ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to USB on the Nios II

Embedded System with Nios II

In Lab 6.1, 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.

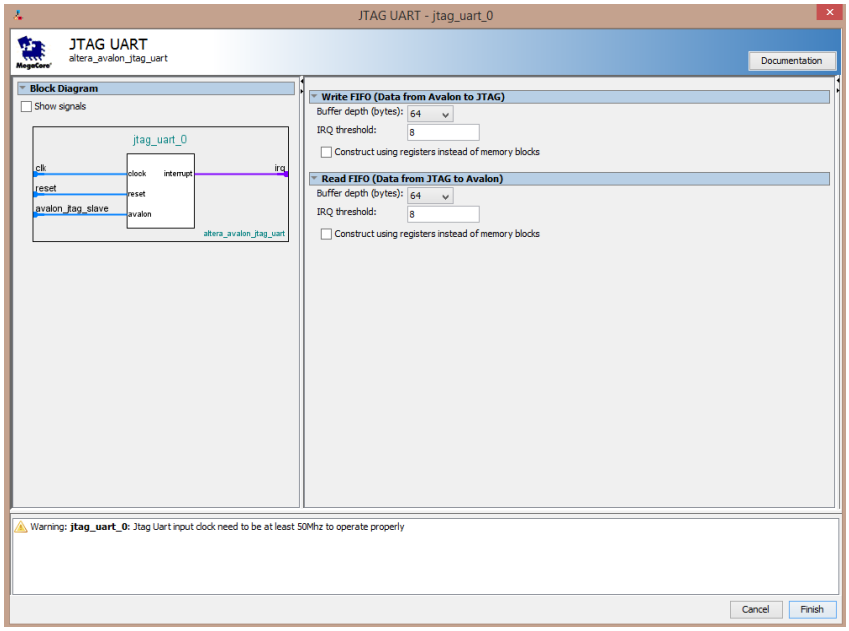
In Lab 6.2, the USB protocol is handled in the software on the Nios II, and the extracted keycode from the USB keyboard is then sent to the hardware for further use. In this tutorial, you will build a Platform Designer interface to handle to I/O interface to communicate with the USB-OTG chip. Then you 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 6.1 Platform Designer setup. If you do not have one, you should follow the tutorial for lab 6.1 to complete the Platform Designer 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 6.1, remove the PIO modules in Platform Designer which correspond to the LEDs and switches, as we will not be using them for lab 6.2. 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

