# ECE385
# DIGITAL SYSTEMS LABORATORY
## Introduction to USB and EZ-OTG on Nios II

### Embedded System with Nios II

In Lab 7, we introduced the Nios II embedded processor. We said in class that the Nios II is ideal for low speed tasks which would require a huge number of states/logic to do in hardware. USB enumeration for a HID device is one of these tasks, since the speed (for a human-interface device such as a keyboard) is very low, but there are a prohibitive number of states/cases to efficiently handle in hardware.

In Lab 8, the USB protocol is handled in the software on the Nios II, and the extracted keycode from the USB keyboard is then sent to the hardware for further use. In this tutorial, we will cover the software part of the project.

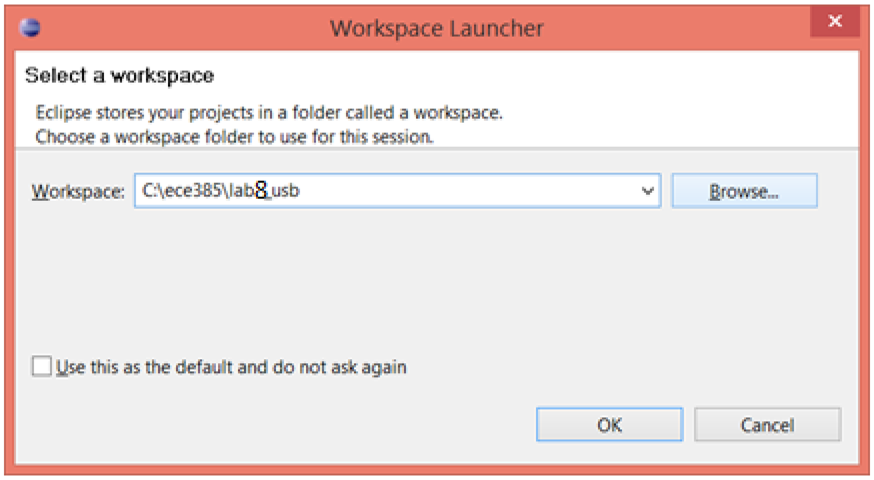
To start, download lab8_usb.zip provided on the course website. Create a folder, say, C:\ece385\lab8_usb, and extract all the zipped files into the folder.

In Quartus II, click on ***File > Open Project…*** and open lab8_usb.qpf in the folder we just created. This loads the hardware part of the project. Go to the ***Files*** tab in the ***Project Navigator*** on the upper left, we will see the included files: a few SystemVerilog source code files (that end with extension .sv), and a IP variation file, usb_system/synthesis/usb_system.qip. The QIP file describes a Nios processor, other IPs that it uses, and the connections between them. The resulting system is equivalent to a SystemVerilog module. In fact, we can see the generated Verilog source code, usb_system.v, by expanding the usb_system.qip item in Project Navigator. You may also view the system in a graphical user interface by going to ***Tools > Qsys*** and opening usb_system.qsys. In this lab, the QIP file is ready to use and requires no further modification.

Compile the project and program it onto the DE2-115 board. Sometimes the programmable file lab8_usb.sof may appear in a subfolder *output_files* instead of the *lab8_usb* root directory.

Go to ***Tools > Nios II Software Build Tools for Eclipse*** to launch the software development environment. Specify the workspace path to be the same as your Quartus II project, as shown in Figure 1.

IUQ. 2



Select a workspace

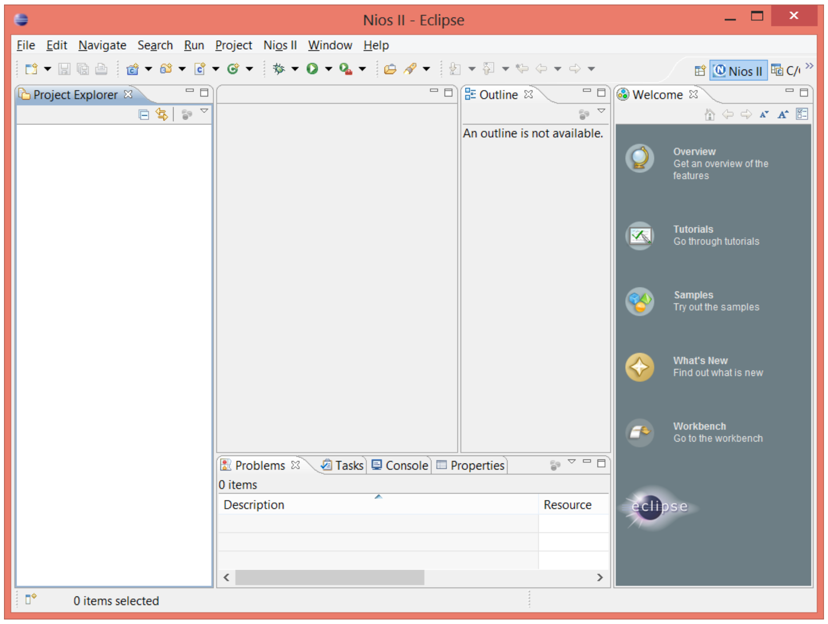
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\ece385\lab8_usb

Use this as the default and do not ask again

Figure 1

The Eclipse window will show up, as in Figure 2.



An outline is not available.

0 items

0 items selected

Figure 2

Set perspective to Nios II by going to ***Menu*** > ***Window***> ***Open Perspective*** > ***Other*** > ***Nios II*** or by clicking on the ***Nios II*** icon on the upper right of the window.

Next, we will import the provided software source code into the workspace. Right click on the blank space in Project Explorer on the left and select ***Import…*** (or alternatively, **Files** > **Import…**). Select ***Nios II Software Build Tools Project*** > ***Import Nios II Software Build Project*** , as shown in Figure 3, and Click ***Next***.

