Lab 3 – PS/2 Keyboard and Serial I/O, Part 1

Objectives

The objectives for this lab are to:

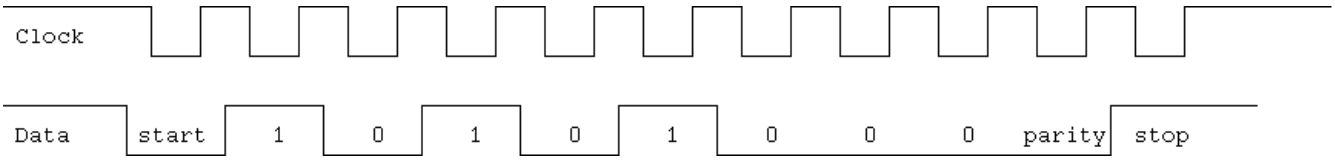
1. control a PS/2 style computer keyboard
2. interface to an external device
3. detect changes in the level of a signal pin
4. collect and interpret data transmitted in serial format
5. implement a finite state machine in assembly language
6. use look-up tables to convert PS/2 key scan codes to ASCII characters
7. use **extrn** and **public** directives to provide access to other modules

Background

This lab focuses on the first external device your project will use, the PS/2 keyboard. You need to read Section 18 of the C8051F120 datasheet (for port and crossbar information) and the UA78M05 voltage regulator datasheet. There are also links to some websites describing how the PS/2 keyboard works that you may find useful.

The PS/2 keyboard for PCs has been around since the mid 1980s when IBM introduced the PS/2 Personal Computer. Over the years it became the de facto standard for PC keyboard interfaces until being replaced by USB keyboards (and many computers still have PS/2 keyboard and mouse connectors). Like USB, the PS/2 keyboard is powered by a +5VDC voltage supplied from the computer. Since our C8051 is a 3.3VDC device and the on-board voltage regulator only outputs that voltage, we need to add a second 5VDC regulator for the keyboard (we will also use this voltage supply for the LCD and some other logic).

The data sent from the keyboard is very similar to the format used by the RS-232 serial port found on PCs of that era. Each byte of data sent from the keyboard is preceded by a start bit and followed by a parity bit and a stop bit. The byte of data is sent LSB first. The parity is odd, meaning the number of one bits in the data and parity bit is an odd number (or, alternatively, if there are an odd number of ones in the data the parity bit is zero, otherwise it is one). The primary difference between an RS-232 serial port and the keyboard is that the keyboard is *synchronous*; a separate clock pin is used to signal when a new bit of data is present. RS-232 uses an *asynchronous* protocol; there is no clock pin (the presence of the start bit signals the beginning of a new transfer) and the clock rate is fixed. The waveforms for the key scan code for the 'A key (*0x15* or *015H* in assembly) is shown on the next page.



Note that the data changes while the clock is high (including the initial stop bit) and is stable while the clock is low. This means we need to detect the falling edge of the clock in the C8051 and use it to initiate the interpretation of the data. How is this done? One technique (the one we will use in this lab) is to busy-wait while the pin is high using the **jb (jump if bit set)** and **jnb (jump if bit clear)** instructions:

```
jnb   CLK, $          ; loop until clock pin goes high
jb    CLK, $          ; loop until clock pin goes low
```

Why do we need to check first for the pin being low? As our code loops to check for each data bit, it's possible that it could run so fast that we finish processing a data bit (say the start bit) before the clock goes high again. Without the **jnb** check, we would incorrectly assume when we loop that the start bit was the LSB of the data. The **jnb** check assures that the clock has transitioned back high before busy-waiting for the next falling edge. Notice that if the clock is in fact already high, the test fails immediately and we proceed to the test-for-falling-edge **jb** check.

The data output by the PS/2 keyboard is unusual (in a number of ways) at first glance. First, the data sent does not directly represent keys, but *key events*. Pressing a key cause one key event to be sent; releasing it cause a different event to be sent. This makes sense when you think about how we use keyboards:

- Holding down a key may cause the key to repeat.
- Pressing shift can cause different values to be sent for the same key
- Multiple keys may be held down simultaneously (Ctl-Alt-Delete, Shift-Ctl, etc).
- Not every key corresponds to an ASCII character; the function keys, shift key, home key, etc.

The PS/2 keyboard uses different *key scan codes* to represent which key was pressed or released. Pressing a key is called a *make* event, while releasing is called a *break* event. Most key scan codes correspond directly to specific keys; however, there are two additional codes which do not. The key scan code *0xf0* (*0F0H*) represents a break event; it is followed by the key scan code of the key that was released. The key scan code *0xe0* (*0E0H*) is an extended key scan code; it is used for some special keys and is followed by a make or break event. A complete list of all the scan codes can be found at http://www.computer-engineering.org/ps2keyboard/scancodes2.html .

