\Cypress\USB\OTG-Host\Source\stand-alone. Figure 3-4 shows a source listing of app.c. The only initialization in the Init_Task warns BIOS that we are using some memory. The button debouncing and press event is handled within Frameworks, so all we have to do is install a Callback_Task, with a pre-defined name, to handle this event.

Frameworks also handles driving the seven segment display, so we just call one of its IO routines. So how do we build and debug our three-line application program?

```
/* Application data. */
// Declare the Hello Message in Seven Segment Display "font"
const uint8 HelloMessage[] = { 0x7F, 0x89, 0x86, 0xC7, 0xC7, 0xC0, 0xFF };
static uint16 HelloIndex;

// Declare the app_init_task
void app_init(void) {
// Tell BIOS that we, and the GDB stub, are in memory.  For DEBUG only
    __asm( "mov r0, 0x1360");
    __asm( "int %0" : : "n" (ALLOC_INT) );
    app_button_handler (0);            // Clear the display
}

// Declare the app_idle_task.  No work here, see Button Callback

// Declare the CallBack routines
void app_button_handler( BUTTON button ) {
// Advance through the display text on any button press
    if (HelloIndex++ > sizeof(HelloMessage)-2) HelloIndex = 0;
    write_cpld (SSD_WRITE, HelloMessage[HelloIndex]);
}
```

Figure 3-4. Our first example application program

Within the se1 directory there are several other files. I created the Frameworks configuration file, fwxcfg.h, and there is no need for you to look at this yet (you will have ample opportunity later). There is a makefile script that defines the rules of how our simple example should be built. The makefile will use a linker script (se1.ld) to locate our program at 1000H (why is described in the next section). There is also a DOS batch file called bash_env.bat that we will use to save a lot of typing.

Target System

Before building our application program we need to consider the capabilities, and constraints, of our target system. Our first example will use the EZ-Host mezzanine board in standalone mode. Note that the EZ-OTG mezzanine board could also be used for this example, and it will produce identical results. Figure 3-5 shows key features of the mezzanine board.

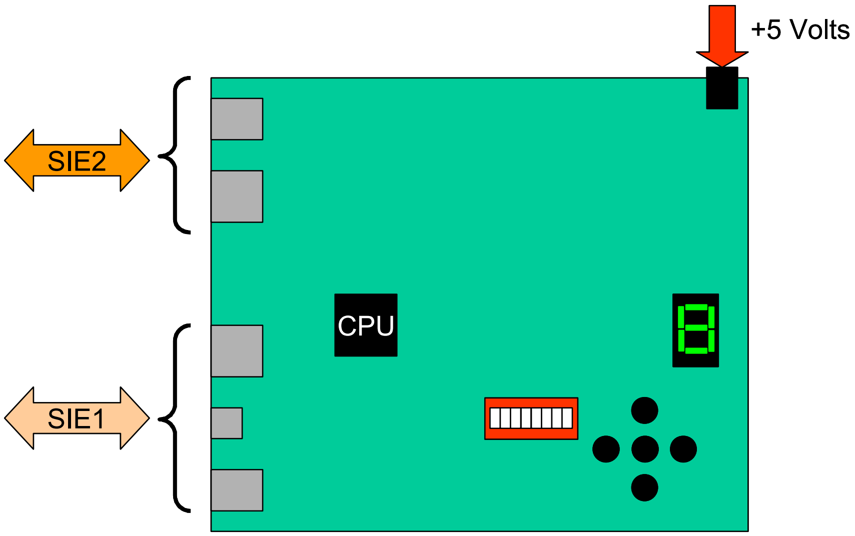


Figure 3-5. Key features of a mezzanine board

We will attach the mezzanine board to our development system using a USB cable to SIE2. Note that all dip switches will be set to OFF. The BIOS contains a default USB device descriptor, and this will be used to enumerate a connection. This default device uses a VID/PID of 0x4B4/0x7200, and this will load a CyUSBGen.sys device driver (this, and the INF files that set up this relationship, were installed when the toolset was installed). CyUSBGen.sys is a custom device driver that creates a DOS device called USBSCAN – this device uses vendor-defined commands to send a byte stream to the mezzanine board (byte stream was discussed in the SCAN section of Chapter 2). This connection will be used by the debugger.

Let's first tell the GNU tools that we will be debugging our software on a remote target (i.e., not using the PC's processor). Double-click the bash_env.bat icon in the se1 directory – this will open a DOS window and then start a bash shell. At the prompt enter "`cy16-elf-libremote -u`." This will start a standardized tool that the debugger will look for when it runs – libremote is the bridge between the PC-hosted debugger application and our remote target.

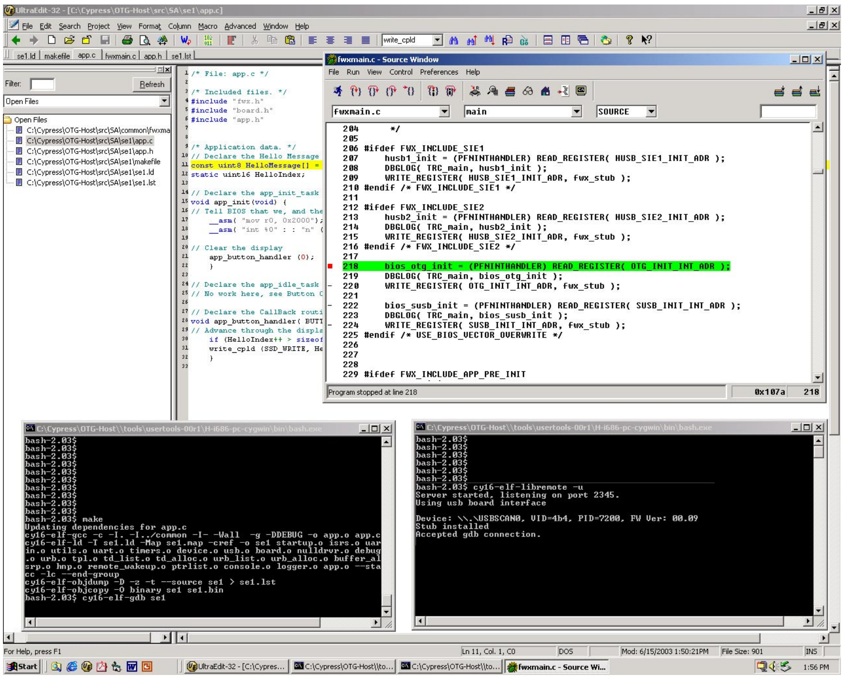
When the debugger runs, libremote will copy a small stub program onto our target board. It uses this to control the operation of the target. The stub installs itself as the BIOS Idle_Task and therefore has complete control of the processor (it calls the Idle_Chain to keep all other tasks operational). This stub loads in free memory at 500H (approximate, different versions load in different places) and extends to 0xA00, so I chose 1000H for the origin of se1 since this is well above the debugger space.

Leave this libremote window open and return to the Windows explorer.

Double-click the bash_env.bat icon again to create another bash window. At the prompt enter "`make`." This will use the makefile script to build se1. There should be no errors. Then enter "`cy16-elf-gdb se1`" at the bash prompt. This will open the Insight debugger (gui shell on top of gdb) in another window. If this is the first time running Insight then you must set the target settings (Target=Remote/TCP, Hostname=localhost, Port=2345); these settings will be remembered. Now click the run icon. Insight will find libremote and establish a connection to our target system (if the libremote window is visible you will see the connection accepted) and will then download our example code to the EZ-Host mezzanine target board. You can now single step through the source code, set breakpoints, view all data using the symbols that were created in the source code, view registers or memory, or whatever you desire to debug the program. Insight, and its base gdb, are very capable debuggers. An online manual was installed when the tools were installed; I would recommend printing it out and studying it now (refer to Cypress/USB/OTG-Host/Docs/OTG-Host Tools/3_debug.pdf).

When we click "continue" with no breakpoints set, the program will run forever until we click the STOP icon. Holding down the top or bottom buttons on the mezzanine board will result in "HELLO" being cycled through the seven-segment display. Pressing any button will advance one letter at a time. This is our first CY16 program.

If you have a large screen then your development environment will look something like Figure 3-6 – an editor window (I use UltraEdit), an Insight main window with several support windows, and two bash windows. The Windows environment makes it easy to switch between these different programs.



```
/* File: app.c */

/* Included files. */
#include "fwx.h"
#include "board.h"
#include "app.h"


/* Application data. */
// Declare the Hello Message
const uint8 HelloMessage[] =
static uint16 HelloIndex;

// Declare the app_init_task
void app_init(void) {
// Tell BIOS that we, and the
    __asm( "mov r0, 0x2000");
    __asm( "int %0" : : "n" (

// Clear the display
    app_button_handler (0);
    }

// Declare the app_idle_task
// No work here, see Button C

// Declare the CallBack routi
void app_button_handler( BUTT
// Advance through the displa
    if (HelloIndex++ > sizeof
    write_cpld (SSD_WRITE, He
    }
```

```
204      */
205
206 #ifdef FWX_INCLUDE_SIE1
207     husb1_init = (PFNINTHANDLER) READ_REGISTER( HUSB_SIE1_INIT_ADR );
208     DBGLOG( TRC_main, husb1_init );
209     WRITE_REGISTER( HUSB_SIE1_INIT_ADR, fwx_stub );
210 #endif /* FWX_INCLUDE_SIE1 */
211
212 #ifdef FWX_INCLUDE_SIE2
213     husb2_init = (PFNINTHANDLER) READ_REGISTER( HUSB_SIE2_INIT_ADR );
214     DBGLOG( TRC_main, husb2_init );
215     WRITE_REGISTER( HUSB_SIE2_INIT_ADR, fwx_stub );
216 #endif /* FWX_INCLUDE_SIE2 */
217
218     bios_otg_init = (PFNINTHANDLER) READ_REGISTER( OTG_INIT_INT_ADR );
219     DBGLOG( TRC_main, bios_otg_init );
220     WRITE_REGISTER( OTG_INIT_INT_ADR, fwx_stub );
221
222     bios_susb_init = (PFNINTHANDLER) READ_REGISTER( SUSB_INIT_INT_ADR );
223     DBGLOG( TRC_main, bios_susb_init );
224     WRITE_REGISTER( SUSB_INIT_INT_ADR, fwx_stub );
225 #endif /* USE_BIOS_VECTOR_OVERWRITE */
226
227
228
229 #ifdef FWX_INCLUDE_APP_PRE_INIT
```

Program stopped at line 218 0x107a 218

```
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$ make
Updating dependencies for app.c
cy16-elf-gcc -c -I. -I../common -I- -Wall  -g -DDEBUG -o app.o app.c
cy16-elf-ld -T se1.ld -Map se1.map -cref -o se1 startup.o isrs.o uar
in.o utils.o uart.o timers.o device.o usb.o board.o nulldrvr.o debug
.o urb.o tpl.o td_list.o td_alloc.o urb_list.o urb_alloc.o buffer_al
srp.o hnp.o remote_wakeup.o ptrlist.o console.o logger.o app.o --sta
cc -lc --end-group
cy16-elf-objdump -D -z -t --source se1 > se1.lst
cy16-elf-objcopy -O binary se1 se1.bin
bash-2.03$ cy16-elf-gdb se1
```

```
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$
bash-2.03$ cy16-elf-libremote -u
Server started, listening on port 2345.
Using usb board interface

Device: \\.\USBSCAN0, VID=4b4, PID=7200, FW Ver: 00.09
Stub installed
Accepted gdb connection.
```

UltraEdit-32 - [C:\Cypress\OTG-Host\src\SA\se1\app.c]
File Edit Search Project View Format Column Macro Advanced Window Help
se1.ld makefile app.c fwxmain.c app.h se1.lst
Filter: Refresh
Open Files
C:\Cypress\OTG-Host\src\SA\common\fwxma
C:\Cypress\OTG-Host\src\SA\se1\app.c
C:\Cypress\OTG-Host\src\SA\se1\app.h
C:\Cypress\OTG-Host\src\SA\se1\makefile
C:\Cypress\OTG-Host\src\SA\se1\se1.ld
C:\Cypress\OTG-Host\src\SA\se1\se1.lst
fwxmain.c - Source Window
File Run View Control Preferences Help
fwxmain.c
main
SOURCE
C:\Cypress\OTG-Host\\tools\usertools-00r1\H-i686-pc-cygwin\bin\bash.exe
For Help, press F1
Ln 11, Col. 1, C0
DOS
Mod: 6/15/2003 1:50:21PM
File Size: 901
INS
Start
UltraEdit-32 - [C:\Cypres...
C:\Cypress\OTG-Host\\to...
fwxmain.c - Source Wi...
1:56 PM

Figure 3-6. Firmware development uses multiple windows

When you finish debugging, select exit from Insight's file menu. This closes all of Insights windows. You may need to enter a Control+C in the libremote window to regain control of the bash shell. While in the libremote window hit the up-arrow key – this will retrieve the previously entered command (cy16-elf-libremote -u) and hit the enter key again. This sets up libremote for the next debug session.

Similarly, in the other bash window, hitting the up-arrow key will retrieve "`cy16-elf-gdb se1`," and hitting it again will retrieve "`make`". You can use this shortcut to rapidly move around the edit-build-debug loop. For now however, type "exit" to close this bash window.

Note, if the up-arrow does not retrieve the previous command, enter "`set –o vi`" to enable the command history/editing features. We will use about 2% of the capability of a bash shell while running the examples – an interested reader should review "Learning the Bash Shell" by Newham and Rosenblatt.

Our first simple example used almost no CY16 features so that we could focus on the TOOLS and the PROCESS. I wanted to do a USB device example next, but we have to go on a slight diversion (still very educational) due to the default operation of BIOS.

Simple Example #2 – Using Scan Records

During initialization BIOS checks the state of two GPIO pins to discover which mode it should start in. These two lines, GPIO30 and GPIO31, are also used for I2C communications so they will have a power-on state of high due to the pull-up resistors. This selects stand-alone mode, which is what we want (note that all of the dipswitches should be OFF). BIOS scans the I2C EEPROM, then initializes SIE2 for the debug tools, and then initializes SIE1 with the same default configuration.

We do not want BIOS to initialize SIE1 since our application program will do that with different parameters. We therefore need to change the default operation of BIOS by adding scan records to the I2C EEPROM. This is the subject of se2 (simple example #2). The se2 directory contains all of the files we need, and the main program, eeprom.s, is repeated in Figure 3-7 for discussion.

```
.section .init
; First define the code that needs to be loaded
; It will be prefixed with a Scan Header
        .short  ScanSignature
        .short  Length+2
        .byte   LoadCommand
        .short  _start
.global _start
_start:
; Initialize SIE2 for the GDB debugger.  Use the Cypress default
; configuration
        mov r2, 2              ; Choose SIE2
        mov r0, 0              ; Full speed
        int SUSB_INIT_INT      ; Let the BIOS initialize SIE2
; I now need to give control back to BIOS
    mov r15, 0x400        ; Reset the stack
        sti                    ; First time interrupts are enabled
        int IDLER_INT          ; This will not return
.equ    Length, .-_start
; Now define a scan record that will transfer control to my program
        .short  ScanSignature
        .short  2
        .byte   JumpCommand
        .short  _start
; Scan will not return so I need not have an EndOfScanRecords
```

Figure 3-7. Using scan records to modify BIOS operation

Since we only need SIE2 initialized, we must take control from BIOS using the two scan records shown in Figure 3-7. We first initialize SIE2 (most of the work is done by BIOS) for the debugger, and we must now own the idle chain or give control back to BIOS so that it will own the idle chain. I chose to restart the idle chain by resetting the stack pointer (R15) and calling BIOS using the IDLER_INT.

The makefile and linker script are included in the se2 directory for your review. Double click bash_env.bat in the se2 directory to open a bash window. Enter “make” at the bash prompt to build eeprom.bin.

The qtui2c utility program is used to download eeprom.bin into the I2C EEPROM. Recall that all of the dipswitches were initially OFF when we powered up the board. This brings it up in a default condition, which is necessary when we are going to program an EEPROM. Dipswitches 6, 5, 4, and 3 must now be set to ON to enable the I2C EEPROM. If you refer to the CY3663 Hardware User’s Manual that was installed with the kit CD-ROM, you will note that this switch configuration selects stand-alone mode with EEPROM 4 active. EEPROM 4 is currently unused, and so it is available for storing our code. In a bash window, enter “`qtui2c eeprom.bin f`” to program the EEPROM. With dipswitches 6, 5, 4, and 3 set the EZ-Host will read the scan records from the EEPROM at power on and implement them. Our scan records will cause BIOS to skip SIE1 initialization. Note how easy it was to change/augment the operation of BIOS using scan records. We will later place example code into the EEPROM so that it too runs on power on.

Simple Example #3 – Buttons and Lights Device

Now that we are familiar with the tools used to create an application program, and we know how to change/augment BIOS using scan records, we can tackle a USB device example.

Our first USB device will be a “Buttons and Lights” Human Interface Device (HID). I chose to use a standard class driver rather than a custom device driver since there will be less new topics to learn. BIOS includes default handling of standard USB requests using default descriptors. It also includes default handling of class requests (it stalls them all), so we will have a small amount of code to write to support the HID. This example will forward mezzanine board button presses to the PC via a HID input report. The seven-segment display will be controlled via HID output reports. A PC-based application is provided to test its operation.

The structure of simple example #3 (se3) is shown in Figure 3-8. The Init_Task must setup BIOS to use our descriptors and must configure Callback routines to implement HID-specific tasks that are not handled by BIOS. There is no idle task in this example since button-presses are handled by Frameworks on our behalf.

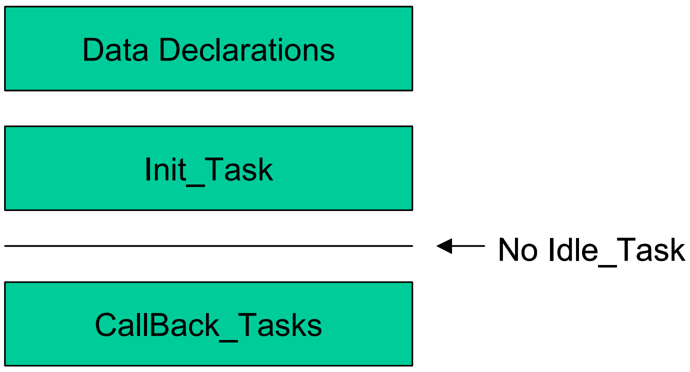


Figure 3-8. Structure of our first USB Device Example

Since this is our first USB example I want to move slowly to ensure that every facet is understood. We will discover that BIOS handles a lot of the USB specification requirements work for us, so our energies can be focused upon the application program.

Figure 3-9 shows the declaration of our USB device descriptors – this is a standard HID class declaration that specifies that HID input reports are supplied on endpoint 1 (EP1) and HID output reports are supplied on EP2. The report descriptor defines each report as a single byte.

```
// Define the descriptors for our "Buttons and Lights" HID example
USB_DEVICE_DESCRIPTOR const device_descriptor = {
    18, 1, 0x200, 0, 0, 0, 64, 0x4242, 0xc003, 1, 1, 2, 0, 1
};

uint8 const report_descriptor[] = {
    6, 0, 0xFF,  // Usage_Page (Vendor Defined)
    9, 1,        // Usage (IO Device)
    0xA1, 1, // Collection (Application)
    0x19, 1, //   Usage_Minimum (1)
    0x29, 8, //   Usage_Maximum (8)
    0x15, 0, //   Logical_Minimum (0)
    0x25, 1, //   Logical_Maximum (1)
    0x75, 1, //   Report_Size (1)
    0x95, 8, //   Report_Count (8)
    0x81, 2, //   Input (Data,Var,Abs) = Buttons
    0x19, 1, //   Usage_Minimum (1)
    0x29, 8, //   Usage_Maximum (8)
    0x91, 2, //   Output (Data,Var,Abs) = Lights
    0xC0     // End_Collection
};

USB_ALL_DESCRIPTORS const configuration_descriptor = {
    {   /* config_descriptor header */
        9, 2, sizeof(USB_ALL_DESCRIPTORS), 1, 1, 0, 0xC0, 1 },
    {   /* interface */
        9, 4, 0, 0, 2, 3, 0, 0, 3 },
    {    /* class_descriptor */
    9, 0x21, 0x100, 0, 1, 0x22, sizeof(report_descriptor) },
    {   /* EP1_In */
        7, 5, 0x81, 3, 8, 100 },
    {   /* EP2_Out */
        7, 5, 2, 3, 8, 100 }
};
```

Figure 3-9. Example 3 Descriptors

The Init_Task, shown in Figure 3-10, adjusts the set of pointers that BIOS uses to describe a USB device. We first change three data pointers so that BIOS will use our descriptors rather than its default descriptors. We then change three function pointers so that we modify BIOS's default operation. The new functions and callback routines that handle the IO are shown in Figure 3-11.

```
// Declare the app_init_task
void app_init(void) {
// Update the descriptor pointers that BIOS uses
    WRITE_REGISTER (SUSB1_DEV_DESC_VEC, (PFNINTHANDLER) &device_descriptor);
    WRITE_REGISTER (SUSB1_CONFIG_DESC_VEC, (PFNINTHANDLER) &configuration_descriptor);
    WRITE_REGISTER (SUSB1_STRING_DESC_VEC, (PFNINTHANDLER) &strings_descriptor);

// Chain a routine before BIOS's standard request handler
    BIOSStandardRequestHandler = (PFNINTHANDLER) READ_REGISTER (SUSB1_STANDARD_INT*2);
    WRITE_REGISTER (SUSB1_STANDARD_INT*2, (PFNINTHANDLER) &InterceptStandardRequest);

// Add a Class Request Handler
    BIOSClassRequestHandler = (PFNINTHANDLER) READ_REGISTER (SUSB1_CLASS_INT*2);
    WRITE_REGISTER ( SUSB1_CLASS_INT*2, (PFNINTHANDLER) &HandleClassRequest);

// We need to know when a Set_Configuration is received
    BIOSConfigurationChange = (PFNINTHANDLER) READ_REGISTER (SUSB1_DELTA_CONFIG_INT*2);
    WRITE_REGISTER ( SUSB1_DELTA_CONFIG_INT*2, (PFNINTHANDLER) &SetConfigurationRequest);

// Now initialize SIE1, this will result in it enumerating with the PC Host
    susb_init( SIE1, USB_FULL_SPEED );
}
```

Figure 3-10. Operation of BIOS is changed by replacing pointers

We first intercept a standard USB Request. BIOS does not know about HID class so does not check for GetDescriptor (Interface). We need to do this and, if we detect this command, then we must supply the HID Report Descriptor. If we do not detect this command then we call the BIOS handler that implements all of the other standard USB commands for us.

