Lab 8e Interrupting Keyboard Interface and Calculator


Goals
- Redesign the hardware interface between a keyboard and a microcomputer using interrupts,
- Study the concept of critical sections and nonintrusive debugging,
- Develop routines for fixed-point arithmetic.

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 Motorola Reference Manual.

Starter files
- OC3 and IC projects, RXFIFO.H, and RXFIFO.C

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 control a keyboard matrix. The hardware for the keyboard is shown in Figure 8.1. 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 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 columns, and any keys are pressed in that row will be identifies 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 lay, when we type “1,2,3”, we push “1”, release “1”, push “2”, release “2”, push “3”, then release “3”.

Figure 8.1. 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

Jonathan W. Valvano
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` 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.

In this keyboard 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 system 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 example, 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

   **Fifo** FIFO queue, with `Clr`, `Put`, `Get` functions
   - static storage initialized by `Key_Open`
   - linkage between Keyboard interrupt and `Key_InChar`

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

   **Key_Open** Initialization of keyboard port
   - Sets `OpenFlag` to true
   - Initialized hardware, size of FIFO queues
   - 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
     ```c
     if(!Key_Open()) error();
     ```

   **Key_Close** Release of keyboard port
   - Sets `OpenFlag` to false
   - Returns an error code if not previously open
   - Output Parameter(error code)
   - Typical calling sequence
     ```c
     if(!Key_Close()) error();
     ```

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
   - Returns data if successful
   - Returns an error code if unsuccessful
     - device not open, Fifo empty, hardware failure (probably not applicable here)
   - Output Parameter(data, error code)
   - Typical calling sequence (you are free to change it so `Key_InChar` waits for next input)
     ```c
     while(!Key_InChar(&data)) process();
     ```
Key_Status  Returns the status of the keyboard port (checks FIFO to see if data is waiting)
  Returns a true if a call to Key_InChar would return with a key
  Returns a false if a call to Key_InChar 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(Key_Status()) Key_InChar(&data);

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
ICHan0, ICHan1, ICHan2, ICHan3
  Occurs when a key is touched or released
This handler disarms all input captures, and arms an OC handler to occur 20 ms from now
OChan
  Occurs 20 ms after a key is touched or released
Scans the matrix, if exactly one key, it puts ASCII code into the Fifo
This handler disarms itself and arms all input captures

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 printf 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 subroutines access the same memory-resident data
structure. Consider the situation where two concurrent threads are communicating with a FirstInFirstOut (FIFO)
queue. What would happen if the PUTFIFO subroutine executed in between any two assembly instructions of the
GETFIFO routine (or vice versa.)

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.
inccounter  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.
ldacounter  where counter is a global variable
inca
staca 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: \texttt{Money +=100;} which may be implemented in assembly as
\begin{verbatim}
  ldd  Money  where Money is a global variable
  add $100
  std Money  Money=Money+$100
\end{verbatim}

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:
\begin{verbatim}
  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
\end{verbatim}

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:
\begin{verbatim}
  short position[2]; // (x,y) location
  void function(void){
    position[0] = PTT; // x position
    position[1] = PTM; // y position
  }
\end{verbatim}

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 must be able to recognize potential sources of bugs due to nonreentrant code. 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:
\begin{verbatim}
  time++;
\end{verbatim}

The following read-modify-write operation is reentrant when using Metrowerks, because it is atomic:
\begin{verbatim}
  PTT = PTT | 0x01;  // set PT0
\end{verbatim}

\textbf{Preparation}

Show the required hardware connections. Label all hardware chips, pin numbers, and resistor values. You will need 10 k\ohm pull-up resistors on the column inputs. You should look at the voltage versus time signals on a scope to determine if hardware drivers are required, and to check if your particular keyboard has switch bounce. \textit{Please check for valid (0 to +5V) digital signals on your external hardware before connecting them to your computer.}

The first main program you write will be used to test the keypad device driver. You are allowed to add lots of SCI 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 the LCD display.

In the second main program you will design a four-function 16-bit signed fixed-point calculator. All numbers will be stored in signed 16-bit fixed-point format with a constant of 0.001. The full-scale range is from -32.767 to +32.767. You should be able to use the fixed-point routines developed in a previous lab, by converting to keyboard input and LCD output. The matrix keyboard will include the numbers ‘0’-‘9’, and the letters ‘+’, ‘-’, ‘*’, ‘/’, ‘=’ and ‘.’. The HD44780 LCD display will show both a 16-bit global accumulator, and a 16-bit temporary register. You are free to design the calculator functionality in any way you wish, but you must be able to: 1) clear the accumulator and temporary; 2) type numbers in using the matrix keyboard; 3) add, subtract, multiply, and divide; 4) display the
results on the HD44780 LCD display. No SCI input/output is allowed in the calculator program. Figure 8.2 shows the data flow graph of the calculator.

![Diagram of data flow](image1)

**Figure 8.2.** Data flows from the keyboard to the LCD.

Figure 8.3 shows a possible call graph of the system. Dividing the system into modules allows for concurrent development and eases the reuse of code.

![Diagram of call graph](image2)

**Figure 8.3.** A call graph showing the four modules used by the calculator.

**Procedure**

Configure the keyboard and connect it to the system. Once again, test the device driver software in small pieces. You can use output ports and a scope to visualize when interrupts are occurring, when data is put into the Fifo, and when data is get from the Fifo. Collect some latency data (time from key touch to Fifo put) measurements and discuss them in your report. The exact time the key is touch will be recorded in the timer latch by the input capture hardware.

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

A) Objectives (1/2 page maximum)
B) Hardware Design
   - keyboard interface, showing all external components
C) Software Design (no software printout in the report)
   - Explain how your software removes switch bounce
   - If you organized the system different than Figure 8.2 and 8.3, then draw its data flow and call graphs
D) Measurement Data
   - Keyboard latency data
E) Analysis and Discussion (1 page maximum)
Checkout

You should be able to demonstrate the calculator functions. You should show the TA your method(s) to nonintrusively visualize the background thread interrupting the critical section of the foreground thread. Prove to your TA that your Fifo implementation has no critical sections (proof could be theoretical or experimental.)

Your software files will be copied onto the TA’s zip drive during checkout.

Hints
1) Try using the debugging techniques developed in earlier labs.
2) Look at how the RxFifo is used to pass data from the SCI input interrupt to the SCI_InChar function in the file SCIa project.
3) The time executing in an interrupt service routine must be small and bounded. It is not appropriate to wait 20 ms inside an ISR.