UNIVERSITY OF NEW BRUNSWICK - FREDERICTON
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE3221 Computer Organization

Lab #5.

2025-01-06

The PS2 Keyboard - Introduction to C Programming

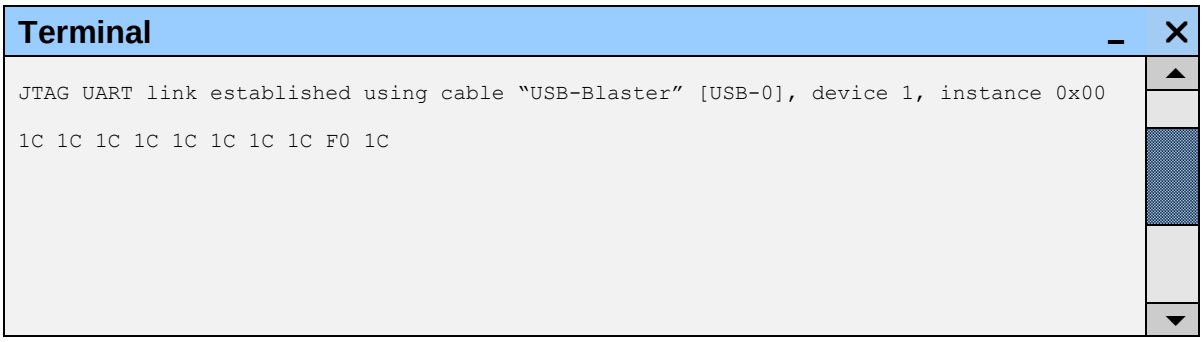
Introduction

In this lab you will use the NIOS-II processor on the DE2 board to explore the use of the C language as an alternative to assembly language for developing low level code. You will read keycodes from a computer keyboard over the standard PS2 serial communications interface.

The procedure to use C-language in the lab is identical to assembly language except that your source code text file now has the *.c* extension and in the monitor program you must choose *C-Language* in the pull down menu when including your program.

1. You will first assess the speed of the NIOS-II processor and the performance of the C compiler by timing a small loop. The red hexadecimal display will serve to show the contents of a register variable as it is rapidly incremented in an infinite loop.

2. You will then extend the above program to read scancodes one bit at a time from the attached keyboard. Whenever a key is pressed on the keyboard, you will display the corresponding scancode on the hexadecimal display and on the monitor screen. You will then recognize some specific keys to perform an action.

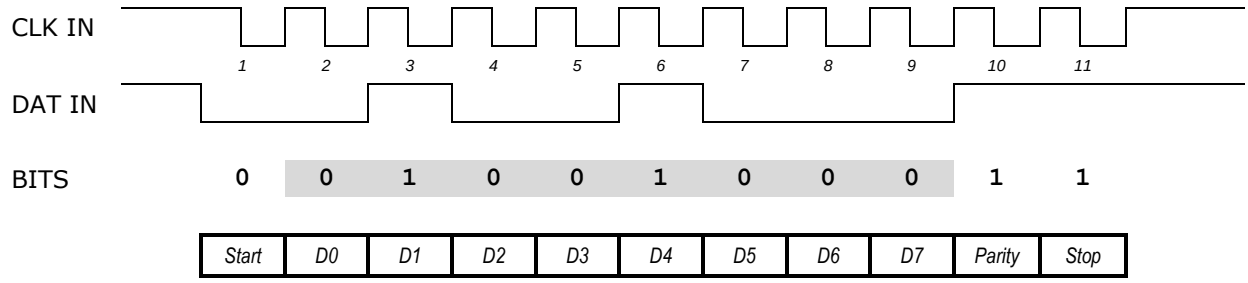


*When the terminal display is ready to receive characters, the "*JTAG UART link established*…" message appears, as shown above. Any characters sent to this display before that time will be ignored.*

The PS2 Keyboard

The PS2 keyboard has its keys wired in a matrix and scanned by an embedded microprocessor that detects and debounces a keystroke before sending scancodes to a host computer. These scancodes are not ASCII codes; as such, the same keyboard can readily be used for different alphabets. Even the SHIFT key is simply a scancode; pressing the **A** key, for example, generates the same scancode whether or not SHIFT has been pressed. The final interpretation of each keystroke must be handled in software on the host computer.

The PS2 connector (I/O port 0x88F0) uses three wires (clock, data, ground) to communicate *serially* in both directions between the keyboard and the host computer. Whenever a key is pressed on the keyboard, a scancode is transmitted from the keyboard to the PS2 connector. Each scancode is one or more 8-bit values sent one after the other. Each 8-bit value is transmitted in a commonly used transmission format using 11-bits sent one after the other as shown in the timing diagram below. The exact clock rate is unspecified, but may be expected to exceed 10 kHz.



This figure shows the byte 0x12 (shaded) as each bit is sent, one for each clock falling edge starting with the LSB (*i.e.* the byte appears to be reversed). The clock line is held high until a byte is to be transmitted. The first bit preceding the data byte is the *start bit* (always 0) and the eleventh bit is the *stop bit* (always 1). Bit 10 is a *parity bit* and is set so that the total number of ones in the data and parity bits works out to be an odd number (in this case, there are three ones and 3 is an odd number) which is referred to as *odd parity*. In this lab, the various scancodes are found in the eight data bits. This overall format is known as *start/stop serial transmission*.

To receive each 8-bit scancode from the keyboard, it will be necessary to read one bit from the data line every time a falling edge is seen on the clock line. Initially, wait for the first falling edge on the clock line that has a 0 input bit (start bit). Then at each falling edge to follow, shift the corresponding input bit into a register. After the final bit has been received, the stop bit is expected, where not finding the stop bit is called a *framing error*. The register may now be examined to see the eight data bits and (optionally) to check the parity bit for a *parity error* (usually caused by reading a bad data bit). If either the framing check or the parity check fails then a reception error has occurred and the data is wrong.

