

## Lab 9d Interrupting Keyboard Interface

This laboratory assignment accompanies the book, *Embedded Microcomputer Systems: Real Time Interfacing*, Second edition, by Jonathan W. Valvano, published by Thomson, copyright © 2006.

- Goals**
- Design the hardware interface between a keyboard and a microcomputer using interrupts,
  - Study the concept of critical sections and nonintrusive debugging.

- Review**
- Valvano Chapter 4 on basic interrupt mechanisms and reentrant programming,
  - Valvano Chapter 6 on input capture interrupts,
  - Valvano Section 8.1 on keyboard scanning and debouncing,
  - The chapter on output compare in the Freescale 9S12 Reference Manual

- Starter files**
- **Lab9dv2**, **OC3** and **IC** projects, **RXFIFO.H**, and **RXFIFO.C** (inside **SCIA** project)

### Background

The interface to the keyboard will be performed using input capture 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 input data from a keyboard matrix. The hardware for the keyboard is shown in Figure 9.1. Your computer will drive the rows (output 0 or HiZ) and read the columns. The low level software (inputs, scans, debounces, and saves keys in a FIFO) runs in the background using interrupts. To scan the keyboard, the software drives the first row low (output 0), while the other rows are off (output HiZ). The software then reads the four columns. Any keys are pressed in that row can be identified as zeros in the column position. If no keys are pressed in that row, then all column inputs will be high. In a similar manner the software checks the other three rows. To recognize that a key has been pressed (or released), your software will drive all four rows low (output 0), and detect a rise or fall on any of the column signals using input capture. **Your system will not need to handle two-key rollover.** For example, when some people type “1,2,3”, they push “1”, push “2”, release “1”, push “3”, release “2”, then release “3”. In this lab, when we type “1,2,3”, we push “1”, release “1”, push “2”, release “2”, push “3”, then release “3”.

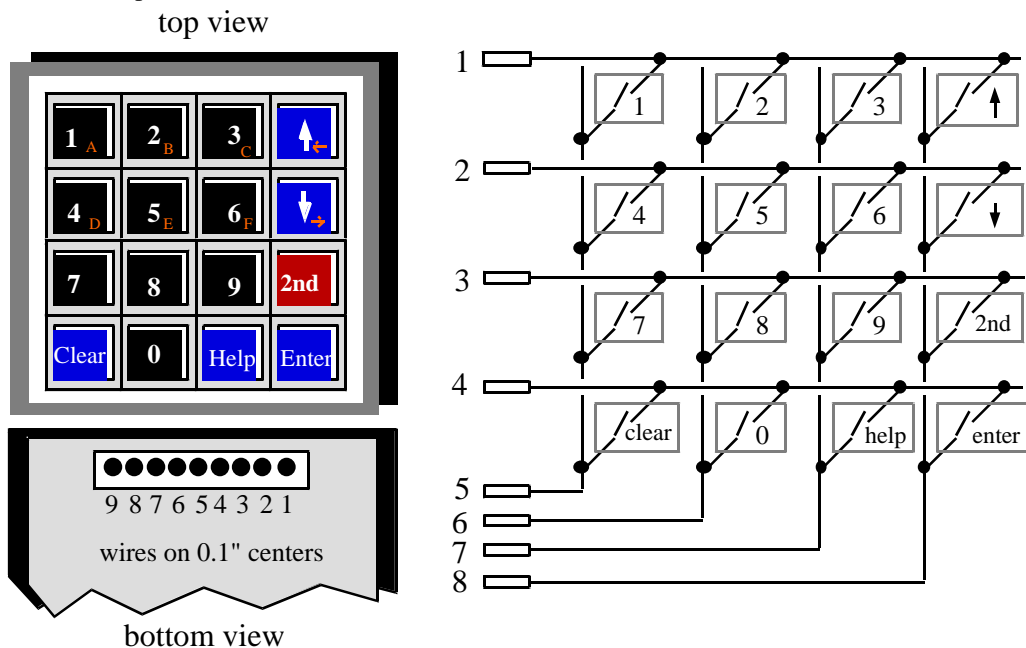


Figure 9.1. 0-9 keyboard, with up arrow, down arrow, 2nd, CLEAR, HELP, and ENTER. Pin 9 is not connected.