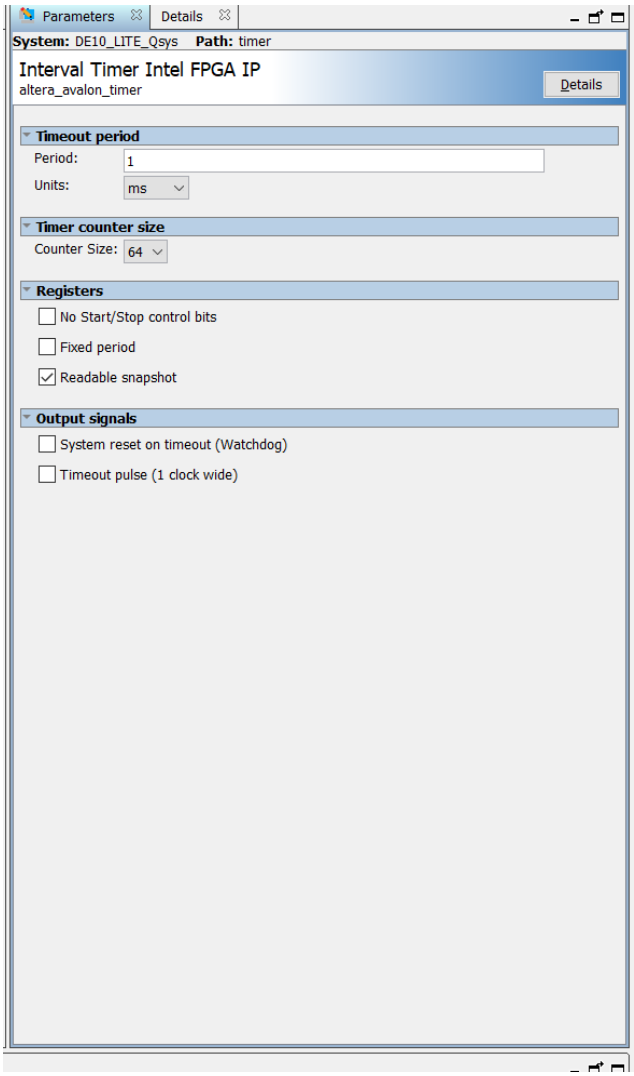


Make the standard connections for clk/reset/data bus, which is what we have been doing for the other peripherals. One difference here is that we must assign an **interrupt** for this, which we will assign IRQ (interrupt request) 1. Connect the interrupt controller to IRQ and give it the number 1 (this 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 do not 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 6.2 setup, these are listed below with the direction of the ports and size of the ports. You should connect these to clk, reset, and the Nios II Avalon data bus as before. Note that you may delete the previous PIOs used for switch input from Lab 6.1. Note that only the three PIOs in bold (along with the timer and SPI port) are necessary for the connection to the USB chipset (MAX 3421E). Therefore, for your Final Project you may remove the other PIOs if you do not need them. Make sure you export them with the export name as shown so that they match up with the provided lab62.sv top-level.

Name	Direction	Width	Export Name
keycode	Out	8	keycode
usb_irq	**In**	**1**	**usb_irq**
usb_gpx	**In**	**1**	**usb_gpx**
usb_rst	**Out**	**1**	**usb_rst**
hex_digits_pio	Out	16	hex_digits
leds_pio	Out	14	leds
key	In	2	key_external_connection

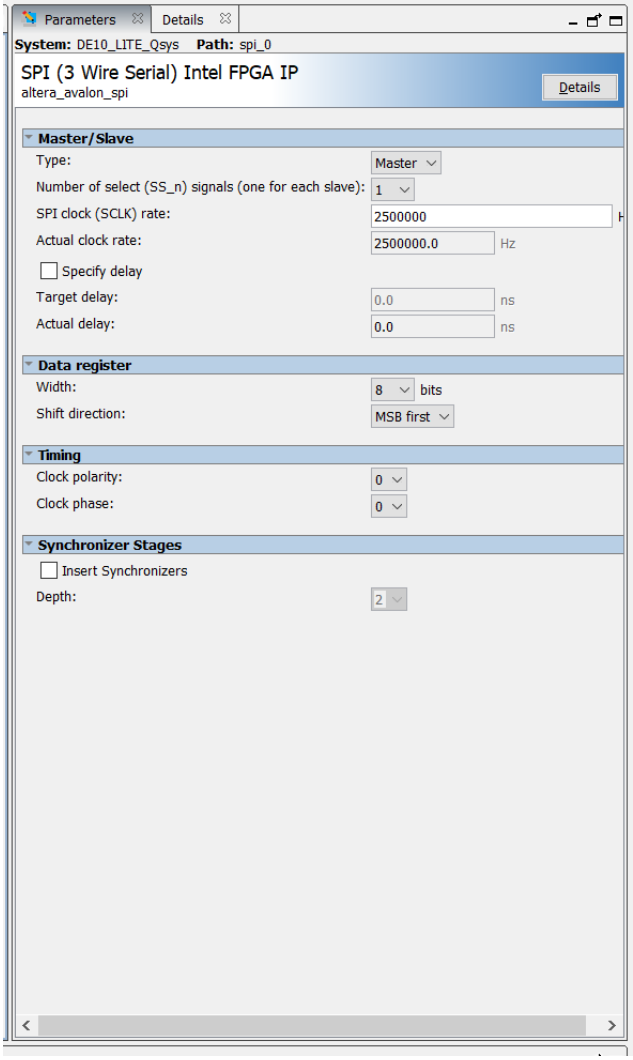
In addition, you should add the Interval Timer Intel FPGA IP. This is found under Processors and Peripherals -> Peripherals -> Interval Timer Intel FPGA IP. Configure it as follows, to count up on millisecond intervals:



And connect it to clock, reset, data bus as with other peripherals. Also assign in IRQ (interrupt) 2. This is needed in the USB driver code in order to keep track of the various time-outs that USB requires.