We use BIOS to send the report descriptor to the PC host. We create an information block that describes our data (it's address, length and a callback routine) and pass it to BIOS to send. In our case, length is only 1 byte – we can put any value in length and BIOS handles the transmission of multiple, endpoint–sized blocks for us. When all of the data has been sent, BIOS will use the CallBack routine to let us know. In this case of supplying a report descriptor we don't have anything special to do, so I specified a 0 which tells BIOS to handle the completion of the transfer on our behalf.

The Handle Class Report CallBack implements the HID specific functions of our application. There is not much to do in this simple example.

We need to know when our device has been successfully configured, so we intercept the BIOS (SUSB1_DELTA_CONFIG) software interrupt. In our routine we first allow BIOS to do its processing and then we can activate our data endpoints. Note that BIOS parsed our descriptors and did things like enable the correct hardware endpoints and other mundane "housekeeping" tasks.

We are interested in receiving reports from the PC host, so we setup a lights_report data structure to be ready to receive data. We specify a lights_report_received CallBack and give this task to BIOS to manage. When an output report arrives from the PC Host then BIOS will call our routine for us. In our lights_report_received routine we update our seven segment display and then post another callback with BIOS.

Frameworks will inform us, via the app_button_handler callback, when any button changes state. This HID device forwards this information as an input report to the PC Host. Our task is to format the data suitable for BIOS, and we can then use BIOS to send the report. We fill out a data information block and pass its address to BIOS.

Notice that our example code only deals with the high-level details of our “buttons and lights” application. The low-level enumeration and sending and receiving of reports are handled by BIOS on our behalf. If you are someone who likes to see ALL of the instructions that get executed then the source of BIOS is available from Cypress upon request (but, be warned, it is in very tightly packed CY16 assembler code). The code implements the sequences and protocols as required by the USB specification. It is not all that exciting. You can change it if you really want to, using SCAN records, but I would not recommend that. It is more productive for you to put energy into your USB application.

```
void SetConfigurationRequest(void) {
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE1_DEV_REQ;
// Let BIOS handle this request first
    BIOSConfigurationChange();
// If I got configured then I can enable my data endpoints
    if ((req->wValue & 0xFF) ==
         configuration_descriptor.config_header.bConfigurationValue) {
         Configured = TRUE;   cpld_set_led(SLAVE_LED);
// Ask BIOS to inform me when a lights report is received
         setup_lights_report_callback();
         }
    else {
         Configured = FALSE;  cpld_clr_led(SLAVE_LED);
         }
    }


void InterceptStandardRequest(void) {
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE1_DEV_REQ;
// BIOS does not handle GetDescriptor(Interface) so check for that
    if ((req->bRequest == USB_GET_DESCRIPTOR_REQUEST) && ((req->bmRequestType & 3) == 1)
         && (req->wValue == 0x2200) ) {
         report_descriptor_info.buffer = &report_descriptor;
         report_descriptor_info.length = sizeof(report_descriptor);
         report_descriptor_info.done_func = 0;      // Let BIOS handle completion
         susb_send ( SIE1, 0, &report_descriptor_info);
         }
    else BIOSStandardRequestHandler();    // Pass the request on to BIOS to handle
    }
void HandleClassRequest(void) {
// Manage the HID Class Requests for the "Buttons and Lights" device
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE1_DEV_REQ;
    switch(req->bRequest) {
        case USB_HID_SET_REPORT_REQUEST:  // Should not get this since I declared EP2_Out
        case USB_HID_GET_REPORT_REQUEST:  // Should not get this since I declared EP1_In
        case USB_HID_SET_IDLE_REQUEST:    // Optional command, not supported
        case USB_HID_GET_IDLE_REQUEST:    // Optional command, not supported
        case USB_HID_GET_PROTOCOL_REQUEST:// Will not get this, we are not a boot device
        case USB_HID_SET_PROTOCOL_REQUEST:// Will not get this, we are not a boot device
        default: BIOSClassRequestHandler();
        }
    }


void buttons_report_sent(void) {
    buttons_report_inuse = FALSE;         // Reuse the same information block
    }


void app_button_handler(BUTTON button) {
// Report this change in button state to the host if we are configured
    if (Configured && !buttons_report_inuse) {
        buttons_report_info.buffer = &buttons_report;
        buttons_report_info.length = sizeof(buttons_report);
        buttons_report_info.done_func = buttons_report_sent;
        buttons_report = button;
        susb_send (SIE1, 1, &buttons_report_info);
        buttons_report_inuse = TRUE;
        }
    }


void lights_report_received(void) {
// Update my display with the new lights value and wait for the next update
    cpld_set_ssd( lights_report );
    setup_lights_report_callback();
    }


void setup_lights_report_callback() {
// Ask BIOS to alert us when the next report is received
    lights_report_info.buffer = &lights_report;
    lights_report_info.length = sizeof(lights_report);
    lights_report_info.done_func = (PFNINTHANDLER) &lights_report_received;
    susb_receive ( SIE1, 2, &lights_report_info);
    }
```

Figure 3-11. The CallBack routines needed for Example 3

Simple Example #4 – BAL Host Program

Just as we did in the first example, double click on bash_env in the se3 directory to create a bash window. Enter "`make`" to build the example then "`cy16-elf-gdb se3`" to start the debugger. Click the RUN icon and note that the debugger automatically inserted a breakpoint at Main ().

Connect a second USB cable from the SIE1 peripheral 1A connector of the mezzanine board to a target PC as shown in Figure 3-12. I show two PCs, a target PC and a development PC for clarity, but they could be the same PC running programs in different windows.

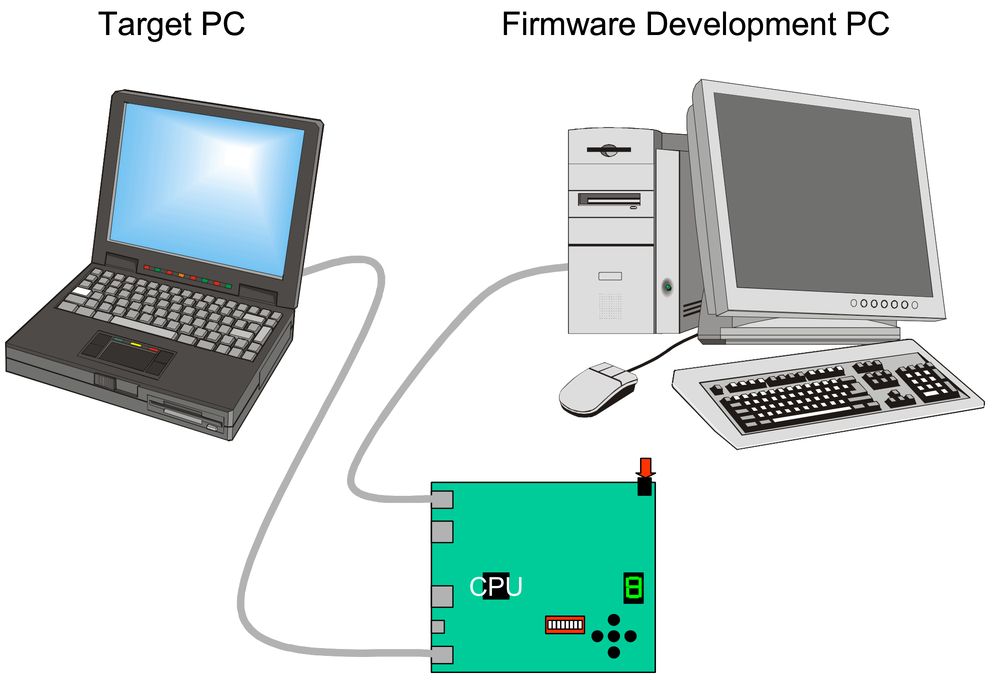
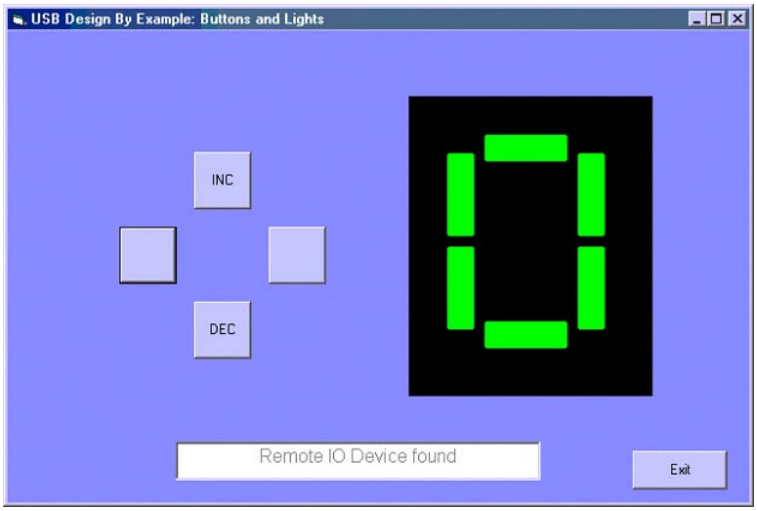


Figure 3-12. Using two PCs to test Example 3

Copy BAL.exe from the se4 directory onto your target PC and start this now. I wrote this PC host test program in Visual Basic since this was the easiest way to get the friendly display shown in Figure 3-13. The source code is available in the se4 directory. The display buttons and the real buttons operate as an OR while the display seven–segment display and the real seven–segment display operate as an AND. Pressing any button will increment both displays.

On the development PC clear all breakpoints and click “continue.” The example 3 program will run, and it will start enumerating on the target PC. It is a standard HID so no special INF file or other operating system software is required. The “Buttons and lights” application will find the newly installed device, and the buttons and display will be operational. This may take several seconds.

Our first EZ-Host device example code is running.



Remote IO Device found

Figure 3-13. Display from Target PC “Buttons and Lights” program

Simple Example #5 – Standalone BAL Device

To round out this chapter, I wanted to introduce you to another tool – scanwrap. Scanwrap takes a binary file and creates a scan record from it. This can be loaded into the I2C EEPROM directly. I recommend programming the EZ-OTG board so that we can use it in subsequent examples. Dipswitches 6, 5, 4, and 3 should be the only ones ON to enable the EEPROM. Double click on bash_env in the se3 directory to open a bash window. Enter “scanwrap se3.bin scan_se3.bin 0x1000” then program the EEPROM using “`qtui2c scan_se3.bin f`.” Once the EEPROM is programmed, the mezzanine board becomes a standalone “Buttons and Lights” device – we shall call this simple example #5 (se5), and it will be used in other examples later in this book.

Chapter Summary

We moved through four simple examples to get used to the EZ-Host/EZ-OTG tools and development environment. We built a standard HID device, and used a PC Host application-program with standard Windows class-drivers to test it.

Building a USB device with the EZ-Host/EZ-OTG components is straight forward since BIOS handles all of the low-level USB details for us. We can focus upon the application layer details. In this example we only had a single device and used the other SIE connection for the debugger – later examples will support multiple independent devices on two independent USB segments.

In the next chapter, we will build a host application.

Chapter 4: Developing a host application

A USB host controller has more work to do than a USB device controller, a **lot** more. USB is a master-slave bus and the host, as master, will control all of the communications. This will involve detecting and identifying connected devices, and then scheduling USB transfers as defined by the requirements of each device. The structure of a host application program is the same as a device application program (refer to Figure 3-3) so we can focus on the functionality of our application program and allow Frameworks and BIOS to handle all of the low-level details of enumeration and scheduling. Since the Frameworks code is pre-written and tested it will not take too much extra effort on our part to write a host applications program.

We shall work through a “Buttons and Lights” host example – essentially replacing the PC and the Visual Basic program that was demonstrated in the previous chapter. We will use the other mezzanine board, and it will talk to the standalone Buttons and Lights HID device we built in example se3.

Most of this chapter describes the inner workings of the Frameworks host controller firmware in sufficient detail that you could modify or tune some parts of it for your particular application. In general, this will NOT be required and many developers will just use Frameworks as is, so these readers may skim the first half of this chapter.

Key Host Controller Concepts

Figure 4-1 shows a comparison between a re-programmable USB host controller, such as a PC, and an embedded host controller as would be implemented using the EZ-Host/EZ-OTG components. The Frameworks code uses many of the same concepts as a Windows-based host implementation.

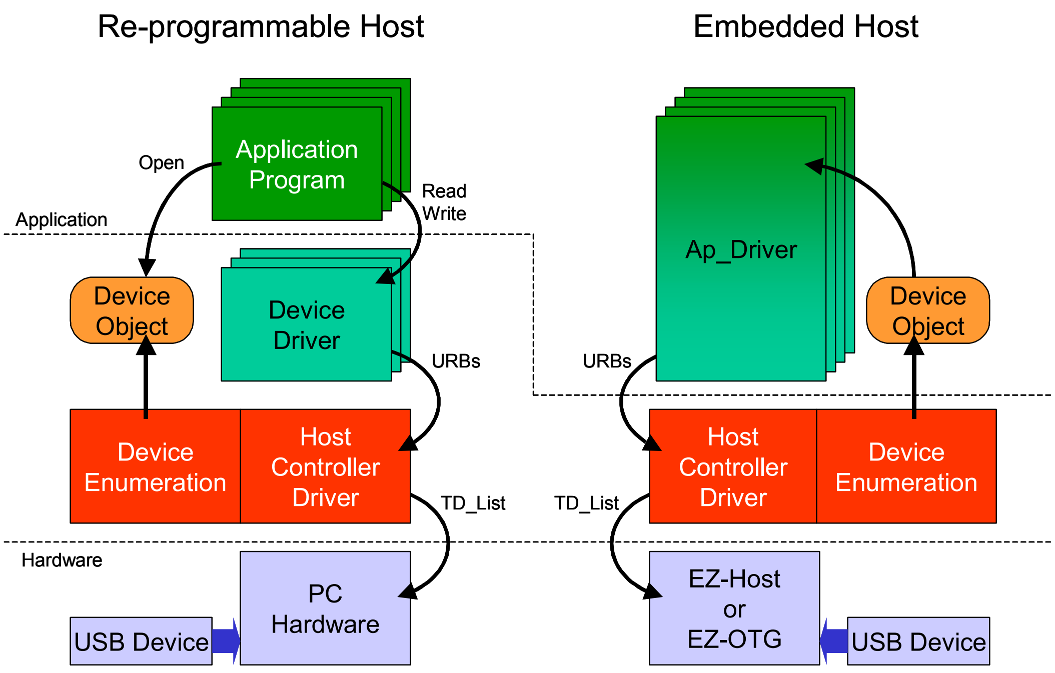


Figure 4-1. A re-programmable host and an embedded host have similar structure

First focus on the USB device being added to the PC at the bottom left of Figure 4-1. The PC will enumerate this device to discover which device driver it should use to manage it. I am assuming the simplest case here where the USB device has a single interface. Many devices have multiple interfaces and will therefore cause multiple device drivers to be loaded by the PC host. We will consider this case later, so, for now, assume that only one device driver is loaded. The operating system will call Start_Device() in the device driver, and a device object will be created. This device object describes all of the attributes of the USB device in a format that is useful for the operating system.

In the PC environment the process stops here. We have a USB device added and a device driver loaded to support this device. The device driver typically doesn't do anything unless requested by an application program, so lets start an applications program that will use this new USB device. Windows has a comprehensive scheme involving GUIDs and Plug-And-Play system tables that allow a user program to rendezvous with the device driver and thus open it (i.e. use Createfile to return a file handle). This scheme, although complex, is very flexible since it allows multiple user programs to share device drivers.

Once the Windows application program has a handle to the device driver it can issue ReadFile, WriteFile, and IOControl commands. The device driver will, in turn, create USB Request Blocks (URBs) that it passes to the USB host Controller driver. The windows host controller driver will accept URBs from many device drivers, and it creates a Transfer Descriptor List (TD_List) that defines the USB operations that must be executed to implement the URBs. USB transfers are scheduled in frames that are 1 msec apart. The host controller software is preparing the *next* frame while the host controller hardware is implementing the *current* frame. The host controller driver passes this TD_List to a host controller component that uses specialized hardware to process this list.

The host controller component processes the TD_List and updates status information indicating the success or failure of each transfer. The host controller driver will use this information as it builds the TD_List for the next frame.

There are three standard USB host controller interfaces, UHCI (Universal Host Controller Interface), OHCI (Open HCI) and EHCI (Enhanced HCI). UHCI and OHCI implement 12 Mb/s signaling while EHCI also includes 480 Mb/s signaling and the protocol enhancements, such as PING and SPLIT, that were added in the USB 2.0 Specification. All three interfaces are documented at www.usb.org.

Windows implements the ReadFile system call as a blocking call. This means that the application program must wait for the read to complete before continuing. So it waits for the device driver to issue a URB, it waits while the TD_List is being built and sent, and it waits while the device is returning NAKs. Eventually the device will send data and the corresponding TD will be marked complete. This propagates all of the way up the stack and the application program can continue.

The operation of an EZ-Host/EZ-OTG embedded host is similar. Frameworks will identify a newly attached device and will create a Device Object for it. Frameworks will then try to match this device object with a list of device drivers that it owns – if a match is found then Frameworks will call Start_Device() in this driver. In an embedded host the application program and device driver are not separate entities, they are combined into a single module which, for explanation purposes, I shall call Ap_driver. The implication for us, since we want to write a host application program, is that we must follow a certain structure – this will be covered in detail in an example later in this chapter.

An embedded host can contain many Ap_driver modules and can thus support many different devices. Since the code space on an embedded host is typically limited, the embedded host will declare a targeted peripheral list that describes the devices that it can support. Our first example supports a single device while later examples will support more.

Once our ap_driver module starts running it will create URBs that it forwards to Frameworks for processing. A Frameworks URB contains a callback routine that ap_driver specifies. Once the URB is completed, this routine is called. The ap_driver can do other processing in the meantime. Frameworks builds a TD_List just like the PC host does – Frameworks uses the scheduling algorithms defined in the UHCI specification. Frameworks passes the TD_List to BIOS for processing, and BIOS will later return the list with updated status. BIOS uses specialized hardware in the EZ-Host/EZ-OTG components to transfer the contents of the TD_List onto the USB wires. Frameworks, meanwhile, creates a TD_List for the next frame.

Frameworks Host Controller Implementation

Host applications developers can skip this section; this section goes into more detail on how the Frameworks host controller firmware is implemented. This section covers key Frameworks data structures and describes memory and buffer allocation. If you are reading through the many source files in the Common directory with the intentions of knowing the finer details, then you will find this section most useful.

The Frameworks host controller firmware has to manage a great deal of information. Data blocks are routinely created, passed through many levels of software and then destroyed. Rather than continually copying data from one place to the next, Frameworks defines fixed format buffers that it allocates and de-allocates. The lifetime of a data buffer varies greatly, so a linked-list is used so that allocations and de-allocations need not occur in order. This linked-list implementation has the side benefit that all allocations and de-allocations take the same fixed time to execute – an important attribute in an embedded real-time system. Frameworks keeps data-copying to an absolute minimum and manages data pointers extensively. Figure 4-2 shows the approximate size and number of fixed-format buffers, all of which are initialized as “free.” Different application programs will use differing numbers of data buffers, so the count for each buffer type is defined in fwxcfg.h.

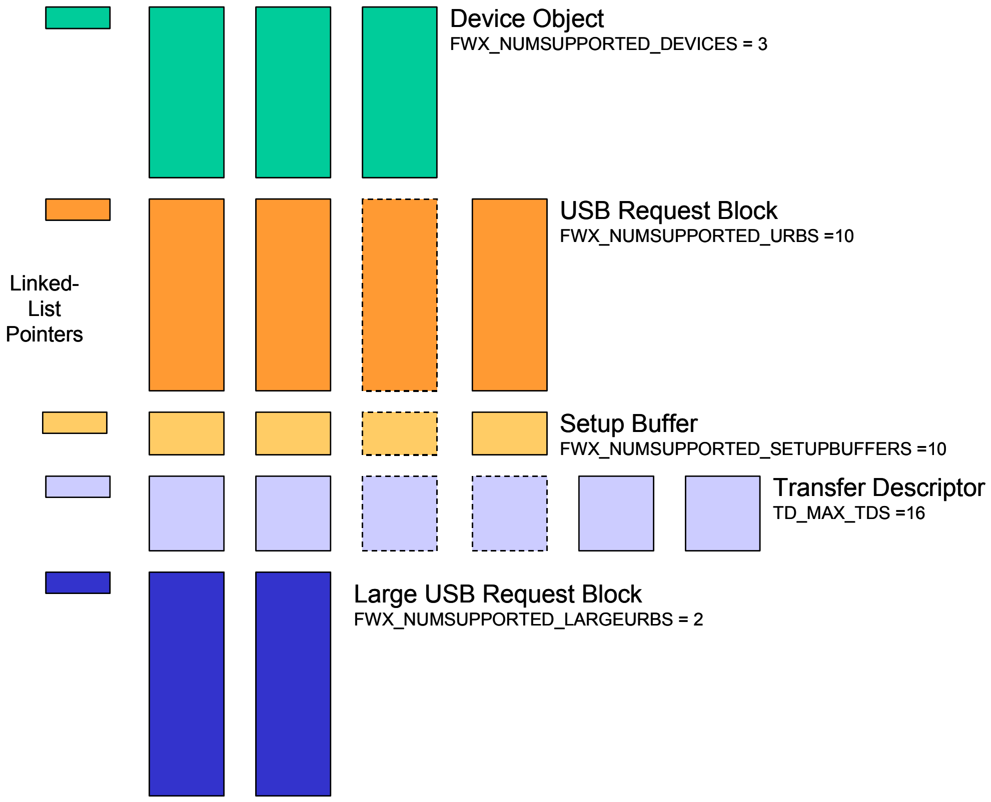


Figure 4-2. Variable Data uses pre-allocated, fixed-format buffers.