Pre-lab

Prior to the lab, you and your lab partner must turn in a “pre-lab” which answers the following questions. Without the correct answers to these questions, you will not be able to get the keyboard to work. You or your partner must upload these answers to the Sakai “Lab 3 Prelab” assignment prior to the deadline.

1. What is the pin name and number (on the J24 connector) for the supply voltage straight from the DC adapter? This pin will be used by the 5V regulator that drives your LCD. You will need to look at the schematic in the C8051F120 development board documentation to determine this.
2. Design the finite state machine described below that will be implemented in **kbprocess**. You can do this with a diagram, flow chart, or pseudo-code. Be specific as to what happens in each state or on each state transition.
3. You will need to use an **extrn** statement to access two code addresses (**keytab** and **keytab2**) and two equates (**minkey** and **maxkey**) in the **scancodes.asm** module. What is the exact assembler directive you need to put into your module? Note: you can use a single **extrn** directive eith multiple operands.
4. Write the assembly code to look up a key scan code in the appropriate look-up table. You need to use **keytab**, **keytab2**, **minkey** (and possibly **maxkey**) to accomplish this. Explain what the code does in your comments.

Lab work

You will need to wire the UA78M05 voltage regulator to provide a +5VDC power supply for the keyboard. There is a pin on the C8051's interface connector which connects to the development board's power supply before the on-board 3.3VDC regulator. You will use this as the input to the UA78M05; to protect everyone involved, you will connect the power supply output to the UA78M05 with a fuse. **Do not apply power to your board until the instructor or TA has verified the wiring is correct. Failure to follow this instruction may result in a significant deduction in points for this entire lab.**

You will be using two inputs for this project; pins P0.5 and P2.2 (these may seem arbitrary, but remember we are anticipating the other devices to be used in the project). The clock signal is on pin P0.5. The pin-outs for the actual PS/2 connector (from the bottom) appear as shown on the left. This would be difficult to wire on the breadboard since pins 3 and 5 are both on the same row, but fortunately the outer casing of the connector is also grounded and can be connected to another pin. Therefore the actual connections you will use on your header are shown on the right.

Actual PS/2 pin out

Pin 4 (Vcc)	Pin 2 (NC)	Pin 1 (Data)	Pin 3 (GND)
Pin 6 (NC)	---	---	Pin 5 (Clock)

Header adapter pin out

Pin 4 (Vcc)	Pin 2 (GND)	Pin 1 (Data)	Pin 3 (NC)
Pin 6 (NC)	---	---	Pin 5 (Clock)

There are three procedures required for your project (you may decide to add more, but these are required):

1. **kbinit:** – This procedure handles the initialization for the keyboard. To do this, you need to initialize variables within this module for the finite state machine. You should also enable the crossbar here (every initialization procedure you will write for the project should enable the crossbar, just to be sure).
2. **kbcheck:** – This procedure checks to see if the ring buffer contains a key. If so, it removes the key and returns the value in R7; otherwise it return -1 (or *0xFF* or *0FFH*) in R7. It is provided for you. Why use R7? This is dictated by the C compiler we will use later in lab; it expects a result to be returned in R7.
3. **kbprocess:** – This procedure handles processing the serial data received from the keyboard. It is called on every falling edge of the clock signal. Your code should perform the necessary operations for the FSM, including processing make/break events, looking up and converting key scan codes, and placing ASCII characters into the ring buffer. The procedure should not loop waiting for the entire character to be sent by the keyboard; it processes one bit at a time and returns to the caller. **This procedure must not modify any general registers (especially accumulator, DPTR, registers R0-R7, or the PSW). This is important**.

These procedures need to be declared **public** within your module. You will also need to include the **extrn** statements for the look-up tables and key scan code ranges.

To make this task tenable, we will use three techniques in the implementation; a finite state machine (FSM), look-up tables, and a ring buffer. Each of these is discussed in more detail below. It will be easier if you do not try to test the FSM with the actual keyboard, since you can't stop it from transmitting in the middle of a key scan code. You can instead test by **carefully** grounding each pin when you want to input a 0 (the internal pull-up resistor will input a one if the pin is not connected to anything), but be aware that you might experience switch bounce when plugging and unplugging the wires.