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 saved as object code files (e.g., DLL) 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** source code files that the user can compile with their application. **Linking** is the process of resolving addresses to code and programs that have been compiled 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.

Labs 9 and 10 together will create a fixed-point calculator. In this lab, you will design the keyboard interface using interrupt synchronization. You will use both input capture and output compare interrupts to read and debounce the switch. There are two advantages of interrupts in an application like this. Placing the key input into a background thread, frees the main program to execute other tasks while the software is waiting for the operator to type something (unfortunately this calculator doesn't have anything else to do). The second advantage of interrupts is the ability to create accurate time delays even with a complex software environment. In particular, the output compare interrupt can be used to accurately wait for the bouncing to stop. A prototype keyboard device driver follows. As always, you are encouraged to modify this driver specification and define/develop/test your own format. This time we have all four categories of the device driver software.

### 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 **Key\_Open**, set to false by **Key\_Close**

static storage (permanent allocation, local scope)

**Fifo** queue, with **Clr**, **Put**, **Get** functions

static storage initialized by **Key\_Open** (permanent allocation, local scope)

linkage between Keyboard interrupt and **Key\_InChar**

**ErrorCode** specifies if a device driver error has occurred

static storage initialized by **Key\_Open** (permanent allocation, local scope)

0 means no error, bit0, bit1, bit2, ... signify specific errors that have occurred. For example, you could define

Bit0 = 1 if any hardware fault occurs (by comparing **PTT** to **PTIT** and **PTAD** to **PTIAD**)

Bit1 = 1 if user tried to open the driver when the driver was already open

Bit2 = 1 if user tried to perform I/O without first opening the driver

Bit3 = 1 if data is lost because **Fifo** became full

Bit4 = 1 if user tried to close the driver when the driver wasn't previously open

### 2. Initialization routines (called by user)

**Key\_Open** Initialization of keyboard port

Sets **OpenFlag** to true

Initializes hardware, initializes FIFO queue

Clears the **ErrorCode** if successful

Sets bit 1 in the **ErrorCode** if already open; Sets bit 0 in the **ErrorCode** if hardware failure

Input Parameters(none)

Output Parameter(none)

Typical calling sequence

```
Key_Open();
```

**Key\_Close** Release of keyboard port

Sets **OpenFlag** to false

Sets bit 4 in the **ErrorCode** if not previously open

Input Parameters(none)

Output Parameter(none)

Typical calling sequence

```
Key_Close();
```

**Key\_ErrorCheck** Returns any previously recorded errors

Returns the **ErrorCode**

Typical calling sequence

```
if(err = Key_ErrorCheck()) Display(err);
```

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

**Key\_InChar** Input an ASCII character from the keyboard port

Tries to **Get** a byte from the **Fifo**

Sets bit 2 in the **ErrorCode** and returns with 0xFF if the driver is not open

Returns data if successful

Returns 0 if **Fifo** is empty

Output Parameter(data)

Typical calling sequence (you are free to change it so **Key\_InChar** waits for next input)

```
if(data = Key_InChar()) process(data);
```

**Key\_Status** Returns the status of the keyboard port (checks **Fifo** to see if data is waiting)

Returns the number of elements in the **Fifo**

Returns a 0 if a call to **Key\_InChar** would not return data (**Fifo** is empty)

Returns a 1,2,3,... if a call to **Key\_InChar** would return data

Sets bit 2 in the **ErrorCode** and returns with 0xFF if the driver is not open

Typical calling sequence

```
while(Key_Status()==0){}; // wait for keyboard input
data = Key_InChar();
```

### 4. Support software (protected, not directly accessible by the user).

There are five interrupt service handlers. A separate input capture interrupt is attached to each column

**ICHan7, ICHan6, ICHan5, ICHan4**

Occurs when a key is touched or released

This handler disarms all input captures, and arms an OC handler to occur 20 ms later

