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, you will build a Qsys interface to handle to I/O interface to communicate with the USB-OTG chip. Then you will learn how to import existing NIOS II software and make changes to it to get the keyboard to work correctly.

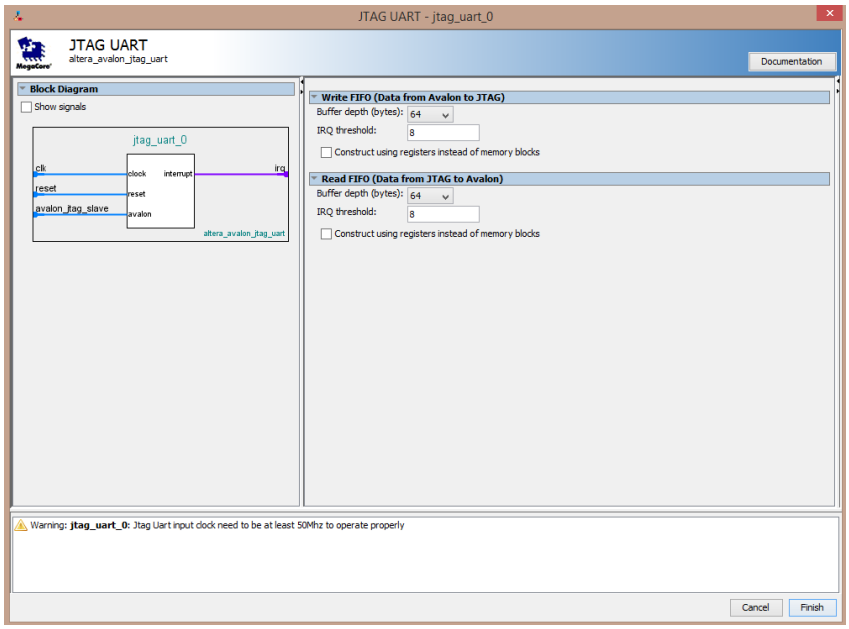
You should start with a working lab 7 Qsys setup. If you don't have one, you should follow the tutorial for lab 7 to complete the Qsys setup. Always keep a backup copy of old files that you are reusing especially if you know if they were working with the older version.

Starting from lab 7, remove the PIO modules in Qsys which correspond to the LEDs and switches you used in lab 7, as we will not be using them for lab 8.

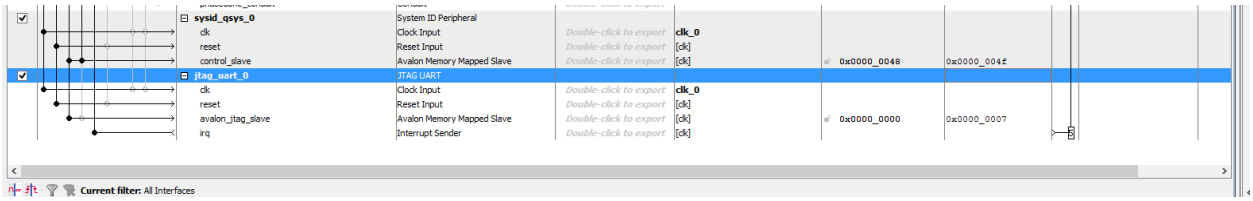
You will then need to add the JTAG UART peripheral. This is found under Interface Protocols->Serial. This is so that you can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

You can simply leave the settings here as a default. What this block does is give you the ability to use console (printf) commands from the Nios II which go through the programming cable via USB. While this is typically not a good user interface for an embedded system (as it requires the programming cable to be connected and the user to have all the Altera software installed), this is an excellent way to debug your software while in development.

