Lab #1 - Frequency and speed measurement

Leonardo Fusser (1946995)

WARNING Lack of diligence may result in board damage. Students are financially liable for any damage to the board.

Objective:

- Write an interrupt-driven program.
- Program a Change Notification (CN) pin.
- Measure a frequency in Hz using an interrupt driven method.
- Measure a motor speed in RPS and RPM.
- Use MPLABX simulator and stimulus.

Hardware: 1 motor-encoder-HBridge, Explorer16/32, jump wires, Oscilloscope and probe, handheld optical tachometer.

Motor characteristics:

12V brushed DC motor

19:1 metal gearbox ratio

Integrated quadrature encoder: 64 CPR

To hand in:
These sheets including answers to all questions on **Teams**.

C code on **GitHub**:

i. Must be indented and commented properly.
ii. Function prolog header must be included.
iii. Abstract all configuration related code by creating functions: e.g., initT2() .
iv. The use of macros and pre-processor directive (#ifdef) is now mandatory.
v. main.c prolog header: Name and lab number, a short description, author, date, and version.
vi. You must keep a version history of your project.

Example of history:

```
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Author          Date              Comments on this revision
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Serge Hould     August 22 2016    First version of source file, tested on the target    - v1
* Serge Hould     October 29 2016   Add LUT tables                                          -v1.1
* Serge Hould     October 30 2016   Multiple files system created                           -v1.2
```

Use representative name for variable:
E.g., `sec` rather than `s`

Use of macro:
E.g., `#define   FCY   80000000`

All private variables must be accessible through public functions only. This way, global broadcast is replaced by public setter-getter functions.
Example:

Isr_driven.c	**Main.c**

Global variable:
`signed int64_t period;`
 	
Must be replaced by a local variable
`static signed int64_t period;`
 	
Definition of the public protected getter to access the variable:
`signed int64_t getPeriod(void){`
`  signed int64_t temp;`
`  Disable interrupt`
`  temp = period;`
`  Enable interrupt`
`  return temp;`
`}`
	Accessed using to extern keyword:
	`extern signed int64_t period;`
	
	Is now accessed through a public getter:
	`static signed int64_t local_period;`
	`local_period = getPeriod();`

Table 1: Template file content

Files	**Content**
`labx/generator.c`	Contains all function implementations to generate a square wave. Already populated be the teacher.
`labx/initBoard.c`	Includes its own header file: `initBoard.h` Contain all functions related to initializing the board: oscillator, timers, IOs, etc. Partially populated be the teacher.
`labx/isr_driven.c`	Contains the bulk of the solution. Contains ISR driven code and function implementations to measure the frequency of the input signal. Must encapsulate as much as possible.

`labx/isr_driven.h`
`labx/initBoard.h`
`labx/generator.h`
	Includes all dependencies, macros, function prototypes and structure definitions.

`labx/main.c`	Initializes the resources used: IOs, timers, tick lib, generator, LCD, etc. Executes the main super loop at a specific rate to display speed and frequency on the LCD.
`labx/Tick_core.c`	Contains the complete Tick library service layer. Already populated be the teacher.
`labx/console32.c`	Contains the complete LCD driver layer. Already populated be the teacher.

Use the following function for simulation purpose, see `console32.c` file:

```
Function: int  fprintf2(int mode, char *buffer)

   Precondition:
      initUartx must be called prior to calling this routine.

   Overview:
      This function prints a string of characters to the selected
      console. Possible choices are _UART1, _UART2 and _LCD.

   Input:
      mode: select the console C_UART1, C_UART2 or C_LCD
      buffer: Pointer to a character string to be outputted

   Output: returns the number of characters transmitted to the console

Function: void initUartx( void)
    Overview: this function initializes the UARTx serial port.
```

Example to print to LCD in target mode and to UART2 in simulation mode:

```
#define SIMULATION

#ifdef SIMULATION
    #define CONSOLE     C_UART2
#else
    #define CONSOLE     C_LCD

#endif
initUart1();      // initializes simulation UART1 display tab
initLCD();        // initializes target LCD display

fprintf2(CONSOLE,"debugging message 1\n");
```

Lab organisation:

The first part involves some setups and calculations for the project.

The second part is entirely simulated by an internal square wave thanks to the stimulus tool.

The third part is executed on the target. It measures the frequency of a processor generated square wave frequency.

The fourth part is executed on the real motor.