**OCHan**

Occurs 20 ms after a key is touched or released

Scans the matrix, if exactly one key is pressed, it puts ASCII code into the **Fifo**

This handler disarms itself and arms all input captures

Sets bit 3 in the **ErrorCode** if the **Fifo** becomes full

Sets bit 0 in the **ErrorCode** if hardware failure (by comparing **PTT** to **PTIT** and **PTAD** to **PTIAD**)

**FifoClr, Fifo\_Put, Fifo\_Get** functions implement the first in first out byte-wide queue.

Each 9S12 port has two data registers (**PTT** and **PTIT**, **PTM** and **PTIM**, **PTAD** and **PTIAD**). For example, if a Port AD pin is an output, reading **PTIAD** will return the actual high/low level existing on the pin. However, if **PTAD** is an output, reading **PTAD** will return the high/low level most recently written to **PTAD**. Comparing **PTAD** to **PTIAD** is one way to detect an overload or short-circuit condition on a Port AD output pin

Set the **DDRAD** direction register bit to output

Set the **PTAD** output pin high

Verify that the corresponding bit in both **PTAD** and **PTIAD** are high

Set the **PTAD** output pin low

Verify that the corresponding bit in both **PTAD** and **PTIAD** are low

*Nonintrusiveness* is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a **SCI\_OutString** statement added to your source code and single-stepping are very intrusive because they significantly affect the real time interaction of the hardware and software. When a program interacts with real time events, the performance is significantly altered. On the other hand, dumps, dumps with filter and monitors (e.g., output strategic information on LED's) are much less intrusive. A logic analyzer that passively monitors the address and data by is completely non-intrusive. An in-circuit emulator is also nonintrusive because the software input/output relationships will be the same with and without the debugging tool.

A program segment is *reentrant* if it can be concurrently executed by two (or more) threads. This issue is very important when using interrupt programming. To implement reentrant software, place local variables on the stack, and avoid storing into global memory variables. Use registers, or the stack for parameter passing (normal C call/return method). Typically each thread will have its own set of registers and stack. A nonreentrant subroutine will have a section of code called a *vulnerable window* or *critical section*. An error occurs if

- 1) one thread calls the nonreentrant subroutine
- 2) is executing in the “vulnerable” window when interrupted by a second thread
- 3) the second thread calls the same subroutine or a related subroutine. There are a couple of scenarios
  - A) 2nd thread is allowed to complete the execution of the subroutine  
control is returned to the first thread  
the first thread finishes the subroutine.
  - B) 2nd thread executes part of it, is interrupted and then re-entered by a 3rd thread  
3rd thread finishes  
control is returned to the 2nd process and it finishes  
control is returned to the 1st process and it finishes
  - C) 2nd thread executes part of it, is interrupted and the 1st thread continues  
1st thread finishes  
control is returned to the 2nd thread and it finishes

A vulnerable window may also exist when two different functions access the same memory-resident data structure. Consider the situation where two concurrent threads are communicating with a First-In-First-Out (FIFO) queue. What would happen if the **Fifo\_Put** function executed in between any two assembly instructions of the **Fifo\_Get** routine? In other words, the main thread starts executing **Fifo\_Get** to look for new input data (interrupts are enabled). Then, an output compare interrupt occurs and the background thread calls **Fifo\_Put** to save the next ASCII code. Will data be lost? Will the FIFO become damaged?

An *atomic operation* is one that once started is guaranteed to finish. In most computers, once an instruction has begun, the instruction must be finished before the computer can process an interrupt. Therefore, the following read-modify-write sequence is atomic because it can not be reentered.

```
inc counter           where counter is a global variable
```

On the other hand, this read-modify-write sequence is not atomic because it can start, then be interrupted.

```
ldaa counter         where counter is a global variable
inca
staa counter
```

In general, nonreentrant code can be grouped into three categories all involving *nonatomic writes to global variables*. The first group is the *read-modify-write* sequence.

