Lab 7 Gadfly Keyboard Interface

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Design the hardware interface between a keyboard and a microcomputer,
- Create the low-level gadfly device driver that can be used in other applications,
- Design the hardware interface between a microcomputer and a single LED,
- Implement keyboard security system.

Review
- Valvano Section 1.6 on open collector logic,
- Valvano Section 2.7 on writing software device drivers,
- Valvano Section 8.1 on keyboard scanning and debouncing,
- Data sheets for 7405, 7407, 74LS14 and 74HC14,
- The chapter on the parallel port in the Motorola Reference Manual.

Starter files
- none

Background

The interface to the keyboard will be performed in two parts. In this first lab, you will interface the keyboard using gadfly synchronization, and in Lab 7 you will redesign the interface using interrupts. Microprocessor controlled keyboards are widely used, having replaced most of their mechanical counterparts. This experiment will illustrate how a parallel port of the microcomputer will be used to control a keyboard matrix. The hardware for the keyboard is one of the following. In each case your computer will drive the rows (output 0 or HiZ) and read the columns. The low level software that inputs, scans, debounces, and saves key's in a FIFO runs in a background period interrupt thread. Your system must handle two key rollover. For example, if I were to type "1,2,3", I could push "1", push "2", release "1", push "3", release "2", then release "3".

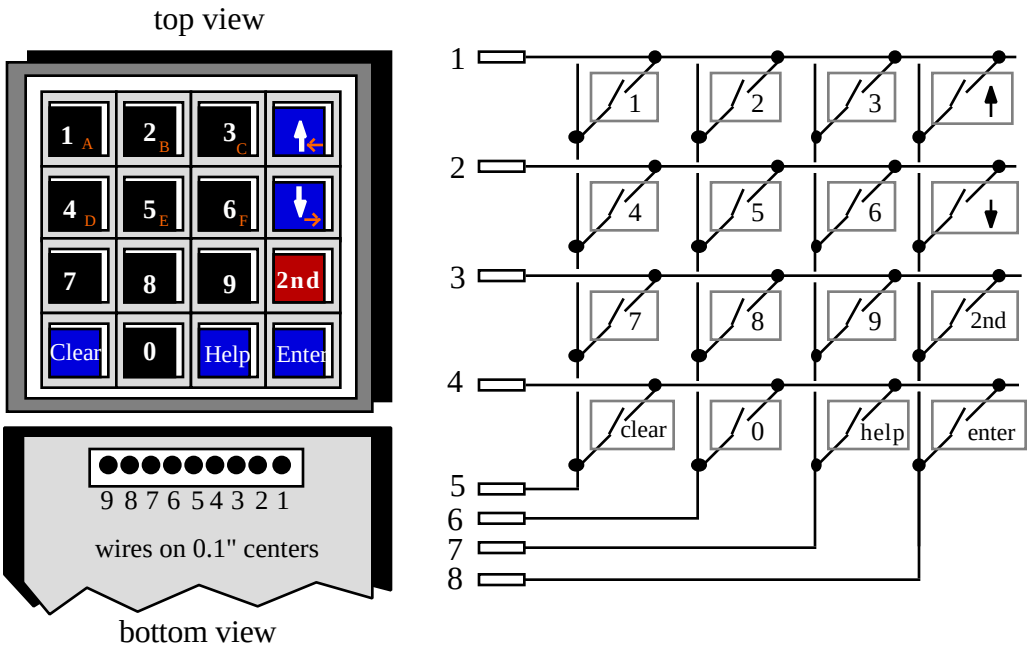

Figure 7.1. 0-9 keyboard, with up arrow, down arrow, 2nd, CLEAR, HELP, and ENTER.