Lab Work

Make sure you are logged into GitHub. Copy-paste the following URL to accept an invitation for Lab1:

https://classroom.github.com/a/czcEtHy1

You will get a private new repository for Lab1:

https://github.com/advanced-programming/lab1_511_a21-yourName

Part 1: Initialisation functions

Modify the following snippets to:

- Initialize Timer2 module: 1:4 pre-scaler and maximum period value.

```
void initT2(void)
{
      //[Initialize and enable Timer2]
      T2CONbits.TON = 0;        //Disable Timer2.
      T2CONbits.TCS = 0;        //Select internal inst. cycle clock.
      T2CONbits.TGATE = 0;      //Disable Gated Timer mode.
      T2CONbits.TCKPS = 0b111; //Prescale.
      TMR2 = 0x00;              //Clear Timer2 register.
      PR2=0x10;                 //Load the period value.
      IPC2bits.T2IP = 0x01;     //Set Timer 2 Inter Priority.
      IFS0bits.T2IF = 0;        //Clear Timer 2 Interrupt Flag.
      IEC0bits.T2IE = 1;        //Enable Timer 2 interrupt.
      T2CONbits.TON = 1;        //Start Timer.

} //init
```

- Implement CN ISR (CN9 on pin p11).

```
void initCN9 (void){
    TRISGbits.TRISG7 =1;               //Input CN9/RG7 pin 11.
    CNENbits.CNEN9 = 1;                //CN9.
    __builtin_disable_interrupts();    //Disable interrupts.
    CNCONbits.ON = 1;                  //Turn on CN.
    IPC6bits.CNIP=2;
    IFS1bits.CNIF=0;
    INTCONbits.MVEC=1;
    IEC1bits.CNIE=1;
    __builtin_enable_interrupts();     //Enable Interrupts.
}
```

- Implement Timer2 ISR:

```
void __ISR( _TIMER_2_VECTOR, IPL1SOFT) T2InterruptHandler( void){
}
```

Calculate Timer2 interrupt frequency if PR2 is set to its maximum value:

Assume sysclk = 80 MHz

$Clock\ tick\ frequency\ (PBCLK) = 80MHz$

$Clock\ tick\ time = \frac{1}{80MHz} = 12nS$

$Since\ we\ are\ using\ Timer2\ module, maximum\ value\ that\ can\ be\ stored\ in\ PR2\ will\ be\ 65'536\ (2^{16}).$

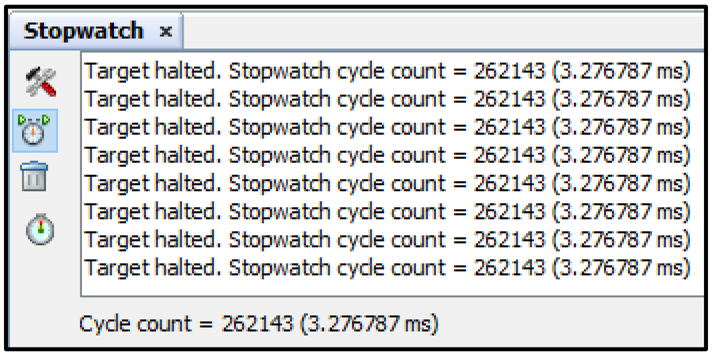
$Timer2\ Interrupt\ period = (PR2 + 1) * PRE * Tick\ time = 65'537 * 4 * 12nS = 3.15mS$

$Timer2\ Interrupt\ frequency = \frac{1}{Timer2\ Interrupt\ period} = \frac{1}{3.15mS} = 317.46Hz$

Set a breakpoint at the beginning of the Timer ISR and using the stopwatch, measure the ISR period and frequency:

Does it match with the calculation?

When running the simulator and setting the breakpoint at the beginning of the Timer2 interrupt (in isr_driven.c), the simulator stops at around every 3.27mS (around every 305.81Hz). This occurs when the Timer2 overflows and the T2IF flag is set, which causes the microcontroller to halt what it is doing and forces it to serve the interrupt. The microcontroller will then serve the interrupt and clears the T2IF flag (prevents ISR trapping) once it's complete with the interrupt, it will resume normal operation. This process repeats forever. The results match with the ISR calculation done above. See screenshot below.



```
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)
Target halted. Stopwatch cycle count = 262143 (3.276787 ms)

Cycle count = 262143 (3.276787 ms)
```

Screenshot taken from MPLAB X IDE. After running the simulator, it stops at around every 3.27mS (around every 305Hz) at the user-set breakpoint (located in beginning of Timer2 interrupt service in isr_driven.c).