- 1) a read of global variable produces a copy of the data
- 2) the copy is modified
- 3) a write stores the modification back into the global variable

Example: **Money +=100;** which may be implemented in assembly as

```
ldd Money   where Money is a global variable
addd #$100
std Money   Money=Money+$100
```

In the second group is the *write followed by read*, where the global variable is used for temporary storage:

- 1) a write to the global variable is used to save the only copy important data
- 2) a read from the global variable expects the original data to still be there

Example:

```
short thePort;
void function(void){
  thePort = PTT; // save in global
  // a bunch of stuff that may modify PTT, but not thePort
  PTT = thePort;} // restore original value
```

In the third group, we have a *non-atomic multi-step write* to a global variable:

- 1) a write part of the new value to a global variable
- 2) a write the rest of the new value to a global variable

Example:

```
short position[2]; // (x,y) location
void function(void){
    position[0] = PTT; // x position
    position[1] = PTM; // y position
}
```

Reentrant programming is very important when writing software in the context of multiple threads (interrupts). Obviously, we minimize the use of global variables. But when global variables are necessary, we must be able to recognize potential sources of bugs due to critical sections. We must study the assembly language output produced by the compiler. For example, we can't determine whether the following read-modify-write operation is reentrant or not without knowing if it is atomic:

```
time++;
```

The following read-modify-write operation is reentrant when using Metrowerks, because it is atomic:

```
PTT = PTT | 0x01; // set PT0
```

### Preparation (do this before your lab period)

1. Design the hardware interface between the keyboard and the 9S12. Label all hardware chips, pin numbers, and resistor values. Draw the circuit using a CAD program like ExpressSCH. You will need 10 k $\Omega$  pull-up resistors on the column inputs. You should not use PAD7, PAD6, PT1 or PT0 to interface the keyboard because of the existence of the switches and LEDs on the docking module.

2. Write the keyboard device driver, creating **Key.h** and **Key.c** files. Put the prototypes for public functions in the **Key.h** file. Look ahead to the next lab to assign appropriate ASCII codes for each of the 16 keys. You can use the '2nd' key to create 30 different codes. Because this input capture interface can not handle 2-key rollover, the user will have to hit '2nd', release '2nd', hit '1', release '1' to generate the secondary code for '1'.

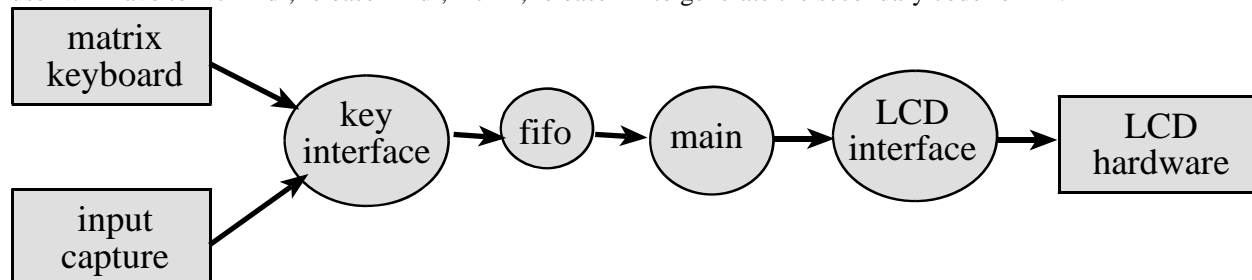


Figure 9.2. Data flows from the keyboard to the main program. The main program can use the LCD or SCI.

