

## Lab #8 - Timer and interruption

Leonardo Fusser (1946995)

### Objective:

- Configure a timer module and external CN interrupt.
- Write a program including a timer and a CN ISR
- Observe Timer and external CN interrupts in simulation mode
- Implement exception traps

---

**Hardware:** Explorer16/32 optional

### To hand in:

These sheets including all commented screenshots and answers to all questions on **Teams**

C code on **GitHub**:

- Indented and commented C code. Do not forget to comment on fcy and ISR frequency.
- The main.c file must include a prolog header: Name and lab number, a short description, author, date, version.

```

/*****
* FileName:      main.c
*
* Name:   Lab#    Timer and interruptions
*
* Description: To complete ...
*
* REVISION HISTORY:
* ~~~~~
* Author      Date              Comments on this revision
* ~~~~~
* MJ          June 15 2018      v1.0.0          To complete...
*
* ~~~~~
*/

```

- Relevant screenshots. Must include explanations.
- All functions must have a prolog header:

```

/*****
* Name: void initT3(void)
* Description: Timer3 initialization and configuration
*   Configure Timer3 to trigger an interrupt every 1mS.
*
*****

```

- An ISR is like a function, therefore should have a function-like prolog header.

```

/*****
* Name: void _ISR_T3Interrupt( void)
* Description: Interrupt Service Routine for Timer 3
*   Interrupts every 1mS
*****

```

**Table 1:** Template file content

Files	Content
<b>Files to populate</b>	
main.c	Initializes the resources used: IOs, timers, UART, etc. Executes the main super loop.
initBoard.c	Includes its own header file: initBoard.h Contain all functions related to initializing the board: oscillator, IOs, etc. Partially populated by the teacher.
Timer3.c	Contains timer3 ISR
CN.c	Contains Change Notification ISR
initBoard.h Timer3.h CN.h	Includes all dependencies, macros, function prototypes and structure definitions.
<b>Libraries</b>	
Tick_core.c Tick_core.h	Contains the complete core timer library service layer. Already populated by the teacher.
Console32.c Console32.h	Contains the complete UART/LCD driver layer. Already populated by the teacher.

## **Lab Work**

### **XC32 compiler:**

Install the XC32 compiler on your machine:

<https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-xc-compilers>

Install XC32 compiler in C:\Program Files\Microchip

Here are the detailed steps: <https://microchipdeveloper.com/xc32:installation>

