Lab 9 Interrupting 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
- Redesign 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 Section 6.2.3 on Output Compare periodic interrupt,
- Valvano Section 8.1 on keyboard scanning and debouncing,
- Lab 7 on the Gadlfy keyboard interface
- The chapter on output compare in the Motorola Reference Manual.

Starter files
- OC3TEST.C, OC3.C, OC3.H, RXFIFO.H, and RXFIFO.C

Background

In this second keyboard lab, you will redesign the keyboard interface using interrupt synchronization. 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 security 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 `KeyOpen`, set to false by `KeyClose`
static storage (or dynamically created at bootstrap time, i.e., when loaded into memory)

`Fifo` FIFO queue, with `Clr, Put, Get`
dynamic storage created by `KeyOpen`
linkage between Keyboard interrupt and `KeyIn`

2. Initialization routines (called by user)

`KeyOpen` 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(Fifo size)
Output Parameter(error code)
Typical calling sequence

```
if(!KeyOpen(100)) error();
```

`KeyClose` Release of keyboard port
Sets `OpenFlag` to false
Release memory of FIFO queues
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
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 KeyIn waits for next input)

```
while(!KeyIn(&data)) process();
```

`KeyStatus` Returns the status of the keyboard port (checks FIFO to see if data is waiting)
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 (protected, not directly accessible by the user).

There is one interrupt service handler:

`KeyHan`

Occurs every 20 ms using output compare
Scans, debounces, deals with 1 or 2 key rollover, puts ASCII code into Fifo

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.

```
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:

```
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:

```
sts  Safe          Save SP in a Safe place
pshx
lds  Safe          Restore SP
```

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:

```
stx  Var           Var is a 32 bit global variable
sty  Var+2
```

Reentrant programming is very important when writing high-level language software too. 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 not determine whether the following read-modify-write operation is reentrant or not without knowing if it is atomic:

```
time++;
```

Preparation

Show any changes to the hardware you wish to make. Label all hardware chips, pin numbers, and resistor values. Write four 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. You will use the same main program that implements the access code based security system.

Write a second main program and periodic interrupt handler to study the concept of *critical sections*. In this system, there will be two threads implementing the producer/consumer problem, like the security system. The background thread, implemented as a periodic interrupt, will produce a simple predictable data stream (e.g., 0,1,2,3,...) and put bytes in the FIFO. The foreground thread, implemented as the main program, will consume the data, and check for the proper sequence. Your job is the develop methodologies for *nonintrusively* detecting when a call to PutFifo (made by the producer thread) has interrupted a call to GetFifo (made by the consumer thread.) Design two implementations of PutFifo and GetFifo: one such that there is a critical section, and one without any critical sections. Make an assembly listing of the GetFifo/PutFifo implementation with the critical section. Show the places within GetFifo that if the background thread were to interrupt then call PutFifo, an error would occur (lost or damaged data.) Full credit will be given to solutions that do not disable interrupts to make the correct operations (i.e., design the FIFO's so that PutFifo and GetFifo are not critical with each other). You do not need to disable interrupts or use semaphores to eliminate critical sections in GetFifo/PutFifo. If the right FIFO algorithm is chosen, then GetFifo and PutFifo will not be critical with each other (i.e. if GetFifo gets interrupted and the interrupt handler calls PutFifo, then no loss or damage of data will occur). Note that this does not mean that either PutFifo or GetFifo will be reentrant, that would require disabling interrupts or using semaphores. You will still need to design two implementations of PutFifo and GetFifo. One with a critical section (e.g., shared global where both routines perform a read modify write) and one without any critical sections (to be used in the system).

Procedure

Configure the keyboard and connect it to the system. Once again, 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 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. Compare these results with the previous data collected with the gadfly implementation.

When running the critical code test, make sure the periodic interrupt rate is fast enough to increase the probability of hitting the critical section, but not too fast so that the fifo gets full.

Checkout

You should be able to demonstrate that any proper key sequence (with 1 or 2 key rollover) causes the LED to activate. 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 correct GetFifo/PutFifo implementation has no critical sections (proof could be theoretical or experimental.)

Hints

1) Try using the debugging techniques developed in Lab 2.
2) There are FIFO examples on the web site (RXFIFO.C RXFIFO.H), in the Example/MC6812/Chap4.C file on the TExaS CD and in the ICC12 folder after TExaS is installed (see the TUT4.C example).