3. The purpose of this step is to determine whether or not your FIFO has a critical section. You will be modifying the **Lab9dv2** project to test your FIFO queue. Just like a typical application that uses a keyboard, this background generates data then calls **Fifo\_Put**. Similarly, this foreground program calls **Fifo\_Get** when it needs data to process. However, the difference between this system and an actual application is that this background calls **Fifo\_Put** at a high rate and with a predictable data sequence (0,1,2,3,...65535,0,1,...). The producer rate is fast, but slow enough to avoid the FIFO becoming full (when running you will verify that **NumFullError** remains zero). However, this test program will put from 1 to 10 elements in the queue at a time, so the FIFO size frequently goes above 1 element. There are two FIFO implementations in this project (the first one does not contain any critical sections, and a second implementation does contain at least one critical section). If your FIFO is different from these two implementations, add it to the project so it too can be tested. Since only the background calls **Fifo\_Put** and only the foreground calls **Fifo\_Get**, the functions themselves need not be re-entrant. However, the issue is "do the **Fifo\_Put** and **Fifo\_Get** functions interfere with each other?" There is a debugging instrument (called **EnteredCount**) that counts the number of times that the main program starts executing **Fifo\_Get**, an interrupt occurs and the ISR executes **Fifo\_Put**. Change the compile flag, **GOOD** to 1, and observe the listing file for **Fifo\_Get** and count the total number of assembly instructions for the good get function. Change the compile flag, **GOOD** to 0, and count the assembly instructions for the bad implementation as well. (Do not count execution time in cycles, but rather count assembly instructions.)

4. Write a main program to test the keyboard device driver. You are allowed to use either SCI or LCD output to assist in testing and debugging the keyboard interface. You could write this main program so that it inputs from your keyboard and outputs to some display the TA can see. You will find it convenient to implement an input string function with echo to either SCI or LCD. You can add a backspace or delete line key to allow the operator to correct typing errors.

Figure 9.3 shows a possible call graph of the system. Dividing the system into modules allows for concurrent development and eases the reuse of code. Notice the details of input capture, port pin assignments and the FIFO queue are hidden from the main program.

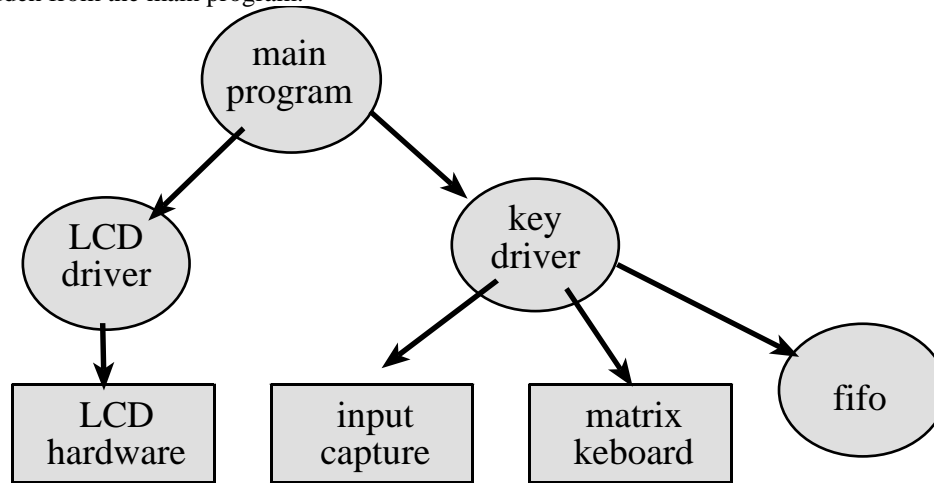


Figure 9.3. A call graph showing the testing of the keyboard device driver, using either the LCD or SCI.

### Procedure

1. Set up the software so one row is low and the other rows are HiZ. Connect two scope probes to two adjacent column signals. Measure the voltage versus time signals to determine if your particular keyboard has switch bounce. Use the measurements the other channel to see if touching one switch causes glitches on adjacent columns.