### **GitHub:**

Make sure you are logged into GitHub. Copy-paste the following URL to accept an invitation for this lab:

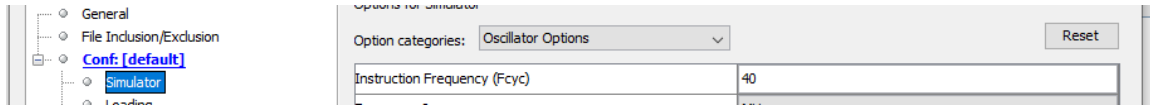
[https://classroom.github.com/a/79D07C\\_F](https://classroom.github.com/a/79D07C_F)

You will get a private new repository for this lab. You will push your project to this repo at the end of this lab.

[https://github.com/System-Programming-Vanier/lab\\_405\\_w21-yourname](https://github.com/System-Programming-Vanier/lab_405_w21-yourname)

## Part1: Timer 3 Module

In simulation mode, set Fcyc to 40 MHz (PBCLK):



In `main.c` create a function `initT3()` that sets all required timer3 registers to enable an ISR at a frequency of 400kHz.

You must set the prescaler to 1:1.

(Assume PBCLK at 40MHz)

Details of the calculation:

$$\text{Tick frequency} = \frac{40\text{MHz}}{1} = 40\text{MHz (1:1 prescale)}$$

$$\text{Tick period} = \frac{1}{40\text{MHz}} = 25\text{nS}$$

$$\text{Interrupt frequency} = 400\text{kHz}$$

$$\text{Interrupt period} = \frac{1}{400\text{kHz}} = 2.5\mu\text{S}$$

$$\text{Timeout} = 25\text{nS} * (x + 1) = 2.5\mu\text{S}$$

$$= \frac{25\text{nS} * (x + 1)}{25\text{nS}} = \frac{2.5\mu\text{S}}{25\text{nS}}$$

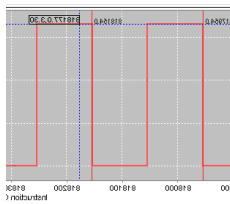
$$x + 1 = 100$$

$$x = 99 \text{ (value of PR3)}$$

Inside the ISR, make `LATA bit7` toggle on every call.

Do not forget to clear the ISR flag at the end of the interrupt.

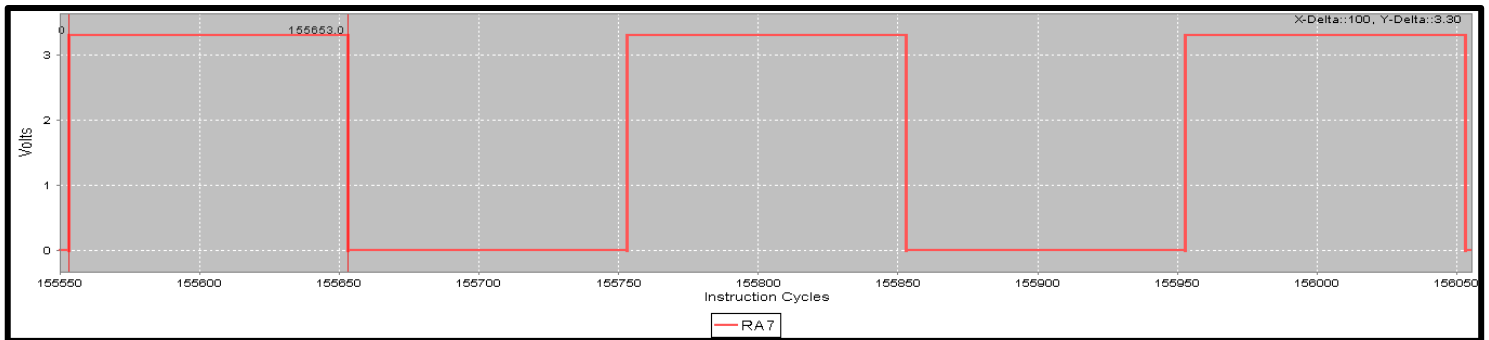
Run your code for a few seconds. Then pause and display pin `RA7` using the **logic analyzer** (Windows -> Simulator-> Logic Analyzer).



How many cycles are there between two toggles? Explain.

- Between two toggles, there are 200 ticks and only half for one toggle (100 ticks). When the program is running, it takes just as much time to run the ISR as the amount of time the ISR is not executed (100 ticks in ISR and 100 ticks outside of ISR). In other words, it takes 100 ticks before the Timer 3 module overflows and enters the ISR to toggle pin RA7. Screenshot below shows result from logic analyzer in MPLABX.

Take a screenshot of the result and explain the number of cycles between toggles.



Logic analyzer result shown above. As shown in the top right corner, it takes around 100 ticks to execute the code in the ISR (toggles pin RA7). It also takes about 100 ticks for the Timer 3 module to overflow and enter the ISR to toggle pin RA7.

