|  |
| --- |
| Addison Elliott  Group member: Jacob Sauls  Spring 2016 ECE 381 Microcontrollers  Final Project: PSoC 5LP PS/2 Keyboard  5/02/2016 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Laboratory Exercise 1: PS/2 Keyboard***

The objective of our final project was to interface the PS2 keyboard with the PSoC5. Initially we wanted to create state machines in Verilog to run the PS2. We wanted to make a user module that a programmer could drag and drop onto the schematic layout feature in the PSoC5. We were able to use Dr. Noble’s code for the PSoC5 PS2 bit bang lab as the basis of this project.

**Materials Needed**

* (1) PSoC 5LP
* (1) PS/2 Keyboard with MiniDIN connector
* (1) External 5.0V supply (for keyboard)
* (1) PS/2 MiniDIN 6-pin socket breakout board
* Serial Console(eg Putty, RealTerm)

**Setup**

The PSoC5 must have the data and clock lines routed from the PS2 breakout board. The miNiDIN connection routes data and clock from the PS2 keyboard to the breakout board. Data was connected to P2[3]. The PS2 clock was routed to the PSoC5 P1[4]. A separate power supply is needed for the PS2 board because it draws more current than the PSoC5 pins can source. The KEYSIGHT E3631A supplies the PS2 with 5 V at the Vcc pin on the breakout board. A common ground connection is needed for the PSoC5 and the external power supply. The external circuit is shown on the next page.