Finally, find and add the SPI port peripheral. Connecting the provided USB host code to the SPI driver will be the majority of the work you will do for Lab 6.2. It is found under Interface Protocols->Serial->SPI (3 Wire Serial) Intel FPGA IP. Configure it as follows, connect it to the data bus, and assign it IRQ 3. Export the SPI port as **spi0**.

IUQ. 4

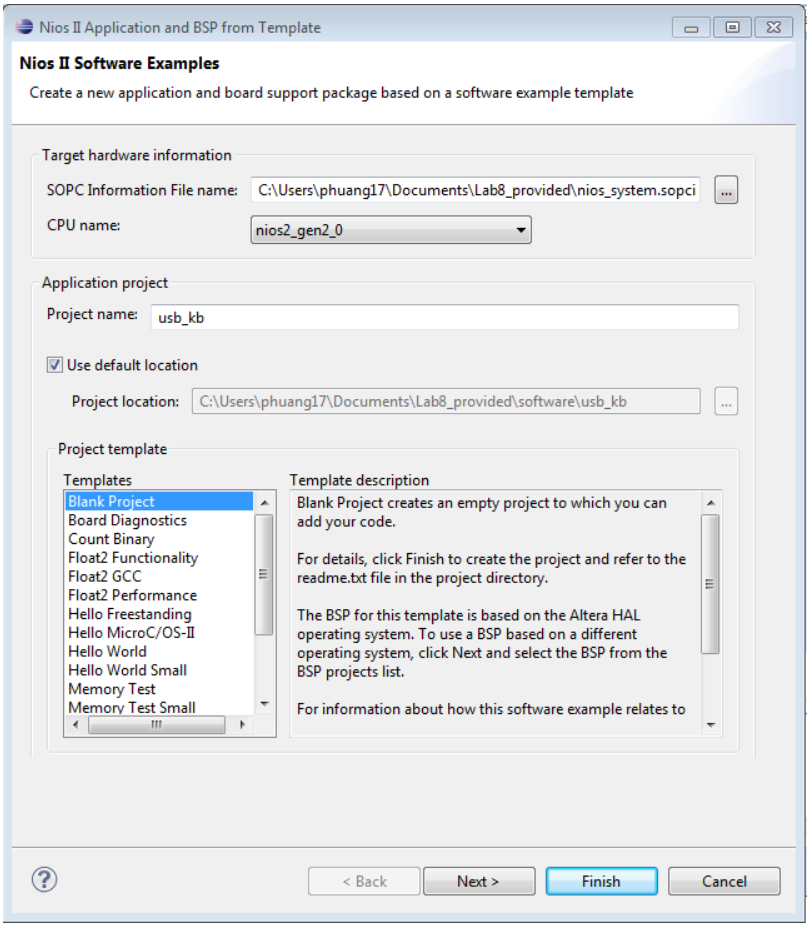


You should save the Platform Designer setup (now is a good time to rename it **lab62_**soc by using Save-As) and generate HDL and make sure you do not have any errors.

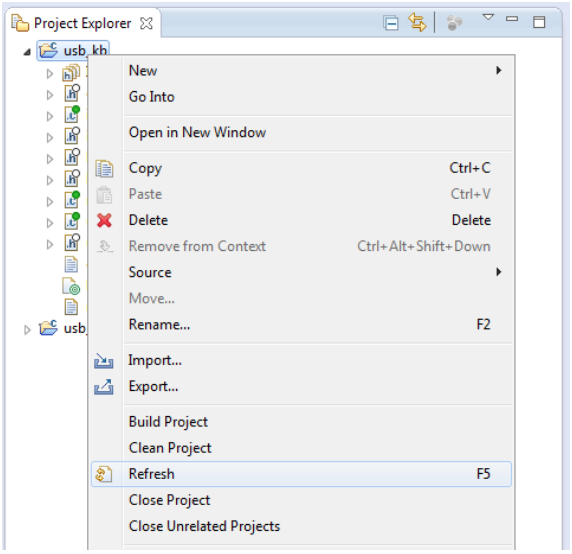
Go to ***Tools > Nios II Software Build Tools for Eclipse*** to launch the software development environment.