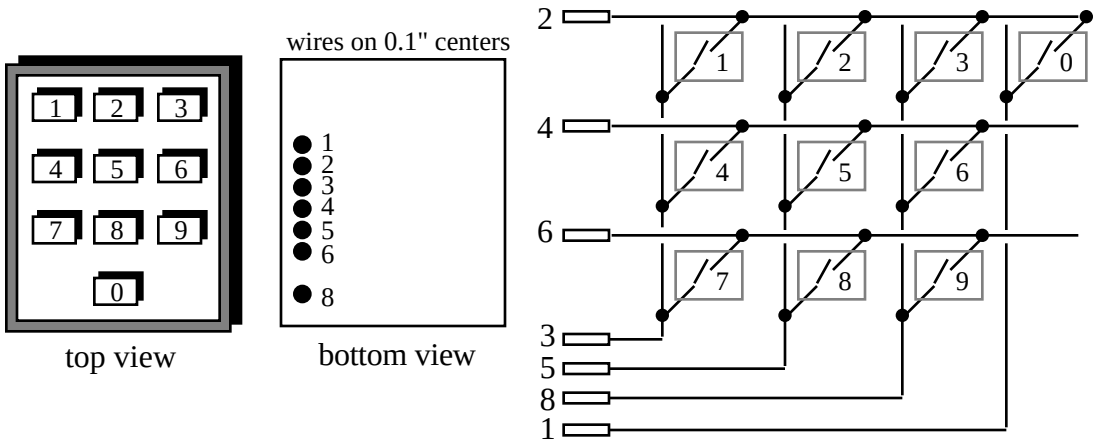

Figure 7.2. 0-9 keyboard.

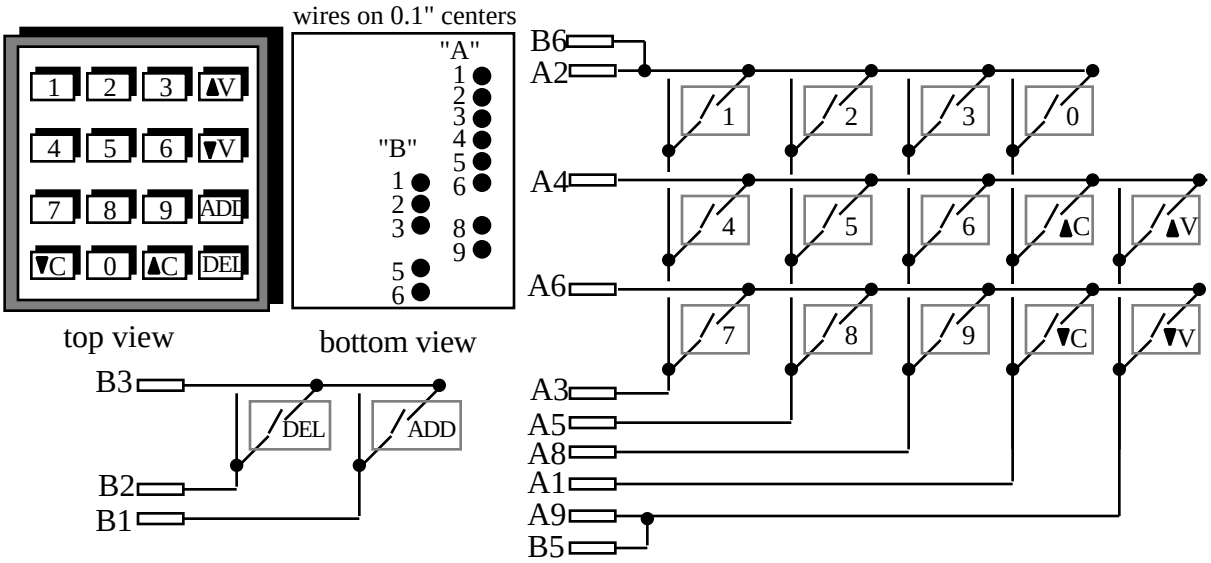

Figure 7.3. 0-9 keyboard with ADD, DEL, _C, _V, _C, _V.