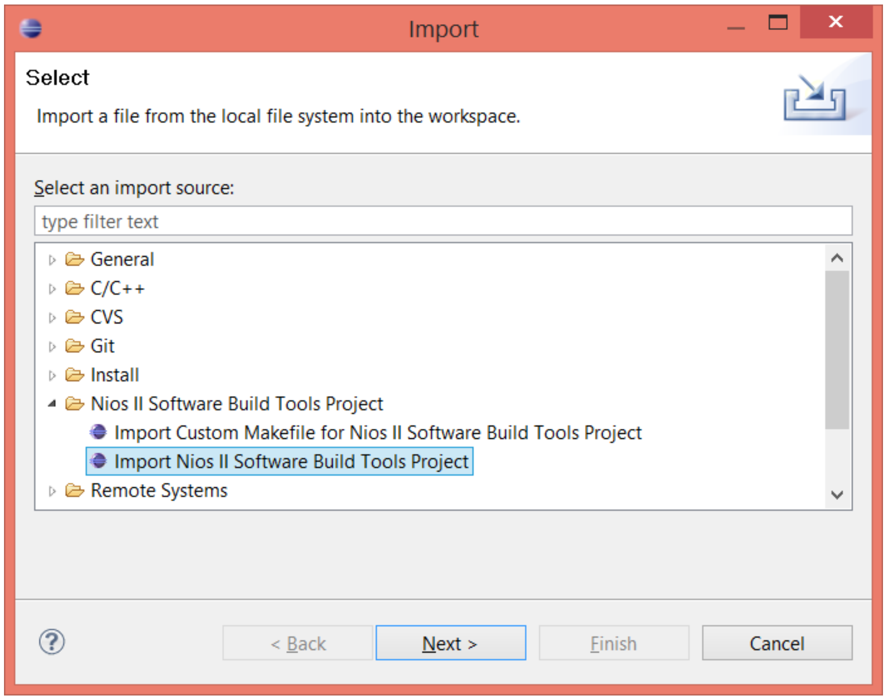
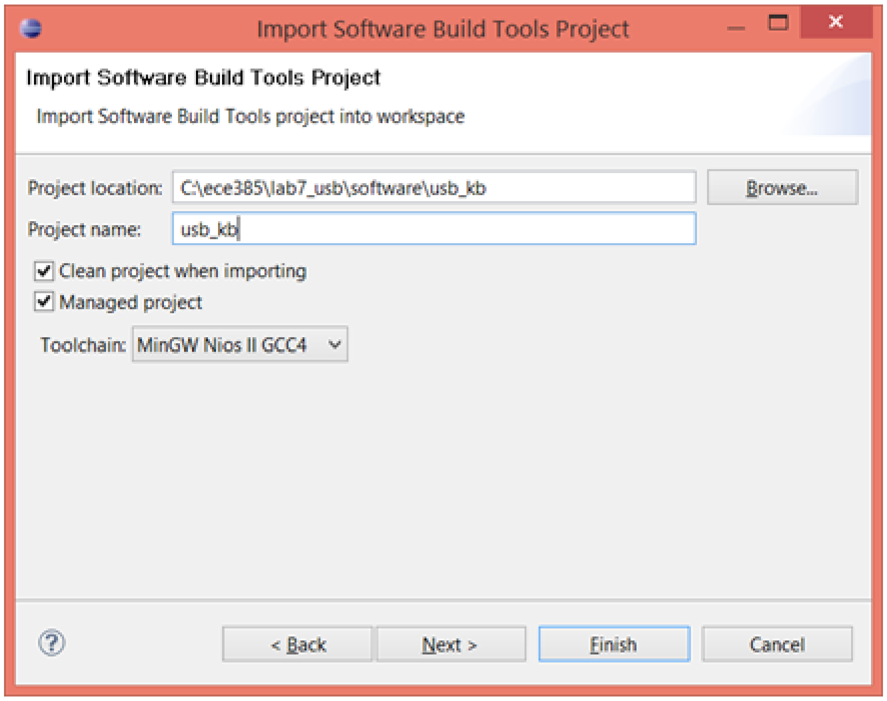

Import
Select
Import a file from the local file system into the workspace.
Select an import source:
type filter text
General
C/C++
CVS
Git
Install
Nios II Software Build Tools Project
Import Custom Makefile for Nios II Software Build Tools Project
Import Nios II Software Build Tools Project
Remote Systems
< Back
Next >
Finish
Cancel

Figure 3

In the next screen Browse for the software\usb_kb folder in the project folder. Type in Project name as usb_kb, as shown in Figure 4. Make sure ***Clean project when importing*** is selected. Click ***Finish***. This is the main software program we will be working on.


Import Software Build Tools Project
Import Software Build Tools Project
Import Software Build Tools project into workspace
Browse...
< Back
Next >
Finish
Cancel

Project location: C:\ece385\lab7_usb\software\usb_kb
Project name: usb_kb
☑ Clean project when importing
☑ Managed project
Toolchain: MinGW Nios II GCC4

Figure 4

IUQ. 4

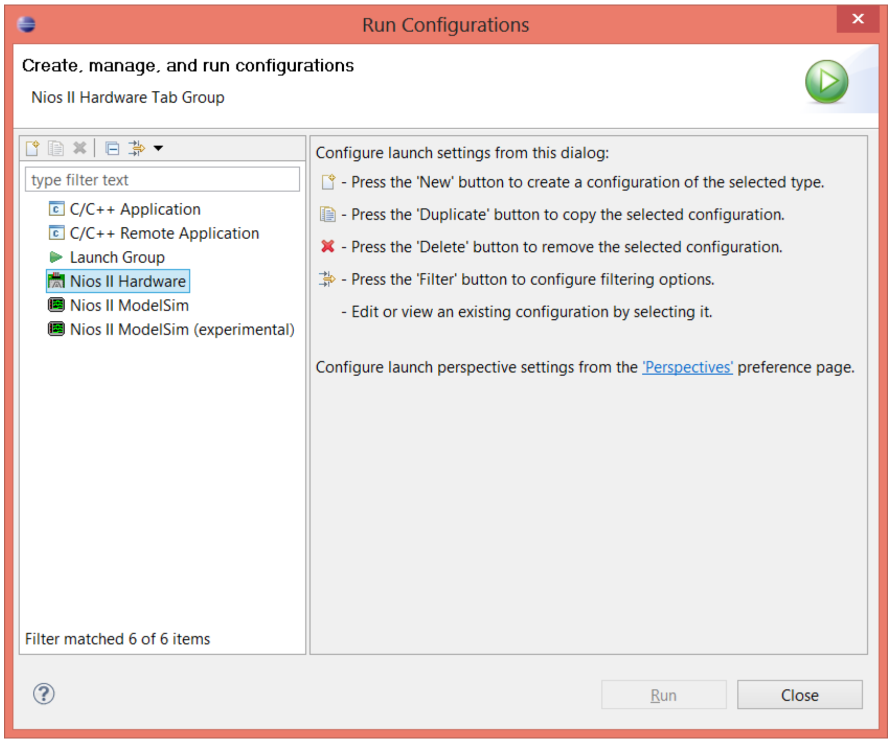
Import another project called *usb_kb_bsp* stored in *software\usb_kb_bsp*. The BSP project defines hardware information so the software can be compiled and run correctly.

Now we have two projects, *usb_kb*, and *usb_kb_bsp* in the Project Explorer. Right click on *usb_kb_bsp* and select ***Nios II > Generate BSP.*** This configures the compilation environment to be compatible with the hardware design. Click on the main project, usb_kb, and then go to ***Project > Build All*** to compile the program.

**IMPORTANT**:
Whenever the hardware part is changed, it needs to be compiled in Quartus II and programmed on the FPGA. If the programmer fails to load the .sof file on the FPGA, it's likely because the software is occupying the USB Blaster port. Simply stop the program or restart the FPGA board if this is the case.

On the software side, make sure to right click on <*project_name_bsp*> and select ***Nios II > Generate BSP*** so the latest hardware information is included in the Makefile. Then, compile the program again (***Build All***). Compatibility errors occur if you fail to do so!

To run the program on Nios II, click on ***Run*** > ***Run Configurations***… to open up a dialog window, as shown in Figure 5.