Following a power-on, Frameworks will look for devices attached to its root hubs. In our example we will continue to use SIE2 as a debugger connection, so we will only consider a single host example using SIE1 (dual host is covered in a later chapter). If a device is found then Frameworks will create a device object, shown in Figure 4-3, and will start to fill in information about the device.

```
typedef struct USB_DEVICE {
    uint8                   sie;
    uint8                   port;
    uint8                   address;
    bool                    direct_connect;
    uint8                   speed;
    uint8                   configuration;
    uint8                   num_endpoints;
    uint8                   endpoint_max_pkt[8];
    uint8                   enum_state;
    uint8                   enum_retry;
    uint8                   otg_attributes;   /* bitmap. */
    struct CLASS_DRIVER     *driver;
    USB_DEVICE_DESCRIPTOR   dev_descr;
    USB_CONFIG_DESCRIPTOR   cfg_descr;
    USB_INTERFACE_DESCRIPTOR inf_descr;
    uint16                  enum_data;
    uint8                   new_address; /* temporary. */
} USB_DEVICE;
```

Figure 4-3. Format of a Frameworks Device Object

Frameworks will use USB Request Blocks (URB) to get information from the new device. The format of this key data structure is shown in Figure 4-4. Our applications code will use the same mechanism to request services from Frameworks.

```
typedef struct URB {
    struct  USB_DEVICE *dev;  /* pointer to associated USB device */
    uint8   dir;              /* in or out end point */
    uint8   usb_dev_addr;     /* device_address */
    uint8   endpoint;         /* end point number */
    uint8   speed;            /* 0 = full or 1 = low speed */
    uint8   type;             /* end point type. iso, intr, control, bulk */
    void   *transfer_buffer;  /* associated data buffer */
    int16   buffer_length;    /* data buffer length */
    int16   actual_length;    /* actual buffer length */
    int16   bandwidth;        /* allocated bandwidth */
    uint8  *setup_packet;     /* setup packet (control only) */
    int16   num_of_packets;   /* number of packets in this request (iso only) */
    int16   interval;         /* polling interval (irq only) */
    int16   error_count;      /* number of errors in this transfer */
    uint16  status;           /* returned status */
    PFNURBCALLBACK callback;  /* callback for URB completion */
    struct URB  *next;        /* To facilitate easy linking of URBs. */
} URB;
```

Figure 4-4. Format of a Frameworks USB Request Block (URB)

Frameworks will create one, or more, Transfer Descriptors (TD) from this URB. The format of a TD exactly matches the format used by the EZ-Host/EZ-OTG hardware and contains several bit fields as shown in Figure 4-5.

```
typedef struct TD {
    uint16      base_address;          /* Base Address of Data Buffer */
    uint16      length       : 14;     /* Port Number /Data Length */
    uint16      port_num     :  2;
    uint16      ep           :  4;
    uint16      pid          :  4;
    uint16      dev_address  :  8;
    uint8       control;
    uint8       status;
    uint16      retry_cnt       : 2; /* Retry Count */
    uint16      retry_xfer_type : 2;
    uint16      retry_active    : 1;
    uint16      unused_3        : 3;
    uint16      residue         : 8; /* Residue */
    struct TD   *next_TD;              /* Points to Next TD Address */
    URB         *urb;                  /* Pointer to URB origin */
    uint16      ctrl_next_state;       /* control pipe state */
    uint16      interval_reference;    /* frame count when TD was last scheduled */
    uint16      nak_retry_count;       /* Limit times a TD is retried after NAK. */
} TD;
```

Figure 4-5. Format of EZ-Host/EZ-OTG Transfer Descriptor

Frameworks will place (a) token(s) for this/these TD(s) onto one of the lists marked NEW in Figure 4-6; it will choose the list that matches the USB packet type. We are now at the heart of the Frameworks host controller firmware. Figure 4-6 shows the multiple input Transfer Descriptor lists (TD_lists) and the single output TD list that Frameworks uses to schedule transfers on the USB wires. Multiple lists are required to manage the scheduling algorithm defined by the UHCI specification – isochronous transfers have the highest priority followed by ready interrupt transfers, and these are allowed up to 90% of the bus bandwidth. If there is time available in the frame then control and bulk transfers will be included.

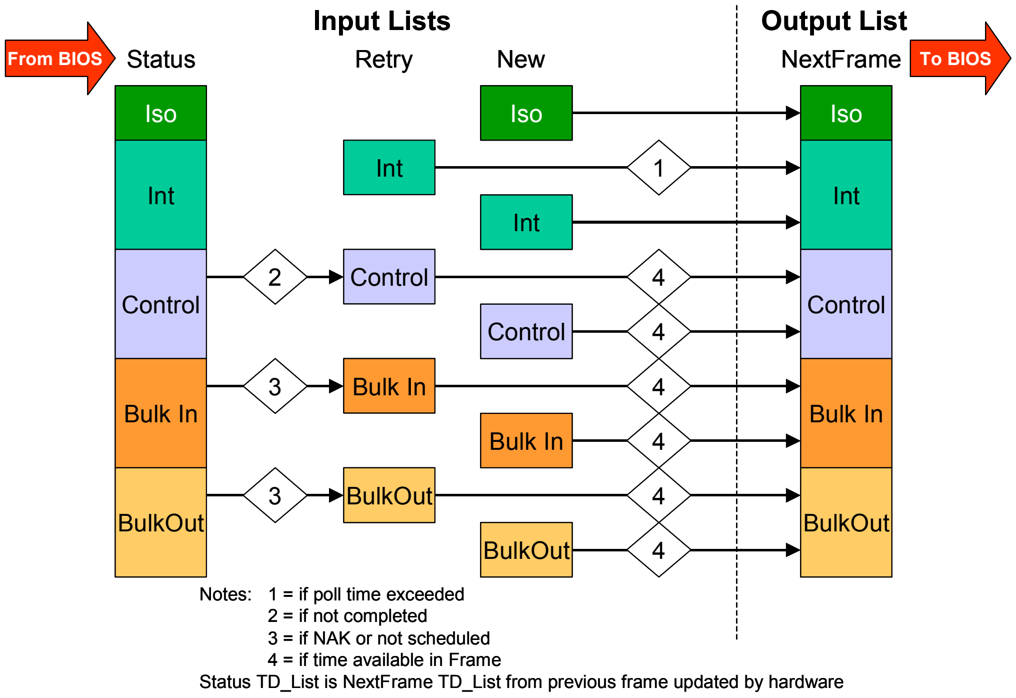

Input Lists
Output List
From BIOS
Status
Retry
New
NextFrame
To BIOS
Iso
Int
Control
Bulk In
BulkOut
1
2
3
4
Notes: 1 = if poll time exceeded
2 = if not completed
3 = if NAK or not scheduled
4 = if time available in Frame
Status TD_List is NextFrame TD_List from previous frame updated by hardware

Figure 4-6. Scheduling Transfers in a frame

Let us assume that the EZ-Host/EZ-OTG has just completed a series of data transfers on USB and that BIOS has updated the status information in the TD_List; I call this the Status_List in Figure 4-6. Frameworks first parses the Status_List looking for successful transfers (it executes the matching callback) or for failed transfers (it places the TD on to the matching Retry_List). It then builds the NextFrame TD_List by adding active items from the New or Retry lists. It processes the lists in the order shown (from top to bottom) and stops either when it has calculated that the frame will be full, or it reaches the end of the input lists. Frameworks then gives this TD_List to BIOS so that it will use specialized hardware within the EZ-Host/EZ-OTG components to move these onto the USB wires. BIOS collects status from these transfers and we repeat this sequence (i.e. re-read this paragraph) until the host is powered down.

So lets elevate from these low-level scheduling details a moment. How did we get here? Yes, an initialization task within Frameworks detected a device present on its root hub and sent URBs to discover the identity of the device. Sending URBs took us to TDs, which took us to scheduling and into the heart of the Frameworks host controller firmware. Notice in the URB structure (shown in Figure 4-4) that the caller specifies a callback routine that is used once the request is completed. The application programmer does not need to know any of these low-level details since the callback function will eventually be called.

Device Identification

Frameworks will use a series of URBs to enumerate the device. It will typically use GetDescriptor(Device), GetDescriptor(ConfigurationHeader), GetDescriptor(Configuration) and will use this gathered information to select a device driver, or in our case, an Ap_driver. All embedded hosts will support a targeted peripherals list, and Frameworks implements this list as an array of registered drivers. Each registered driver is defined by a CLASS_DRIVER structure, shown in Figure 4-7, and this contains pointers to our Ap_driver code which is shown in Figure 4-8.

```
typedef struct CLASS_DRIVER {
    uint8   class;
    uint8   subclass;
    uint8   if_class;                  /* Interface class. */
    uint8   if_subclass;               /* Interface subclass. */
    uint8   protocol;
    uint16  vendor_ID;
    uint16  product_ID;
    uint16 (*start)(USB_DEVICE *dev);  /* Callback for Start_Device */
    uint16 (*stop)(USB_DEVICE *dev);   /* Callback for Stop_Device */
    void (*run)(USB_DEVICE *dev);      /* Callback for idle time processing */
    uint16 (*ioctl)(USB_DEVICE *, uint16, uint16, uint16 );
    uint16  id;                        /* System wide unique device ID. */
} CLASS_DRIVER;
```

Figure 4-7. Ap_driver is described by a Class_Driver structure

Note that the structure of Ap_driver is the familiar three-task model: an Init_Task (that sets up the Idle_Task and the CallBacks), an Idle_Task (that looks for work to do), and a collection of CallBack routines that do the work.

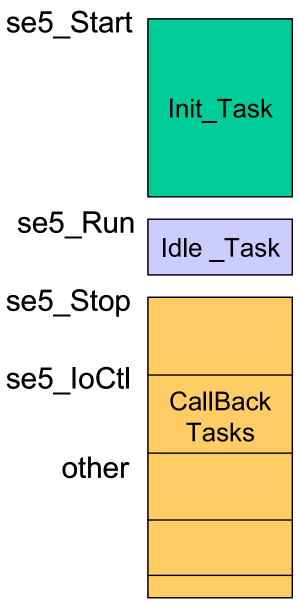


Figure 4-8. Structure of Ap_driver code

The first time our application program knows about the device that it needs to manage is when it receives a call from Frameworks to our Init_Task – the low-level enumeration and driver matching has already been done by Frameworks so we can start our application straight away.

Simple Example #6 – Buttons and Lights Host

The host-side application for our "Buttons and Lights" example is straightforward. We have to create a task that polls the "Buttons and Lights" USB device for input reports, and we have to create a task to monitor local button presses. A single interrupt URB with a callback will wait for, and alert us to, input reports, and we'll use the same button callback routine that we used in the device example. Updating the seven-segment displays is also easy – the local one is a call into the Frameworks IO subsystem and the remote update is a single URB posted to Frameworks. The code for the se6 application is shown in Figure 4-9. As you can see, most of the code deals with initialization.

```
/* File: app.c for Simple Example 6
 * This is a HOST running a BAL program
 */
#include "fwx.h"
#include "board.h"
#include "app.h"
/* Application data. */
// Provide 1 byte lights report on a button change, receive a 1 byte buttons report
uint8 buttons_report ATTR_USB_XFER_BUF_SECTION;
uint8 lights_report  ATTR_USB_XFER_BUF_SECTION;
int DisplayValue;

// My driver will need two URBs - I allocate them in Start_Driver and re-use them
URB *buttons_urb, *lights_urb;
bool lightsUrbInUse = FALSE;

// Describe my driver is a Frameworks compatible format.  Only one may be active
bool DriverInUse = FALSE;
CLASS_DRIVER const se6_driver = {
        0,               // class
        0,               // subclass
        3,               // if_class
        0,               // if_subclass
        0,               // protocol
        0x4242,          // vendor_ID
        0xc003,          // product_ID
        se6driver_start, // (*start)( USB_DEVICE *dev )
        se6driver_stop,  // (*stop)(void)
        se6driver_run,   // (*run)(void)
        se6driver_ioctl, // (*ioctl)( USB_DEVICE *, uint16, uint16, uint16 )
};

uint16 show_error(uint16 error) {
// Helper routine to display errors, should not get any!
        cpld_set_led(ERROR_LED);
        cpld_set_ssd(error);
        return ERROR;
        }

// Declare my Init_Task - this is Driver_Start
uint16 se6driver_start(USB_DEVICE *dev) {
// Frameworks will pass me a driver object for the BAL HID device
// Only allow one copy of the driver to run
    if (DriverInUse) return ERROR;
    DriverInUse = TRUE;
// Get two URBs needed for the interrupt reports
        buttons_urb = alloc_URB(FALSE, sizeof(buttons_report) );
        if (!buttons_urb) return show_error(0xA);
        lights_urb = alloc_URB(FALSE, sizeof(lights_report) );
        if (!lights_urb) return show_error(0xB);
// I have two URBs, initialize them
// NOTE: since I know the device then I know attributes such as endpoint/polling interval
// In the general case I would parse the descriptors to discover this information
// Initialize those elements of the urb that are constant
    lights_urb->dev = dev;
    lights_urb->dir = TD_CTRL_DIR_OUT;
    lights_urb->usb_dev_addr = dev->address;
    lights_urb->endpoint = 2;                             // See NOTE
    lights_urb->speed = dev->speed;
//  lights_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    lights_urb->type = USB_BULK_TRANSFER_TYPE;
    lights_urb->interval = 100;                           // See NOTE
```

```
// Initialize those elements of the urb that are constant
    buttons_urb->dev = dev;
    buttons_urb->dir = TD_CTRL_DIR_IN;
    buttons_urb->usb_dev_addr = dev->address;
    buttons_urb->endpoint = 1;                              // See NOTE
    buttons_urb->speed = dev->speed;
    buttons_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    buttons_urb->interval = 100;                            // See NOTE
    buttons_urb->transfer_buffer = &buttons_report;
    buttons_urb->buffer_length = sizeof(buttons_report);
    buttons_urb->callback = (PFNURBCALLBACK) buttons_report_received;
// Post the buttons_urb to wait for an input report from the device
    if (td_submit_URB(buttons_urb) == ERROR) {
        release_URB(buttons_urb);
        return show_error(0xC);
        }
    else return SUCCESS;
    }


//  Declare Idle_Task - no work to do, all handled via callbacks


//  Declare Callback routines - the three required driver routines first
uint16 se6driver_stop(USB_DEVICE *dev) {
    if (!DriverInUse) return ERROR;
    DriverInUse = FALSE;
// Release any system resources we have
    if (buttons_urb) {
        td_clear_URB(buttons_urb);        // Remove from td processor
        release_URB(buttons_urb);         // Deallocate the urb
        }
    if (lights_urb) {
        if (lightsUrbInUse) td_clear_URB(lights_urb);
        release_URB(lights_urb);
        }
    return SUCCESS;
    }

void se6driver_run(USB_DEVICE *dev) {
// Nothing to do in this IdleTask
    }

uint16 se6driver_ioctl(USB_DEVICE *dev, uint16 cmd, uint16 d1, uint16 d2) {
// Nothing to do since no IOCTLs defined
    if (DriverInUse) return SUCCESS;
    return ERROR;
    }

// Handle local button presses
void app_button_handler(BUTTONbutton) {
    switch (button) {
        case BTN_UP: if (++DisplayValue > 9) DisplayValue = 0; break;
        case BTN_DOWN: if (--DisplayValue < 0) DisplayValue = 9; break;
        case BTN_LEFT: DisplayValue = 0; break;
        case BTN_RIGHT: DisplayValue = 9; break;
        default: break;
        }
    cpld_set_ssd(DisplayValue);
// Need to keep the Device display in sync
    lights_report = DisplayValue & 0x0F;
    if (!lightsUrbInUse) {
        lights_urb->transfer_buffer = &lights_report;
        lights_urb->buffer_length = sizeof(lights_report);
        lights_urb->callback = (PFNURBCALLBACK) lights_report_sent;
        if (td_submit_URB(lights_urb) == ERROR) {
            release_URB(lights_urb);
            show_error(0xD);
            }
        else lightsUrbInUse = TRUE;
        }
    }

// Handle remote button presses
void buttons_report_received(URB *urb) {
    if (urb->status == SUCCESS) {
// Device has just sent me a button press, treat it as a local button press
        app_button_handler(buttons_report);
// Frameworks will continue to monitor INT-IN, I don't need to resubmit the urb
        }
    else {
// There was an error on the urb, so Frameworks will stop scheduling it
// I should try and resubmit it to keep looking for INT-IN
        urb->transfer_buffer = &buttons_report;
        urb->buffer_length = sizeof(buttons_report);
        urb->callback = (PFNURBCALLBACK) buttons_report_received;
        if (td_submit_URB(urb) == ERROR) {
            release_URB(urb);
            show_error(0xC);
            }
        }
    }

// Remote lights update report has been sent
void lights_report_sent(void) {
// Now OK to reuse lights urb
    lightsUrbInUse = FALSE;
    }
```

Figure 4-9. Host application code for simple example #6

There are other files in the se6 directory. The fwxcfg.h file selects a lot more features from the Frameworks subsystem. You can open a bash window and enter "`make`" to build an executable file. Just as before, open another bash window for "`cy16-elf-libremote -u`" and enter "`cy16-elf-gdb se6`" in the first bash window to start the debugger. You should have your hardware set up as shown on Figure 4-10. We are debugging se6 on the EZ-Host mezzanine board; we are using a PC to control the EZ-Host board via a USB cable connected to SIE2; the EZ-OTG mezzanine board with se5 installed in EEPROM will operate as a USB BAL device.

Clear the debugger breakpoints and click "continue." Now press the buttons on the host mezzanine card or the device mezzanine card and watch the seven-segment displays on each of them change.

Our first host application program.

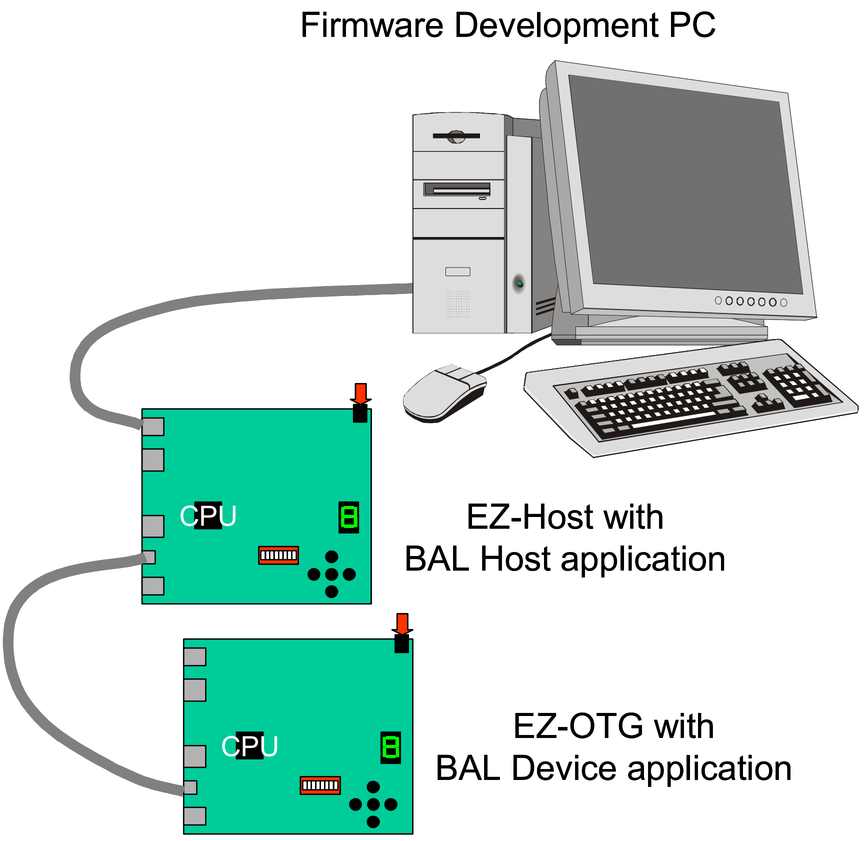


Figure 4-10. Testing the se6 host example

Chapter Summary

The structure of a host application program is the same as the structure of a device application program. The application program focuses on the design logic that we want to implement, and all of the low-level USB interaction, including device identification and transaction scheduling, is handled by Frameworks. We looked inside the operation of the Frameworks host controller firmware and saw how it schedules transfers using pre-allocated, fixed-format buffers and the UHCI algorithm.

In the next chapter we will combine the host and device application programs into one program to build a "two-headed" application that supports host and device functionality concurrently. You will be impressed with what this subsystem can do.

Chapter 5: Concurrent operation as a host and device

As it happens, the concurrent operation of the EZ-Host/EZ-OTG as a host and as a device is simpler to explain than the role-changing operation of a dual-role device, so I decided to cover this topic first. In reality we have already used this mode of operation of the EZ-Host/EZ-OTG since the debugger connection that we have been using is a USB device! But this debug channel has been handled solely by low-level BIOS routines and has therefore not been visible to our applications programs. In this chapter our example program will use both SIEs and we will need to find another method to attach our debugger (several are available).

Figure 5-1 shows the general arrangement of an EZ-Host/EZ-OTG application as a host and as a device. Note that, from this overview, it resembles a hub.

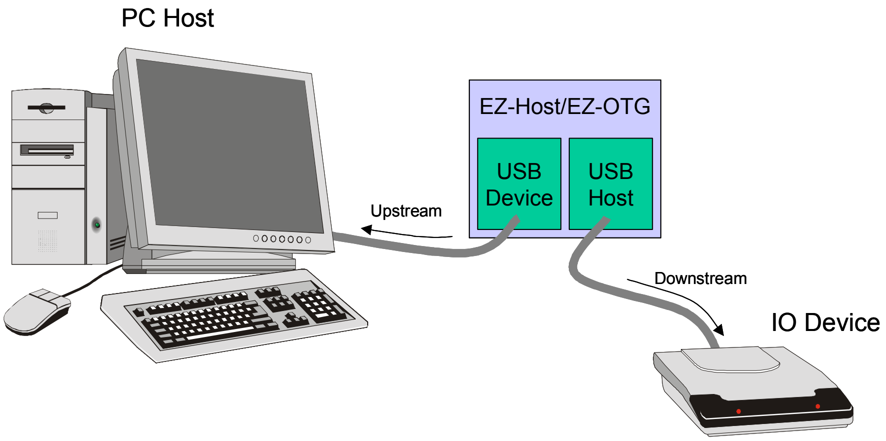


Figure 5-1. Concurrent operation as a host and as a device

The host+device and a hub are similar in that they both have an upstream port facing a PC and a downstream port facing a device. Their operation is quite different however, but we will take advantage of this similarity in a later example.

I am reusing my Buttons and Lights example for this chapter. It has the advantage of being a simple application so that you can focus on the methods and process. You should already be familiar with its operation, so you will be able to focus on the new elements of the examples. It is also readily extensible to other class examples so it is a good learning vehicle. The example we shall build in this chapter is shown in the center of Figure 5-2.