Most of the keys have a single byte scancode. For example, pressing the **A** key results in the single byte scancode 0x1C. When the **A** key is released, the keyboard sends a pair of bytes 0xF0 0x1C, where the first byte signals the release and the second byte indicates which key was released. Some keys have multiple byte scancodes and a corresponding sequence upon release.

Pre-Lab Preparation

1. Review the starting code below and write down how you would do this in assembly language. In the lab, you will see how the C-compiler would write the same assembly language for you.

The following code should be copied into a text file named **lab5.c**

```
/***
CMPE3221 LAB#5 - C LANGUAGE EXERCISE
----------------------------------------------
November 2010     NAME:
----------------------------------------------
This program is the starting point for LAB#5
It enables the hex display and creates a counter.
----------------------------------------------
***/

main ( ) {

volatile int   *const HEXDISP = (int   *)0x000088A0; // display hex digits
volatile short *const HEXCTRL = (short *)0x000088B0; // hex control register

  int  count;                 // define a counter

  *HEXCTRL = 0x01FF;          // enable eight hex digits

  count = 0;                  // initial count value

  while( 1 ) {                // create an infinite loop

      *HEXDISP = count;       // send count to the hex display

      count = count + 1;      // increment the count

  } // end while

} // end main
```

2. Prepare three C-language subroutines as described below by the following three prototypes.

void outchar(char ch) – send the byte **ch** to the Altera Monitor terminal display window.

char bin2hex(char N) – return the ASCII hex character for the 4 least significant bits of N.

void outhex(char N) – use **bin2hex** and **outchar** to display 2 ASCII hex chars = the byte N

What is the meaning of the **void** keyword in front of a subroutine?

Procedures

1. **lab5a.c** - Testing the supplied program & Processor speed calculation.

   - Load and Run the above test program and confirm the operation of the counter and display. *Be sure to select Program Type: **C** when loading your code.*
   - Observe the count on the display and determine how quickly the counter is incrementing. How many loops (counts) per second?

   a. Stop the program. Scroll down in the debug window and find the **main( )** statement. Carefully examine the assembly language code generated by the C compiler. Note that each line of C code appears along with the corresponding assembly language code.
   b. What registers are assigned to the variables (HEXDISP, HEXCTRL, count)?
   c. Write down all the code generated for the **while(1)** loop and discuss how this compares to assembly code you would have written yourself. Comment on how well the compiler did in writing this loop based on your C language source code.
   d. How many assembly language instructions are in the while loop? Referring to your speed calculations above, how many *instructions per second* are executed by the NIOS-II processor?
   e. Change the while loop to rewrite the entire loop compactly as:

      ```
      while(1) *HEX = count++;
      ```

      Run this new program and compare your speed results and examine the resulting assembly code. Has anything changed?

2. Connect a keyboard to the PS2 connector in the DE2 board. Press a key while observing the green LED (LEDG7). This LED is connected to the PS2 CLK IN line and flashes whenever the keyboard is transmitting scancodes. Note that if you hold a key down, the key soon begins to repeat rapidly (try this!). Also, if you momentarily press and release a key, observe that there is activity both when pressed and when released.

lab5b.c - Modify the original program so that the counter is incremented upon each falling edge on the PS2 input clock line (CLK IN). The count will now change every time a bit is sent from the keyboard. Run your program. Note how much the value in the hex display changes when you press the **A** key, and again when you release the key. You will have to do this quickly enough that the key does not begin to repeat. How many bits are sent when this key is first pressed? How many bits are sent when this key is released? Try this several times to confirm your answers.

3. Modify your program to read the state of DAT IN on each CLK IN falling edge. Wait until the first falling edge where the data bit is zero (start bit) and thereafter count the next ten clock edges, gathering a new bit each time into a variable called **buffer** and be sure to *arrange the incoming bits so that the LSB ends up in bit 0* and the parity bit ends up in bit 8. Once a complete data byte is received, isolate the 9-bits (data+parity) and send this value to the hexadecimal display. Press the **A** key, and observe the value read (expect 0x01C). Press various keys and record the scancode and parity bit. Observe and record the value of the parity bit with several examples. In each case, explain how the parity bit relates to the data bits.

4. Modify your program to ignore the parity bit and to generate only the eight data bits. Then rearrange your program to put all the code you used to read the keyboard into a separate subroutine called **getkey()** that returns 8-bit scancodes as they are transmitted from the keyboard. Test this new code with your display program.

5. Modify your program to shift each new scancode byte into a 32-bit display variable and send that value to the hex display so that the most recent four scancodes are visible. What are the scancode(s) sent when the **A** key is pressed and then released? Try a few other keys and record your results. Try the SHIFT keys, both the left one and the right one. What do you observe?

6. Modify your program using the **outhex()** function and the **outchar()** functions you have written and send each scancode to the monitor display terminal, followed by a space. Record the scancodes for the following keys as they are pressed and released.

KEY	**SCANCODE (press)**	**SCANCODE (release)**
A		
B		
C		
Lefthand SHIFT		
Righthand SHIFT		
F1		
PAUSE		

7. Modify your program to recognize specifically when the lefthand SHIFT key or the righthand SHIFT key is pressed. Your program should ignore the scancode generated when they are released. *Hint: set a flag when the byte 0xF0 is encountered and ignore any keystroke immediately following that scancode (i.e. whenever the flag is set).*

Using the 16 LEDs, have a single LED lit up which moves one bit to the left when the lefthand SHIFT key is pressed, and one bit to the right when the righthand SHIFT key is pressed. Decide what your program will do when the LED reaches either endpoint.