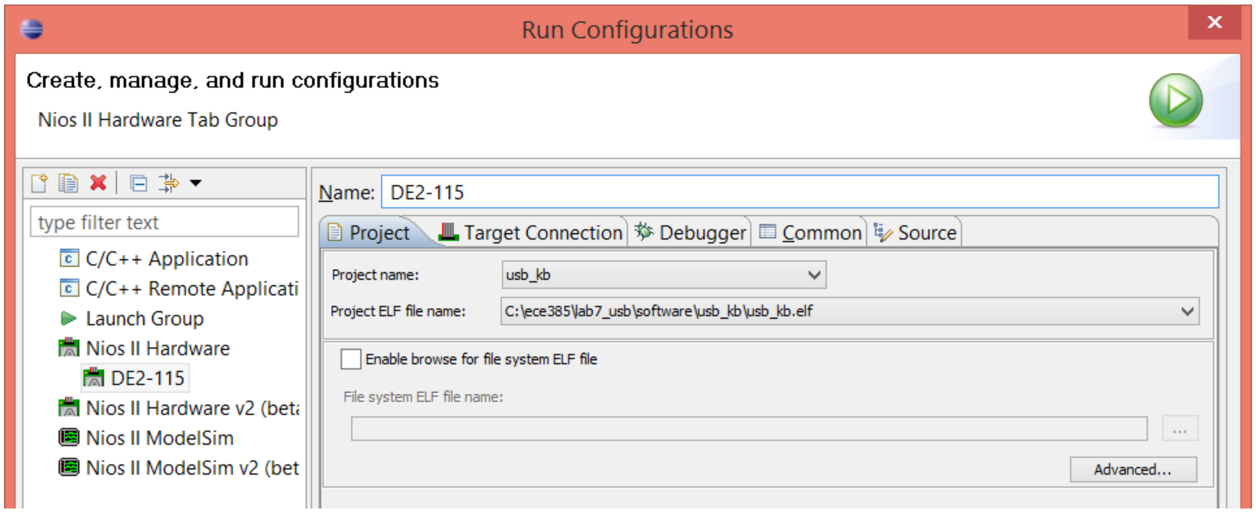


Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Figure 5

Right click on ***Nios II Hardware*** and select ***New***. Type in the name of the configuration as *DE2-115*. In the ***Project*** tab, select Project name to be the project you are working with. The corresponding ELF file should automatically be set up. See Figure 6. The ELF file is the compiled software file that is downloaded into the system memory and run on Nios II.



Run Configurations

Create, manage, and run configurations

Nios II Hardware Tab Group

Name: DE2-115

Project name: usb_kb

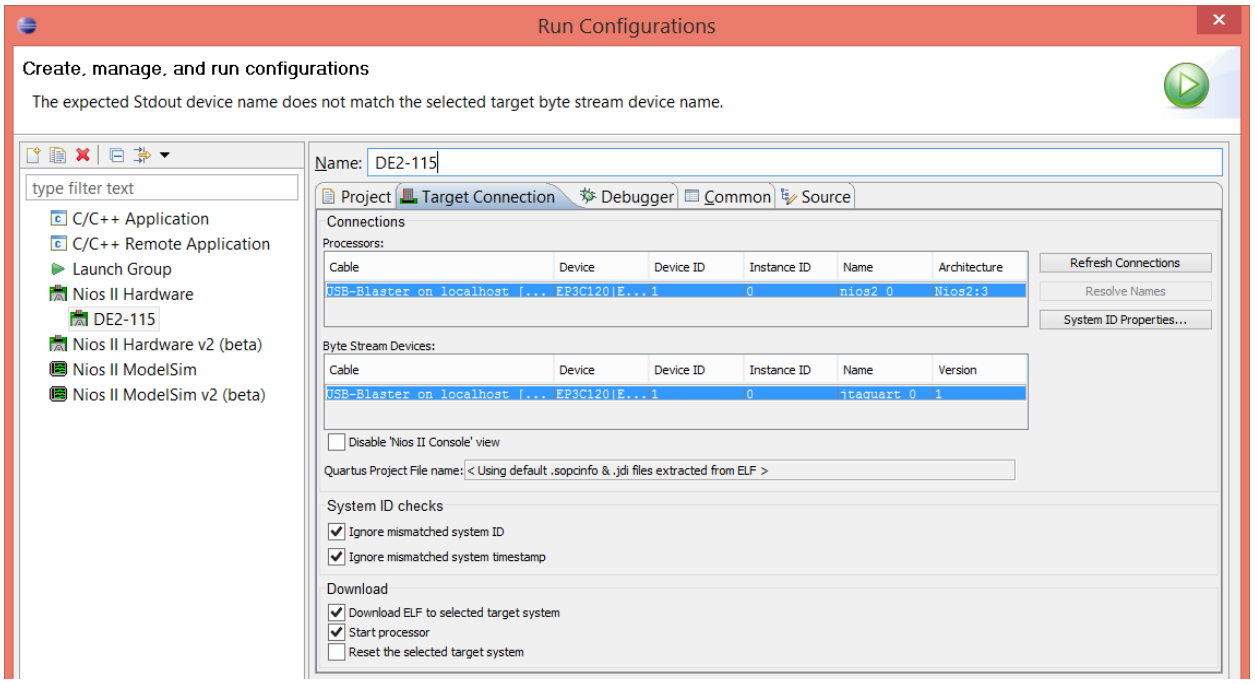
Project ELF file name: C:\ece385\lab7_usb\software\usb_kb\usb_kb.elf

☐ Enable browse for file system ELF file

File system ELF file name:

Figure 6

Go to the ***Target Connection*** tab. Click on Refresh Connections on the right. Make sure the *Processors* and the *Byte Stream Devices* are both *USB-Blaster on localhost*. If an error message "*Connected system ID hash not found on target at expected base address*" appears, check the options of *Ignore mismatched system ID* and *Ignore mismatched system timestamp*. See Figure 7.



Run Configurations

Create, manage, and run configurations

The expected Stdout device name does not match the selected target byte stream device name.

Name: DE2-115

Connections

Processors:

| Cable | Device | Device ID | Instance ID | Name | Architecture |
|---|---|---|---|---|---|
| USB-Blaster on localhost [... | EP3C120\|E... | 1 | 0 | nios2_0 | Nios2:3 |

Byte Stream Devices:

| Cable | Device | Device ID | Instance ID | Name | Version |
|---|---|---|---|---|---|
| USB-Blaster on localhost [... | EP3C120\|E... | 1 | 0 | jtaguart_0 | 1 |

☐ Disable 'Nios II Console' view

Quartus Project File name: < Using default .sopcinfo & .jdi files extracted from ELF >