You should *definitely* develop the code for this module in stages (write a section of the code, then debug it). For example, one approach would be to implement your module in this order:

- the module initialization (set up variable, port pins, etc)
- the ring buffer insertion
- the look-up tables
- the FSM

Finite state machine (FSM)

The FSM will be used to gather the serial data from the keyboard and assemble the key scan codes, and the look-up tables will be used to convert the key scan codes into the corresponding ASCII key or

command key. One way to design the FSM is based on each type of bit received:

1. receive the start bit; remember it
2. receive and store the 8 data bits
3. receive a parity bit; remember it
4. receive the stop bit; check for valid framing and parity

Once the fourth state is entered, if the framing and parity are valid we signal that a new key scan code has been received; otherwise we ignore it. We then return to the first state and start over.

Look-up tables

The look-up tables, which convert the key scan codes into ASCII codes, are provided for you (largely because they're very large and would be tedious for you to type in; it certainly was for me). It also not a straight code-to-code table; some keys (like the 'A' key) have ASCII codes while others (like the Shift key) don't. We also need the table to contain conversions for keys when Shift or Control is pressed. So the table has some special non-ASCII codes which represent when a special key (like Shift or Control) is pressed. Any look-up table entry in the range *0x80-0xfe* (*080H-0FEH*) is one of these special keys. Your code will need to check the value after retrieving it from the table to determine whether it is an ASCII value or a special key which needs further processing.

There are also a number of undefined or unused key scan codes as well; these entries are noted with an *0xff* (*0FFH*). It should be impossible to read one of these codes from the table; if you do, there is something wrong in your code (or, possibly, something wrong in my design of the look-up tables).

You'll notice that the table is broken into two parts (**keytab** and **keytab2**); **keytab** contains the codes when Control is not pressed and **keytab2** the codes when Control is pressed. This is necessary since there are more than 256 possible table entries and the **movc** instruction only allows a 255 byte index in the accumulator. Before looking up a key, your program will need to determine whether or not the Control key is pressed and access the appropriate table. Additionally, there are two entries in each table for each key scan code. These are the values depending on whether or not the shift key is pressed (the first value is the unshifted value). Finally, each table only contains values for keys scan codes 13 to 118 (these are **minkey** and **maxkey** equates, the smallest and largest key scan codes). This makes the tables as small as possible to save code memory. Your code needs to adjust the key scan code appropriately before performing the lookup.

Ring buffers

The ring buffer is used to store each key after its ASCII code have been found in the look-up tables. A ring buffer is a special type of queue data structure. Like a queue, there is a head and tail. New items are inserted into the head of the queue, and removed from the tail. The implementation of a ring buffer is done with a fixed-size array, and uses two indices (or perhaps pointers) for the head and tail. Inserting an item is done by:

```
ring[head] = item;
head = head + 1;
```

This reason this implementation is called a ring buffer is because when the head or tail indices reach the end of the array, they must wrap back around to the first item:

```
if (head == buffer size) head = 0;
```

or

```
head = head % buffer size;      // % is the modulus operator
```

The same is done for the tail index. You must be careful to not overfill the buffer, so usually a count is kept of the number of items in the queue.

The procedure **kpcheck** will perform a similar operation to remove items from the ring buffer. By examining this procedure you can determine which label and variable are used to implement it. You need to be careful about the order of operations to avoid a race condition: remove the key from the ring buffer *before* you decrement the item count.

Testing

Two assembly modules are provided for you; **lab4-test1.asm** and **scancodes.asm**. **lab4-test1.asm** is the main module; it will disable the watchdog timer, set up the stack, and set the system clock to 12.25MHz. It will use your **kbinit**, **kbprocess** and **kbcheck** procedures. Timers 0 and 1 are also used to check how many cycles your **kbprocess** takes to execute and whether you code detects each falling edge (to do this, you will need to connect a temporary wire between the keyboard clock pin and Port pin PX.X). The typical clock frequency for the keyboard is around 13KHz, so your code has a little under 40us to safely perform its work.

The main module will also check that **kbprocess** does not change any of the registers (R0-R7, ACC, B, DPH and DPL). *Note: it does not check **PSW***. If it detects a register has changed, it will busy-wait with a **cjne** instruction. You will then need to fix the problem ***in your code.*** (In a previous year, my instructions did not include "***in your code***", so one "bright" student decided the way to fix the problem was to eliminate the test in my module. So let's be clear; if you encounter a problem here, the problem is *you*, not *me*.)

When a key is detected, the test module will loop as long as R7 contains that value (you should then set R7 to *0xff* **(***0FFH)* in the IDE). The **kbcheck** procedure, as it is provided, assumes your initial implementation does not implement the ring buffer insertion, instead just writing the ASCII key code into the first entry of the ring buffer. Once you implement the ring buffer insertion, be sure to remove the indicated code.