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 Platform Designer 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.



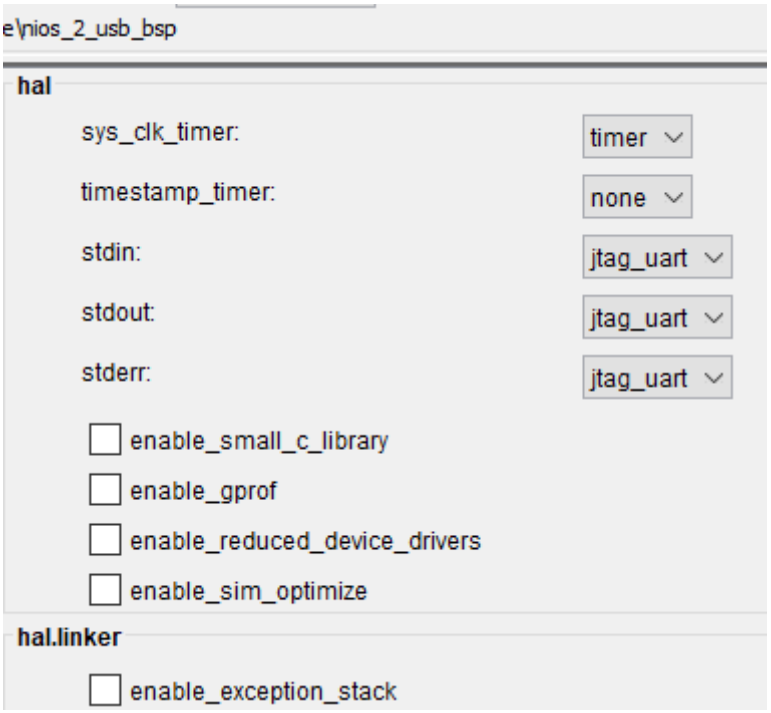
Copy all the files in the provided usb_kb_software.zip to software/usb_kb/, preserving the directory structure (so you will have a software/usb_kb/usb_kb/ directory, which is intended). In Project Explorer, right click on usb_kb project and click on **Refresh**. Eclipse should detect all the provided .h and .c files located in software/usb_kb/ automatically.



IUQ. 6

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 > BSP Editor.***

Verify the HAL Settings in the BSP are as follows and hit Generate:



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 clean and compile the program again (***Build All***). Compatibility errors occur if you fail to do so!

You should now be able to perform a **Build Clean, Generate BSP, and Build All.** For the USB code to work, you must fill in the relevant MAX3421E.c functions:

```
void MAXreg_wr(BYTE reg, BYTE val)

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)

BYTE MAXreg_rd(BYTE reg)

BYTE* MAXbytes_rd
```

