Homework 12 Due: Monday-Tuesday 4/22-23 in Class (turn in one piece of paper)

Exercises are activities you should do, but will not turn in for a grade. Assignments will be turned in. You are allowed to work in groups of two students on homework, but each student must turn in their own solutions.

Assignment 12.1: Go to http://users.ece.utexas.edu/~valvano/Volume1/ and download the grading engine. You should run the HW with the simulator. http://users.ece.utexas.edu/~valvano/Volume1/HW_FIFO.zip

You will need to open a UART#1 window to see the score.
1) Unzip HW_FIFO.zip
2) Launch HW_FIFO.uvproj that starts Keil uVision
3) Build the object code Target->Rebuild all target files
4) Start the debugger and hit run (look in the UART#1 window)

My main program will call your functions multiple times, and will give your solution a performance score of 0 to 100. You should not modify my main program or my example data. When you have written your functions, you should run my main program, which will output the results to the UART.

A FIFO is a first in first out queue. In Lab 9 we will use a FIFO to link a foreground thread with a background thread. The producer will enter data into the FIFO (FIFO_Put), and the consumer will remove data from the FIFO (FIFO_Get). In this FIFO each element is an unsigned 16-bit number, and the FIFO can hold up to 6 elements. Actual data will be restricted to 0 to 65534. The value 65535 will signify error or no data. If there is data in the FIFO, the first entry in the buffer (FIFO[0]) contains the oldest data, which is the data that will be returned on the next call to FIFO_Get.

<table>
<thead>
<tr>
<th>Empty</th>
<th>One</th>
<th>Two</th>
<th>Three</th>
<th>Four</th>
<th>Five</th>
<th>Six</th>
</tr>
</thead>
<tbody>
<tr>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td>65535</td>
<td></td>
</tr>
</tbody>
</table>

The problem is constrained such that FIFO is the only global variable that is allowed. You are allowed to have as many local variables as you wish. The static qualifier has been added so this global variable is accessible only in the FIFO.c file. This means this variable is internal or private to the FIFO module.

#define SIZE 6
static unsigned short FIFO[SIZE];
A second constraint is that the software should be easily scaled. The figure shows a FIFO with a maximum of 6 entries, but your solution should allow the user to select the desired FIFO size by simply changing the `#define SIZE` definition. `SIZE` is not considered a variable, but rather it is a compile-time constant. This definition is placed in the FIFO.h file to facilitate user modification. The figure shows 6 elements, but you will run at 9. The details of how data are passed are included in the comments of these three functions.

Part a) The first function, called `FIFO_Init`, should initialize the FIFO. You will have to put 65535 values into the `FIFO[]` buffer. The 65535 value signifies the entry does not contain data.

Part b) The second function, called `FIFO_Put`, should enter one piece of data into the FIFO. The user will not attempt to put a data value of 65535 into the FIFO. If the FIFO is full at the time of the call, you will return a 65535. If the data is properly stored into the FIFO, then return a 0. **Hint:** if the last entry is not 65535, the FIFO is full. **Hint:** to find a place to store the data, search from the beginning to find the first entry with value 65535 and replace it with the data to be saved.

Part c) The third function, called `FIFO_Get`, should remove one piece of data from the FIFO. If the FIFO is empty at the time of the call, you will return a 65535. If the data is properly removed from the FIFO, then return the data (0 to 65534). **Hint:** if the first entry is 65535, the FIFO is empty. **Hint:** if the FIFO is not empty, the value at `FIFO[0]` is the oldest, which will be the value to return. You will have to move the remaining elements (as illustrated in the figure). The grader engine will disable interrupts while it calls your `FIFO_Get` because put and get have read-modify-write sequences to a shared global data, which produces a critical section. The critical section is removed by disabling interrupts.

After you have debugged your solution, comment out the `DisableInterrupts();` code in the grader engine and notice your FIFO no longer operates properly.

Print one piece of paper that contains some of your HW_FIFO C source code, your name, and the output results showing it runs properly.

Exercise 12.1: Go to [http://users.ece.utexas.edu/~valvano/Volume1/](http://users.ece.utexas.edu/~valvano/Volume1/) and download the grading engine. You can run the HW with the simulator or on your real board.

You will need to open a UART#1 window to see the score.
1) Unzip `HW_FixedPoint.zip`
2) Launch `HW_FixedPoint.uvproj` that starts Keil uVision
3) Build the object code `Target->Rebuild all target files`
4) Start the debugger and hit run (look in the UART#1 window)
My main program will call your functions multiple times, and will give your solution a performance score of 0 to 100. You should not modify my main program or my example data. When you have written your functions, you should run my main program, which will output the results to the UART.

Fixed-point is a means to represent noninteger values on the computer using integer operations. In this homework, all fixed-point numbers have the format
\[ \text{value} = \text{integer} \times 2^{-10}, \]
where integer is signed 32 bits.
Even though the range of values could vary from \(-2^{24}\) to \(+2^{24}\), we will restrict the range from -32 to +32. This is binary fixed-point because the fixed constant is a power of 2.

**Part a)** The first function, called **Add**, should add two binary fixed-point numbers. You will implement ceiling and floor, forcing the output to be between -32 and +32. E.g.,

\[
10.125 + 4.5 = 14.625
\]

You are not allowed to use any floating point operations. Your function will be called with the two integers 10368 and 4608. The integer 10368 represents the value 10.125, because 10368/1024 = 10.125. Similarly, the integer 4608 represents the value 4.5. The return value should be 14976, because 14976/1024 = 14.625. In general, if
\[
x = I \times 2^{-10}
\]
\[
y = J \times 2^{-10}
\]
\[
z = K \times 2^{-10}
\]
Let \(z = x+y\), and solve for \(K\) in terms of \(I\) and \(J\).

**Part b)** The second function, called **Mult**, should multiply two binary fixed-point numbers. You are not allowed to use any floating point operations. You will implement ceiling and floor, forcing the output to be between -32 and +32. For example

\[
5.125 \times 4.5 = 23.0625
\]

Your function will be called with the two integers 5248 and 4608. The return value should be 23616, because 23616/1024 = 23.0625. In general, if
\[
x = I \times 2^{-10}
\]
\[
y = J \times 2^{-10}
\]
\[
z = K \times 2^{-10}
\]
Let \(z = x\times y\), and solve for \(K\) in terms of \(I\) and \(J\).

**Part c)** Write a third function, called **Quad**, which returns the result of this quadratic equation (You are not allowed to use any floating point operations.)

\[
y = -0.9765625 \times x^2 + 1.220703125 \times x + 1.953125
\]

The input and output parameters are binary fixed-point numbers (integer value passed). The input values will be constrained to -1 to +1, so there should be no overflow.
Important Notes:
- Your functions should work for all cases shown in the starter file.
- Handle the simple cases first and the special cases last.

You are allowed to call your Add and Mult functions in part (c) but you are not required to do so.