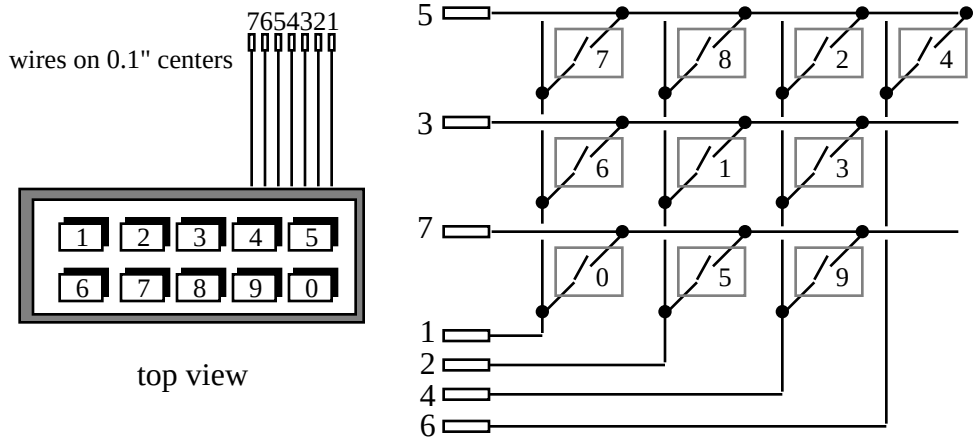

Figure 7.4. 0-9 keyboard.

Low level ***device drivers*** normally exist in the BIOS ROM and have direct access to the hardware. They provide the interface between the hardware and the rest of the software. Good low-level device drivers allow:

- new hardware to be installed;

- new algorithms to be implemented
  synchronization with gadfly, interrupts, or DMA
  error detection and recovery methods
  enhancements like automatic data compression
- higher level features to be built on top of the low level
  OS features like blocking semaphores
  user features like function keys

and still maintain the same software interface. In larger systems like the Workstation and IBM-PC, the low level I/O software is compiled and burned in ROM separate from the code that will call it, it makes sense to implement the device drivers as software TRAP's (SWI's) and specify the calling sequence in assembly language. In embedded systems like we use, it is OK to provide `KEY.H` and `KEY.C` files that the user can compile with their application. ***Linking*** is the process of resolving addresses to code and programs that have been complied separately. In this way, the routines can be called from any program without requiring complicated linking. In other words, when the device driver is implemented with a TRAP, the linking is simple. In our embedded system, the compiler will perform the linking. The concept of a device driver can be illustrated with a prototype device driver. You are encouraged to modify/extend this example, and define/develop/test your own format. We will use interrupts in the later labs. A prototype keyboard device driver follows. The device driver software is grouped into four categories.

1. Data structures: global, protected (accessed only by the device driver, not the user)

`OpenFlag` Boolean that is true if the keyboard port is open
initially false, set to true by `KeyOpen`, set to false by `KeyClose`
static storage (or dynamically created at bootstrap time, i.e., when loaded into memory)

2. Initialization routines (called by user)

`KeyOpen` Initialization of keyboard port
Sets `OpenFlag` to true
Initialized hardware
Returns an error code if unsuccessful
hardware non-existent, already open, out of memory, hardware failure, illegal parameter
Input Parameters(none)
Output Parameter(error code)
Typical calling sequence

```
if(!KeyOpen()) error();
```

`KeyClose` Release of keyboard port
Sets `OpenFlag` to false
Returns an error code if not previously open
Output Parameter(error code)
Typical calling sequence

```
if(!KeyClose()) error();
```

3. Regular I/O calls (called by user to perform I/O)

`KeyIn` Input an ASCII character from the keyboard port
Waits for a key to be pressed, then waits for it to be released (there is bounce and two key rollover)
Returns data if successful
Returns an error code if unsuccessful
device not open, hardware failure (probably not applicable here)
Output Parameter(data, error code)
Typical calling sequence

```
if(!KeyIn(&data)) error();
```