2. Connect the keyboard to the microcontroller. Write a very simple main program that performs these operations over and over

Set row 3 output low, other rows are floating, then read the column

Set row 2 output low, other rows are floating, then read the column

Set row 1 output low, other rows are floating, then read the column

Set row 0 output low, other rows are floating, then read the column

For example, if the four rows are interfaced to PTAD5,4,3,2, and the four columns are interfaced to PTT7,6,5,4 then

```
unsigned char Column;
```

```
void main(void){
```

```
    ATDDIEN = 0xFF;
```

```
    DDRT = 0x03; // PT7,6,5,4 inputs
```

```
    PTAD = 0; // Pin will be 0, when it is output, and HiZ when it is input
```

```
    for(;;){
```

```
        DDRAD = 0x20; // activate only the row connected to PTAD5
```

```
        Column = PTT&0xF0;
```

```
        DDRAD = 0x10; // activate only the row connected to PTAD4
```

```
        Column = PTT&0xF0;
```

```
        DDRAD = 0x08; // activate only the row connected to PTAD3
```

```
        Column = PTT&0xF0;
```

```
        DDRAD = 0x04; // activate only the row connected to PTAD2
```

```
        Column = PTT&0xF0;
```

```
    }
```

```
}
```

Using the debugger, single step this **main** program, and using a voltmeter observe the voltages on the column signals. Compare the measured voltages to the digital data (**Column**) collected by the program.

3. The purpose of this step is to experimentally determine whether or not your FIFO has a critical section. Using the **Lab9dv2** project, download and run with **GOOD** equal to 1. Verify the producer rate (frequency of the calls to **Fifo\_Put**) does not exceed the consumer rate (number of bytes/sec that can be removed and checked for proper sequence) by noticing the FIFO never becomes full. Let **n** be the number of times **Fifo\_Put** interrupts **Fifo\_Get**, debugging variable **EnteredCount**. Let **m** be the total number of assembly instructions in **Fifo\_Get**. Run this test until **n** greatly exceeds **m** (e.g., let **n** approach 1 million times.) Think of this corresponding probability question. Assume you have **m** different cards in a deck. Now, you will select one card at random, with replacement. What is the probability after **n** selections (with replacement) that a particular card was never selected? Similarly, what is the probability that all cards were selected at least once? Use these measurements to guarantee (at least be pretty certain) that the **Fifo\_Put** function was executed in between each pair of **Fifo\_Get** assembly instructions at least once.

Change **GOOD** to 0, compile, download and run again. Notice the out of sequence errors (debugging variable **Errors**). Does the critical section cause errors immediately? Observe the assembly listing and identify specific assembly instructions involved in the critical section. There are two ways to fix this bug, both involving making the read-modify-write sequence atomic. Fix the bug and retest the next FIFO programs. If your FIFO is different from these two, use this procedure to test your FIFO too.

4. Test the device driver software in small pieces.

5. Measure the interface latency (time from key touch to FIFO put) and discuss this data in your report. The exact time the key is touch will be recorded in the timer latch by the input capture hardware. Measure the maximum, minimum and average latency over about 20 key touches.

#### **Deliverables (exact components of the lab report)**

A) Objectives (1/2 page maximum)

B) Hardware Design

Keyboard interface, showing all external components (Preparation 1)

C) Software Design (a hardcopy software printout is due at the time of demonstration)

Explain how your software removes switch bounce

If you organized the system different than Figure 9.2 and 9.3, then draw its data flow and call graphs

D) Measurement Data

Keyboard bounce data (Procedure 1)

Give the **EnteredCount**, and the number of instructions in your **Fifo\_Get**.

Keyboard latency data (Procedure 5)

E) Analysis and Discussion (1 page maximum)

Do these data prove your Fifo has no critical section? Or are they just extremely convincing?

Is the keyboard interface real time?

#### **Checkout**

Demonstrate the functionality of your keyboard interface. Convince your TA that your FIFO implementation has no critical sections (could be theoretical proof or experimental observations.) You will be asked to describe the two ways the critical section could have been removed from the bad FIFO implementation.

**A hardcopy printout of your software will be given to your TA, and graded for style at a later time.**

#### **Hints**

1) Try using the debugging techniques developed in earlier labs.

2) The time executing in an interrupt service routine must be small and bounded. It is not appropriate to wait 20 ms inside an ISR. However, many students find adding a 1-4us delay between setting the rows and reading the columns increases the accuracy of the keyboard scanning. This short wait will more likely be required when running at 24 MHz.

3) There are internal pull-ups that can be activated on digital input pins, but some students find it more reliable to use external 10 k $\Omega$  resistors.