IUQ. 2



Make the connections as shown below:



Make the standard connections for clk/reset/data bus, which is what swe've been doing for the other peripherals. One difference here is that we must assign an **interrupt** for this, which we will assign IRQ (interrupt request) 5. Connect the interrupt controller to IRQ and give it the number 5 (this is on the far right of the row. The reason for using interrupts here is that transmitting or receiving text over the console is in general very slow, and we don't want this procedure to block on the CPU. Therefore, the typical way in which printf is implemented is that control is returned program as soon as printf is called, but an interrupt is set up at the end of transmission for each buffer. This way, the CPU does not have to block while waiting for each of the characters to be transmitted, it is only interrupted whenever the peripheral (the JTAG UART) needs more data.

You need to add multiple PIO connections to the lab 8 setup, these are listed below with the direction of the ports and size of the ports.

Name	Direction	Width
keycode	Out	8
otg_hpi_address	Out	2
otg_hpi_data	Inout	16
otg_hpi_r	Out	1
otg_hpi_w	Out	1
otg_hpi_cs	Out	1
otg_hpi_reset	Out	1

You don't need the LED and switches PIO's anymore for this lab, so you can uncheck it to disable it.

This is how all the PIO's and JTAG UART should look when connected up (this only include additions from lab 7).

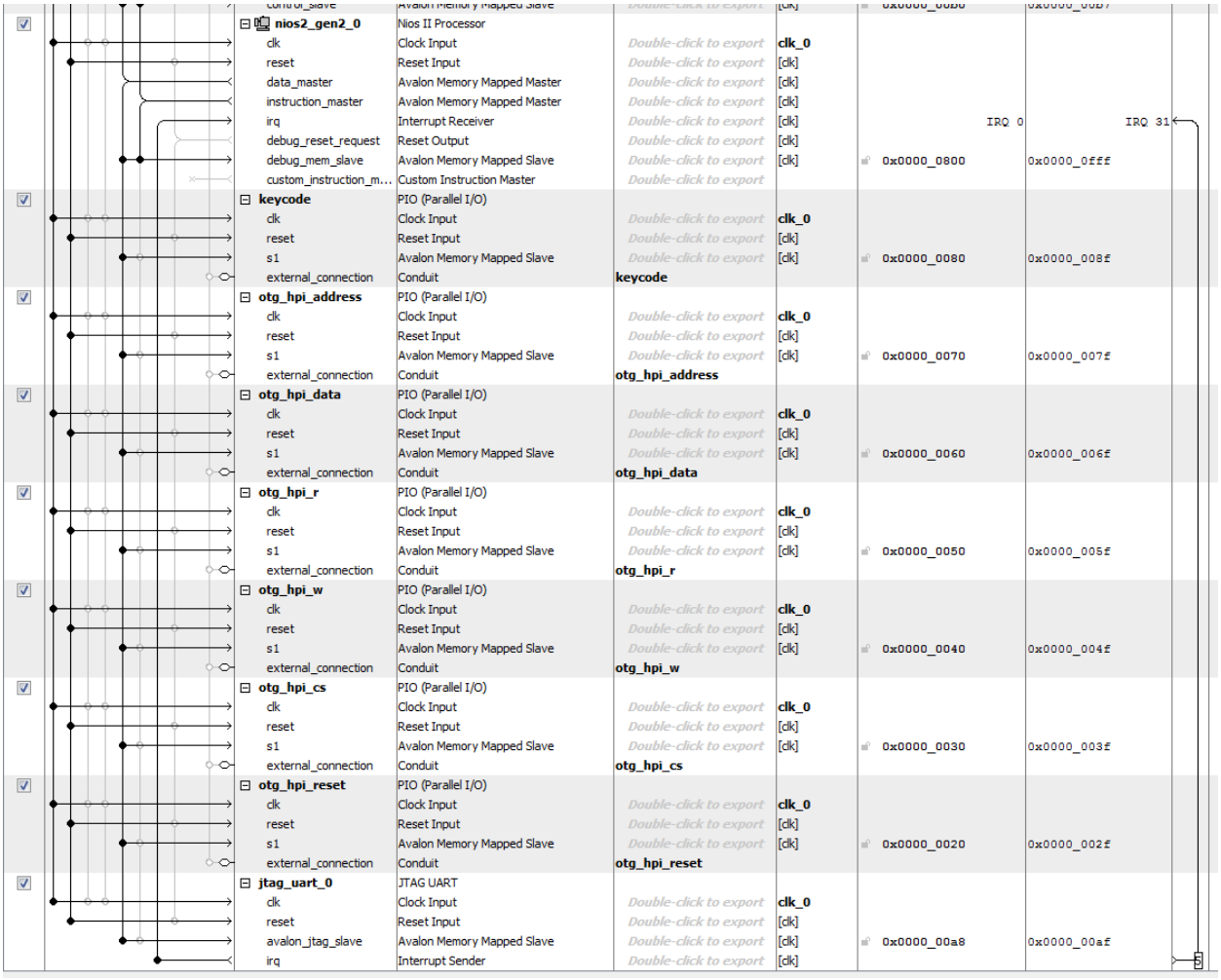


Figure 0

You should save the Qsys setup and generate HDL and make sure you don't have any errors.

For this next part you need to have a complete hardware top level entity. You need to make sure that all the pin connections mentioned in the lab 8 experiment section exist and are connected to the correct entities. **Remember that your HDL will need to pass the PIO ports to the physical pins on the FPGA which are physically connected to the USB chip. The easiest way to do this is to map them in HDL to the names found in the standard pin assignment file.** To complete this you also need to add in the HDL version of the Qsys system to the project by adding the .qip file like you did in lab 7. If all the connections to the qip are made correctly, you should be able to compile the project and program it onto the board.

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.