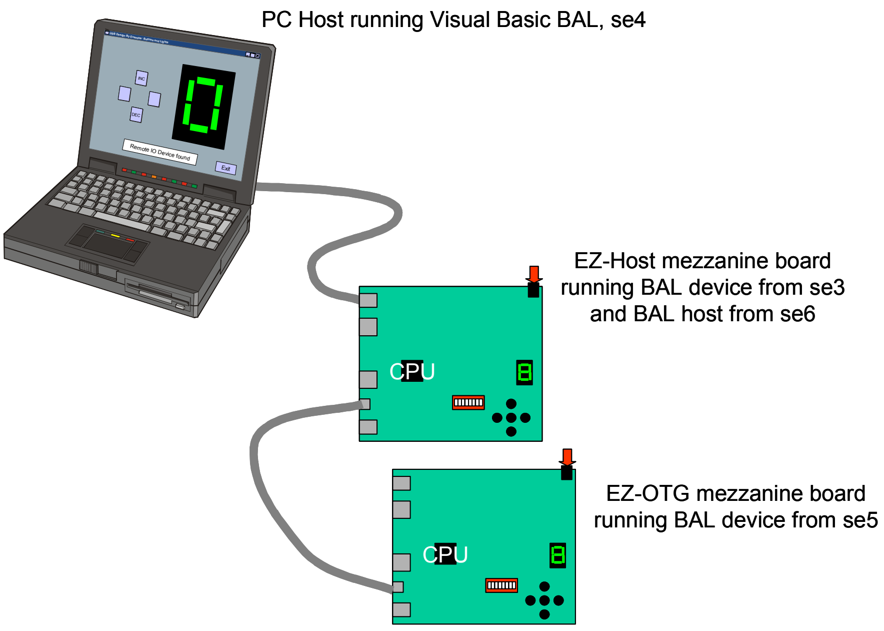


Figure 5-2. Buttons and Lights example with two hosts and two devices

We will use a PC running the Visual Basic Buttons and Lights program from Chapter 3. We will also use an EZ-OTG mezzanine board running as a Buttons and Lights device from Chapter 3 (= se5). Our target will be the EZ-Host mezzanine board running both the BAL host and the BAL device applications programs. We have most of the code for our host+device example, so we “just” have to integrate it into a single application program.

Our first design decision comes when we look at the target for our example. The mezzanine board only has one set of buttons and one seven segment display. Should I share the hardware amongst the two programs or should I add more hardware? This is a trivial example of a more complex issue – should the host application program and the device application program be independent of each other or should they cooperate to solve the current design challenge? We are in total control here – we can simply pass packets on the upstream segment to the downstream segment (similar to a hub) or we could process the data in both directions and selectively forward packets in either direction (an intelligent hub?).

In this first example I decided to share the host+device lights but not the buttons. The seven-segment display will be updated from the PC host OR from the BAL device. A button press will be passed to the PC host but not to the BAL device. This design decision will result in the three sets of buttons (PC host, host+device and device) each having a different effect. In the next example I shall change the algorithm such that all three displays stay in sync.

Simple Example #7 – Concurrent BAL Host and Device

We wrote the code for the Buttons and Lights host program in Chapter 4 – we will reuse it. We also wrote the code the for Buttons and Lights device program in Chapter 3 – we need to change this so that it uses SIE2 rather than SIE1 (since this is being used by the host program). The combined structure of Init_Task, Idle_Task and CallBack_Tasks is shown in Figure 5-3. I also edited fwxcfg.h in the se7 directory to include host and device features from Frameworks.

```
/* File: app.c.           Simple Example 7
 * BAL host+device,       host on SIE1, device on SIE2
 */

#include "app.h"
#include "sie1.h"

/* Application data. */
// Declare the Host data first
// Provide a 1 byte lights report on a button change, receive a 1 byte buttons report
uint8 host_buttons_report ATTR_USB_XFER_BUF_SECTION;
uint8 host_lights_report  ATTR_USB_XFER_BUF_SECTION;
int DisplayValue = 0;

// Support both host ports on SIE1
// Port changes are detected in an ISR and serviced in the IdleTask
// In this example se5 is directly connected to se6
// It can be attached on either host port (try it!)
static bool port_change[2] = {FALSE, FALSE};
static bool direct_connect_present[2] = {FALSE, FALSE};

// My driver will need two URBs - I allocate them in Start_Driver and re-use them
URB *buttons_urb, *lights_urb;
bool lightsUrbInUse = FALSE;

// Describe  my driver is a Frameworks compatible format.  Only one may be active
bool DriverInUse = FALSE;
CLASS_DRIVER const se7_driver = {
    0,               // class
    0,               // subclass
    3,               // if_class
    0,               // if_subclass
    0,               // protocol
    0x4242,          // vendor_ID
    0xc003,          // product_ID
    se7driver_start, // (*start)( USB_DEVICE *dev )
    se7driver_stop,  // (*stop)(void)
    se7driver_run,   // (*run)(void)
    se7driver_ioctl, // (*ioctl)( USB_DEVICE *, uint16, uint16, uint16 )
};

// Define how the Host and Device share buttons
bool share_local_buttons = FALSE;                  // Keep buttons private
bool passthru_downstream_buttons = FALSE;          // Keep buttons private

// Now declare the device data, see se3 for more details
```

```
bool Configured = FALSE;
extern uint8 strings_descriptor;
uint8 device_buttons_report ATTR_USB_XFER_BUF_SECTION;
uint8 device_lights_report  ATTR_USB_XFER_BUF_SECTION;
USBTXRXINFO report_descriptor_info, buttons_report_info, lights_report_info;
bool buttons_report_inuse = FALSE;  // I recycle the same buffers
PFNINTHANDLER BIOSConfigurationChange, BIOSStandardRequestHandler,
BIOSClassRequestHandler;

USB_DEVICE_DESCRIPTOR const device_descriptor ATTR_SIE1_DESCR_SECTION = {
    18, 1, 0x200, 0, 0, 0, 64, 0x4242, 0xc003, 0x100, 1, 2, 0, 1
    };

uint8 const report_descriptor[] ATTR_SIE1_DESCR_SECTION = {
    6, 0, 0xFF,      // Usage_Page (Vendor Defined)
    9, 1,            // Usage (IO Device)
    0xA1, 1,         // Collection (Application)
    0x19, 1,         //   Usage_Minimum (1)
    0x29, 8,         //   Usage_Maximum (8)
    0x15, 0,         //   Logical_Minimum (0)
    0x25, 1,         //   Logical_Maximum (1)
    0x75, 1,         //   Report_Size (1)
    0x95, 8,         //   Report_Count (8)
    0x81, 2,         //   Input (Data,Var,Abs) = Buttons
    0x19, 1,         //   Usage_Minimum (1)
    0x29, 8,         //   Usage_Maximum (8)
    0x91, 2,         //   Output (Data,Var,Abs) = Lights
    0xC0             // End_Collection
    };

USB_ALL_DESCRIPTORS const configuration_descriptor ATTR_SIE1_DESCR_SECTION = {
    {   /* config_descriptor header */
        9, 2, sizeof(USB_ALL_DESCRIPTORS), 1, 1, 0, 0xC0, 1 },
    {   /* interface */
        9, 4, 0, 0, 2, 3, 0, 0, 3 },
    {    /* class_descriptor */
    9, 0x21, 0x100, 0, 1, 0x22, sizeof(report_descriptor) },
    {   /* EP1_In */
        7, 5, 0x81, 3, 8, 100 },
    {   /* EP2_Out */
        7, 5, 2, 3, 8, 100 }
    };

uint16 show_error(uint16 error) {
// Helper routine to display errors, should not get any!
    cpld_set_led(ERROR_LED);
    cpld_set_ssd(error);
    return ERROR;
    }


// Declare my Init_Tasks - there are several in this example
// sie1_init getc called early and I initialize sie1 as a host
// sie2_init gets called early - I wait and initialize sie2 in App_Init
// se7driver_start is called to initialize Host side once BAL device has been enumerated
// App_Init is called to initialize the Device side of this example

void sie1_init(void) {
// Initialise the Status LEDs
    cpld_set_led(HOST_LED);
    cpld_set_led(SESSION_ACTIVE_LED);
    cpld_clr_led(SLAVE_LED);
    cpld_clr_led(ERROR_LED);
// Now spin up the SIE as a Host
```

```
    device_map_init();
    WRITE_REGISTER(HUSB_pEOT_ADR, 1200);
    husb1_init();
    INPLACE_OR(HOST1_IRQ_EN_REG, (VBUS_IRQ_EN | A_CHG_IRQ_EN | B_CHG_IRQ_EN) );
    }

void sie2_init(void) {
// Frameworks gives me an opportunity to initialize the sie here, I wait until app_init()
    }

uint16 se7driver_start(USB_DEVICE *dev) {
// Frameworks will pass me a driver object for the BAL HID device
// Only allow one copy of the driver to run
    if (DriverInUse) return ERROR;
    DriverInUse = TRUE;
// Get two URBs needed for the interrupt reports
    buttons_urb = alloc_URB(FALSE, sizeof(host_buttons_report) );
    if (!buttons_urb) return show_error(0xA);
    lights_urb = alloc_URB(FALSE, sizeof(host_lights_report) );
    if (!lights_urb) return show_error(0xB);
// I have two URBs, initialize them
// NOTE: since I know the device then I know attributes such as endpoint/polling interval
// In the general case I would parse the descriptors to discover this information
// Initialize those elements of the urb that are constant
    lights_urb->dev = dev;
    lights_urb->dir = TD_CTRL_DIR_OUT;
    lights_urb->usb_dev_addr = dev->address;
    lights_urb->endpoint = 2;                              // See NOTE
    lights_urb->speed = dev->speed;
//  lights_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    lights_urb->type = USB_BULK_TRANSFER_TYPE;
    lights_urb->interval = 100;                            // See NOTE
// Initialize those elements of the urb that are constant
    buttons_urb->dev = dev;
    buttons_urb->dir = TD_CTRL_DIR_IN;
    buttons_urb->usb_dev_addr = dev->address;
    buttons_urb->endpoint = 1;                             // See NOTE
    buttons_urb->speed = dev->speed;
    buttons_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    buttons_urb->interval = 100;                           // See NOTE
    buttons_urb->transfer_buffer = &host_buttons_report;
    buttons_urb->buffer_length = sizeof(host_buttons_report);
    buttons_urb->callback = (PFNURBCALLBACK) buttons_report_received;
// Post the buttons_urb to wait for an input report from the device
    if (td_submit_URB(buttons_urb) == ERROR) {
        release_URB(buttons_urb);
        return show_error(0xC);
        }
    else return SUCCESS;
    }

void app_init(void) {
// This is se3 but using SIE2
// Update the descriptor pointers that BIOS uses
    WRITE_REGISTER (SUSB2_DEV_DESC_VEC, (PFNINTHANDLER) &device_descriptor);
    WRITE_REGISTER (SUSB2_CONFIG_DESC_VEC, (PFNINTHANDLER) &configuration_descriptor);
    WRITE_REGISTER (SUSB2_STRING_DESC_VEC, (PFNINTHANDLER) &strings_descriptor);
// Chain a routine before BIOS's standard request handler
    BIOSStandardRequestHandler = (PFNINTHANDLER) READ_REGISTER (SUSB2_STANDARD_INT*2);
    WRITE_REGISTER (SUSB2_STANDARD_INT*2, (PFNINTHANDLER) &InterceptStandardRequest);
// Add a Class Request Handler
// Actually, since I stall all requests anyway, I may as well let BIOS do that!
//  BIOSClassRequestHandler = (PFNINTHANDLER) READ_REGISTER (SUSB2_CLASS_INT*2);
//  WRITE_REGISTER (SUSB2_CLASS_INT*2, (PFNINTHANDLER) &HandleClassRequest);
// We need to know when a Set_Configuration is received
    BIOSConfigurationChange = (PFNINTHANDLER) READ_REGISTER (SUSB2_DELTA_CONFIG_INT*2);
    WRITE_REGISTER (SUSB2_DELTA_CONFIG_INT*2, (PFNINTHANDLER) &SetConfigurationRequest);
// Now initialize SIE2, this will result in it enumerating with the PC Host
    susb_init(SIE2, USB_FULL_SPEED);
    }

/* Declare the Idle_Tasks */
// There are several, but the only one that does work is sie1_idle
// During Idle we look for device connect/disconnects on the host ports
void sie1_idle(void) {
    if (port_change[0] || port_change[1]) sie1_check_for_connected_devices();
    }
void sie2_idle(void) {
    }
void se7driver_run(USB_DEVICE *dev) {
    }

/*  Declare Callback routines */
void sie1_check_for_connected_devices(void) {
    int16 port, reg;
// Disable the insert/remove interrupts
    INPLACE_AND(DEV1_IRQ_EN_REG, ~(A_CHG_IRQ_EN) );
// Check for connected devices. */
    for (port = 0; port < 2; ++port) {
        reg = husb_reset(20, port);
        if (!reg & 2) enumerate_device(SIE1, port, &direct_connect_present[port],
sie1_enumeration_notify);
        port_change[port] = FALSE;
        }
// Prior to re-enabling the interrupts, make sure the insert/remove interrupt is cleared.
    INPLACE_OR( HOST1_STAT_REG, A_CHG_IRQ_EN );
// Enable the insert/remove interrupts. */
    INPLACE_OR( DEV1_IRQ_EN_REG, A_CHG_IRQ_EN );
    }

void sie2_check_for_connected_devices(void) {
    }        // Null since SIE2 is a device

uint16 se7driver_stop(USB_DEVICE *dev) {
    if (!DriverInUse) return ERROR;
    DriverInUse = FALSE;
// Release any system resources we have
    if (buttons_urb) {
        td_clear_URB(buttons_urb);        // Remove from td processor
        release_URB(buttons_urb);         // Deallocate the urb
        }
    if (lights_urb) {
        if (lightsUrbInUse) td_clear_URB(lights_urb);
        release_URB(lights_urb);
        }
    return SUCCESS;
    }

uint16 se7driver_ioctl(USB_DEVICE *dev, uint16 cmd, uint16 d1, uint16 d2) {
// Nothing to do since no IOCTLs defined
    if (DriverInUse) return SUCCESS;
    return ERROR;
    }

// Handle button presses from device
void buttons_report_received(URB *urb) {
    if (urb->status == SUCCESS) {
// Do I process the button locally or just pass it on?
```

```
        if (passthru_downstream_buttons) app_button_handler(host_buttons_report);
        else button_press(host_buttons_report);
        }
    else {
// There was an error on the urb, so Frameworks will stop scheduling it
// I should try and resubmit it to keep looking for INT-IN
        urb->transfer_buffer = &host_buttons_report;
        urb->buffer_length = sizeof(host_buttons_report);
        urb->callback = (PFNURBCALLBACK) buttons_report_received;
        if (td_submit_URB(urb) == ERROR) {
            release_URB(urb);
            show_error(0xC);
            }
        }
    }

// Device lights update report has been sent
void lights_report_sent (void) {
// Now OK to reuse lights urb
    lightsUrbInUse = FALSE;
    }

void button_press( BUTTON button) {
// Process the button press locally and update the display
    switch (button) {
        case BTN_UP: if (++DisplayValue > 9) DisplayValue = 0; break;
        case BTN_DOWN: if (--DisplayValue < 0) DisplayValue = 9; break;
        case BTN_LEFT: DisplayValue = 0; break;
        case BTN_RIGHT: DisplayValue = 9; break;
        default: break;
        }
    update_display(DisplayValue);
    }

void update_display(uint16 value) {
// Update the local display and relay the information downstream
    cpld_set_ssd(value);
    host_lights_report = value & 0x0F;
    if (!lightsUrbInUse) {
        lights_urb->transfer_buffer = &host_lights_report;
        lights_urb->buffer_length = sizeof(host_lights_report);
        lights_urb->callback = (PFNURBCALLBACK) lights_report_sent;
        if (td_submit_URB(lights_urb) == ERROR) {
            release_URB(lights_urb);
            show_error(0xD);
            }
        else lightsUrbInUse = TRUE;
        }
    }

// This function is called when device enumeration is complete or failed
void sie1_enumeration_notify(USB_DEVICE *dev) {
// This example host only supports the BAL device, check that this is it!
    if ((dev->enum_state == ES_COMPLETE) && (dev->dev_descr.idVendor == 0x4242) && (dev-
>dev_descr.idProduct == 0xc003)) {
        cpld_set_led(SESSION_ACTIVE_LED);
        if (dev->direct_connect) direct_connect_present[dev->port] = TRUE;
        }
    else {
        cpld_set_led(ERROR_LED);
        cpld_set_ssd(0xF);
        direct_connect_present[dev->port] = FALSE;
        tpl_unlink_all_port( dev->port );
        device_cleanup( dev->sie, dev->port );
        dealloc_device( dev );
         }
    }

// Now declare the device callbacks - this is se3 but using SIE2
void SetConfigurationRequest(void) {
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE2_DEV_REQ;
// Let BIOS handle this request first
    BIOSConfigurationChange();
// If I got configured then I can enable my data endpoints
    if ((req->wValue & 0xFF) ==
configuration_descriptor.config_header.bConfigurationValue) {
    Configured = TRUE;
        cpld_set_ssd(0);
        cpld_set_led(SLAVE_LED);
// Ask BIOS to inform me when a lights report is received
        setup_lights_report_callback();
        }
    else {
        Configured = FALSE;
        cpld_clear_ssd();
        cpld_clr_led(SLAVE_LED);
        }
    }

void InterceptStandardRequest(void) {
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE2_DEV_REQ;
// BIOS does not handle GetDescriptor(Interface) so check for that
    if ((req->bRequest == USB_GET_DESCRIPTOR_REQUEST) && ((req->bmRequestType & 3) == 1)
&& (req->wValue == 0x2200) ) {
        report_descriptor_info.buffer = &report_descriptor;
        report_descriptor_info.length = sizeof(report_descriptor);
        report_descriptor_info.done_func = 0;      // Let BIOS handle completion
        susb_send(SIE2, 0, &report_descriptor_info);
        }
    else {
// Pass the request on to BIOS to handle
        BIOSStandardRequestHandler();
        }
    }

// The device owns the buttons
void app_button_handler(BUTTON    button) {
// Report this change in button state to the host if we are configured
    if (Configured) {
        buttons_report_info.buffer = &device_buttons_report;
        buttons_report_info.length = sizeof(device_buttons_report);
        buttons_report_info.done_func = 0;         // Let BIOS handle completion
        device_buttons_report = button;
        susb_send(SIE2, 1, &buttons_report_info);
        }
// If the device is sharing the buttons then initiate a host_lights_report
    if (share_local_buttons) button_press(button);
    }

// The device has just received a lights_report
void lights_report_received (void) {
    update_display(device_lights_report);
// Wait for the next update
    setup_lights_report_callback();
    }
```

```
void setup_lights_report_callback() {
    lights_report_info.buffer = &device_lights_report;
    lights_report_info.length = sizeof(device_lights_report);
    lights_report_info.done_func = (PFNINTHANDLER) &lights_report_received;
    susb_receive(SIE2, 2, &lights_report_info);
    }
```

Figure 5-3. Host+Device Application Program

Simple Example #8 – Using Scan Records 2

The default operation of BIOS initializes both SIEs using default descriptors. We do not want either of them initialized in this example so we must preload the short program shown in Figure 5-4 into the I2C eeprom of the EZ-Host mezzanine board while we are debugging the program with Insight/gdb.

```
.section .init
; First define the code that needs to be loaded
; It will be prefixed with a Scan Header
        .short  ScanSignature
        .short  Length+2
        .byte   LoadCommand
        .short  _start
.global _start
_start:
; Give control back to BIOS, this skips SIE1 and SIE2 initialization
mov r15, 0x400       ; Reset the stack
        sti                  ; First time interrupts are enabled
        int IDLER_INT        ; This will not return
.equ    Length, .-_start
; Now define a scan record that will transfer control to my program
        .short  ScanSignature
        .short  2
        .byte   JumpCommand
        .short  _start
```

Figure 5-4. se8 used to modify BIOS operation

Click bash_env in the se8 directory to create a bash window. Then enter "`make`" to build se8.bin. Set dipswitches 6, 5, 4, and 3 on and enter "`qtui2c se8.bin f`" to program the eeprom then enter "exit" to close the bash window. We have now given up our USB debug port so we must connect a serial cable between the EZ-Host mezzanine board and our development PC. We connect the USB cable to our target PC and connect the EZ-OTG mezzanine board that has been programmed to look like a buttons and lights device to SIE1. Your hardware should now look like Figure 5-5.

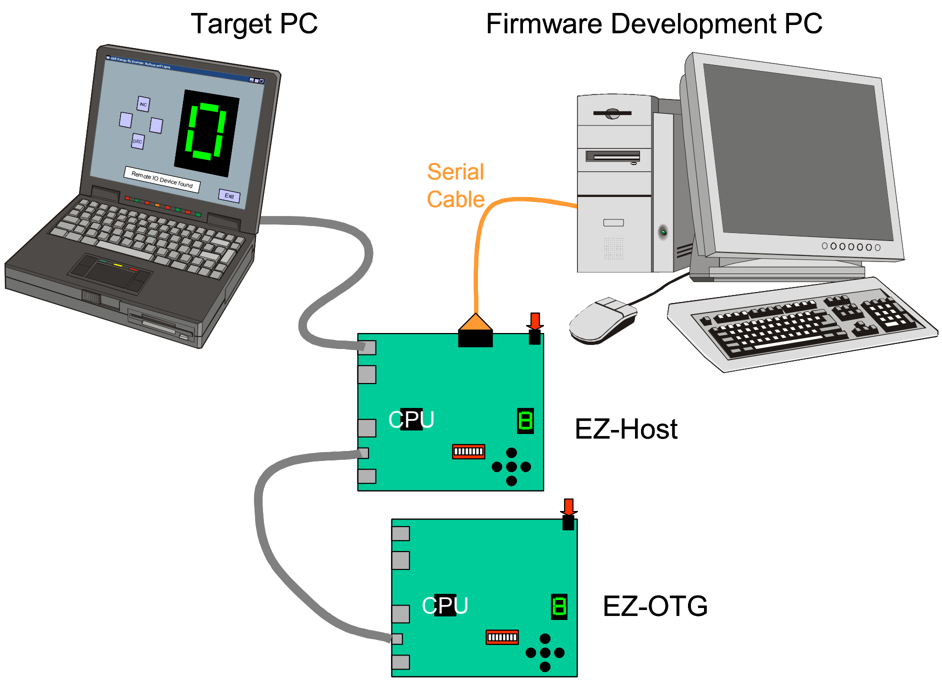


Figure 5-5. Hardware staged for debug

Click on bash_env in the se7 directory to create a bash window. Enter "`make`" to build our host+device example se7. Click on bash_env again to create a second bash window and enter "`cy16-elf-libremote -s -P com1 -b 28800`" to create a serial connection to our target system. In the first bash window enter "`cy16-elf-gdb se7`" to start the Insight debugger and click the RUN icon. Your target settings for this example will be the same as the previous examples, even though we are shifting to using the serial port. The program will be downloaded to the EZ-Host mezzanine board via the serial cable and the program will break at Main().