Use **stop/watch** tool (Windows -> Debugging-> Stopwatch) to measure the ISR period and overhead (do not forget to set the simulator clock frequency - fcy).

Period: 100 ticks (2.5uS)

Overhead: 33 ticks (825nS)

Explain the ISR period/frequency:

- The ISR period/frequency is the amount of time that the ISR uses to execute whatever is within the ISR. As seen above, the amount of time that the ISR takes to execute its code is 2.5uS (100 ticks).

Explain the ISR overhead:

- The ISR overhead is the amount of time the PIC's CPU spends to handle anything associated with the ISR. It is associated with the CPU detecting that an interrupt has occurred, vectoring to the ISR, clearing any status flags and returning from the ISR. As seen above, the amount of time that this takes is 825nS (33 ticks). Ideally, we would want this to be a very small amount to avoid problems in our program.

### Profiling

Calculate the percentage of time the system executes the ISR:































$$ISR\% = \frac{Overhead}{Period} = \frac{825nS}{2.5uS} * 100 = 33\%$$

Calculate the percentage of time the system executes the main loop:

$$Main\ Loop\% = 100\% - 33\% = 67\%$$

## 247-405-VA Circuits and Embedded Systems

Open the Watches window (Windows -> Debugging-> Watches) and find the IFS0 register values in binary before the interrupt clears the flag and after the flag has been cleared.

Watches	Breakpoints	Variables	Output	Stopwatch	Debugger Console	Logic Analyzer	Call Stack
							
Name			Hexadecimal			Binary	
 IFS0	<input checked="" type="checkbox"/>		0x00000000			00000000 00000000 00000000 00000000	
 CTIF			0x00000000			0	
 CS0IF			0x00000000			0	
 CS1IF			0x00000000			0	
 INT0IF			0x00000000			0	
 T1IF			0x00000000			0	
 IC1IF			0x00000000			0	
 OC1IF			0x00000000			0	
 INT1IF			0x00000000			0	
 T2IF			0x00000000			0	
 IC2IF			0x00000000			0	
 OC2IF			0x00000000			0	
 INT2IF			0x00000000			0	
 T3IF			0x00000000			0	
 IC3IF			0x00000000			0	
 OC3IF			0x00000000			0	
 INT3IF			0x00000000			0	
 T4IF			0x00000000			0	
 IC4IF			0x00000000			0	
 OC4IF			0x00000000			0	
 INT4IF			0x00000000			0	
 TSIF			0x00000000			0	

Comment the results:

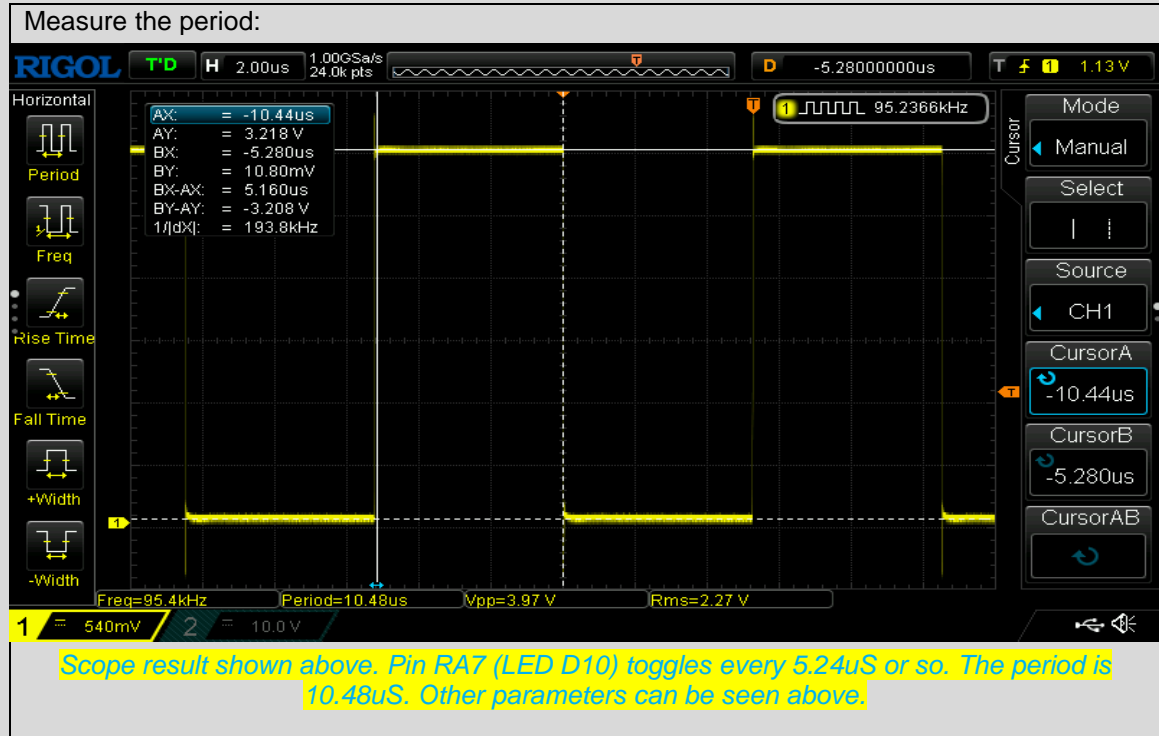
Before the ISR clears the flag: 0x01.

After the ISR clears the flag: 0x00.

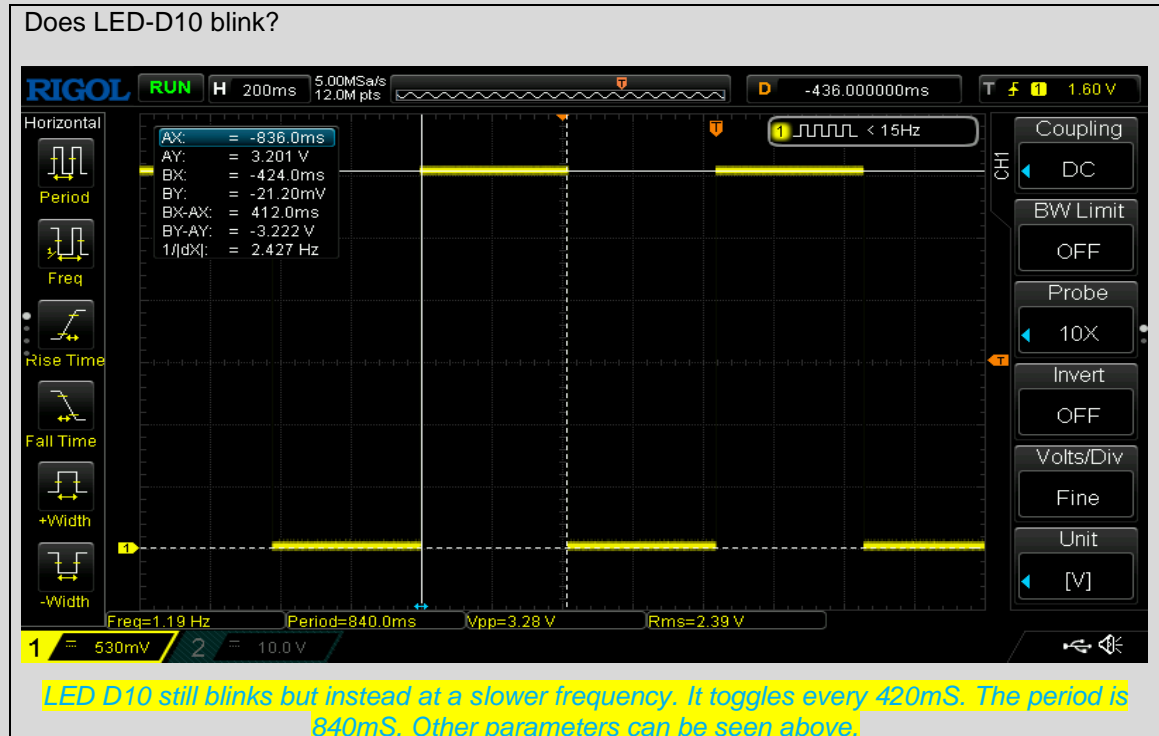