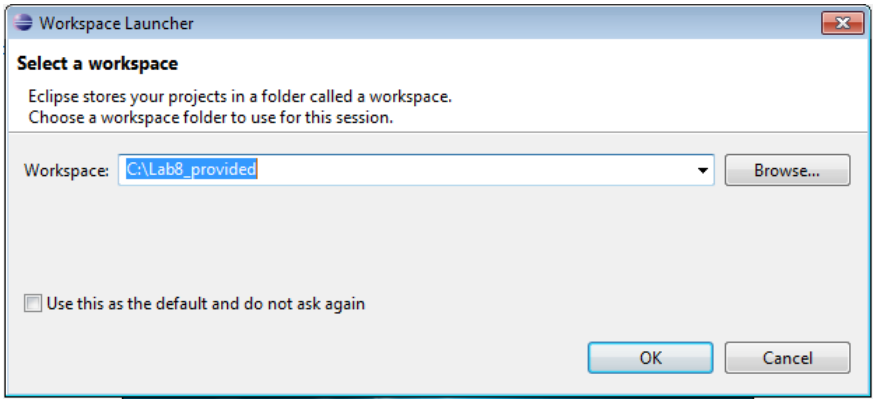


Figure 1

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

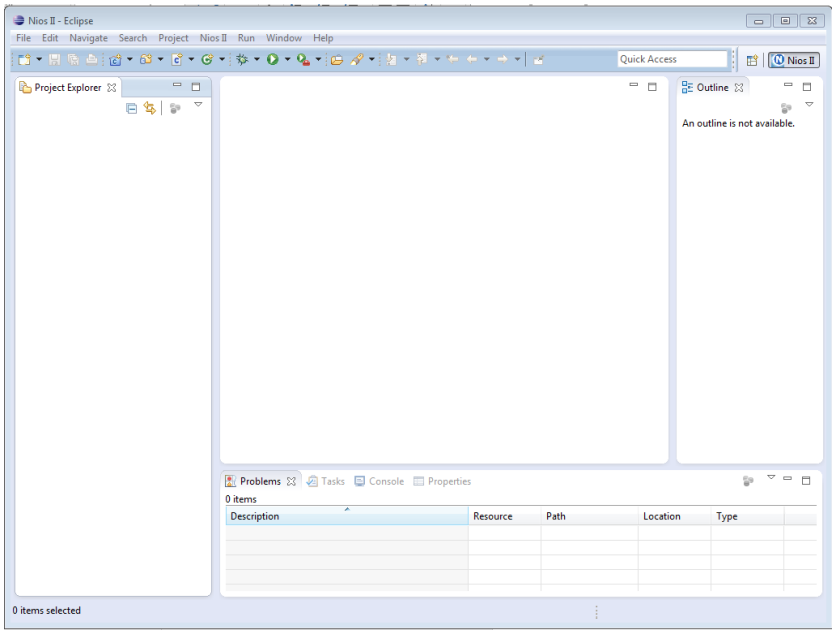


Figure 2

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 create a new Eclipse project. Go to ***Menu > New > NIOS II Application and BSP from Template***. In the pop-up window, select the .sopcinfo file generated by Qsys in *SOPC Information File name*, and the system should automatically detect the NIOS CPU that you