Clear all breakpoints and click on continue.

On the target PC start the Buttons and Lights host program and verify that clicking buttons on the PC host display or on the EZ-Host mezzanine board cause the seven segment display to track. The device is working!

On the EZ-OTG mezzanine board click buttons to ensure that the seven-segment display advances. Click buttons on the EZ-Host mezzanine board to ensure that both seven segment displays track. The displays may not start in sync but they will track after the first button press. The host is working!

Stop the debug session and exit Insight. You should also reset the libremote window (i.e. Control+C, up-arrow, enter).

The three seven segment displays did not remain in sync since the host application program keeps button presses it receives from the EZ-OTG mezzanine board as private. Some applications will work this way. We will make a small modification to our host program to forward these button presses to the device program that will, in turn, forward these to the PC host. The device program will also forward seven-segment display changes down to the EZ-OTG mezzanine board.

Open app.c in the se7 directory and search for *share_local_buttons* and *passthru_downstream_buttons*. These two boolean variables are initially set to false. You can change one or both of these variables to change how button press events are routed around the system. Note that the same approach would be used for larger reports or data movement. Try setting “passthru_downstream_buttons = TRUE;” and save app.c. In the bash window enter “`make`,” then “`cy16-elf-gdb se7`” and in the Insight window click on RUN. Clear all breakpoints and click “continue.”

Check the operation of all the buttons and observe all of the seven-segment displays now stay in sync.

The host+device application is working!

Smart USB Devices

I have redrawn our example in Figure 5-6 so that we can better appreciate the capability we have created.

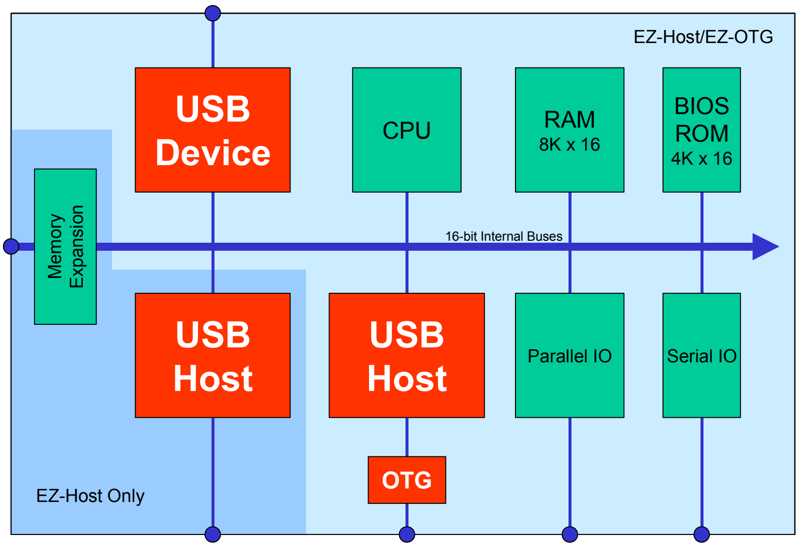


Figure 5-6. EZ-Host/EZ-OTG is a ***very*** smart USB device

To the PC, the EZ-Host/EZ-OTG in this configuration looks just like a standard USB device. It is fully programmable to enable it to look like any USB device. It also has host capability – one host port on the EZ-OTG and two host ports on the EZ-Host. This means that you can plug any standard USB device into this subsystem.

Have you designed a USB device and wished that you could attach a USB keyboard or a USB mouse to it? Or needed mass storage, so longed for an A socket to attach a mini Flash Drive? Well, with this EZ-Host/EZ-OTG sub system you can now do that!

Before we get too carried away with the solution possibilities that this opens up I must remind you that the EZ-Host/EZ-OTG must contain a device driver for whatever USB device that you want to attach to the host port(s). We wrote a device driver in Chapter 4 and saw that it was not difficult. This subsystem will be used as an embedded host and need only support a few specific USB devices – this is the targeted peripherals list.

I have two diverse examples to demonstrate the wide range of solutions that this host+device subsystem can create. The first is a remote data acquisition and control system and the second is a video “black-box.”

Data Acquisition Example

USB has had limited adoption in remote data acquisition and control systems due to the limits in cable length and device count. The USB specification limits the maximum length of a cable to 5 meters, the maximum hub depth to 5 and the maximum number of devices to 126. This means that all 126 devices must be within a 30 meter radius of the PC host. Our EZ-Host/EZ-OTG subsystem is a device so it must be within 30 meters of the PC host, but it is ALSO a host and, as such, can support ANOTHER 126 devices at a radius of 30 meters. And we could do this again. And again. We are, in effect, creating multiple USB "sub-nets" as shown in Figure 5-7.

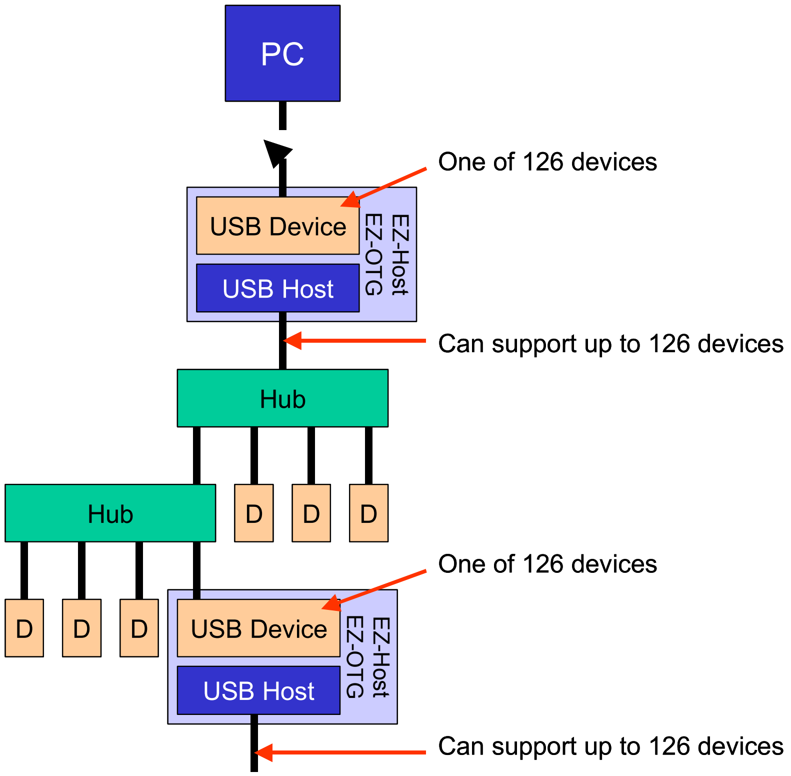


Figure 5-7. Example of a smart USB device: data acquisition and control

In this DAQ example the EZ-Host/EZ-OTG is acting as a data concentrator for inbound data and a distribution point for outbound control information. The data collection/control elements can be simple USB devices supporting a standard protocol such as HID. The EZ-Host/EZ-OTG concatenates the data from its slave devices and forwards this upstream. Similarly it receives concentrated control information from upstream and redistributes it to its slave control devices. The constant polling of the devices would quickly identify any broken connections and the plug-and-play nature of USB would enable the sub-net to be dynamically changed or repaired.

Simple Example 7 has most of the functionality that we need for this data acquisition example. We will want, of course, to support multiple devices and will therefore need a hub driver. The structure of this code will be just like se6_driver, and a working example will be presented in Chapter 7.

Additionally the se6_driver code must be extended to support multiple running devices: the code can be extended using two alternate methods, both of which separate the driver code from the driver data. We need to run multiple copies of the driver code (ie multiple identical tasks) but each will operate on a different set of data. We could declare the procedures as reentrant so that the working data was stored on the stack, or we could extend the device object to include the variables that the driver requires.

I would recommend using the “Buttons and Lights” device as your first data-acquisition and control device and then add features to it. Once your example code is built you would use the hardware setup shown in Figure 5-8 to test it.

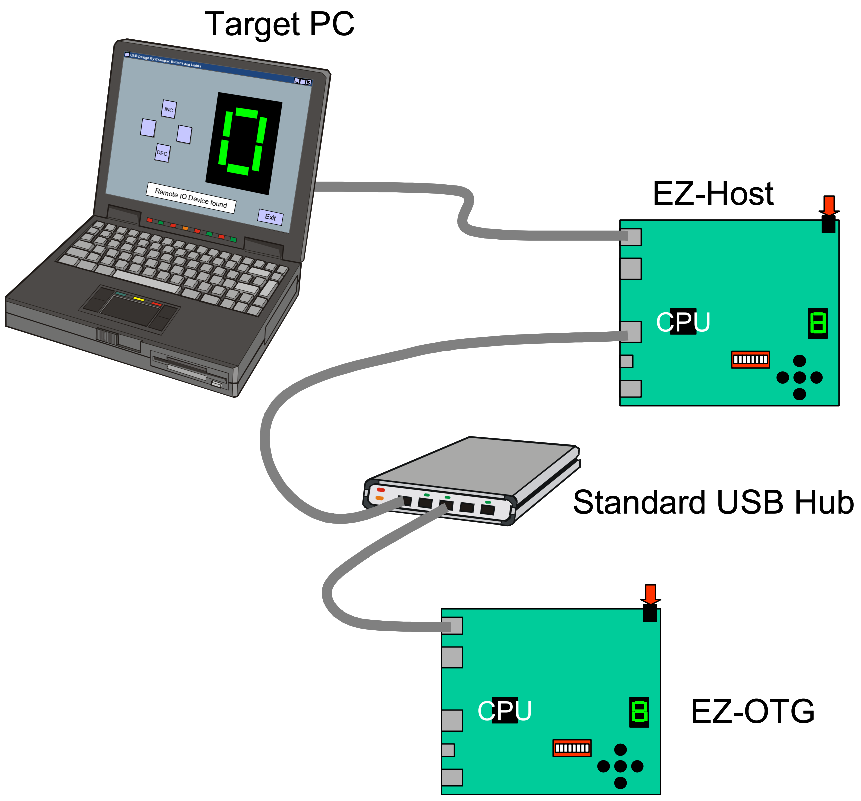


Figure 5-8. Data acquisition and control example

One of my reviewers pointed out that the device side of the EZ-Host/EZ-OTG host+device need not use USB. If the data rate were low then an RS232 connection could be used – this is a little more work but it would mean that our data acquisition system could be a long way from the PC!

Another reviewer pointed out that the device connection need not be a permanent one. The EZ-Host/EZ-OTG could gather data and store it. An operator could visit the sub system and connect it to a laptop and upload the collected data and download new control parameters.

I'm sure that you too will think up many applications for this device+host subsystem.

Video Black Box Example

My second host+device example takes a standard USB device, in this case a video camera, and adds features to it. Lets first consider the case where the PC is attached as shown in Figure 5-9 – we will remove it later.

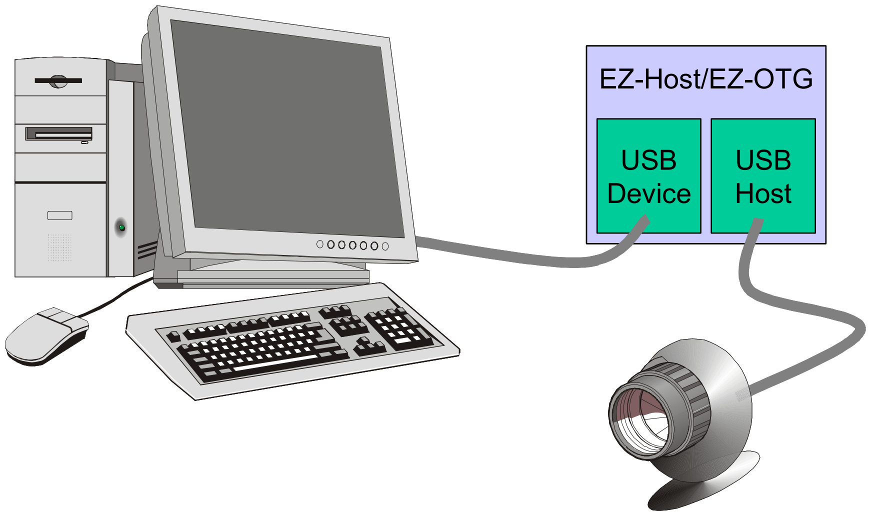


Figure 5-9. Smart USB device: video black box using standard USB camera

At power-on the EZ-Host/EZ-OTG host-side enumerates the video camera but does not enable it yet. It passes this descriptor information to the device-side. The device-side then connects to the PC host and uses the enumeration information gathered from the camera. The PC host, assuming that it is talking directly to the camera, loads an appropriate device driver that will instruct the camera to start sending video data. Our host+device forwards this command to the camera and then forwards the video data from the camera to the PC host. Neither the PC host or USB video camera is aware that we are intercepting and relaying information in both directions – this means that no extra software needed to be written at either end.

But we do more than just pass the video data through. Our application program keeps a buffer of the last X second in its internal memory. Note that the video data is not in a useful format such as frames. All video cameras that operate at 12 Mb/s use some kind of proprietary data compression on the video data and use bulk transfers to transport the data to the PC. This encoded data is decompressed by a camera device driver at the PC. So we cannot process the data but we can store it.

Should the video stop for some reason then the EZ-Host/EZ-OTG host+device could be requested to play back the stored data. This may tell us why the video stopped. Or the host+device could be instructed to save data at strategic times for later playback. Once the system has been set up, the PC host could be removed – it could be reconnected at a later time to view stored video.

This video black box solution would be attractive in security and safety applications.

What other USB device can you think of that would benefit from a smart “front-end?”

Chapter Summary

It is straight forward to build a subsystem that supports a host connection and a device connection concurrently. The host application program and the device application program can cooperatively share and process data thus enabling you to build a new range of smart devices. A simple example of a buttons and lights HID device was worked in detail and two other examples, a remote data acquisition system and video black box, were outlined.

The simplicity with which the EZ-Host/EZ-OTG components enable you to construct feature-rich smart devices will quickly extend the range of USB solutions.

In the next chapter we shall look at a dual-role device that can operate as a host or as a peripheral and dynamically switch between the two roles. This is the world of OTG.

Chapter 6: Designing a dual-role device

We have seen, in the previous chapters, that the design of host capability and the design of device capability with the EZ-Host/EZ-OTG is straightforward, and we have several working examples. These designs used an A connector for a host and a B connector for a device, and some of these examples used multiple connectors. In this chapter we will design a dual-role device that is characterized by its single Mini-AB connector – this device is sometimes a host (other devices plug into it) and it is sometimes a device (it plugs into a host). The firmware is more complex, especially since we have to support the swapping of roles, but we have most of the building blocks that we need to complete the design, from previous examples, so this project too will be straightforward.

A dual-role device is typically battery powered. In fact, this was the model that the OTG Supplement was written around. Figure 6-1 shows an overview of the dual-role example that we will develop. We will re-use the host and device "buttons and lights" examples code so that we can focus on the new elements required for the dual-role functionality.

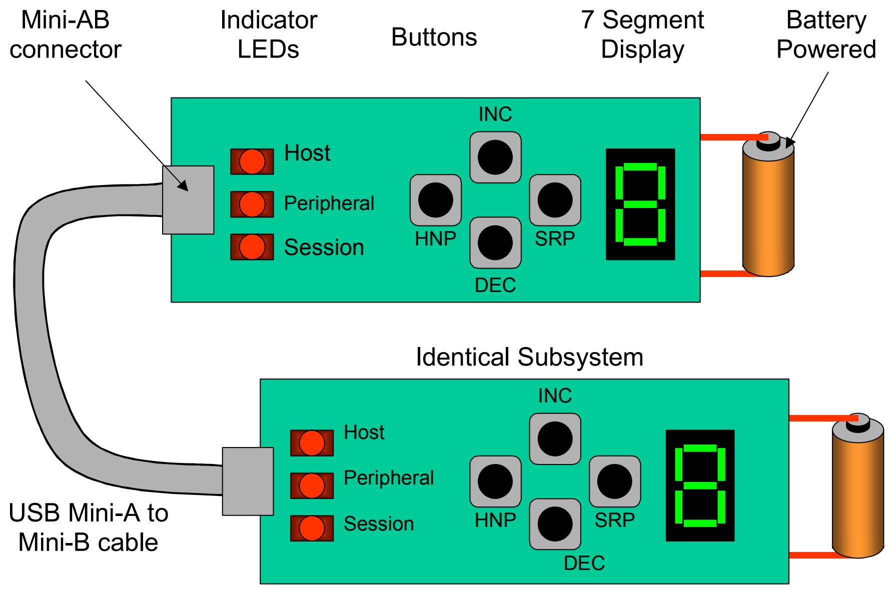


Figure 6-1. Overview of dual-role example

The mezzanine boards do not support battery-powered operation so we have to trust that this operates correctly (it does, I prototyped a smaller, battery-powered board to test the firmware).

New dual-role concepts

The OTG Supplement that defines a dual-role device introduced several new concepts that I will demonstrate in this chapter's example. A USB cable has two ends, an A-end and a B-end. This allows us to define two terms that describe the two roles of a dual-role device. The cable defines the start-up configuration of each dual-role device: at the A-end of the cable is the default A-device or start-up host and at the B-end of the cable is the default B-device or start-up peripheral. Note that if a cable is not inserted into a Mini-AB connector then both dual-role devices default to be a peripheral since the ID pin is floating (refer back to Figure 1-9 if required).

Battery-powered devices always manage themselves to minimum power consumption, so an A-device will turn off Vbus when it has finished using the USB cable. If the B-device later decides that it needs to initiate some USB transfers then it will, if enabled, use Session Request Protocol (SRP) signaling to ask the host to re-power the cable. It will use Host Negotiation Protocol (HNP) to swap roles so that it can control the USB communications for a while. When completed it will close the session and revert back to being a peripheral. From an application program perspective these protocols are very easy as Frameworks handle them. The Frameworks code is quite elaborate since it involves interacting hardware and firmware on the two devices, and we will cover what is happening "behind-the-scenes" later in this chapter. The application program must be extended to allow SRP and HNP sequences to be generated, and we shall do this via button presses.

Simple Example #9 - Dual-role Buttons and Lights Device

The structure of the dual-role firmware will be the familiar Init_Task, Idle_Task and Callback routines as used in the other examples. I defined a global variable, FWX_SYSTEM_MODE, which has values of STOP, HOST and SLAVE, so the firmware knows which role it is currently implementing. I combined the "buttons and lights" device application code (se3) and the "Buttons and Lights" host application code (se6) to create the dual-role application code shown in Figure 6-2.

I added code to the device-role buttons callback routine to detect two additional button presses, SRP and HNP. I also added an OTG descriptor to the device configuration, and code to handle the SetFeature (HNP) command.

On the host side, I added code to handle the additional features by defining them in fwxcfg.h. This will result in the host giving the peripheral an opportunity to implement an SRP and then an HNP. The host enables the device to swap roles via a SetFeature (HNP) prior to suspending itself.