`KeyStatus` Returns the status of the keyboard port
Returns a true if a call to KeyIn would return with a key
Returns a false if a call to KeyIn would not return right away, but rather it would wait
Returns a true if device not open, hardware failure (probably not applicable here)
Typical calling sequence

```
if(KeyStatus()) KeyIn(&data);
```

4. Support software (none in this lab, later you will add interrupt handlers here).

Preparation

Show the required hardware connections. Label all hardware chips, pin numbers, and resistor values. You could connect the keyboard with only pull-up resistors on the inputs (the 6812 can be configured to have internal pull-ups), or you may choose to drive the rows with an open collector 7405 and sense the columns with a Schmitt Trigger 74LS14. You should look at the voltage versus time signals on a scope to determine if the hardware drivers are required. *Please check for valid (0 to +5v) digital signals on your external hardware before connecting them to your computer.*

Write three categories of the low-level keyboard device driver. You must have a separate KEY.H and KEY.C files to simplify the reuse of these routines.

The main program will implement an access code based security system. There will be no SPI input/output in this lab. Each access code will consist of 4 digits between 0-9. The security system can recognize up to five access codes. You will specify these codes in global memory. The keyboard will be used to enter these access codes. If this access code is one of the valid codes, checked by searching the access code database, the single LED is turned on. The LED will remain on until the new key is typed. The main program will need its own data structure to hold the last four keys typed. Assume "1257" and "2222" are valid codes. Following example shows the LED status (0=off, 1=on) after each key hit.

1 2 1 2 5 7 8 9 2 2 2 2 2 2 6 1 2 5 7 4
0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0

Procedure

Configure the keyboard and connect it to the system. Use the scope to measure the bounce time and verify the sharpness of the digital inputs/outputs. Collect some bounce time measurements and discuss it in your report. If the signals spend excess time in the transition range (0.5 to 2v) or if there are voltages above +5 or below 0, then digital driver gates are required. *Test these signals before connecting them to the microcomputer*. Creating the appropriate signals on the output pins can test the keyboard. A key can be pressed and the input is examined to determine if the character is being received. This debugging procedure can be done with a simple software routine before the scanning software is tested.

Build the LED display and perform operational tests. Using an output port and writing a simple program that toggles the LED can do this.

Test the device driver software in small pieces. You could write a main program that inputs from your keyboard and outputs to the computer display. You can use output ports and a scope (like lab 2) to visualize critical real time events. In particular, collect some latency data (time from key touch to computer returning the keycode) measurements and discuss them in your report.

Run the access code security system with the normal PC serial input/output. After the complete system has been observed working in separate blocks, load the complete software package and implement all individual functions.

Checkout

You should be able to demonstrate that any proper key sequence (with 1 or 2 key rollover) causes the LED to activate.

Hints

1) The keys bounce both when pressed and released. Make sure you debounce them before returning data.

2) You can not use trace or breakpoints, or printf statements to debug time-critical software because of the real time nature of the signals. Try using the debugging techniques developed in Lab 2, like defining

```
int DEBUG=0;  //Debugger counter
```

Then place `DEBUG++` instructions at strategic locations in your interrupt software to determine the program execution pattern.

3) Please pay careful attention as to how many details you place in the KEY.H. A good device driver separates the policy (overall operation, how it is called, what it returns, what it does etc.) from the implementation (access to hardware, how it works etc.) In this lab you place the policies in the KEY.H and the implementations in the KEY.C file. Think of it this way, if you wrote commercial software that you wished to sell for profit and you delivered the KEY.H and object file KEY.O, how little information could you place in the KEY.H and still have the software system be fully functional. In object-oriented terms the policies will be public, and the implementations will be private.

4) You may wish to read lab 9, so that the organization of this software (KEY.H) can remain constant even though the implementation (KEY.C) changes.