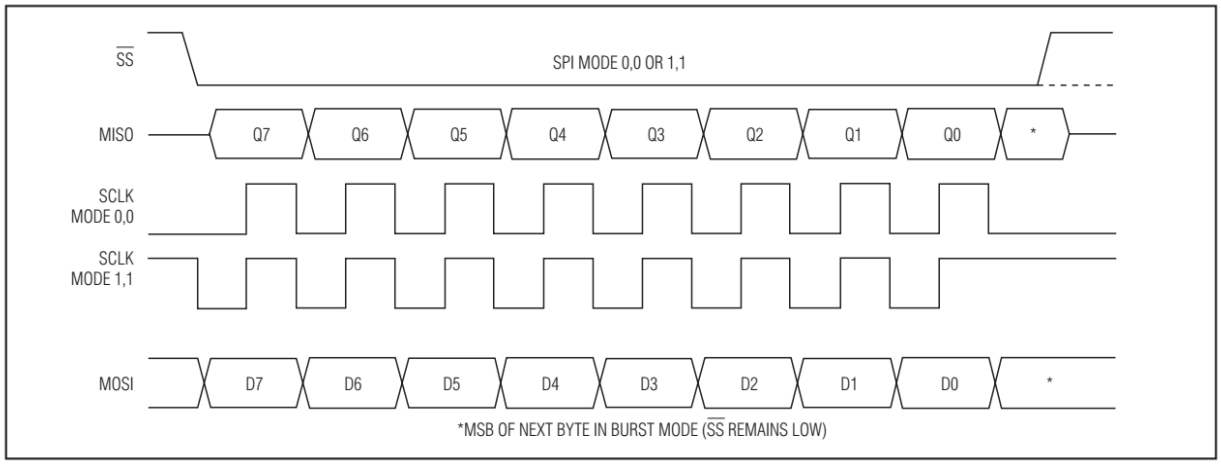
USB Host Programming with the MAX3421E Chip

The provided USB host driver is mostly complete, with support for USB keyboards and USB mice. However, you will be required to fill in some functions inside the MAX3421E.c file. These functions pertain to reading and writing registers through SPI to the MAX3421E chip.

SPI is a synchronous serial bus, consisting of 4 signals called CLK, MOSI, MISO, and SS. CLK is the clock signal, which is transmitted from the master device (Nios II) to the slave devices (MAX3421E). MOSI is a data signal which stands for master-out slave-in, which transmits data from the Nios II -> MAX3421E – synchronous to the CLK. MISO is a data signal which stands for master-in slave-out, which transmits data from the MAX3421E -> Nios II – also synchronous to the CLK. SS stands for slave select, which allows a specific slave device to be selected when SS is low. Multiple slave devices may share the same SPI bus by sharing MOSI, MISO and CLK lines, but SS must be a unique connection between the master device and each slave device.

The timing diagram for the MAX3421E generally looks as follows. Note that there are 4 SPI modes corresponding to the various polarities between clock and data, but the MAX3421E supports both 0,0 and 1,1. Our SPI peripheral (above) should be pre-configured to operate in mode 0,0 with a clock rate of 2.5 MHz. Note that the following is a *full duplex* transfer, as 8 bits are read and written (through MISO and MOSI) respectively during the same 8 clock cycles. The MAX3421E starts up in half duplex mode, but the first SPI operation (from MAX3421E.C) configures the FDUPSPI bit, switching it to full duplex mode. You should understand this behavior in the MAX3421E by reading the datasheet starting from page 19. Note that to (for example) read a register on the MAX3421E, first you need to write the register address through SPI and then perform a read through SPI.

IUQ. 8



Once you understand the various operations the USB device drivers need to communicate via SPI (the comments in the provided function stubs will tell you what each function should do), you must implement them using the IntelFPGA SPI peripheral. To do this, you should read the Embedded Peripherals User Guide, in the section which describes the SPI Core (Chapter 5). Although you can accomplish this task by programming the individual registers for the SPI peripheral (as you did in Lab 6.1), the easier way to do this is to go through the driver, e.g.: altera_avalon_spi.h and use alt_avalon_spi_command(…) function. Make sure you understand the example provided at the very end of Chapter 5.

Finally, take a look at the provided main.c for Lab 6.2. Although for this lab, it is only necessary to read 1 keycode (the list of keycodes is provided below), the USB HID protocol actually supports up to 6 simultanous keycodes. The provided main function will only output the first one to the PIO named ‘keycode’, but for your final project you may wish to use all 6. Note: you may have to modify the keycode memory address in the provided main function to reflect the address that Platform Designer assigned your ‘keycode’ PIO. Finally, the provided USB driver will also accept a standard USB mouse which you may make use of for your final project.

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				

List of Keycodes