System ID checks

☑ Ignore mismatched system ID

☑ Ignore mismatched system timestamp

Download

☑ Download ELF to selected target system

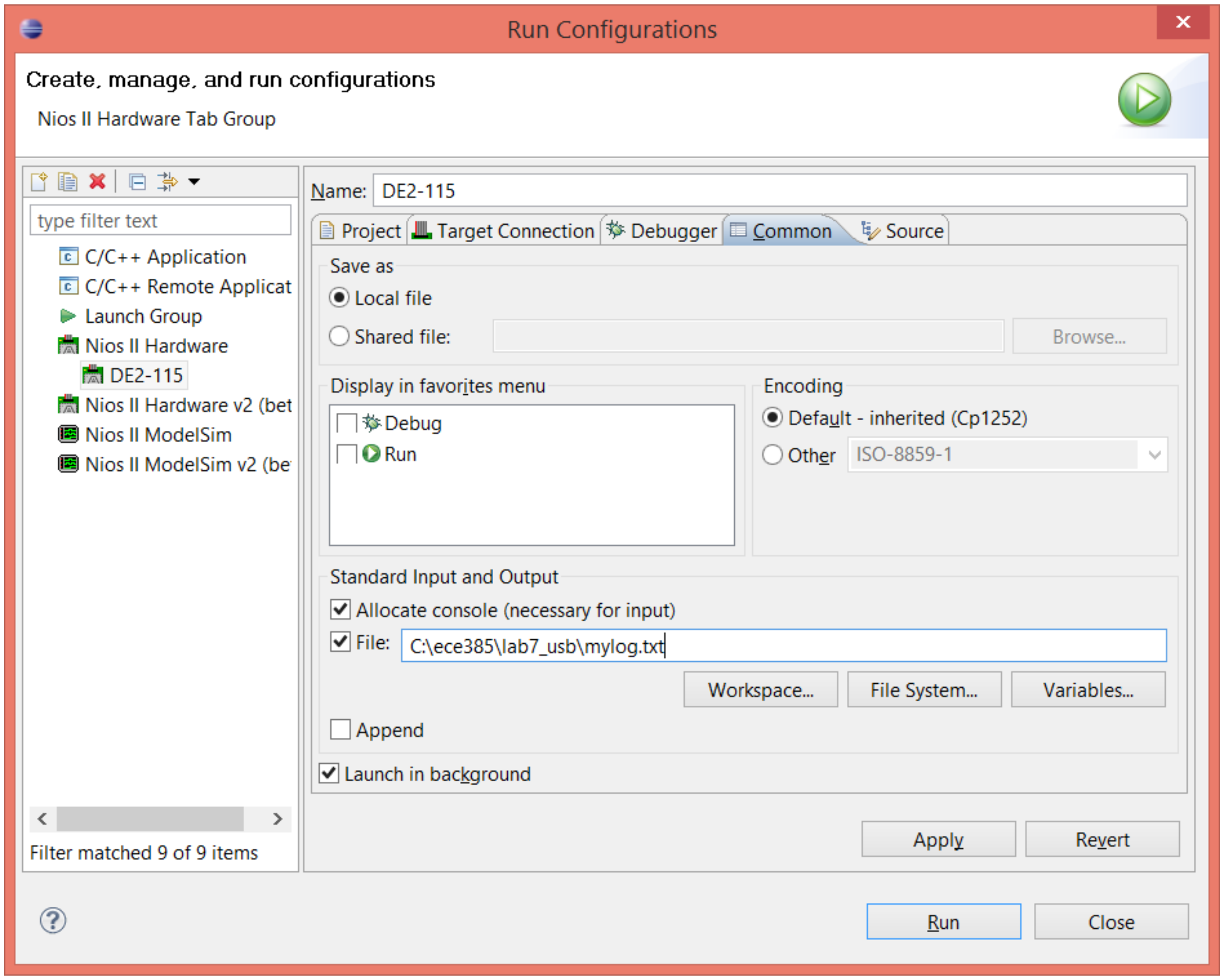
☑ Start processor

☐ Reset the selected target system

Figure 7

IUQ. 6

Go to the ***Common*** tab. Check both ***Allocate console*** and ***File*** under *Standard Input and Output*. Type in a path for your log file, such as *C:\ece385\lab8_usb\mylog.txt*, as shown in Figure 8. This will log some of the error/warning messages in case you encounter them, which helps a lot in troubleshooting.



Name: DE2-115

Save as
- [✓] Local file
- [ ] Shared file:

Display in favorites menu
- [ ] Debug
- [ ] Run

Encoding
- [✓] Default - inherited (Cp1252)
- [ ] Other ISO-8859-1

Standard Input and Output
- [✓] Allocate console (necessary for input)
- [✓] File: C:\ece385\lab7_usb\mylog.txt
- [ ] Append

- [✓] Launch in background

Figure 8

Click ***Run*** to run the program. A Nios II console should appear on the bottom of the Eclipse environment. Standard I/O is made on the Nios II console on your PC, transmitted via the USB Blaster cable to the DE2-115 board and to the program running on the Nios II processor.

## USB Host Programming with the EZ-OTG (CY7C67200) Chip

The Cypress EZ-OTG (CY7C67200) chip handles the USB protocol. OTG stands for On-The-Go, which is an add-on specification on top of the standard USB 2.0 standard that allows not only PCs but also other mobile devices to act as a USB Host. The EZ-OTG chip itself contains a RISC microprocessor (CY16), RAM, and ROM (BIOS), as shown on the left in Figure 9. The RAM can be accessed through direct memory access (DMA) at addresses 0x0000 – 0x3FFF. The Serial

Interface Engines (SIEs) are the front end of the USB controller and are where the USB ports are attached. The connections between EZ-OTG and the DE2-115 FPGA board are made through the Host Port Interface (HPI), which are shown as the connections in the center of Figure 9. The noteworthy pins are HPI_D[15:0], which is the 16-bit parallel data bus; HPI_INT, which indicates the event of interrupt; HPI_A[1:0], which indicates the addresses to four port registers on the chip. Each of the port registers control some important functionalities on EZ-OTG, some of which will be used later. The port registers and the addresses are listed in Table 1.