The fwxcfg.h configuration file in the se9 directory enables the SRP and HNP modules within Frameworks.

```
/* File: app.c.           Simple Example 9
 * BAL host+device,       dual-role device on SIE1 (device on SIE2 for debugger)
 */

#include "app.h"
#include "sie1.h"

/* Application data. */
// Most of the OTG functionality is handled by Frameworks
// We control it's operation via the otg data structure (declared in otg.c)
// OTG support is only available on Host port 0
#define OTG_Port 0
// Port changes are detected in an ISR and serviced in the IdleTask
bool port_insert = FALSE;
bool port_remove = FALSE;
bool direct_connect_present = FALSE;

// Declare the Host specific data
// Provide lights report on button change and receives a buttons report
uint8 host_buttons_report ATTR_USB_XFER_BUF_SECTION;
uint8 host_lights_report  ATTR_USB_XFER_BUF_SECTION;
int DisplayValue = 0;
uint16 previous_button_state = ~0;

// My Host driver needs two URBs - I allocate them in Start_Driver and re-use
URB *buttons_urb = 0, *lights_urb = 0;
bool lightsUrbInUse = FALSE;

// Describe  my driver is a Frameworks compatible format.  Only one may be active
bool DriverInUse = FALSE;
CLASS_DRIVER const se9_driver = {
    0,                    // class
    0,                    // subclass
    3,                    // if_class
    0,                    // if_subclass
    0,                    // protocol
    0x4242,               // vendor_ID
    0xc003,               // product_ID
    se9driver_start,      // (*start)( USB_DEVICE *dev )
    se9driver_stop,       // (*stop)(void)
    se9driver_run,        // (*run)(void)
    se9driver_ioctl,      // (*ioctl)( USB_DEVICE *, uint16, uint16, uint16 )
    };

// Now declare the device data, see se3 for more details
extern uint8 strings_descriptor;
uint8 device_buttons_report ATTR_USB_XFER_BUF_SECTION = 0;
uint8 device_lights_report  ATTR_USB_XFER_BUF_SECTION = 0;
USBTXRXINFO reply = {0}, buttons_report_info = {0}, lights_report_info = {0};
bool buttons_report_inuse = FALSE;  // I recycle the same buffers

USB_DEVICE_DESCRIPTOR const device_descriptor ATTR_SIE1_DESCR_SECTION = {
    18, 1, 0x200, 0, 0, 0, 64, 0x4242, 0xc003, 0x100, 1, 2, 0, 1
    };
```

```
uint8 const report_descriptor[] ATTR_SIE1_DESCR_SECTION = {
    6, 0, 0xFF,  // Usage_Page (Vendor Defined)
    9, 1,        // Usage (IO Device)
    0xA1, 1,     // Collection (Application)
    0x19, 1,     //   Usage_Minimum (1)
    0x29, 8,     //   Usage_Maximum (8)
    0x15, 0,     //   Logical_Minimum (0)
    0x25, 1,     //   Logical_Maximum (1)
    0x75, 1,     //   Report_Size (1)
    0x95, 8,     //   Report_Count (8)
    0x81, 2,     //   Input (Data,Var,Abs) = Buttons
    0x19, 1,     //   Usage_Minimum (1)
    0x29, 8,     //   Usage_Maximum (8)
    0x91, 2,     //   Output (Data,Var,Abs) = Lights
    0xC0         // End_Collection
    };

USB_ALL_DESCRIPTORS const configuration_descriptor ATTR_SIE1_DESCR_SECTION = {
    {   /* config_descriptor header */
        9, 2, sizeof(USB_ALL_DESCRIPTORS), 1, 1, 0, 0xC0, 50 },
    {   /* interface */
        9, 4, 0, 0, 2, 3, 0, 0, 3 },
    {   /* class_descriptor */
        9, 0x21, 0x100, 0, 1, 0x22, sizeof(report_descriptor) },
    {   /* EP1_In */
        7, 5, 0x81, 3, 8, 100 },
    {   /* EP2_Out */
        7, 5, 2, 3, 8, 100 },
    {   /* OTG */
        3, 9, USB_OTG_SRP_SUPPORT | USB_OTG_HNP_SUPPORT }
    };

// Declare my Init_Tasks - there are several in this example
// Frameworks expects to call these named routines
void sie1_init(void) {
    otg_init();
    }

void app_pre_init(void) {
    INPLACE_AND(IRQ_EN_REG, ~TMR0_IRQ_EN);    // Disable timer0
    }

void sie1_init_slave(void) {
// I usually do this in App_Init, but Frameworks needs it done here
// Update the descriptor pointers that BIOS uses
    WRITE_REGISTER (SUSB1_DEV_DESC_VEC, &device_descriptor);
    WRITE_REGISTER (SUSB1_CONFIG_DESC_VEC, &configuration_descriptor);
    WRITE_REGISTER (SUSB1_STRING_DESC_VEC, &strings_descriptor);
// I #define susb1_standard_handler and susb1_delta_cfg_handler in fwxcfg.h
// Now initialize SIE1
// Results in enumerating with the Host side of this app on other board
    susb_init(SIE1, USB_FULL_SPEED);
    INPLACE_OR(HOST1_IRQ_EN_REG, VBUS_IRQ_EN);
    }

void sie1_init_host(void) {
// Initialise as a host
    WRITE_REGISTER(HUSB_pEOT_ADR, 1200);
    husb1_init();
    INPLACE_OR(HOST1_IRQ_EN_REG, (VBUS_IRQ_EN | A_CHG_IRQ_EN | B_CHG_IRQ_EN) );
    otg.a_set_b_hnp_en = FALSE;
    port_insert = FALSE;
    port_remove = FALSE;
    }
```

```
uint16 se9driver_start(USB_DEVICE *dev) {
// Frameworks will pass me a driver object for the BAL HID device
// Only allow one copy of the driver to run
    if (DriverInUse) return ERROR;
    DriverInUse = TRUE;
    cpld_set_led(HOST_LED);
// Get two URBs needed for the interrupt reports
    buttons_urb = alloc_URB(FALSE, 0 );
    if (!buttons_urb) return FALSE;
    lights_urb = alloc_URB(FALSE, 0 );
    if (!lights_urb) return FALSE;
// I have two URBs, initialize them
// NOTE: since I know the device then I know it's attributes
// In the general case I would parse the descriptors to discover this information
// Initialize those elements of the urb that are constant
    lights_urb->dev = dev;
    lights_urb->dir = TD_CTRL_DIR_OUT;
    lights_urb->usb_dev_addr = dev->address;
    lights_urb->endpoint = 2;                                      // See NOTE
    lights_urb->transfer_buffer = &host_lights_report;
    lights_urb->buffer_length = sizeof(host_lights_report);
    lights_urb->speed = dev->speed;
    lights_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    lights_urb->interval = 100;                                    // See NOTE
// Initialize those elements of the urb that are constant
    buttons_urb->dev = dev;
    buttons_urb->dir = TD_CTRL_DIR_IN;
    buttons_urb->usb_dev_addr = dev->address;
    buttons_urb->endpoint = 1;                                     // See NOTE
    buttons_urb->speed = dev->speed;
    buttons_urb->type = USB_INTERRUPT_TRANSFER_TYPE;
    buttons_urb->interval = 100;                                   // See NOTE
    buttons_urb->transfer_buffer = &host_buttons_report;
    buttons_urb->buffer_length = sizeof(host_buttons_report);
    buttons_urb->callback = (PFNURBCALLBACK) buttons_report_received;
// Post the buttons_urb to wait for an input report from the device
    if (td_submit_URB(buttons_urb) == ERROR) {
        release_URB(buttons_urb);
        return FALSE;
        }
    else return SUCCESS;
    }

/* Declare the Idle_Tasks */
// Operation depends upon whether I am a Host or a Device
void sie1_idle(void) {
    FWX_SYSTEM_MODE mode;
    mode = fwx_get_system_mode(SIE1);
    switch(mode) {
        case SYSTEM_MODE_HOST:
// During Idle a host must look for device connect/disconnects
            if ((port_insert || port_remove) && otg_is_host() ) {
                enumerate_device(SIE1, OTG_Port, &direct_connect_present,
                sie1_enumeration_notify);
                port_insert = FALSE;
                port_remove = FALSE;
                }
            break;
        case SYSTEM_MODE_HOST_INACTIVE:
        case SYSTEM_MODE_SLAVE_INACTIVE:
            port_insert = FALSE;
            port_remove = FALSE;
            break;
        case SYSTEM_MODE_SLAVE:
        default: break;
        }
    }

void se9driver_run(USB_DEVICE *dev) {
    }


void app_task(FWX_SYSTEM_MODE mode[2]) {
// Handle button presses in my Idle_Task as a Device
    uint16 button_state;
    button_state = CPLD_READ_BUTTONS();
    CPLD_CLEAR_BUTTON(button_state);
    if (button_state != previous_button_state) {
        previous_button_state = button_state;
        if (button_state) app_button_handler(button_state);
        }
    }


/*  Declare Callback routines */
uint16 se9driver_stop(USB_DEVICE *dev) {
    DriverInUse = FALSE;
// Release any system resources we have
    if (buttons_urb) {
        td_clear_URB(buttons_urb);    // Remove from td processor
        release_URB(buttons_urb);     // Deallocate the urb
        buttons_urb = 0;
        }
    if (lights_urb) {
        td_clear_URB(lights_urb);
        release_URB(lights_urb);
        lights_urb = 0;
        }
    lightsUrbInUse = FALSE;
    return SUCCESS;
    }

uint16 se9driver_ioctl(USB_DEVICE *dev, uint16 cmd, uint16 d1, uint16 d2) {
// Nothing to do since no IOCTLs defined
    if (DriverInUse) return SUCCESS;
    return ERROR;
    }


// Handle button presses from device
void buttons_report_received(URB *urb) {
    if (urb->status == SUCCESS) {
// The device sent me a button press (so I must be a host at the moment!)
// Treat this as a local button press
        app_button_handler(host_buttons_report);
        }
    else {
// There was an error on the urb, so Frameworks will stop scheduling it
// I should try and resubmit it to keep looking for input reports
        urb->transfer_buffer = &host_buttons_report;
        urb->buffer_length = sizeof(host_buttons_report);
        urb->callback = (PFNURBCALLBACK) buttons_report_received;
        if (td_submit_URB(urb) == ERROR) {
            release_URB(urb);
            }
        }
    }
```

```
// Device lights update report has been sent
void lights_report_sent (void) {
// Now OK to reuse lights urb
    lightsUrbInUse = FALSE;
    }


void update_display(uint16 value) {
// Update the local display
    cpld_set_ssd(value);
    if (otg_is_host()) {
// Relay the information downstream if I am a host
        host_lights_report = value & 0x0F;
        if ( lights_urb != 0 && !lightsUrbInUse) {
            lights_urb->callback = (PFNURBCALLBACK) lights_report_sent;
            if (td_submit_URB(lights_urb) == ERROR) {
                release_URB(lights_urb);
                }
            else lightsUrbInUse = TRUE;
            }
        }
    }

// This function is called when device enumeration is complete or failed
void sie1_enumeration_notify(USB_DEVICE *dev) {
// This example host only supports the BAL device, check that this is it!
    if ((dev->enum_state == ES_COMPLETE) && (dev->dev_descr.idVendor == 0x4242)
                             && (dev->dev_descr.idProduct == 0xc003)) {
        if (dev->direct_connect) {
            direct_connect_present = TRUE;
            if (otg.id == A_DEV) otg.b_conn = TRUE; else otg.a_conn = TRUE;
            }
        }
    else {
        direct_connect_present = FALSE;
        tpl_unlink_all_port(dev->port);
        dealloc_device(dev);
        }
    }


// Now declare the device callbacks
void susb1_delta_cfg_handler(void) {
    USB_DEVICE_REQUEST  *req;
    req = (USB_DEVICE_REQUEST *) SIE1_DEV_REQ;
// BIOS signals me on ALL "set" commands, look for "Set Configuration"
    if ((req->bmRequestType == 0) && (req->bRequest == USB_SET_CONFIGURATION)) {
// If I got configured then I can enable my data endpoints
        if ((req->wValue & 0xFF) ==
                configuration_descriptor.config_header.bConfigurationValue) {
// Ask BIOS to inform me when a lights report is received
            setup_lights_report_callback();
            }
        }
    }


bool susb1_standard_handler(USB_DEVICE_REQUEST *req) {
// BIOS does not handle GetDescriptor(Interface) so check for that
    if ((req->bRequest == USB_GET_DESCRIPTOR_REQUEST) &&
            ((req->bmRequestType & 3) == 1) && (req->wValue == 0x2200) ) {
        reply.buffer = &report_descriptor;
        reply.length = sizeof(report_descriptor);
        reply.done_func = 0;      // Let BIOS handle completion
        susb_send(SIE1, 0, &reply);
        }
    else {
// BIOS does not handle OTG requests, so check for these too
        if ((req->bRequest == USB_GET_DESCRIPTOR_REQUEST) &&
                ((req->bmRequestType & 3) == 0) && (req->wValue == 0x0900) ) {
            reply.buffer = (void *) &configuration_descriptor.otg_descriptor;
            reply.length = sizeof(USB_OTG_DESCRIPTOR);
            reply.done_func = 0;      // Let BIOS handle completion
            susb_send(SIE1, 0, &reply);
            }
        else {
            if ((req->bRequest == USB_SET_FEATURE_REQUEST) &&
                                    (req->wValue == B_HNP_ENABLE) ) {
                otg.b_hnp_en = TRUE;
                susb1_finish();
                }
            else if ((req->bRequest == USB_SET_FEATURE_REQUEST)
        && (req->wValue == A_HNP_SUPPORT || req->wValue == A_ALT_HNP_SUPPORT)) {
                susb1_finish();
                }
            else {
// Pass the request on to BIOS to handle
                return FALSE;
                }
            }
        }
    return TRUE;
    }


void app_button_handler(BUTTONbutton) {
// Operation of the buttons depends upon what mode I am in
    FWX_SYSTEM_MODE mode;
    mode = fwx_get_system_mode(SIE1);
    switch (mode) {
        case SYSTEM_MODE_SLAVE_INACTIVE:
// The Left and Right buttons are used to initiate some action
            switch(button) {
                case BTN_LEFT:
// This is interpreted as a "Request HNP"
                    switch(otg.state) {
                        case a_peripheral: otg.a_bus_req = TRUE; break;
                        case b_peripheral: otg.b_bus_req = TRUE; break;
                        default: otg.b_bus_req = FALSE;
                        }
                    break;
                case BTN_RIGHT:
// This is interpreted as a "Request SRP"
                    if (otg.state == b_idle) otg.b_bus_req = TRUE;
                    break;
                default: break;       // Other buttons are ignorred
                }
            break;
        case SYSTEM_MODE_SLAVE:
// React to the INC and DEC buttons
            switch(button) {
                case BTN_UP:
                case BTN_DOWN:
// Forward these buttons to the host
                    if ((otg.state == a_peripheral) ||
                        (otg.state == b_peripheral)) {
                        buttons_report_info.buffer = &device_buttons_report;
                        buttons_report_info.length =
                            sizeof(device_buttons_report);
                        buttons_report_info.done_func = 0;
                        device_buttons_report = button;
                        susb_send(SIE1, 1, &buttons_report_info);
                        }
                    break;
                default: break;      // Other buttons are ignorred
                }
            break;
        case SYSTEM_MODE_HOST_INACTIVE:
        case SYSTEM_MODE_HOST:
            switch(button) {
                case BTN_UP:
                    if (++DisplayValue > 9) DisplayValue = 0;
                    update_display(DisplayValue);
                    break;
                case BTN_DOWN:
                    if (--DisplayValue < 0) DisplayValue = 9;
                    update_display(DisplayValue);
                    break;
                case BTN_LEFT:
// This is interpreted as a "Request HNP"
// Ask Frameworks to initiate a swap roles and call me back with the result
                    handle_hnp(app_hnp_notify);
                    break;
                case BTN_RIGHT:
// This is interpreted as a "End SRP Session"
                    if (otg_is_host()) {
                        otg.a_bus_drop = TRUE;
                        otg.a_bus_req = FALSE;
                        }
                    else if (otg.state == a_idle) otg.a_bus_req = TRUE;
                    break;
                default: break;
                }
            break;
        default:
            break;
        }
    }


// Was the HNP request successful?
void app_hnp_notify(uint16 status) {
    if (status != SUCCESS) otg.a_bus_req = TRUE;
    }


// The device has just received a lights_report
void lights_report_received (void) {
    update_display(device_lights_report);
// Wait for the next update
    setup_lights_report_callback();
    }

void setup_lights_report_callback() {
    lights_report_info.buffer = &device_lights_report;
    lights_report_info.length = sizeof(device_lights_report);
    lights_report_info.done_func = lights_report_received;
    susb_receive(SIE1, 2, &lights_report_info);
    }
```

```
// File sie1.c
// Some "housekeeping" routines from app,c since this was getting a little long

// Setup OTG to switch roles
void app_switching_otg_roles(uint16 new_device_role) {
    switch(new_device_role) {
        case A_DEV: otg.a_bus_req = TRUE; break;
        case B_DEV: otg.a_bus_req = FALSE; break;
        }
    }

void sie1_host_cleanup(void) {
    direct_connect_present = FALSE;
    }

void set_port_change( uint16 port ) {
    port_insert = TRUE;
    }

void sie1_slave_reset_isr(void) {
    otg_slave_reset_isr();            // Pass on this reset
    }

void sie1_host_ins_rem_isr(uint16 status_register) {
// Handle plug/unplug events when operating as a Host
uint16 high_count, loop_count, reg;
#define min_high_count 250
#define max_loop_count 8000
    switch (otg.id) {
        case A_DEV:
            if (status_register & A_CHG_IRQ_FLG) {
                if (status_register & A_SE0_STAT) {
                    otg.b_conn = FALSE;       // Disconnected
                    port_remove = TRUE;
                    }
                else {
                    otg.b_conn = TRUE;
                    port_insert = TRUE;
                    if (otg.state == a_idle) {
                        high_count = 0;
                        loop_count = 0;
                        while ((high_count < min_high_count) &&
                                (loop_count < max_loop_count)) {
                            reg = READ_REGISTER(HOST1_STAT_REG) & A_SE0_STAT;
                            if (reg) high_count = 0; else ++high_count;
                            ++loop_count;
                            }
                        if (high_count >= min_high_count) {
                            otg.a_srp_det = TRUE;         // SRP detected
                            otg.a_bus_req = TRUE;
                            }
                        else otg.b_conn = FALSE;
                        }
                    }
                }
            break;
        case B_DEV:
            if (status_register & A_CHG_IRQ_FLG) {
                if (status_register & A_SE0_STAT) {
                    otg.a_conn = FALSE;       // Disconnected
                    port_remove = TRUE;
                    }
                else {
                    otg.a_conn = TRUE;
                    port_insert = TRUE;
                    }
                }
            break;
        }
    if (port_insert) otg_insert_remove_isr(TRUE); // Tell Frameworks about event
    }
```

Figure 6-2. Dual-role “buttons and lights” example.

Figure 6-2 contains a combined listing of app.c and sie1.c from the se9 directory edited to better fit the size of these book pages. The source file was getting a little large so I partitioned it into two elements. The Design Examples on the Cypress release CDROM take this one additional step by breaking out the host code into deXdrvr.c.

As usual, click “bash_env” in the se9 directory to create a bash window, and enter “`make`” to build the example. Review se9.lst and notice that the se9 object file is about 28KB – this is too big to fit into the EZ-OTG internal memory. I used a few simple steps to reduce the size of the memory image of se9 to about 15KB so that we can debug this example using our two mezzanine boards.

I created this compressed variant in the se10 directory so that you can follow the steps I took.

Simple Example #10 – Standalone Dual-role Buttons and Lights Device

Compressing se9 involved a few simple steps: by removing DEBUG from the compilations and using code optimization I saved 12KB! This approach has the disadvantage that I cannot use the GDB debugger on se10 but, since the application code is the same as se9 that will run on the EZ-Host mezzanine card, then I believe that we are adequately covered. With another 5KB of reduction required I focused on the Frameworks code. Cypress provides the source code of all of the Frameworks elements in the /Common directory and this code is feature-rich supporting ALL aspects of a host/device/dual-role/multi-role design. I removed some features that were not required for this simple dual-role HID example and easily got below 15KB. The edited versions of these Frameworks files are also in the se10 directory. I did have to make a few minor edits to app.c to match the edits I made in the Frameworks files and this too is in the se10 directory – I made no functional changes. Build se10 by clicking on “bash_env” in the se10 directory and then entering “`make`” in the bash window. The search paths defined in the makefile cause the edited Frameworks files in the project directory to be used in preference to the standard Frameworks files in the /Common directory.

Once se10.bin is built we must copy it into the EEPROM of the EZ-OTG mezzanine board. Create an EEPROM image using:

```
scanwrap se10.bin scanse10.bin 0x4a4
```

Finally, with DIP switches 6, 5, 4, and 3 ON enter:

```
qtui2c scanse10.bin f
```

This will copy the scan record file into the EEPROM. Now, when the EZ-OTG mezzanine board is reset, it will be a standalone, dual-role, Buttons and Lights device. We are now ready to test the dual-role example. The EZ-OTG mezzanine board will be operating independently and we will control the EZ-Host mezzanine board using the debugger. Set up the hardware as shown in Figure 6-3 and note that the only cable connection initially made is the debugger connection of SIE2 on the EZ-Host board to the development system.

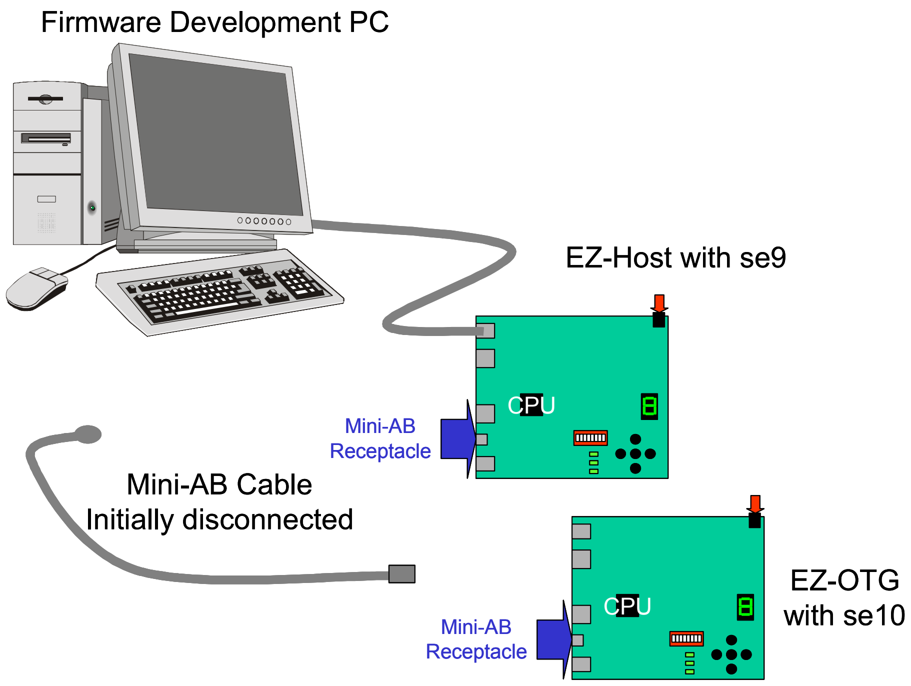


Figure 6-3. Hardware used to debug the dual-role example.

This example uses both SIEs of the EZ-Host mezzanine board: SIE1 as a dual-role connection and SIE2 for the debugger. We will need the BIOS startup modification code we developed in Chapter 3 loaded into the EZ-Host mezzanine board EEPROM before continuing. So, open a bash window using bash_env from the se2 directory and enter "`qtui2c eeprom.bin f`" to program the EEPROM. Close this bash window.

Open two bash windows on the development PC. In one window start the remote debug driver using "`cy16-elf-libremote -u`" and in the other window start the debugger using "`cy16-elf-gdb se9`." Click the RUN icon of Insight to get the program loaded, clear all breakpoints and click continue.

Initially, with no interconnecting cable, both boards will initialize as a peripheral – this is shown by the mezzanine board's PERIPHERAL LED. Now insert the A-end of the mini-AB cable into one of the boards, notice how it transforms itself into a host. Removing the cable forces the board to revert to being a peripheral device.

The host-side of the example powers Vbus in anticipation of starting a session – note that the SESSION LED comes on at the same time as the HOST LED comes on. Now, with the A-end of the mini-AB cable attached to one board plug the B-end of the cable into the other board – it's SESSION LED will come on alongside it's PERIPHERAL LED. Pressing the INC and DEC buttons will cause both displays to change.

The host will terminate the session if you press the SRP/SESSION END button. It will also start a new session if you press it again.

The host can change its local display even when a session is not active and this will cause the displays to become out of sync. Once a session is started then the displays will resynchronize.

The device can also start a session if you press its SRP/SESSION END button but the host must terminate the session.

While a session is active, press the HNP button on the default host (i.e. the one with the A-plug inserted). I implemented this as an "Offer HNP" and this gives the default peripheral an opportunity to swap roles and become a temporary host. This swapping of roles is indicated on the HOST and PERIPHERAL LEDs.