use. Then, type "usb_kb" in Project name, select Blank Project in Templates, and click on **Finish**. This should create two projects *usb_kb*, and *usb_kb_bsp* in the Project Explorer.

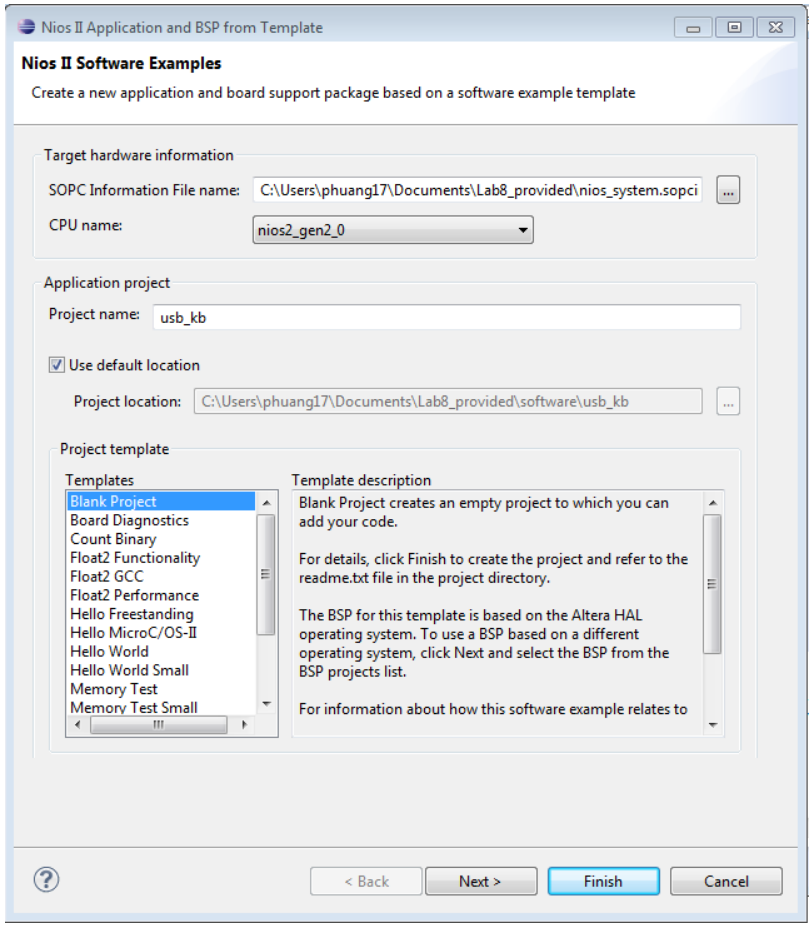


Figure 3

Copy all the files in *software_codes/* to *software/usb_kb/*. In Project Explorer, right click on usb_kb project and click on **Refresh**. Eclipse should detect all the provided .h and .c files located in *software/usb_kb/* automatically.