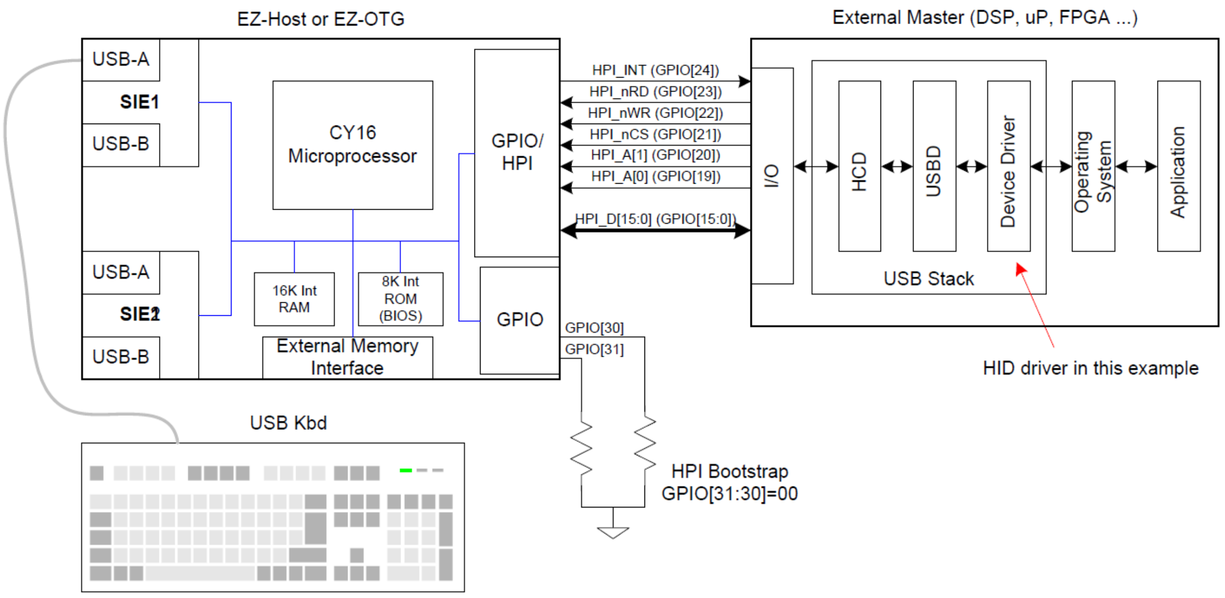


Figure 9

| Port Registers | HPI A [1] | HPI A [0] | Access |
|---|---|---|---|
| HPI DATA | 0 | 0 | RW |
| HPI MAILBOX | 0 | 1 | RW |
| HPI ADDRESS | 1 | 0 | W |
| HPI STATUS | 1 | 1 | R |

Table 1

We need to program and configure EZ-OTG in order to make it act as a Host Controller and establishes a connection with the USB keyboard. In the program, we will do the following:

- Set up the EZ-OTG chip.
- Detect connection and assign an address to the connected device.
- Identify device type from descriptors.
- Set configuration.
- Poll keyboard data.

In the Nios II system, most of the peripheral IPs are memory-mapped, that is, we are able to read and write from these IPs via memory access functions. Two important functions (provided by Altera, defined in *HAL/inc/io.h*) that read and write from the peripherals are introduced here:

`IORD(base, offset):`
Read and return data from the memory location specified by (base address + offset). Offset is word-wise, that is, an offset of 1 is equivalent to a 32-bit or 4-byte offset in the address.
`IOWR(base, offset, data):`
Write data to the memory location specified by (base address + offset).

Hint: In Eclipse, right click on a function/variable name and choose Open Declaration will lead you to the function/variable. This is very helpful when you need to trace your source code in multiple files.

By using these functions, we can communicate with the EZ-OTG. The standard procedure of reading is:

1. Write the address to access to HPI_ADDR.
2. EZ-OTG will fetch the data from the specified address (e.g. an address in the RAM) and make it ready to be transferred via HPI_DATA.
3. Read from HPI_DATA.
4. (Optional: continuous read) EZ-OTG will load the data at the next available address to HPI_DATA, that is, if more data are to be read from the next address, we can simply read again from HPI_DATA without giving the next address.

Similarly, the procedure of writing is:

1. Write the address to access to HPI_ADDR.
2. Write data to HPI_DATA in 16-bit little endian words.
3. EZ-OTG will store the data to the specified address (e.g. an address in the RAM).
4. (Optional: continuous write) If more data are to be written into the next address, we can simply write again to HPI_DATA. EZ-OTG will store the data into the next available address.

To further simplify the procedure, we define two helper functions in USB.c as follows:

```
alt_u16 UsbRead(alt_u16 Address)
{
      IOWR(CY7C67200_BASE,HPI_ADDR,Address);
      return IORD(CY7C67200_BASE,HPI_DATA);
}

void UsbWrite(alt_u16 Address, alt_u16 Data)
{
      IOWR(CY7C67200_BASE,HPI_ADDR,Address);
      IOWR(CY7C67200_BASE,HPI_DATA,Data);
}
```

These functions perform a single read/write without utilizing the continuous read/write feature. Here, alt_u16 is Altera's built-in data type of a 16-bit unsigned integer.

## USB Keyboard Enumeration

The keyboard enumeration procedure is done in *main.c*, following the **Example Data Transfer to Enumerate a USB** section in the **Cypress Using HPI in Coprocessor Mode with OTG-Host (AN6010)** document. The steps are summarized as follows:

Step 1a: Initialize EZ-Host/EZ-OTG registers.
Step 1b: Configure SIE1 as a host.
Step 2: USB_RESET.
Step 3: Set address.
Step 4: Get device descriptor.
Step 5: Get configuration descriptor (1).
Step 6: Get configuration descriptor (2).
(Get device info.)
Step 7: Set configuration.
(Make class requests.)
Step 8: Get HID descriptor (Class 0x21).
Step 9: Get report descriptor (Class 0x22).
And finally poll keyboard data.

Each step requires a series of reads and writes on the EZ-OTG registers and RAM. In the end of Steps 1-2, COMM_EXEC_INT (0xCE01) is written to HPI_MAILBOX to issue an execution, as shown in Figure 6 of the **AN6010** document. In the rest of the steps, a list of Transaction Descriptors (TDs) is written into the RAM, and then UsbWrite(HUSB_SIE1_pCurrentTDPtr, Address) is called to specify the start address of the TDs in the RAM so EZ-OTG can process the TDs. After the TDs are read and processed, EZ-OTG will set the SIE1msg bit of the HPI_STATUS register to 1 to indicate successful execution. The bit fields in HPI_STATUS are explained in Table 4 of the **AN6010** document. If the SIE1msg bit is not set to 1, the program should make another attempt to transfer the TDs and have EZ-OTG process them again.

**TASK: Complete Steps 3 through 9 in main.c by calling the appropriate functions defined in USB.c and USB.h.**

Hint: All tasks in the software part are marked with comment "// TASK:" in the source code.