Basically, when leaving the ISR, the interrupt flag for the Timer 3 module (T3IF) is always cleared. The only time the interrupt flag for the Timer 3 module is set is when entering the ISR (when it is triggered), otherwise the PIC would not know when to enter the ISR and execute the code in it. On the other hand, if the interrupt flag is not cleared before leaving the ISR (it is always constantly set), the PIC would constantly be stuck in the ISR (like a forever loop). This is not what we want when programming an ISR. This behaviour is shown in the next part.

**On the Target Board**

Connect a scope probe to p92 at J46 connector.



For visual effect, set the pre-scaler to 256 and set PR3 to its maximum value.



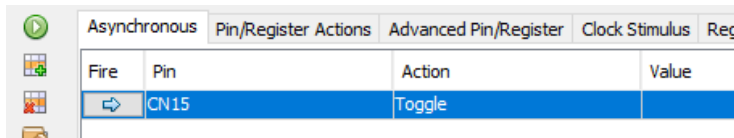
Disable Timer3 (by not calling the initializing function!)

### Part 2: CN external interrupt

Write a function `initCN15()` that enables CN15 interrupt.

Also write the CN ISR. Make `LATA bit3` toggle on every call of CN ISR.

Test CN ISR by opening the stimulus window (Windows -> Simulator-> Stimulus) and stimulate pin CN15, in toggle mode.



Click on the Fire button to trigger the CN interrupt:

Explain the effect:

- When the fire button is triggered, the PIC will halt whatever it is doing, and it will enter the CN ISR. When in the ISR, pin RA3 will be toggled. The PIC will resume normal operation when the fire button is pressed again (will return to main while loop) and will clear the corresponding interrupt flag (CNIF - Change Notification Interrupt Flag) before leaving the ISR so that the PIC will not remain in the ISR.

### On the Target Board

Press PB S3.

What happens to the LED D10 when you press S3?

- Upon every time the push-button S3 is pressed, the state of LED D10 changes. Initially, the state of LED D10 is off and after every press of push-button S3, the state is toggled (from off to on and on to off and so on).

Now change your ISR code so the CNIF is not reset anymore. Then Fire CN15 again.

Explain the effect:

- When the CNIF (Change Notification Interrupt Flag) is not set to clear anymore upon leaving the CN ISR, the PIC will never leave the CN ISR once the fire button is pressed for the first time. The state of pin RA3 will remain the same after triggering the ISR for the first time. All other attempts thereafter will be meaningless as the PIC will be constantly stuck in the CN ISR. The only way the PIC will know when to resume normal operation is if the interrupt flag is reset to 0.

### Part3: Traps and exceptions

In your main(), declare a type int variable and then divide it by zero.  
Reset the board and run the code.

Explain the behavior:

```
main.c: In function 'main':
main.c:40:24: warning: division by zero [-Wdiv-by-zero]
    variable = variable/0;
```

*Warning in MPLABX (right after compiling code) shown above.*

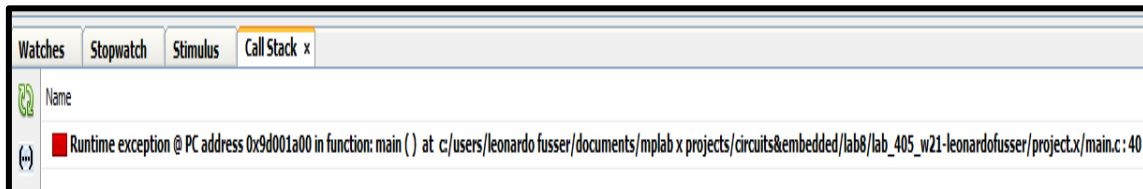
```
PIC32 runtime exception detected. Use Call Stack window to resolve source line location.
Launching
Initializing simulator
User program running
User program stopped
```

*Error in MPLABX (during runtime) shown above.*

- What happens is basically what is shown in the two screenshots above. The user can make the mathematical mistake by dividing by 0 in the code, but the first sign of problems occur when the user compiles the code. As seen above in the first screenshot, the compiler gives a warning about the issue, but the user can still proceed and try to run the code. When this occurs, the program will almost immediately lock up and the program will halt. This is seen above in the second screenshot where the compiler throws a “runtime exception” error. Although we know where the problem is occurring in the code, the exact location of where the error occurs can be seen in the call stack window below (assuming you do not know where it is in the code).

Open the Call Stack window to find out the Runtime exception (Window -> Debugging -> Call Stack)

Explain the message provided by the Call Stack Window:



*Runtime exception mentioned above shown in call stack window above.*

- The source of the problem mentioned in the previous question above can be viewed in the call stack window. As shown above, the exact location of the error can be found at the PC (program counter) address and more details about the location follow it.

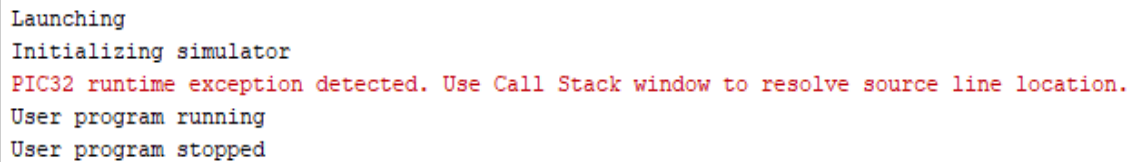


**Do not divide by zero** anymore.

Now declare an array of two bytes and write to a wrong location:

```
int my_array[2];  
my_array[10000]=1; // writes to a wrong location - buffer overflow
```

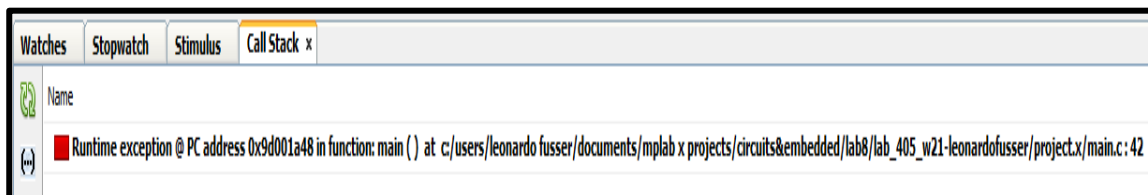
Explain the behavior:



```
Launching  
Initializing simulator  
PIC32 runtime exception detected. Use Call Stack window to resolve source line location.  
User program running  
User program stopped
```

*Error in MPLABX (during runtime) shown above. Similar to previous page.*

Explain the message provided by the Call Stack Window:



*Runtime exception shown in call stack window above. Similar to previous page.*

- Similar to the previous question above, since the program halts (because of another ALU error – writing to an illegal location in array), a runtime exception error is thrown (as seen in first screenshot above). The source of the problem can be found by looking at the call stack window. As seen in the above second screenshot, the location of the error can be found by looking at the PC address, the location of the file (in this case main.c) and the line in the code (42 in this case). Theoretically, a programmer can take this information and use it to solve the runtime exception error and prevent it from occurring again.

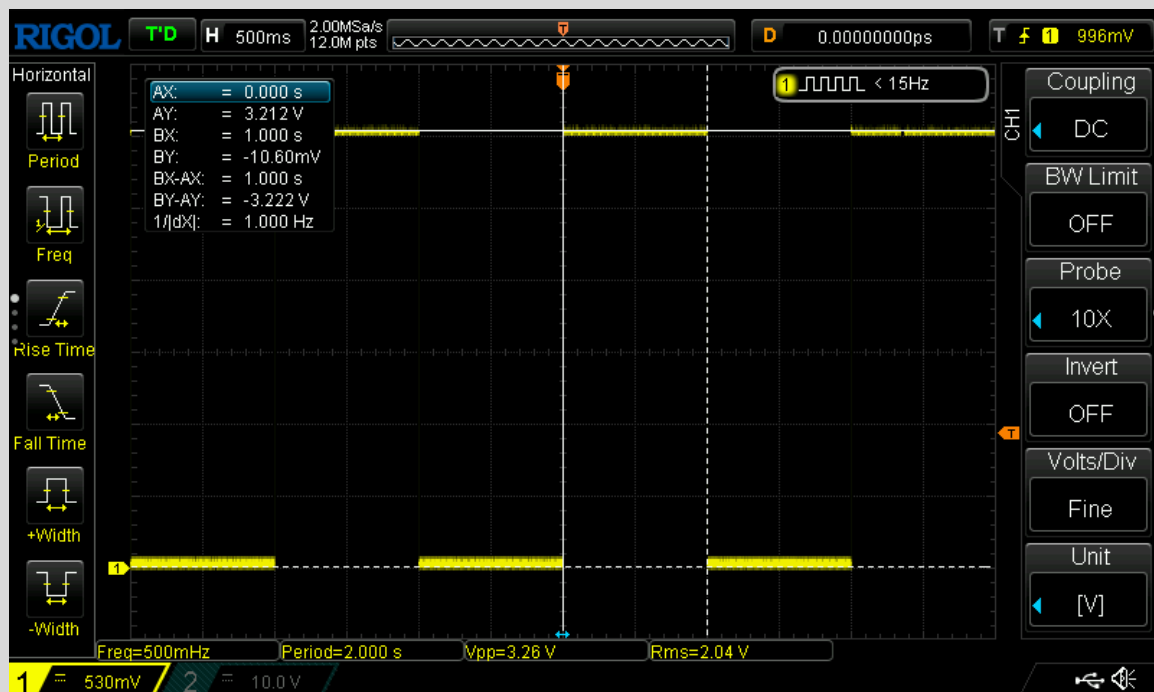
**On the Target Board**

In **release mode** at the top of the main and before the error occurs, turn on/off LED D10 (port RA7) for 1 second. Use `delay_us()`