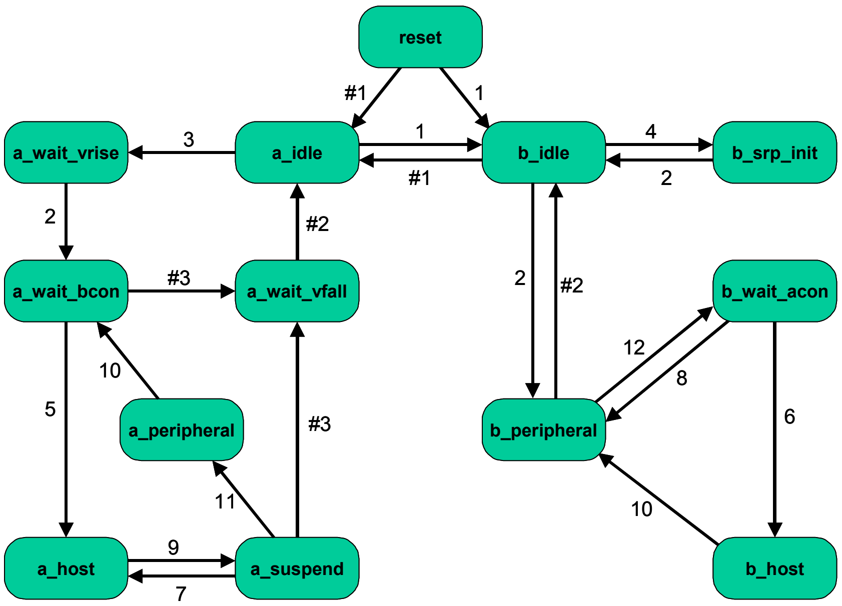
You can modify the operation of this dual-role device by changing the logic of the app_button_handler procedure. Have fun!

This simple example showed that adding dual-role capability, via SRP and HNP, to an application program is straightforward since the complexity is handled within Frameworks. The next section goes "behind-the-scenes" to explain what

Frameworks is doing. There is no need for you to modify this code – in fact; I would recommend that you do not. It has passed the USB-IF OTG protocol test suite and therefore is known to match the OTG Supplement Specification.

OTG behind-the-scenes

The OTG Supplement describes the two OTG protocols using a combination of state machines and text. This definition is very complete and covers all situations including error conditions. But this completeness does make it more difficult to explain, so I must thank my colleague Lane Hauck for producing a simplified version by removing exceptional conditions. I combined Lane's simplified A-device and simplified B-device state diagrams to produce the simplified dual-role device state diagram shown in Figure 6-4. I also redrew the diagram to better show the similarity of operation of both devices. Note that Frameworks implements the full protocol as required by the OTG Supplement Specification, and I am only using this simplified diagram for explanation purposes.



Key:	1 = OTGID, defined by plug	2 = Vbus Valid
	3 = A or B want to use the bus	4 = B wants to start a session
	5 = B signals a connect	6 = A signals a connect
	7 = B signals a resume	8 = B signals a resume
	9 = A finished, offer HNP	10 = B finished, suspend bus
	11 = HNP success, swap role	12 = B accepts HNP

# used to signify NOT

Figure 6-4. Simplified dual-role device OTG state machine

Figure 6-4 is in two halves – the left-hand side describes the operation of the A-device, or default host, and the right-hand side describes the B-device, or default peripheral. The only time a device would traverse from one side to the other is following a major event such as plugging or unplugging the A-end of a USB cable. In all other cases the A-device stays on the left and the B-device stays on the right. Note that each side has stable states for host and peripheral operation.

The diagram shows states, whose names are called out in the OTG Supplement, in rounded rectangles and shows state transitions as numbered arrows. A simplified description of the transition is shown in the key. States that have "wait" in their name have an associated timer, and these timers may cause states transitions also.

Both OTG protocols, SRP and HNP, use voltage levels on the USB wires to signal progress through the state machine. The default connection of two dual-role devices, shown in Figure 1-9 and repeated in Figure 6-5 for convenience, will also be used in this discussion.

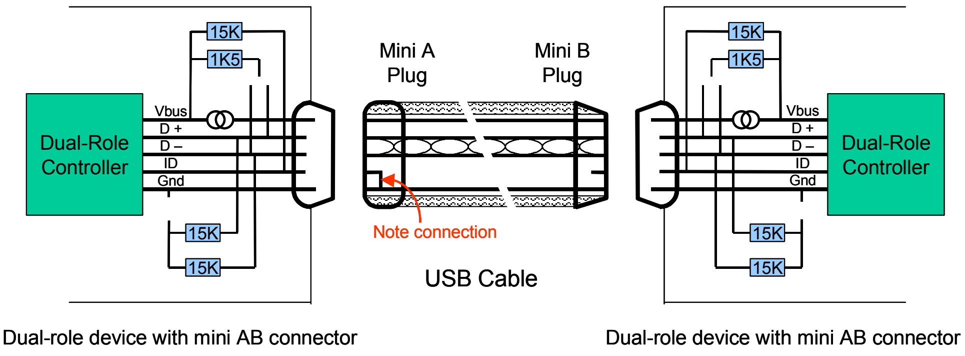


Figure 6-5. Default dual-role device connection

The cable in Figure 6-5 is shown adjacent to the connectors for clarity. This discussion assumes that the USB cable is connected at both ends. The A-end of the cable defines the default A-device and the B-end of the cable defines the default B-device. Note that no data line biasing resistors are connected at initial system power-on. The A-device will be in the ***a_idle*** state and the B-device will be in the ***b_idle*** state. In the idle state both devices will turn on their pulldown resistors so the bus will be in a SE0 state. This is point 1 in Figure 6-6, which shows time advancing down the page and signals increasing positively to the right.

Since this is the first time the A-device has been powered up it will start an OTG session to discover if anything is connected to its root hub. It turns on Vbus and moves to ***a_wait_vrise*** waiting for Vbus to be valid. Once Vbus is valid the A-device will transition to ***a_wait_bcon***. The B-device meanwhile will also detect Vbus as valid and will transition to ***b_peripheral***, where it removes its pulldown resistors and attaches a pullup resistor to D+. This is point 2 in Figure 6-6. The A-device sees this connection as a rise in D+ voltage and transitions to ***a_host***. The A-device will enumerate the B-device and will issue standard USB requests, which, in our example, will be the interchange of HID reports created by the pressing of the INC and DEC buttons. This is the standard operating mode of USB, and this too is shown in Figure 6-6.

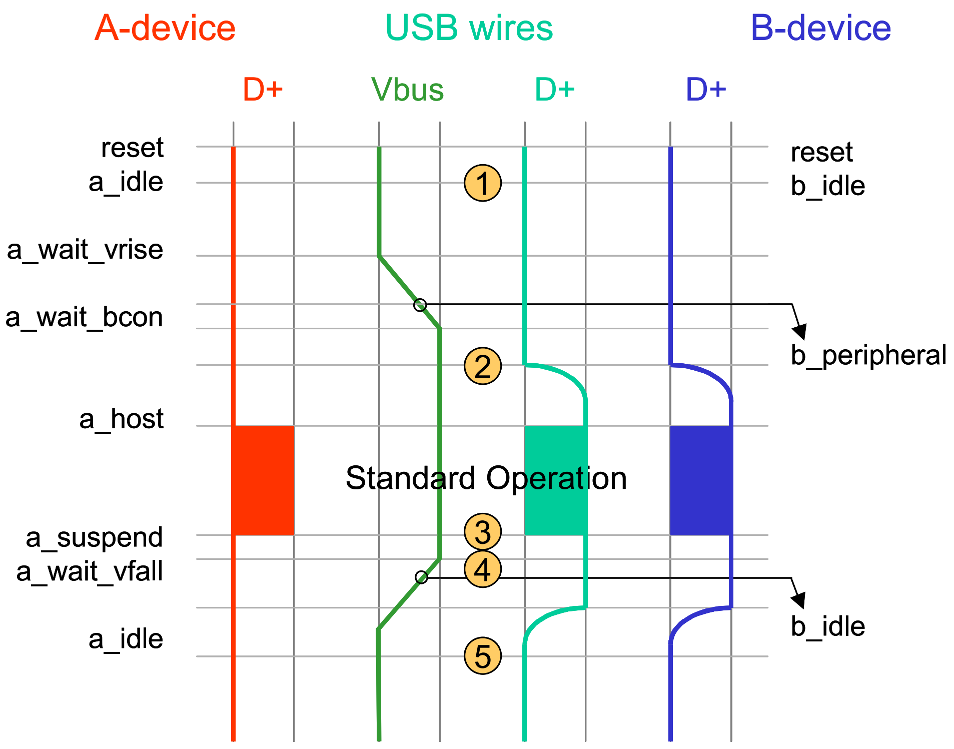


Figure 6-6. State transitions from power-on

In our example, we defined responses to button presses to mean that the host has completed its task and can remove Vbus. It first issues a SetFeature(HNP) command to the device, which will allow it to take control of the bus if it desires. The A-device will then transition to ***a-suspend***. This is point 3 in Figure 6-6.

On arrival at ***a_suspend*** the A-device will start a timer, ***a_aidl_bdis_tmr***. If the B-device wishes to take control of the bus, then it must indicate its intent by disconnecting its D+ pullup resistor before this timer expires. In our example the B-device does not need the bus yet so it does not disconnect. This causes the A-device to transition to ***a_wait_vfall*** where it turns off Vbus to end the session. This is point 4 in Figure 6-6. Once Vbus has fallen below the valid session voltage the A-device will transition to ***a_idle***. The removal of Vbus will cause the B-device to transition to ***b_idle***. We are back where we started at power on! This is point 5 in Figure 6-6.

We can start a new session by pressing the SRP button on either board. If we press the A-device button the sequence will run exactly as it did for the power-on case. Note that a real-life A-device does not need to initiate SRP since it is the host and therefore may start a session whenever it desires – the button-press is for demonstration purposes in this example. In real-life it will be the B-device that initiates a session using SRP; this example uses a button-press as the starting action. The B-device, realizing that Vbus is absent, must initiate a Session Request Protocol. It will transition to ***b-srp-init***.

Session Request Protocol

A B-device must employ two methods to signal an SRP to an A-device – the first is “data-line pulsing” and the second is “Vbus pulsing.” The A-device is required to respond to at least one of these methods. The B-device must wait until Vbus is lower than (VA_SESSION_VLD min) and the data lines have been in a SE0 state for at least 2 msec before it is allowed to start signaling.

Data-line pulsing is the simpler method since the B-device need only attach its D+ pullup for a period of 5 to 10 msec. Unfortunately, some non-compliant devices can cause this method to fail. This is shown as point 1 in Figure 6-7.

Vbus pulsing relies on time constants to charge and discharge a known capacitance at the A-device. A dual-role device will have a maximum capacitance of 6.5uF while a standard host will have a minimum capacitance of 97uF. By driving the Vbus line with a constant current of 8mA the B-device can successfully generate a signal pulse that meets the OTG Supplement Specification. The EZ-Host/EZ-OTG components integrate a Vbus source, and Frameworks uses an internal timer to generate a pulse of the correct width. This is point 2 in Figure 6-7.

After generating the SRP signal, the B-device returns to ***b_idle*** and waits for Vbus to turn on. This is point 3 in Figure 6-7.

In response to the SRP signaling the A-device will turn on Vbus and will transition to ***a_wait_vrise***. Once the session voltage is valid, the A-device will transition ***to a_wait_bcon*** and the B-device will transition to ***b_peripheral***. The A-device will detect the B-device and will transition to ***a_host***. Following enumeration, since the A-device has no pending USB transfers to implement, it will transition to ***a_suspend***. This is the same sequence as the power-on case described in Figure 6-6. This is point 3 in Figure 6-6 and point 4 in Figure 6-7.

The B-device will detect this suspend but this time, since it has some USB transfers it needs to implement, and it has received a SetFreature(HNP) from the A-device, it transitions to ***b_wait_acon***. In this state the B-device detaches its pullup resistor causing the bus to fall to an SE0 state. This is point 5 in Figure 6-7.

The host detects this SE0 state on the bus and treats it as positive acknowledgement that the peripheral wants to switch to a host role. The host therefore transitions to ***a_peripheral*** where it attaches a pullup to its D+ line. This is point 6 in Figure 6-7.

The B-device detects the attachment of the pullup resistor and transitions to ***b_host*** where it operates as a standard USB host in control of all USB transfers. This is point 7 in Figure 6-7.

We have swapped roles!

When the B-device has completed its data transfers it will suspend the bus and transition into ***b_peripheral***. The A-device will detect the suspended bus and will transition to ***a_wait_bcon***. Since the A-device doesn't need the bus either, it then transitions to ***a_wait_vfall*** where it turns off Vbus. When Vbus drops below the valid session voltage both devices will return to their idle states, ***a_idle*** and ***b_idle***.

We are back at the power-on state. This is point 8 in Figure 6-7.

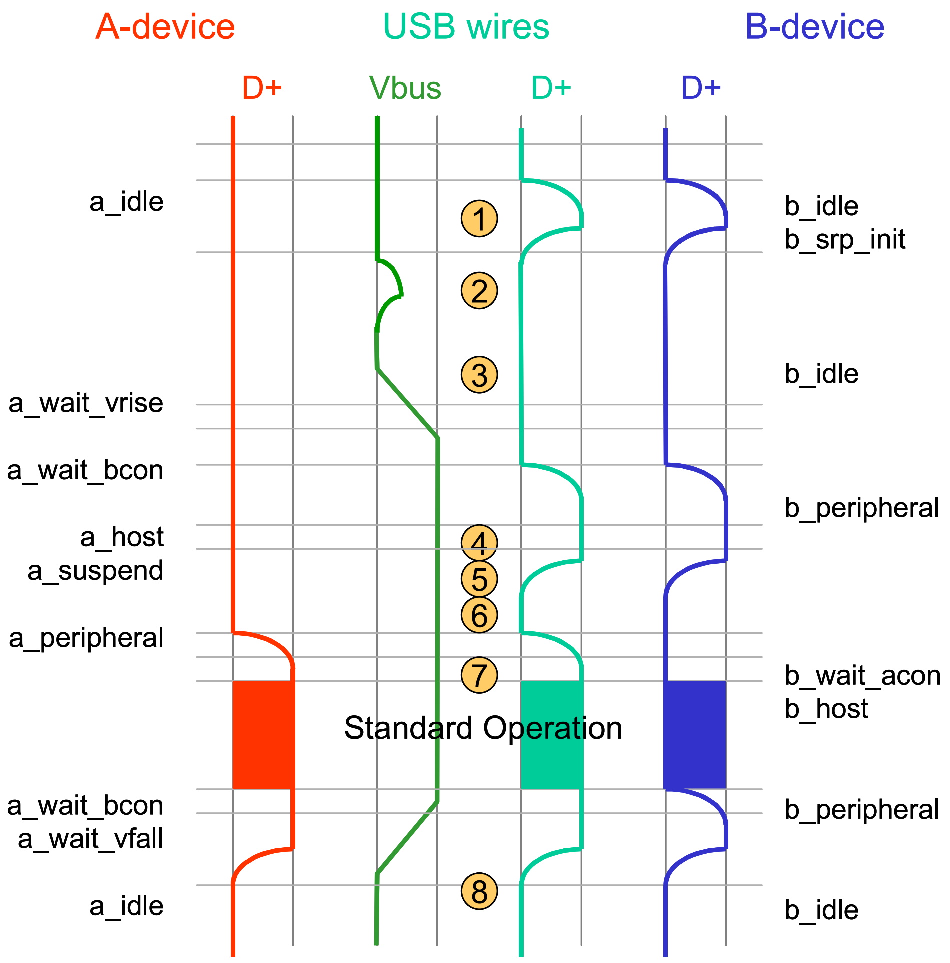


Figure 6-7. State transitions with B-device initiating SRP signaling

Chapter Summary

Designing a dual-role device with the EZ-Host/EZ-OTG is straightforward since the complexity is handled by integrated hardware and the Frameworks reference code. We worked through a simple example that used a “buttons and lights” host application program and a “buttons and lights” peripheral application program. The simple nature of the example allowed us to focus on the method, the process and the new elements of SRP and HNP. A more complex dual-role device, such as a digital still camera, would follow the same method – the host application program would resemble a printer class driver and the peripheral application program would resemble a mass storage class driver.

In the next chapter we will explore the Host Port Interface (HPI) that both the EZ-Host and the EZ-OTG components support. A “main” processor will use the EZ-Host/EZ-OTG as a co-processor. The HPI interface is an orthogonal choice with respect to the USB interfaces; therefore the examples that we have implemented so far would all operate. We will be able to, of course, do many more examples using this co-processor mode.

Chapter 7: Using EZ-Host/EZ-OTG in co-processor mode as a USB host controller

The previous chapters have described applications where the EZ-Host/EZ-OTG component has been used in standalone mode. In standalone mode the EZ-Host/EZ-OTG is the only processor in the system and it is responsible for running the application program and for managing the USB connections. These USB connections have been host, device or a combination of the two roles. The EZ-Host/EZ-OTG also support a co-processor mode, and this is an orthogonal choice with respect to the USB modes: this means that ALL of the examples that we have worked in this book so far could be re-partitioned into a “main-CPU” section that handled the application program and a “co-processor” section that handled the details of the USB connections.

We have seen that the EZ-Host/EZ-OTG are very capable sub-systems so they could handle ALL of the USB awareness of a project.

In co-processor mode the EZ-Host/EZ-OTG is a slave device to a main processor. The main processor is running the application program and, most likely, an operating system, and this processor is using the EZ-Host/EZ-OTG to manage a USB subsystem on its behalf. The communications channel between the main processor and the EZ-Host/EZ-OTG component can be implemented as a parallel interface using HPI or as a serial interface using HSS or SPI. My example will use HPI, but the software has been written such that a swap to HSS or SPI only affects a single module.

Embedded Linux was chosen as target for this example since this is popular in the intended application range of the EZ-Host/EZ-OTG components and the development environment is readily available. The examples use Linux release 2.4.18, and an overview of the layered structure of this Linux release is shown in Figure 7-1.

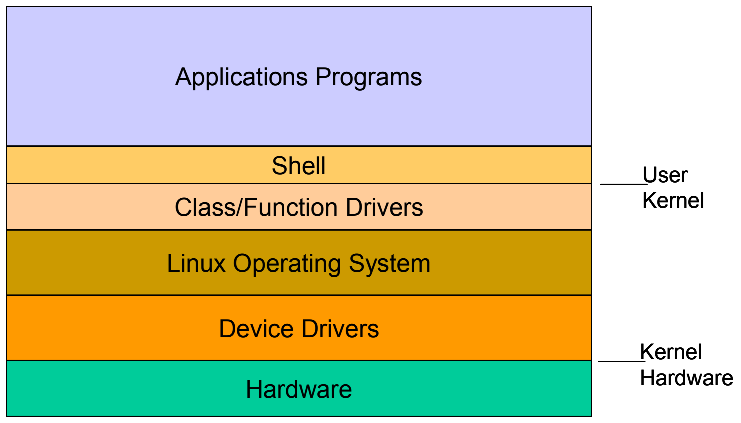


Figure 7-1. Linux is implemented in layers

Linux 2.4.18 is a USB-aware release: device drivers are included for a UHCI host controller and for an OHCI host controller. There is also an EHCI host controller in experimental release 2.5.x, and the source code for this is downloadable for review if required. The 2.4.18 release also contains several USB class drivers such as HID, audio, hub and mass-storage, and we shall utilize this capability later in this chapter.

Our project for this chapter, then, is to write a host controller driver for the EZ-Host/EZ-OTG component that will replace the UHCI/OHCI driver in the standard release. This is a well-defined problem since all of the software interfaces to Linux are already defined – we will be writing a standard Linux device driver, and many examples, books and tools are available to help us.

A USB host controller always has a root hub – this is part of the USB specification. In our case, the EZ-Host can have up to four root hubs, so our host controller driver will have to manage these. Four root hubs means that the EZ-Host can support four separate USB segments with up to 126 devices on each for a total of 504 devices. The EZ-OTG has two root hubs so it can support 252 USB devices.

We chose a StrongArm platform for the target system and the CY3663 co-processor development board is shown in Figure 7-2. StrongARM is well supported by Linux, and there are many device drivers and examples available. Many PDAs use StrongARM + Linux to deliver a capable hand held device.

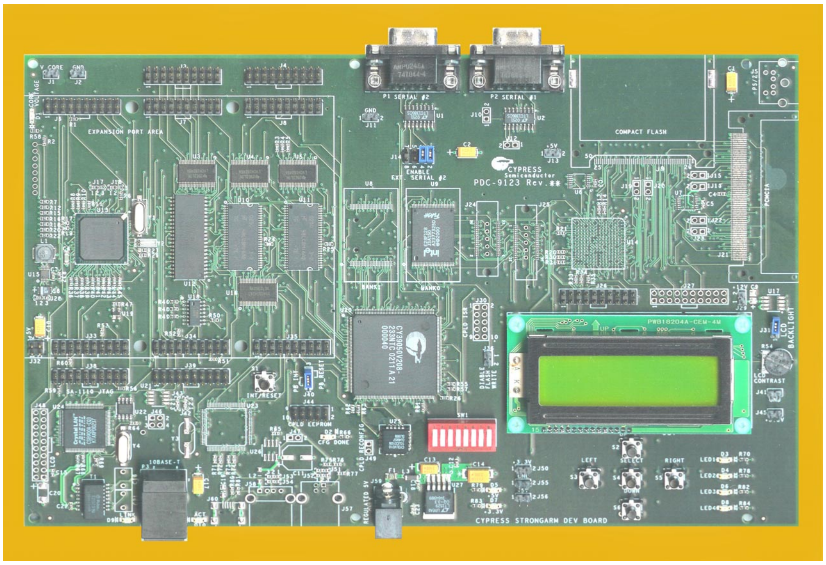


Figure 7-2. StrongARM co-processor development board.

A block diagram of the CY3663 development board is shown in Figure 7-3. The 133MHz StrongArm processor is supported with 16MB of Flash Memory, 512KB of SRAM, 32MB of DRAM, local IO including buttons, LEDs, seven segment display and a 2 line LCD display, an Ethernet connection and, most importantly for us, an expansion connector where the EZ-Host or EZ-OTG mezzanine boards can be attached.