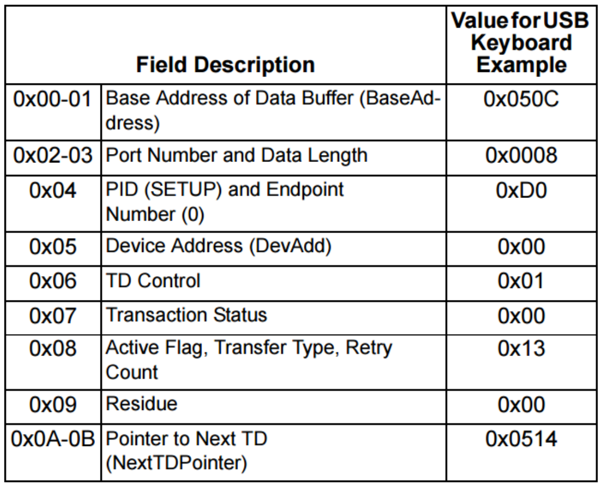
Most of these functions are provided and defined, but we will complete Step 3's UsbSetAddress() here to have a better understanding of how to work with TDs. Below is the pseudo code for UsbSetAddress().

```
void UsbSetAddress()
{
    Specify the start address of the TDs as 0x0500 by writing this value to HPI_ADDR.

    // TD #1
    Write the base address of data buffer (bit fields 0x00-0x01).
    Write the port number (bit field 0x02) and data length (bit field 0x03).
    Write the device address (bit field 0x05), PID as SETUP and endpoint number as 0 (bit field 0x04).
    Write the transaction status (bit field 0x07) and TD control (bit field 0x06).
    Write the residue (bit field 0x09) and active flag (bit field 0x08).
    Write the pointer to next TD (bit fields 0x0A-0x0B).
     (See Tables 6 and 7 in AN6010 for more details.)

    // TD #2
    Write the bRequest (bit field 0x01) and bmRequestType (bit field 0x00).
    Write the wValue (bit fields 0x02-0x03).
    Write the wIndex (bit fields 0x04-0x05).
    Write the wLength (bit fields 0x06-0x07).
     (See Tables 8 and 9 in AN6010 for more details.)

    // TD #3
    Write the base address of data buffer (bit fields 0x00-0x01)
    Write the port number and data length (bit fields 0x02-0x03)
    Write the device address (bit field 0x05), PID as IN and endpoint number as 0 (bit fields 0x04)
    Write the transaction status (bit field 0x07) and TD control (bit field 0x06).
    Write the residue (bit field 0x09) and active flag (bit field 0x08).
    Write the pointer to next TD (bit fields 0x0A-0x0B).
     (See Tables 10 and 11 in AN6010 for more details.)

    UsbWrite(HUSB_SIE1_pCurrentTDPtr,0x0500); //HUSB_SIE1_pCurrentTDPtr
}
```

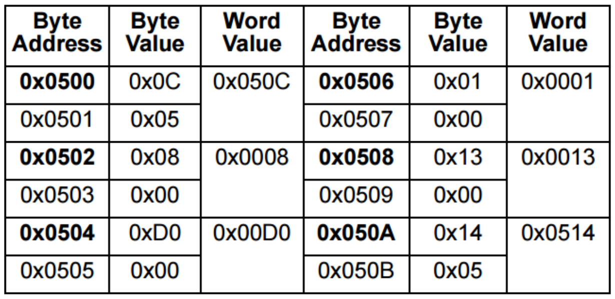
Note that each data write is a 16-bit word in little endian, which means two 8-bit fields should be combined to form a 16-bit word, with the first bit field placed in the lower bits. For example, two fields 0xAA followed by 0xBB should be combined into a word as 0xBBAA.

**TASK: Complete the UsbSetAddress() function in USB.c. Do this by following the above pseudo code and the values in the table below from the AN6010 documentation.**

Hint: Use the code given in UsbGetDeviceDesc1() with Tables 12 and 13 in the **AN610** document as a guide to complete USBSetAddress().



| Field Description | | Value for USB Keyboard Example |
|---|---|---|
| 0x00-01 | Base Address of Data Buffer (BaseAddress) | 0x050C |
| 0x02-03 | Port Number and Data Length | 0x0008 |
| 0x04 | PID (SETUP) and Endpoint Number (0) | 0xD0 |
| 0x05 | Device Address (DevAdd) | 0x00 |
| 0x06 | TD Control | 0x01 |
| 0x07 | Transaction Status | 0x00 |
| 0x08 | Active Flag, Transfer Type, Retry Count | 0x13 |
| 0x09 | Residue | 0x00 |
| 0x0A-0B | Pointer to Next TD (NextTDPointer) | 0x0514 |



| Byte Address | Byte Value | Word Value | Byte Address | Byte Value | Word Value |
|---|---|---|---|---|---|
| **0x0500** | 0x0C | 0x050C | **0x0506** | 0x01 | 0x0001 |
| 0x0501 | 0x05 | | 0x0507 | 0x00 | |
| **0x0502** | 0x08 | 0x0008 | **0x0508** | 0x13 | 0x0013 |
| 0x0503 | 0x00 | | 0x0509 | 0x00 | |
| **0x0504** | 0xD0 | 0x00D0 | **0x050A** | 0x14 | 0x0514 |
| 0x0505 | 0x00 | | 0x050B | 0x05 | |

Ref. Table 6 and 7 from the **AN610** document needed to complete **USBSetAddress()** TD#1

| Field Description | | Value for USB Keyboard Example |
|---|---|---|
| 0x00 | bmRequestType | 0x00 |
| 0x01 | bRequest | 0x05 |
| 0x02-03 | wValue | 0x0002 |
| 0x04-05 | wIndex | 0x0000 |
| 0x06-07 | wLength | 0x0000 |

| Byte Address | Byte Value | Word Value | Byte Address | Byte Value | Word Value |
|---|---|---|---|---|---|
| **0x050C** | 0x00 | 0x0500 | **0x0510** | 0x00 | 0x0000 |
| 0x050D | 0x05 | | 0x0511 | 0x00 | |
| **0x050E** | 0x02 | 0x0002 | **0x0512** | 0x00 | 0x0000 |
| 0x050F | 0x00 | | 0x0513 | 0x00 | |

Ref. Tables 8 and 9 from **AN610** document needed to complete **USBSetAddress()** TD#2

| Field Description | | Value for our USB Keyboard Example |
|---|---|---|
| 0x00-01 | Base Address of Data Buffer (BaseAddress) | 0x0000 (don't care) |
| 0x02-03 | Port Number and Data Length | 0x0000 |
| 0x04 | PID (IN) and Endpoint Number (0) | 0x90 |
| 0x05 | Device Address (DevAdd) | 0x00 |
| 0x06 | TD Control | 0x41 |
| 0x07 | Transaction Status | 0x00 |
| 0x08 | Active Flag, Transfer Type, Retry Count | 0x13 |
| 0x09 | Residue | 0x00 |
| 0x0A-0B | Pointer to Next TD (NextTDPointer) | 0x0000 (NULL) |

| Byte Address | Byte Value | Word Value | Byte Address | Byte Value | Word Value |
|---|---|---|---|---|---|
| **0x0514** | 0x00 | 0x0000 | **0x051A** | 0x41 | 0x0041 |
| 0x0515 | 0x00 | | 0x051B | 0x00 | |
| **0x0516** | 0x00 | 0x0000 | **0x051C** | 0x13 | 0x0013 |
| 0x0517 | 0x00 | | 0x051D | 0x00 | |
| **0x0518** | 0x90 | 0x0090 | **0x051E** | 0x00 | 0x0000 |
| 0x0519 | 0x00 | | 0x051F | 0x00 | |