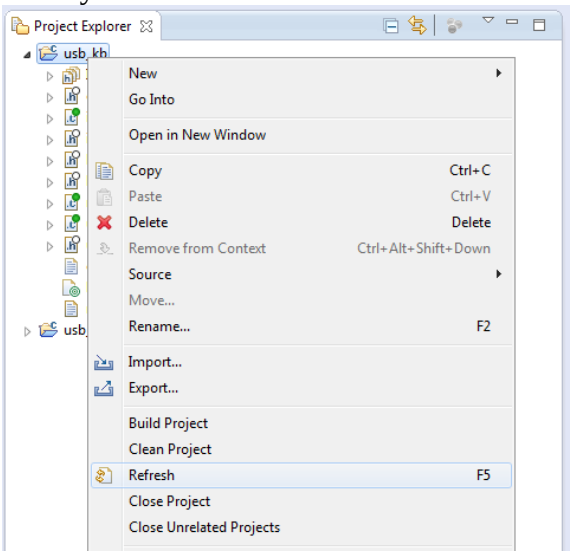


Figure 4

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.

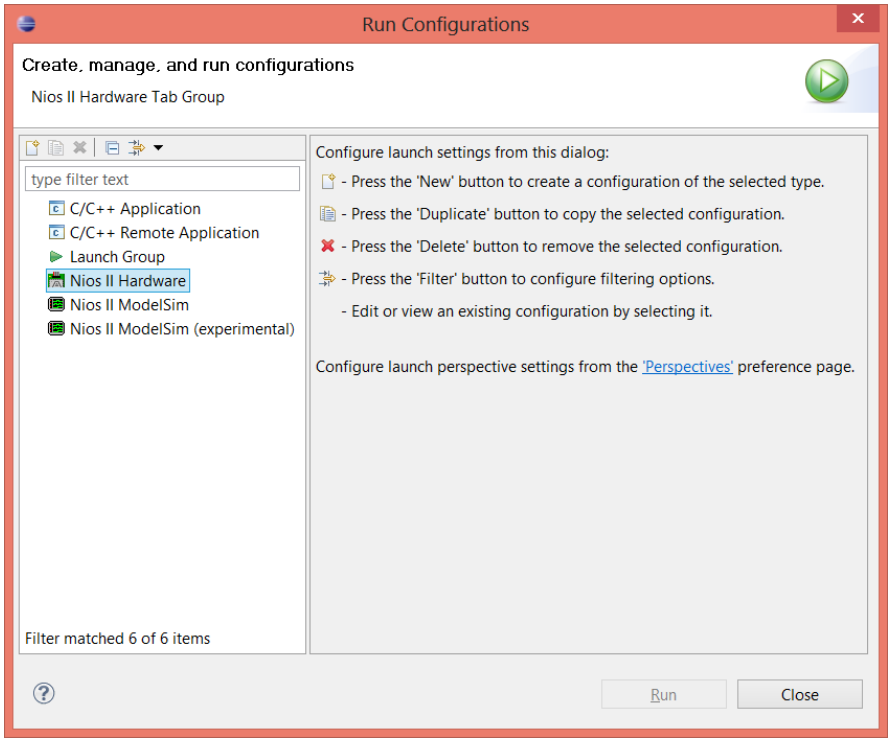


Figure 5

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.