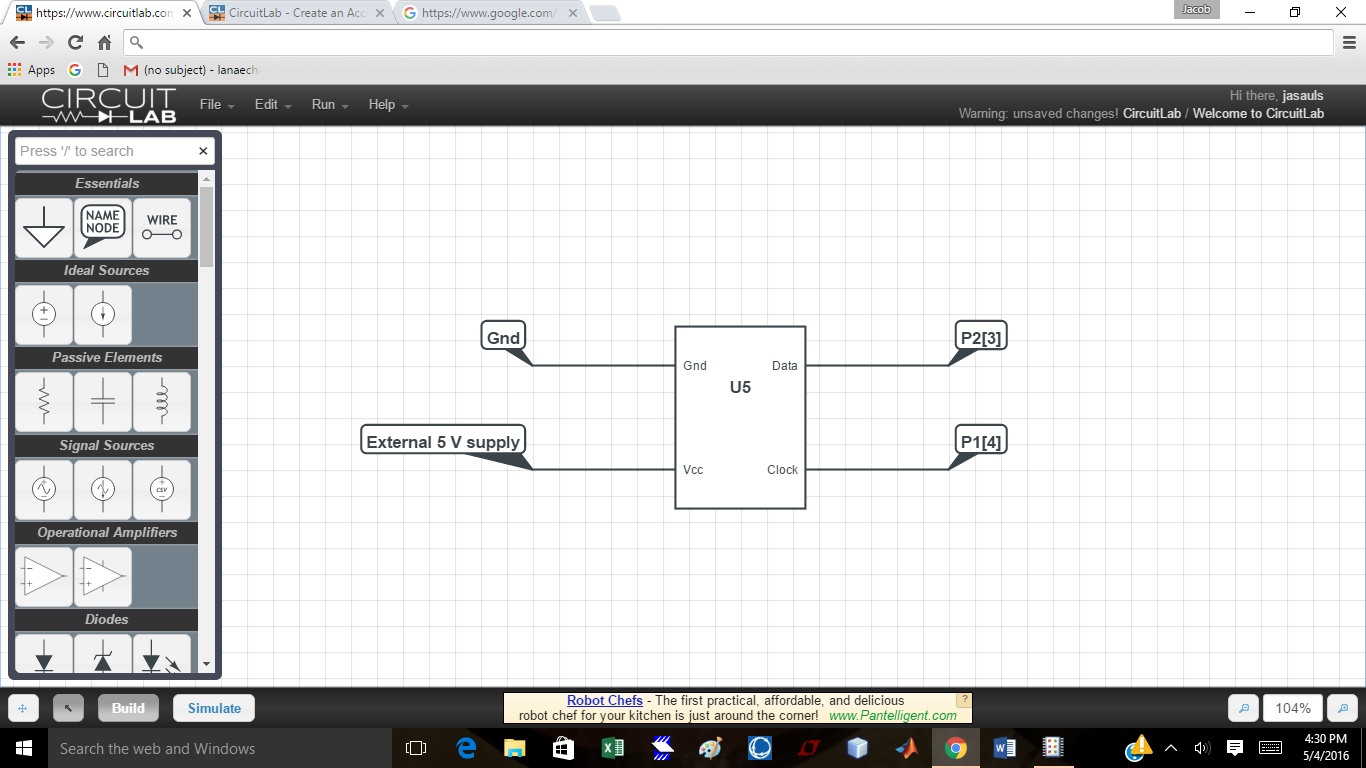


Figure 1: The external circuit for the breakout board to the PSoC5

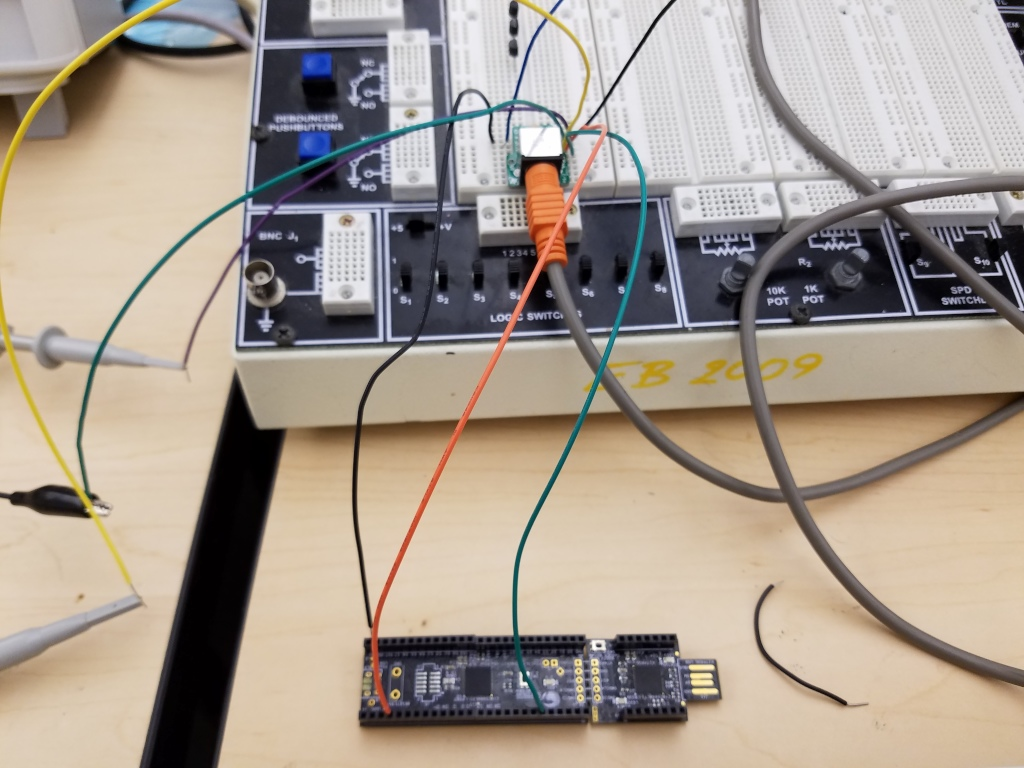


Figure 2: The physical circuit for the PS2 lab. Common ground (black), Data (orange), and Clock (green)

The code for this project can be found in the attached zip file named “PS2Keyboard.zip.” The components that make up this project are:

* PS2 – Component that links the PS2\_Core with a timer and registers to be able to read/write information from a PS/2 device
  + PS2.cysch – Schematic that contains the PS2\_Core, timer, and various registers to create a PS/2 user module
  + PS2.c – Source file for the PS2 module
  + PS2.h – Header file for the PS2 module
* PS2\_Core – Component that contains a state machine to read/write a byte from a PS/2 device
  + PS2\_Core.v – Verilog file that contains the code to read/write a byte from a PS/2 device
* TopDesign – Actual project that utilizes the PS2 component to communicate with a PS/2 device
  + TopDesign.cysch – Schematic that contains the PS2 module and connects the necessary clocks and pins to the user module.
  + Keyboard.h – Header file for PS/2 keyboard
  + Keyboard.c – Source file for PS/2 keyboard
  + Main.c – Main source code that makes up the project

This project is built up in a hierarchical way such that each component depends on another. I will start at the bottom of this hierarchy and work my way to the top with discussing it.

The most basic element is the PS2\_Core module. The purpose of this module is to utilize a state machine created in Verilog to read and write a byte from a PS/2 device. The module has a ***rw*** pin which sets whether the module is to read(rw=0) or write(rw=1) information out. When the negative edge of the ***ps2\_clock*** is detected, the state machine will either read from ***ps2\_data*** and store the bit in ***data\_out*** or write to ***ps2\_data\_out*** with the information in ***data***. In addition, before data is read or written, the state machine will output or read the start bit. After the 8 bits of data are clocked, the state machine will check the parity bit. Finally, the stop bit is clocked along with the ACK bit if the machine is writing. If any error occurs during the process such as incorrect parity, start/stop bit, the ***error***flag will be set HIGH and the state machine will go into an ERROR state. If everything goes smoothly, the ***done*** flag will be set HIGH instead and the state machine will go into a DONE state. The only way to escape these states and continue the process is to toggle the ***reset*** bit. The current ***state*** of the state machine is also outputted to check the progress of the read/write process.