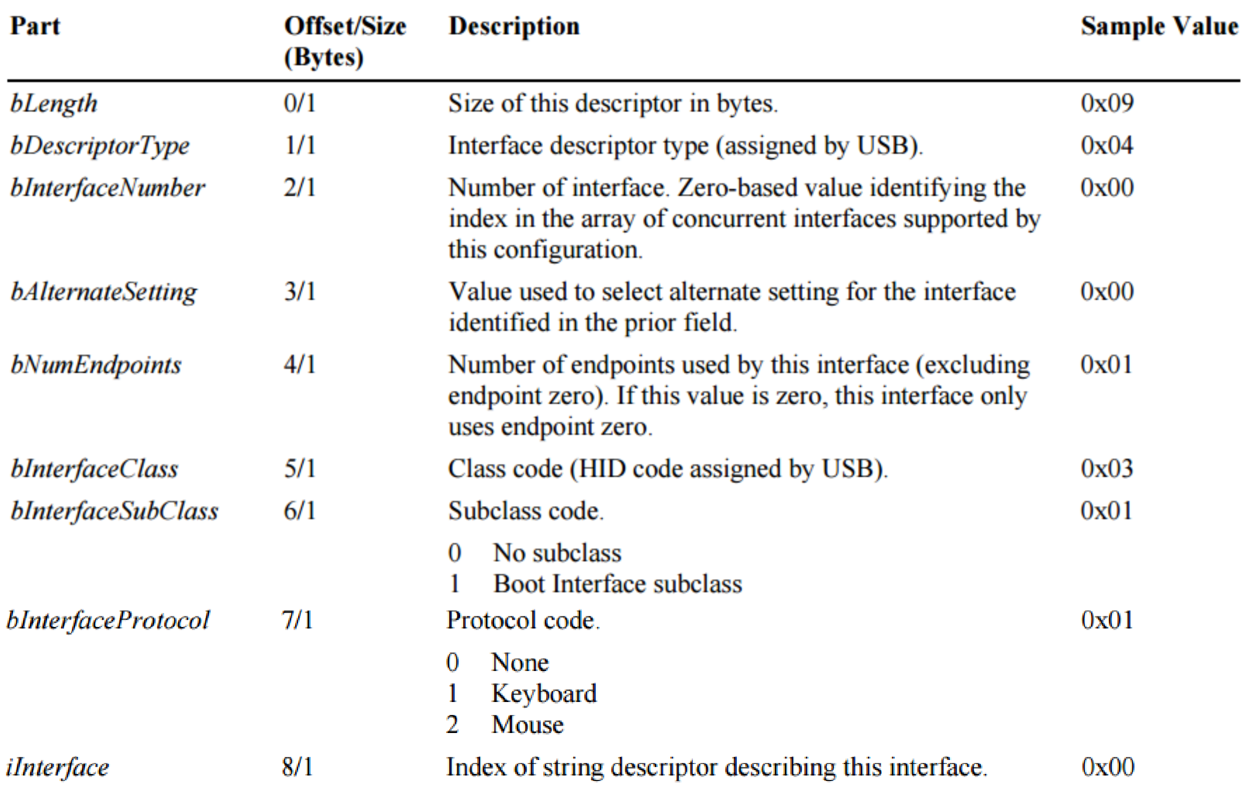
Ref. Tables 10 and 11 from **AN610** document needed to complete **USBSetAddress()** TD#3

Next, we will go to the section "get device info" for keyboard identification. The keyboard interface descriptor is stored in the RAM starting from address 0x0565 after Step 6 is done. We need to check if the protocol code (byte 7) is 0x01 to determine whether the connected device is indeed a keyboard.

**TASK: Check the device type by reading from the appropriate memory location for byte 7 of the interface descriptor.** Since HID devices can only have protocol code of 0x00, 0x01, or 0x02, you may check only the last two bits by ANDing the variable with 0x03 to obtain this information.

Hint: A helper function, UsbPrintMem(), is provided so you can print out the content of the EZ-OTG RAM. This can help you identify the correct byte to read by comparing the content with the HID specification.

IUQ. 12



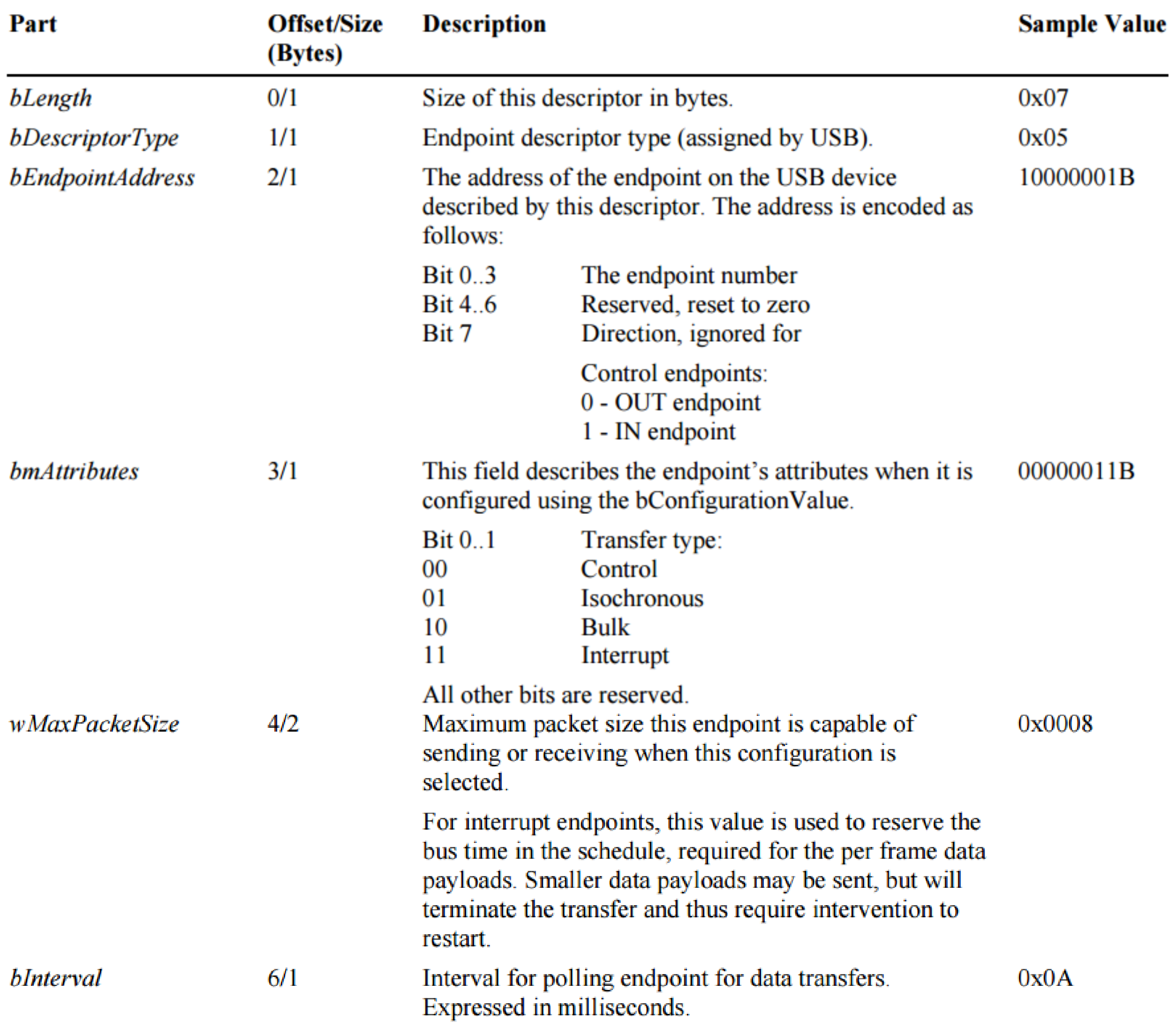
| Part | Offset/Size (Bytes) | Description | Sample Value |
|---|---|---|---|
| *bLength* | 0/1 | Size of this descriptor in bytes. | 0x09 |
| *bDescriptorType* | 1/1 | Interface descriptor type (assigned by USB). | 0x04 |
| *bInterfaceNumber* | 2/1 | Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration. | 0x00 |
| *bAlternateSetting* | 3/1 | Value used to select alternate setting for the interface identified in the prior field. | 0x00 |
| *bNumEndpoints* | 4/1 | Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses endpoint zero. | 0x01 |
| *bInterfaceClass* | 5/1 | Class code (HID code assigned by USB). | 0x03 |
| *bInterfaceSubClass* | 6/1 | Subclass code.<br>0 No subclass<br>1 Boot Interface subclass | 0x01 |
| *bInterfaceProtocol* | 7/1 | Protocol code.<br>0 None<br>1 Keyboard<br>2 Mouse | 0x01 |
| *iInterface* | 8/1 | Index of string descriptor describing this interface. | 0x00 |