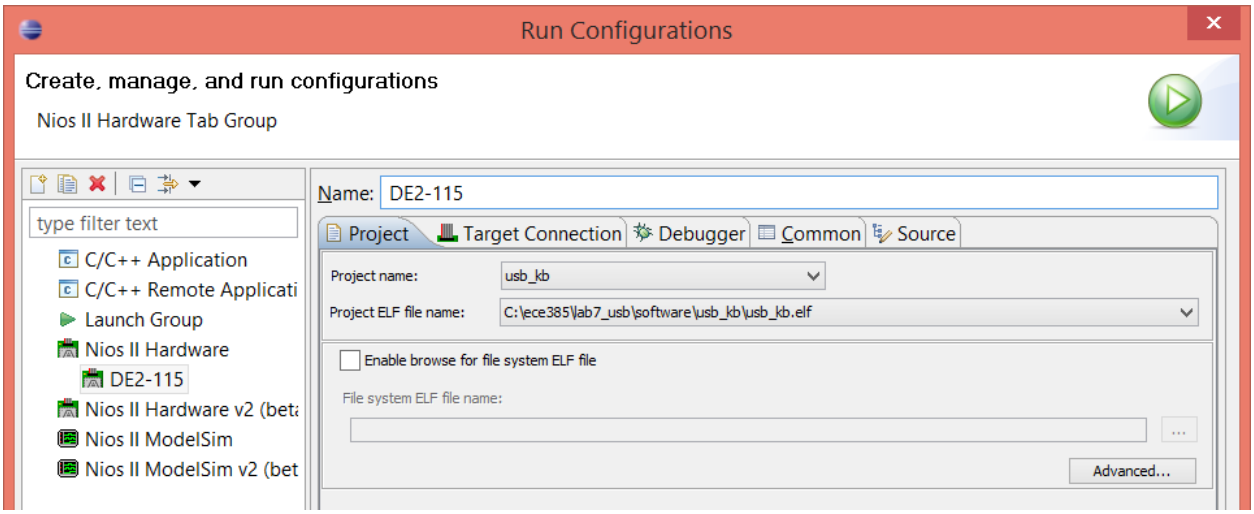


Figure 6

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.

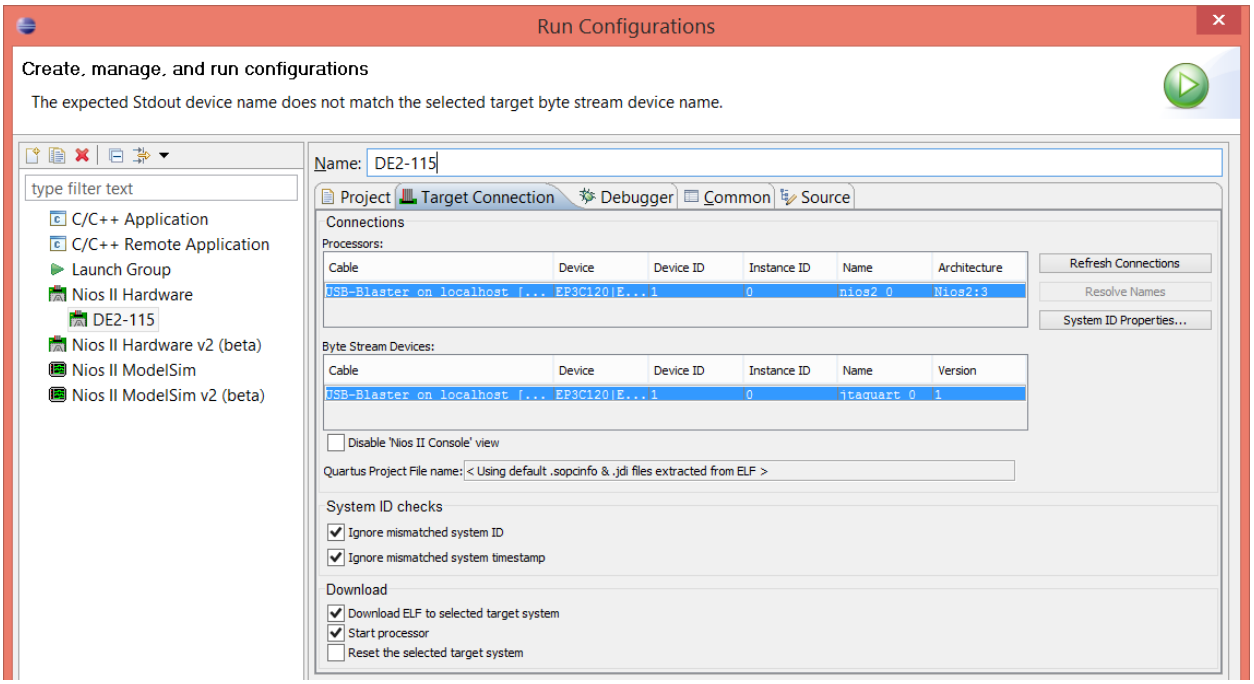


Figure 7

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.