```
main() {
    initIO();
    LATAbits.LATA7 =1;    //ON
    delay_us(1000000);
    LATAbits.LATA7 =0;    //OFF
    delay_us(1000000);
    ...
}
```

What happens to the LED D10? Explain the behavior:

- What happens with LED D10 is that it will constantly blink on and off. This is because when the processor encounters the errors that have been shown above in the previous questions (divide by 0 or writing to illegal location), it will constantly reset and attempt to recover from the beginning of the program. Since the LED toggle (for LED D10 on 16/32 board) is programmed before the error is supposed to occur, it should only toggle once but since the processor constantly resets itself (when it reaches the point when the error occurs), the code for the LED to toggle will always execute no matter what. Thus, it feels like it is done on purpose but in our case, it signals us a serious problem that needs to be addressed. The screenshot below shows the behaviour of the LED toggle.



*Toggle behaviour from LED D10 on 16/32 board shown above. Screenshot above shows that the LED is on for 1 second and off for 1 second. The period is 2 seconds, and the frequency is 500mHz. Other parameters can be seen above.*

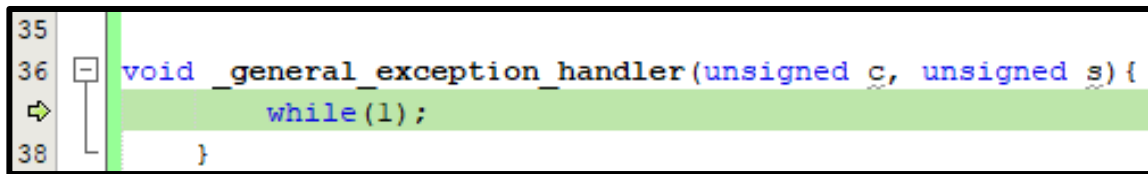
**Trap**

In the main implement the following exception handler trap:

```
void _general_exception_handler( unsigned c, unsigned s)
{
    while (1);
} // exception handler
```

Again, reset the board and run the code.

Explain the behavior:

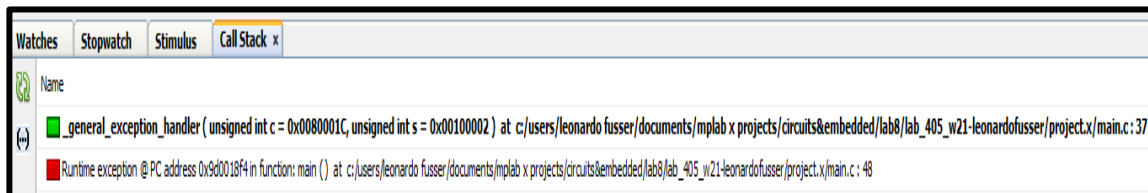


*Result of having exception handler implemented in code shown above.*

Does the processor still keep stopping and resetting?

- With the exception handler implemented in the code, the processor stops continuously resetting and halting. The result is that when an error is to occur, instead of doing what was seen before in previous questions, the processor will instead be stuck in the while loop in the exception handler (as shown in green in the above screenshot).

Explain the message provided by the Call Stack Window:



*Runtime exception and result of it shown in call stack window above.*

- Almost similar to what was shown in previous call stack windows, an additional message is present (shown as green square in above screenshot). From what was said about previous call stack windows, the same applies here, but this time around the program is not halted immediately and instead keeps running by staying stuck in the exception handler. This of course only happens when an error (division by 0 or writing to illegal location) occurs. The location of the exception handler in the code is shown in the above screenshot as well (line 37 in this case).

**On the Target Board**

In release mode, what happens to the LED D10? Explain the behavior:

- When in release mode this time around, LED D10 only toggles one time because the processor does not continuously reset and start from the beginning of the program (this was predicted in the previous grey question). When the program runs for the first time, only one toggle will occur and when the processor encounters an error, it will then jump to the exception handler to remain forever there until the user quits or restarts the program. The LED does not toggle in the exception handler when the processor is stuck there because there is no code there to handle the toggling behaviour.

Commit **only** the modified files and then push your code to GitHub.

You must provide a prologue header with a version number.

Approval before leaving!

**Check list:**

- ☐ Indented code.
- ☐ Commented code – function and file prolog headers.

Versioning

**After lab questions:**

1- Explain what you learned in this lab. What went wrong and what did you learn from your mistakes.

- There were multiple errors that occurred in this lab, most of them were minor (syntax mistakes) but a few were major. Some of the major ones were dealing with the IO configuration for PIC32. For example, some initialization was missing, and I was not able to see some waveforms on my scope display. After this, I learned to check and verify all configuration that needs to be done before the program should run. Another problem that was encountered was when the exception handler for handling processor errors was not working as expected (processor kept resetting); problem was that the exception handler was put in the wrong place in the main.c file. After this, I learned to thoroughly test different scenarios for errors with the processor to see how it will handle it (to make sure that the exception handler was doing what it was supposed to do).

2- Name internal interruption(s) used in this lab.

- An internal interruption that was used in this lab was waiting for the Timer 3 module to overflow/roll-over on PIC32.

Name external interruption(s) used in this lab.

- An external interruption that was used in this lab was waiting for a logic state change on pin CN15 (change notification interrupt for pin CN15 on PIC32).

3- What is the value of implementing a trap? Explain.

- The value of implementing a trap is that it avoids serious problems from occurring, some of which we saw in the previous questions. It is a more controlled way for processors to handle errors instead of them just resetting and halting all together; this is very inefficient because we cannot tell what is causing the problem to occur. Implementing traps also allows for programmers to include error-handling techniques in their code so that they can narrow down errors when they occur.