Ref. Appendix E.3 in the **USB HID Specification**

Next, we will fetch the keyboard's packet size by reading the endpoint descriptor, available starting from address 0x0577. The endpoint descriptor's format is available in Appendix E.5 in the **USB HID Specification**.

**TASK: Get the data packet size by reading from the appropriate memory location for bytes 4-5 of the endpoint descriptor.**

Hint: Unfortunately, some keyboards do not generate a correct endpoint descriptor. Since the data packet size is required to correctly fetch keyboard data, our program will fail to read keycodes if this information is incorrect. Utilize the UsbPrintMem() function to check the content being read.
In the worst case, hard code the data size according to the keycode format to make the program work. i.e. set data_size = 0x08;



| Part | Offset/Size (Bytes) | Description | Sample Value |
|---|---|---|---|
| *bLength* | 0/1 | Size of this descriptor in bytes. | 0x07 |
| *bDescriptorType* | 1/1 | Endpoint descriptor type (assigned by USB). | 0x05 |
| *bEndpointAddress* | 2/1 | The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows:<br>Bit 0..3 The endpoint number<br>Bit 4..6 Reserved, reset to zero<br>Bit 7 Direction, ignored for<br>Control endpoints:<br>0 - OUT endpoint<br>1 - IN endpoint | 10000001B |
| *bmAttributes* | 3/1 | This field describes the endpoint's attributes when it is configured using the bConfigurationValue.<br>Bit 0..1 Transfer type:<br>00 Control<br>01 Isochronous<br>10 Bulk<br>11 Interrupt<br>All other bits are reserved. | 00000011B |
| *wMaxPacketSize* | 4/2 | Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected.<br>For interrupt endpoints, this value is used to reserve the bus time in the schedule, required for the per frame data payloads. Smaller data payloads may be sent, but will terminate the transfer and thus require intervention to restart. | 0x0008 |
| *bInterval* | 6/1 | Interval for polling endpoint for data transfers. Expressed in milliseconds. | 0x0A |

Ref. Appendix E.5 in the **USB HID Specification**

Finally, we get to the "get keycode value" section. This section mainly consists of a big while loop which continuously sends interrupt requests to the USB keyboard and then fetch data from it. The line IOWR(CY7C67200_BASE,HPI_DATA,0x051c) indicates that the report descriptor containing key press information generated by the keyboard should be stored at address 0x051c. The report descriptor is formatted in Table 2.

| Byte | Description |
|---|---|
| 0 | Modifiers |
| 1 | Reserved (0) |
| 2 | Keycode 1 |
| 3 | Keycode 2 |
| 4 | Keycode 3 |
| 5 | Keycode 4 |
| 6 | Keycode 5 |
| 7 | Keycode 6 |

Table 2
(Ref. Appendix B.1 in the **USB HID Specification**)

Each field contains a 16-bit value, which is two bytes. In this lab, we only need to consider the event in which only one key is pressed at a time, which means we only have to look at byte 2, which corresponds to the lower 8 bits of rbuf[5]. The keycodes for W, A, S, and D are 26, 04, 22, and 07, respectively. Upon release, the keycode becomes 0x00. A full list of keycodes is shown in Table 3.

**TASK: Get the keycode by reading from the appropriate memory location for byte 2 of the report descriptor.**

The line IOWR(KEYCODE_BASE, 0, keycode & 0xff) writes the fetched keycode to a parallel I/O peripheral, which is sent to a register (keycode) in the hardware.

The rest of Lab 7 should be completed in hardware. Note that the VGA has a slow clock of 60 Hz, which is used in the ball routine as its clock source, while the Nios II and the rest of the system has the regular 50 MHz clock, it's possible that a very short key press event can be missed. **Design a buffer register to latch the latest keycode to make sure it is read by the ball routine for extra robustness.**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| - | err | err | err | A | B | C | D | E | F | G | H |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| I | J | K | L | M | N | O | P | Q | R | S | T |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| U | V | W | X | Y | Z | 1 | 2 | 3 | 4 | 5 | 6 |
| 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 7 | 8 | 9 | 0 | Enter | Esc | BSp | Tab | Space | - / _ | = / + | [ / { |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| ] / } | \ / \| | ... | ; / : | ' / " | ` / ~ | , / < | . / > | / / ? | Caps Lock | F1 | F2 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | PrtScr | Scroll Lock |
| 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 |
| Pause | Insert | Home | PgUp | Delete | End | PgDn | Right | Left | Down | Up | Num Lock |
| 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 91 | 91 | 94 | 95 |
| KP / | KP * | KP - | KP + | KP Enter | KP 1 / End | KP 2 / Down | KP 3 / PgDn | KP 4 / Left | KP 5 | KP 6 / Right | KP 7 / Home |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
| KP 8 / Up | KP 9 / PgUp | KP 0 / Ins | KP . / Del | ... | Applic | Power | KP = | F13 | F14 | F15 | F16 |
| 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| F17 | F18 | F19 | F20 | F21 | F22 | F23 | F24 | Execute | Help | Menu | Select |
| 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 |
| Stop | Again | Undo | Cut | Copy | Paste | Find | Mute | Volume Up | Volume Down | Locking Caps Lock | Locking Num Lock |
| 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 |
| Locking Scroll Lock | KP , | KP = | Internat | Internat | Internat | Internat | Internat | Internat | Internat | Internat | Internat |
| 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 |
| LANG | LANG | LANG | LANG | LANG | LANG | LANG | LANG | LANG | Alt Erase | SysRq | Cancel |
| 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 |
| Clear | Prior | Return | Separ | Out | Oper | Clear / Again | CrSel / Props | ExSel | | | |
| 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | | | | |
| LCtrl | LShift | LAlt | LGUI | RCtrl | RShift | RAlt | RGUI | | | | |

Table 3