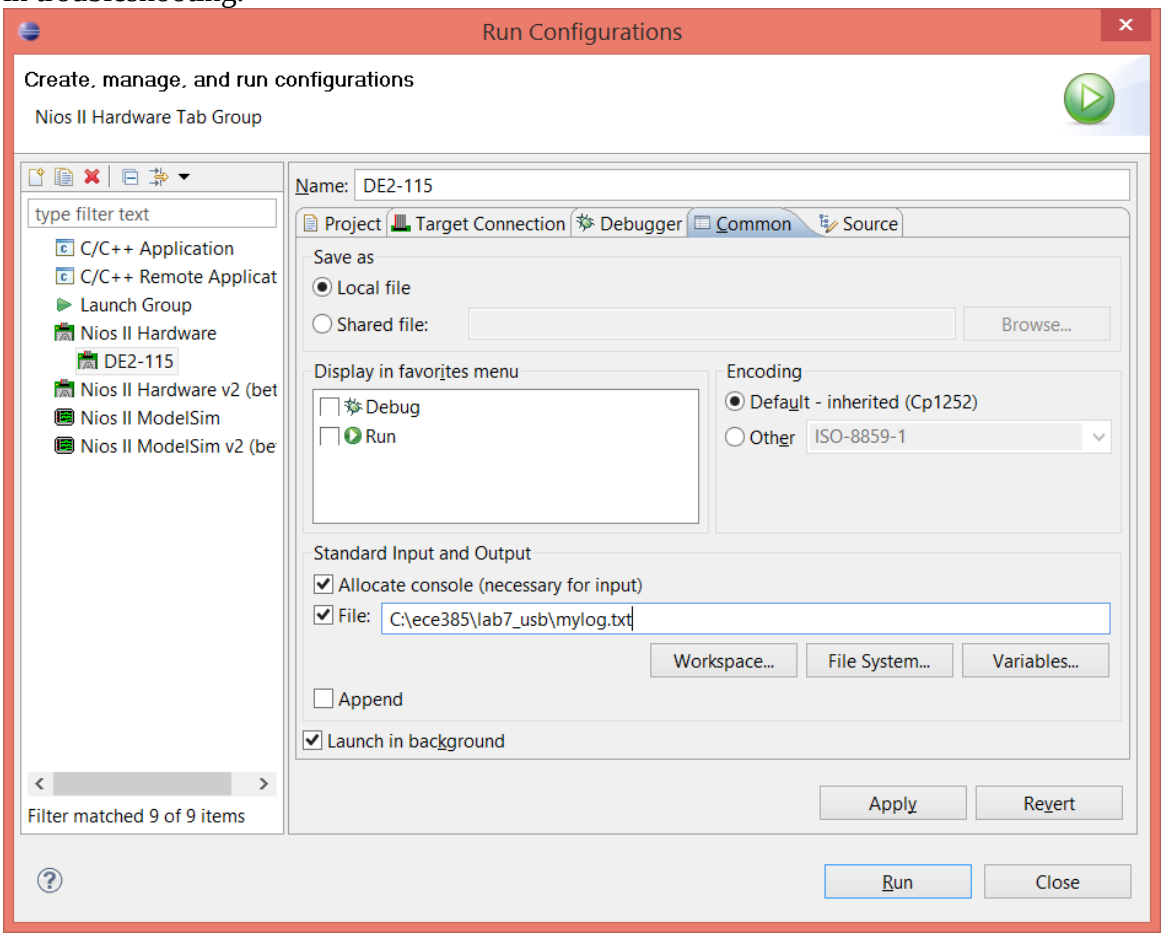


Figure 8

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.

