CMPE1250 – ICA #7 Real Time Interrupt (RTI)

We have so far implemented some means of time intervals based on the delay cycles. By doing this, we have noticed that approach brings two difficulties: it changes depending on the BUS speed we are using, and it blocks the main code doing nothing.

As we have discussed in class, Interrupts provide with the ability of breaking from main program to perform some critical action in the ISR (Interrupt Service routine) and then come back to the same line, previously stored in the program counter (see notes on interrupts).

Interrupts can bring quite some complexity, especially when they are combined across different peripherals. As this is an introductory course, we will try to implement this the simplest way possible.

Use the *rti.h* header provided to implement the following functions:

- ***void RTI_Init(void)* –** Initializes the RTI to 1[ms] interrupt.
- ***void RTI_Delay_ms(unsigned int ms)* –** Implements a blocking delay in [ms] based on the RTI.

As you may have already noticed, the header contains this line:

```
//This has to be declared as a global variable in rti.c
extern volatile unsigned long rtiMasterCount;
```

The extern keyword in C "extends" the visibility of a variable, in this case ***rtiMasterCount.*** You will declare this variable as global in *rti.c*. If you were to try to use this variable in *main.c*, the compiler would say it cannot find it. The way of solving this problem is to let the compiler know such variable is declared somewhere else (*rti.c*). That would be done using the syntax shown above.

Lastly, to avoid typing this line in every program where we want to use this variable in *main.c*, we added it to *rti.h* so it would be automatically available when *rti.h* is included. Remember the #include clause adds the content of the library to the main compilation process.

You may also wonder, what is the ***volatile*** keyword doing. In simple terms, the volatile keyword is generally used for any global variable that is being modified in an ISR because it could change at any point.

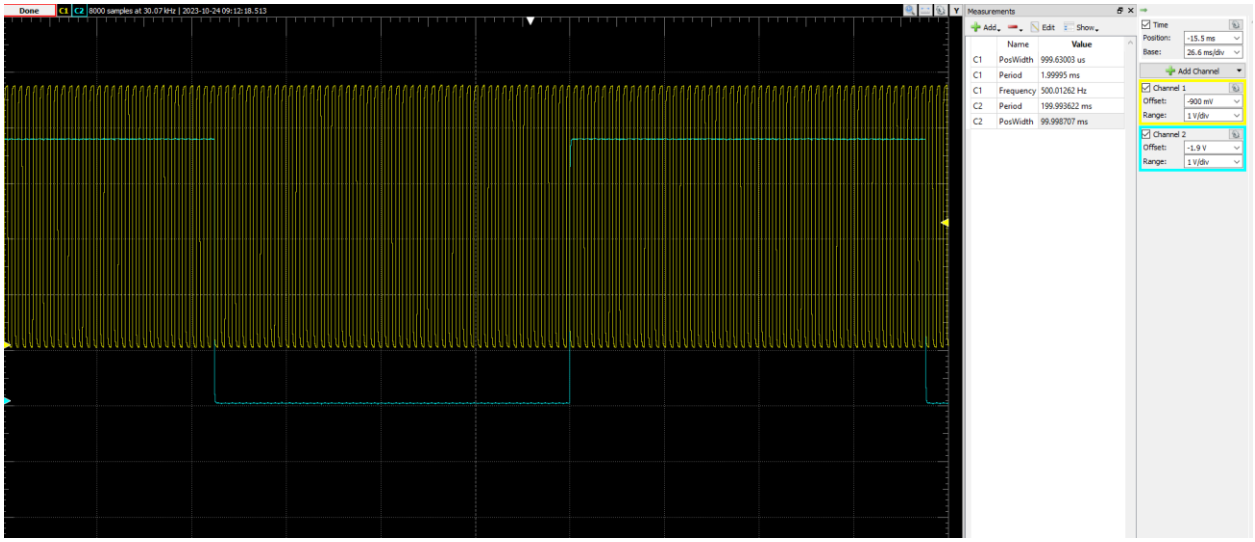
Part 1

Implement the function required to initialize the RTI to 1[ms] interrupt (***RTI_Init)***. Have the ISR coded in main, so we can add some statements to it.

As we discussed in class, the first line you need to have in the ISR is the one that clears the flag. Also, to have a means for counting in milliseconds, you will have to increment *rtiMasterCount* every time the ISR is called. Add a statement that toggles the RED LED in the ISR. Verify this functionality using an oscilloscope or the AD2 to verify the RED LED is toggling every 1[ms].

Part 2

Now it is time to implement and check the function that performs a blocking delay in milliseconds (***RTI_Delay_ms***). If you have done things correctly up to this point, you have noticed *rtiMasterCount* is incrementing every 1[ms] in the ISR and it can be used to implement a blocking delay in [ms]. To verify this function is working correctly, have a blocking delay of 100[ms] in main then toggle the GREEN LED. Scope both signals (red and green LEDs) and verify it looks like the following.



The timing should be very accurate and independent of the BUS speed (you might want to verify that!).

Part 3

Change the delay in the main loop to **1[s]** and add a functionality such that when the **LEFT SWITCH** is pressed, the **GREEN LED** goes **OFF** and the **1[s] toggling** happens with the **YELLOW LED**. Pressing the **RIGHT SWITCH** reverts the process to the original: **YELLOW LED OFF** and **GREEN LED toggling every 1[s]**.

Challenge

You may have already noticed that the responsiveness of the button press is not the best. That is because we are blocking the code for 1[s]. Implement a solution that fixes this problem but keeps the same functionality.

Hint: Alternative to a blocking delay?