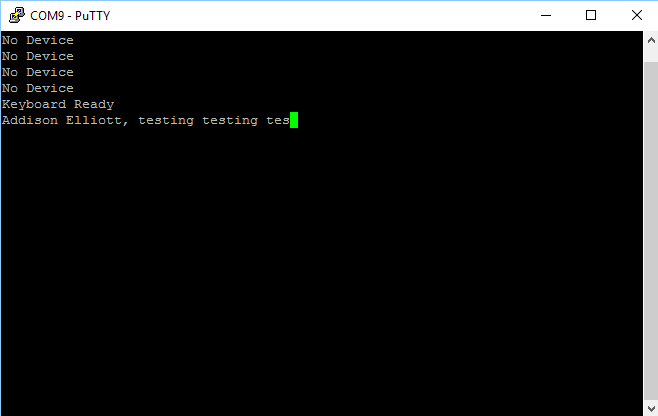
The PS2 module incorporates the PS2\_Core module into a useable user module. This module contains a timer for the PS/2 timing constraints as well as registers to allow the API to read and write information. A ***clock\_10us*** pin is fed into the 8-bit Timer module to allow precise timing. There is a timer interrupt that is triggered when the timer is done. There are buffoe modules that take the bidirectional ***ps2\_clock*** and ***ps2\_data***pins and split them into individual input and output. In addition, the input portion of the ***ps2\_data*** pin is MUXed so that it can be written in the API and by the PS2\_Core module(***ps2\_data\_out)***. The ***dataOut***, ***dataIn***, and ***stateReg*** are direct connections to the ***data\_out, data,*** and ***state*** pins on the PS2\_Core module. There is a ***statusReg*** which uses bit 1 and 0 for the ***error*** and ***done*** pins on the core module. The control register(***controlReg***) controls various things. Bit 0 connects to ***rw*** to control whether a byte should be read or written. Bit 1 is used to reset the core module. Bit 2 is the input of the ***ps2\_data*** line when in software mode. This is set by the API when it needs to set the output on the data line. Bit 4 is the input for the ***ps2\_clock*** line always. The last bit, Bit 5, is the select for the ***ps2\_data*** MUX. The purpose of this is to set whether the ***ps2\_data*** input should be software or hardware. Software refers to bit 2, and hardware refers to the output from the state machine.

Now that we understand the hardware components of the PS2 module, the API can be discussed. In ps2.h, various macros are made for reading/writing to the registers. In addition, there is a wait/timeout macro that runs the timer module for x seconds and waits until a condition is true OR the timer times out. There is also a macro that will just wait for the timer to trigger. The Start function simply initializes the necessary components to the PS2. This includes setting the reset, initializing the timer, etc. The next function is DetectDevice. It attempts to read a byte from the PS/2 device within 1000ms which is what the BAT test consists of. If this function previously failed, it will send a RESET command to the device first. The return value is the result of the read byte such as timeout, communication error, device error, or bad BAT. WriteByte simply writes the specified byte to the PS/2 device. However, SendCommand is slightly different. It uses WriteByte to write a command, but it also continuously sends the byte until an ACK or error is received. ReadByte will block until a byte is read from the PS/2 device while cReadByte will return a byte if one is available and 0 if no data is available.

The TopDesign module utilizes the PS2 component to provide functionality to the PS/2 device. The PS2 module just provides the capability to communicate with it. The TopDesign specifies how to communicate with the device. In our case, we are using a PS/2 keyboard. The keyboard.h and keyboard.c files allow us to interpret the data sent from the keyboard and obtain the key pressed with it. They implement another state machine for Scancode Set 2 on the PS/2 keyboard. KeyboardAction actually takes the key and performs an action based on it. This includes things such as changing the three status LEDs on the keyboard, recording whether control, alt, or shift are being held down currently. From there, the KeyboardToASCII function can convert the keypress into ASCII. The main source code implements all of these functions. It first attempts to detect a device in a continuous while loop with 1 second delay. Once detected, the PSoC continuously reads from the PS/2 keyboard and processes any data. If a valid character was entered, it is outputted to the console via UART.

**Test Procedure**

Start UART when you turn on the program. You should see a “No Device” or “Keyboard Ready” string depending on whether the +5V power supply is turned on. If its not on yet, turn it on and you should see the “Keyboard Ready” text. Now, press keys on the PS/2 keyboard and you should see them display in the console. Your screen should look similar to Figure x below.

****

**Participation:**

I setup the circuit for this lab. It was fairly easy, especially since we could just go off of the previous PS/2 lab. Jacob and I programmed together. We spent many hours diagnosing the state machine and working our way up to a workable PS/2 lab in PSoC 5.

**Conclusion:**

The PSoC5 offers more functionality than the PSoC1 which have used in all of the 381 labs. PSoC1 allows the programmer to drag predefined user modules onto the empty chip level blocks. We intended to recreate the means of interface for the PS2 keyboard with a microcontroller. The microcontroller was the PSoC5, and this interface was made successfully using the approach previously described. An all-out Verilog implementation of the Dr. Nobel’s code was not realized, but interface was successful. A core module was created entirely in Verilog. That module may be added to another schematic, where external clocks, control and status registers, pass through, and multiplexers were added to finish the interface.