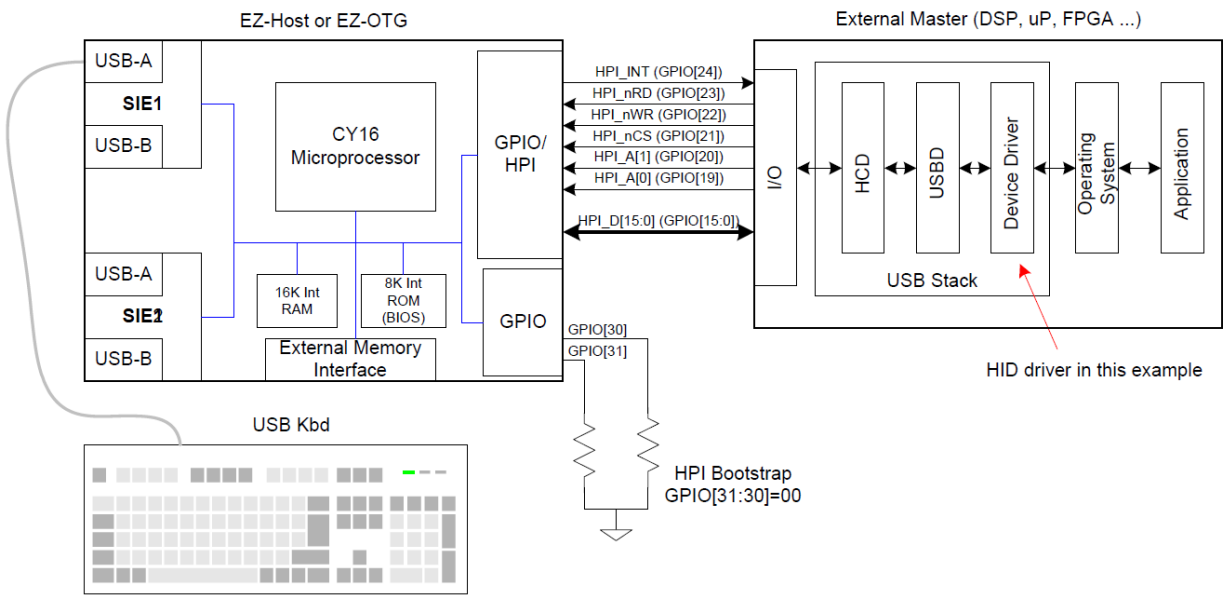


Figure 9

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):`

IUQ. 10

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.

However in this lab we will **not** using these functions (IORD and IOWR), since the EZ-OTG's memory space is **not** memory mapped to the NIOSs II address space. Instead you will write the following helper functions which the provided C code will call to talk to the EZ-OTG.

`IO_read(address)`:
Read and return data from the memory location specified by the address. The address should be 2 bits wide as described in the CY7C67200 datasheet.

`IO_write(address, data)`:
Write data to the memory location specified by the address. The address should be 2 bits wide while the data should be 16 bits wide as described in the CY7C67200 datasheet.

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, you will need to complete the definition of two helper functions in USB.c as follows:

```
alt_u16 UsbRead(alt_u16 Address)

void UsbWrite(alt_u16 Address, alt_u16 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. Therefore here the address is 16 bits wide and so is the data.

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.

The Keyboard Enumeration has been completed for you in this lab and you don't need to worry about it. One thing to keep in mind is that the data bus to the EZ-OTG chip is only 16 bits wide. That means with this setup you can only get information about two keys on the keyboard simultaneously. To add the ability to get more information about more keys you need to add another UsbRead call in main.c.

Below is a table with a description of how keycode correspond to the key pressed.

Code that performs this is given as part of the C code for this lab you, this description is for your reference and to assist any debugging.

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