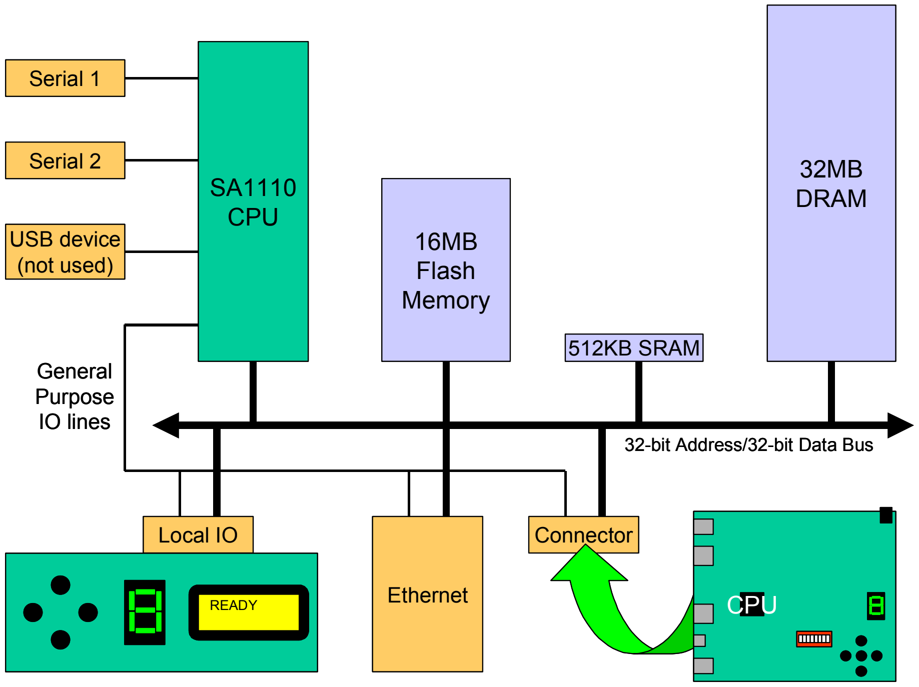


Figure 7-3. Block diagram of StrongArm coprocessor platform

The StrongArm processor is a 32-bit RISC CPU with a 32-bit address bus. The development platform sparsely populates this memory map. The flash memory is at 0 and is organized as shown in Figure 7-4. A Linux system requires a file system, and this example uses a Flash File System driver – the bootloader is in the first 128KB Flash block (with some configuration parameters in the next two blocks), and this loads a Linux image from Flash blocks 4 through 31. Flash blocks 32 through 127 are used as a 12MB disk drive. All of the Linux sources and build scripts are provided on the Cypress release CD, so we can rebuild the Linux kernel and copy new boot images to the Flash memory as often as required.

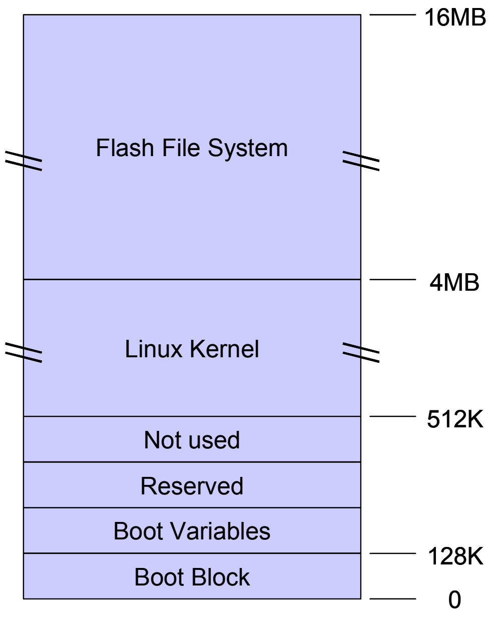


Figure 7-4. The flash memory operates as a disk drive.

The static Ram occupies physical memory space from 800000H to 87FFFFH, the dynamic memory occupies memory space from C0000000H to C1FFFFFFH and the IO is in memory page 48xxxxxxH. The StrongArm processor has several serial ports, and two are made available on the development platform. An Ethernet controller is also available. A pre-configured version of Linux is included in the Flash memory to support this hardware platform. The elements configured into the kernel are shown in Figure 7-5.

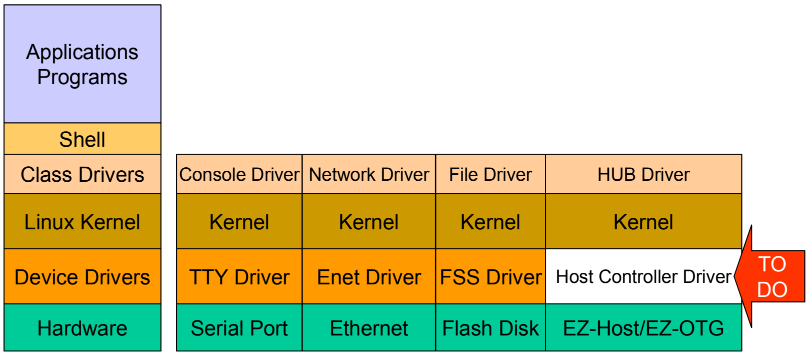


Figure 7-5. Starting Linux kernel implementation

USB Host Controller Driver

The EZ-Host/EZ-OTG host controller driver is built using three major pieces as shown in Figure 7-6. The host controller driver (hcd) accepts USB Request Blocks (URBs) from the kernel; hcd uses a low-level communications driver (lcd) to communicate with the EZ-Host/EZ-OTG component; and the virtual root hub driver (hcd_rh).

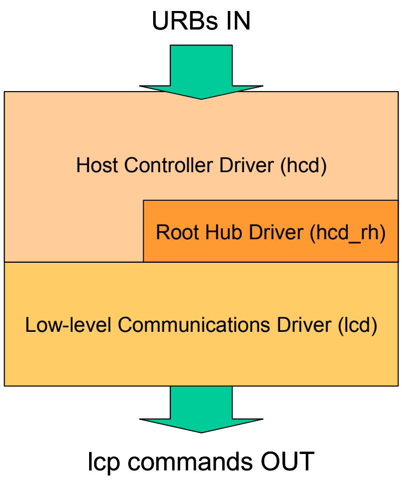

URBs IN
Host Controller Driver (hcd)
Root Hub Driver (hcd_rh)
Low-level Communications Driver (lcd)
lcp commands OUT

Figure 7-6. Structure of EZ-Host/EZ-OTG host controller driver.

We studied the operation of a host controller driver in Chapter 5. The Linux implementation in this chapter essentially does the same task but has been expanded to support up to four root hubs. This driver is processing many lists as shown in Figure 7-7.

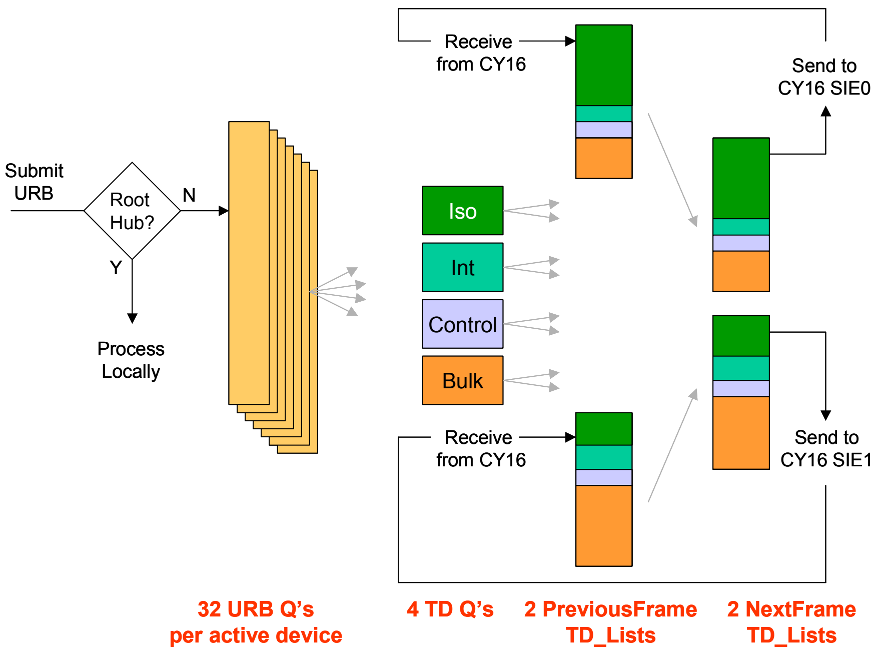


Figure 7-7. Host controller driver processes lists

The Linux kernel calls Submit_URB to initiate a USB transfer. The hcd first checks if the URB is targeted for one of the root hubs and diverts it if necessary. The hcd supports four root hubs, and this processing is described later. It is essential that multiple URBs sent to the same device endpoint are kept in order so the hcd manages up to 32 queues per active device – this equates to 16 input endpoints and 16 output endpoints as required by the USB specification.

The hcd will scan through the URB queues during its Idle_Task and will create one or more Transfer Descriptors (TD) for each new URB. The hcd will add these TDs to the TD_List matching the type of transfer that the URB requires (isochronous, interrupt, control or bulk).

Every 1 msec the hcd must supply a new NextFrame TD_List for each SIE within the EZ-Host/EZ-OTG component. It creates this list by examining the status of the TD_List from the previous frame and also including new transactions from the queued TD_Lists. It allocates TD's in order as defined by the UHCI specification: isochronous first, followed by interrupt, then control and bulk if time is available in the frame. Even though each SIE on the EZ-Host component has two host ports, the 12Mb/s bandwidth is shared, so the hcd need only be concerned that it is filling each NextFrame_TD_List to maximum 12 Mb/s capacity. Once the NextFrame_TD_List is built the hcd will use the lcd module, described next, to send the list to the EZ-Host/EZ-OTG component.

Low-level Communications Driver

The example low-level communications driver (lcd) uses the Host Port Interface (HPI) of the EZ-Host/EZ-OTG component to transfer data. If your application requires low USB bandwidth then the High-Speed-Serial (HSS) or Serial-Peripheral-Interface (SPI) could be used. All three mechanisms use the same Link-Control-Protocol (lcp) that is implemented by the EZ-Host/EZ-OTG BIOS. Figure 7-8 shows the hardware detail of HPI. From the main CPU's perspective, this is four 16-bit memory locations and an interrupt line.

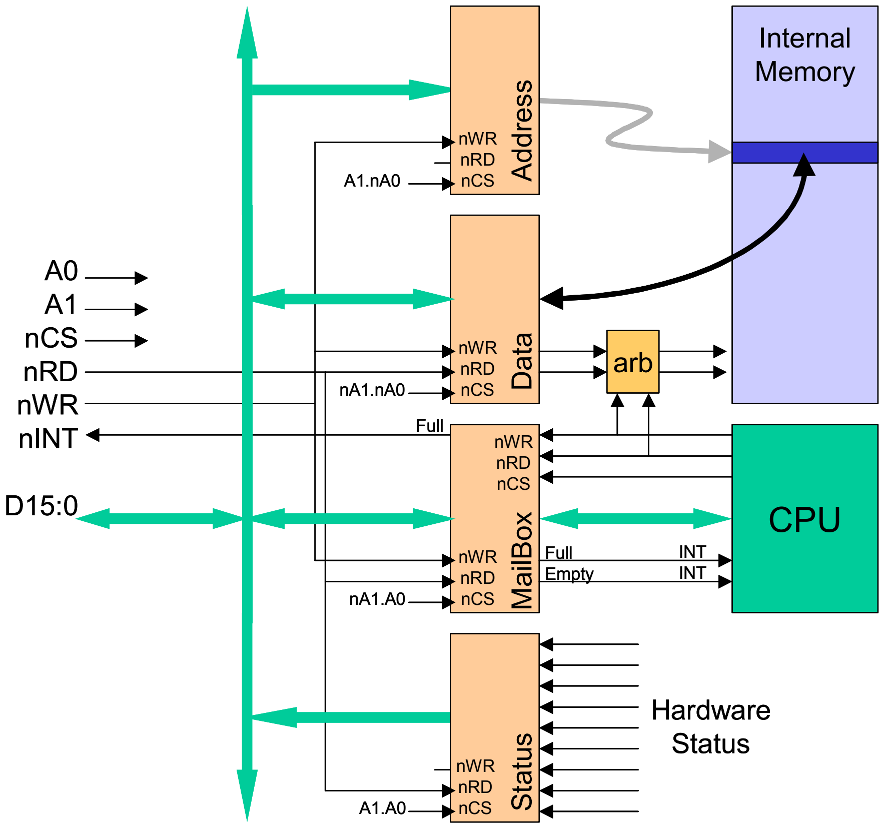


Figure 7-8. HPI hardware detail

The main CPU writes to the address register to setup a pointer into the internal memory of the EZ-Host/EZ-OTG. When the main CPU reads from, or writes to, the data register it is actually accessing internal memory locations. The address register is auto-incremented on each data register access so the main CPU can efficiently read or write blocks of internal memory. This HPI channel has priority access to the internal memory, and a transfer rate of 16MB/sec is achievable.

A main CPU write to the mailbox register will generate an interrupt to the EZ-Host/EZ-OTG CPU informing it that it should come and read the command that the main CPU has sent. When the EZ-Host/EZ-OTG has completed its command, it writes a response into the mailbox register, and this generates an interrupt to the main CPU. When the main CPU reads this response from the mailbox the EZ-Host/EZ-OTG CPU will be alerted via a separate interrupt. For those readers who remember the 8042 used in the early PC AT design to control the keyboard, this is the same mechanism but at least two orders of magnitude more capable!

The main CPU can also read a status register that summarizes the state of the pending interrupts of the EZ-Host/EZ-OTG component.

Link Control Protocol

BIOS implements a Link Control Protocol (LCP) to assure reliable data transfer using HPI (or HSS or SPI). The base set of commands is shown in Figure 7-9. You have learned, from previous chapters, that you can change or augment this command set to better suit your application.

```
COMM_RESET
COMM_JUMP2CODE
COMM_CALL_CODE
COMM_WRITE_CTRL_REG
COMM_READ_CTRL_REG
COMM_READ_XMEM
COMM_WRITE_XMEM
COMM_EXEC_INT
```

Figure 7-9. Base LCP Commands

Additionally a set of parameter registers, COMM_REG 0 through COMM_REG13, is also defined since all of the commands require data values. BIOS only implements a single parameter block, and this means that lcp commands must be executed serially. Typically hcd will generate many lcp commands, so lcd implements a queue and passes lcp commands to BIOS in the time order that they were requested. BIOS will generate a response for each command, and lcd will stage the next lcp command while executing the callback for the previous lcp command.

Root hub functionality

A root hub is a special case – it has all of the attributes of a standard hub (as defined by the USB specification), but it is not connected downstream of the USB host controller; it is embedded inside the host controller. From a software perspective this means that hcd should route URBs targeted at the root hub directly to a local hub driver rather than create TDs to be scheduled on the bus.

During initialization hcd will call rh_connect. The root hub module provides all of the descriptors required for a hub and simply does a USB_connect. The Linux USB core software will enumerate this device in the standard way and discover that its descriptors define it to be a hub. The kernel will therefore match the root hub with its hub class driver and will initialize it! It will create a device object and assign a USB_device ID to it. The Linux kernel is USB aware, and its core supports USB device enumeration and several USB class drivers. Our example uses the Linux kernel code and, therefore, there is not a lot of new code that we have to write to support root hub operation. Our example will actually call rh_connect four times to support the four root hubs on the EZ-Host component.

The root hub module will make calls into lcd to send commands and read status from the EZ-Host/EZ-OTG component.

Testing our host controller

The hcd example code is written to be part of the kernel code. During development we started writing the code as loadable modules but had problems with the order that the Linux kernel would initialize the subsystems – we fixed this with static binding into the kernel. We found that it took only about a minute to rebuild a kernel image after making changes to hcd and so continued on this route.

Open the Cypress/USB/OTG-Host/Source/coprocessor/linux directory and identify a Linux kernel configuration file called .config. I ran "`$make config`" from a bash window and selected options from the main build menu to create this example. Figure 7-10 summarizes the options chosen.

Console Driver	Network Driver	File Driver	HUB	HID	Audio	Mass Storage
Kernel	Kernel	Kernel	Kernel			
TTY Driver	Enet Driver	FSS Driver	Host Controller Driver (hcd, lcd, hcd-rh)			
Serial Port	Ethernet	Flash Disk	EZ-Host/EZ-OTG			

Figure 7-10. Linux configuration for coprocessor example.

I then ran "`$make dep`" to create the required dependency lists and then "`$make Image`" to create a bootable Linux kernel image which I named linux.img. Full instructions on building a kernel and downloading this into the Flash memory of the StrongArm development board are detailed in the Cypress document "CY3663 Hardware User's Manual." Follow these instructions to download vmlinux.img. Attach the OTG mezzanine board to the StrongArm board and an RS232 cable to serial port 1 of the StrongArm board. This RS232 cable should be attached to your development PC that is running a terminal program, such as Hyperterminal, at 115200 baud. Your hardware should look like the setup shown in Figure 7-11.

As a download alternative you could set up your Firmware Development PC to be a tftp host. This Ethernet connection is explained in the "CY3663 User's Manual" and is worth the effort to set up if you plan on creating a range of Linux images for development and debug. This is also shown in Figure 7-11.

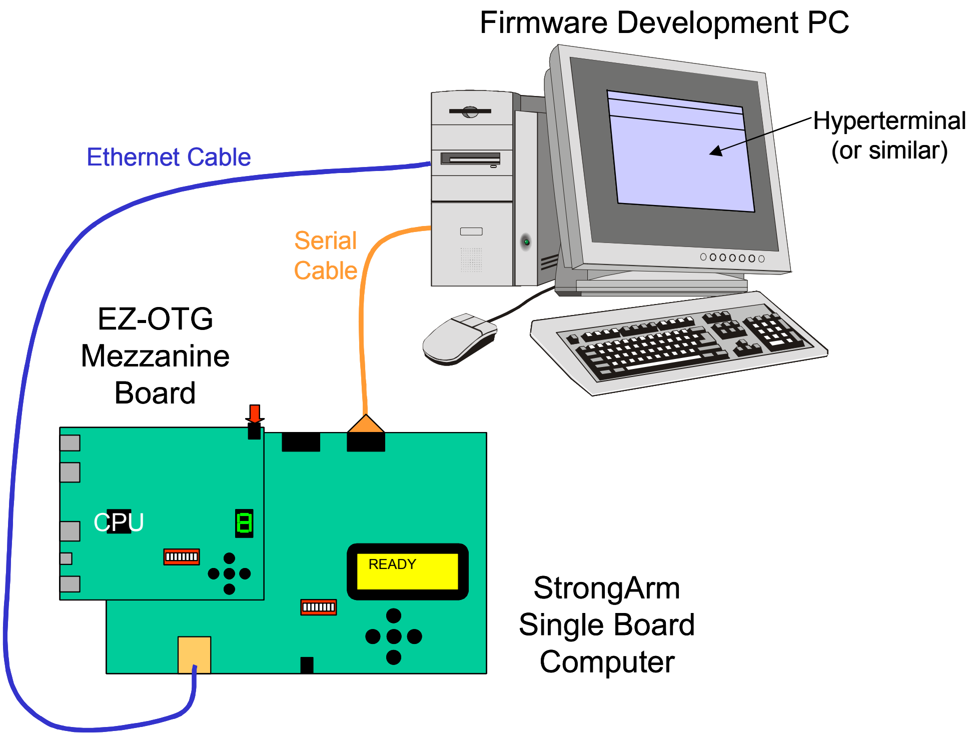


Figure 7-11. Hardware ready to test coprocessor example

Set the DIP switches on the StrongArm board to all OFF and set the mezzanine DIP switches to all OFF. Now attach power to the mezzanine board or the StrongArm board. The mezzanine board will be held in RESET while the StrongArm board boots the Linux kernel from its Flash memory. The LCD display will display "Initializing…", and a verbose collection of kernel messages will be displayed on the terminal. These messages will indicate the progress of the Linux initialization. The LCD display will then change to "Ready", and you will be prompted to Login. Use the username "root." You now have a complete embedded Linux system ready for action! Explore the directory structure and files within the Flash file system, and note that you have a USB hcd installed.

You can now plug USB devices into the mezzanine board. They will be identified and a description will be displayed on the terminal. You can try whatever you have in your lab but I would recommend attaching a set of USB speakers to one mezzanine port and a portable Flash drive into the other. Both will be recognized and will match class drivers integrated into the vmlinux.img kernel. The Flash drive may not automatically mount its filesystem – if not enter "`mount –t vfat /dev/sda1 /mnt/usbhd`." Now copy cypress.wav from the root file system onto your flash drive. If you have a large wav file on your flash drive already then you can use that in the next step.

Now enter the following:

```
./bplay -d /dev/dsp -s 11000 -b 8 /mnt/usbhd/cypress.wav
```

You will hear sound on your speakers.

Bplay is using bulk transfers (via the mass storage class driver) to read data from the Flash disk and is using isochronous transfers (via the audio class driver) to send this data to your speakers. The transactions were set up using control and interrupt transfers. All of this data is passing through our hcd and being passed to the mezzanine board by lcd.

The EZ-Host/EZ-OTG based host controller is working!

The Cypress Release CDROM contains many more examples. Starting from C:/Cypress/USB/OTG-Host/Source/coprocessor, look in the following subdirectories:

- de_app
- linux/drivers/usb/cy7c67300/dedrv
- linux/drivers/usb/cy7c67300/usbd/dedev

Again all of the source code and build scripts are provided so that you can get a head start on your project. Some of these examples use Linux as a device and some as a dual-role device. A device driver template, originally written by Lineo, was used to develop a device-side function driver that was readily integrated into the Linux kernel. Linux has rich USB support and we are beginning to see this same underlying support being added to other embedded operating systems.

Chapter Summary

We integrated the EZ-Host/EZ-OTG into a system as a co processor. We chose a system that was already USB-aware so that we could focus on the EZ-Host/EZ-OTG aspects of the project. As a co processor the EZ-Host/EZ-OTG managed up to four root hubs (two when using the EZ-OTG) on behalf of an embedded Linux implementation. This off-loading of the USB communications task gave the Linux "main" CPU more time to implement other tasks.

We essentially swapped out a PC-based UHCI controller driver for an embedded EZ-Host/EZ-OTG based driver. The project was well defined since standard Linux device driver interfaces were used. We developed a low-level communications driver to isolate the hardware dependencies from the host controller driver. Since Linux is USB-aware it includes many USB class drivers and we demonstrated moving a wav file from a mass storage device to an audio device.

Throughout this book the EZ-Host/EZ-OTG components have been used in a wide range of applications. Example code has been provided for each application and the source code and build scripts will enable you to choose an example close to your intended use and tune it to best fit your application solution.

I trust that this book has given you a head start with unlocking the potential within the EZ-Host and EZ-OTG components.

I wish you success in your USB Design projects

John Hyde