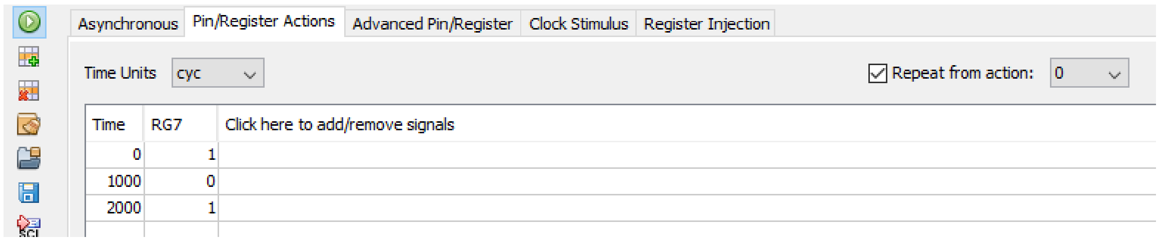
Part 2: Measure a frequency in Hz (stimulus tool generated square wave)

Don't forget to set the instruction frequency of the simulator at 80 MHz.

Now, you need to develop an interrupt-driven program to measure the frequency of an emulated square wave generator.

To emulate a 100 Hz square wave, in simulator mode, enable the stimulus to trigger pin RG7 at a 100 Hz frequency. Use the PIN/Register Actions tab.

The example below emulates a square wave at pin RG7 with a period of 2000 instruction cycles (1000 cycles ON and 1000 cycles OFF):



Time Units: cyc

Repeat from action: 0 (checked)

Time	RG7
0	1
1000	0
2000	1

Calculate the period in instruction cycles to generate a 100 Hz square wave knowing that the instruction frequency of the simulator is set at 80 MHz:

$$Square\ wave\ period = \frac{1}{100Hz} = 10mS$$

Knowing that after every 80MHz (80 million ticks) 1-second elapses, we can solve the following situation: How many ticks elapse after 10 milliseconds?

$$\frac{80MHz}{x} = \frac{1Sec.}{10mS}$$
$$\frac{1x}{x} = \frac{800'000}{x}$$
$$x = 800'000\ ticks\ (square\ wave\ period)$$

$$T_{ON} = 400'000\ ticks\ (50\%\ D.C.)$$
$$T_{OFF} = 400'000\ ticks\ (50\%\ D.C.)$$

This result shows that it takes a total of 800'000 ticks to generate the 100Hz square wave when the instruction clock is set at 80MHz in MPLAB X IDE simulator.

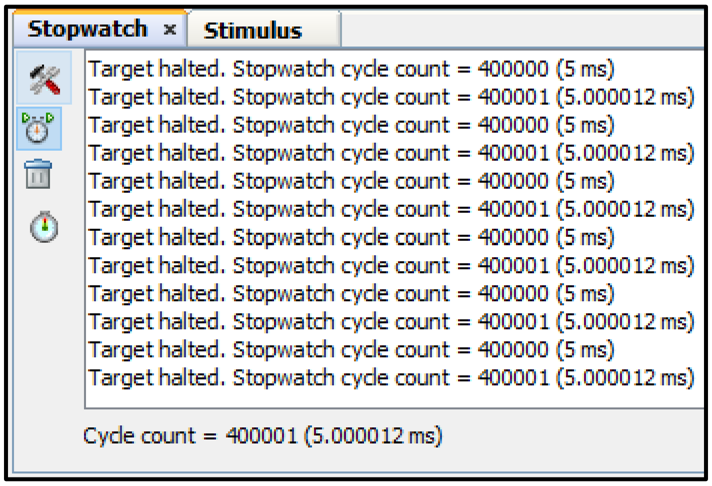
Apply the stimulus and set a breakpoint at the beginning of the external interrupt ISR - Click on the green triangle to apply the stimulus.

Comment on the result. Does it interrupt if you apply the stimulus?

When running the simulator and setting the breakpoint at the beginning of the CN9 (pin RG7 -> 11) interrupt service routine (in isr_driven.c), the simulator stops at unique time intervals when the stimulus is applied. This is because after every change of state (on RG7 -> where the 100Hz square connects to), the CNIF flag is set, causing the microcontroller to halt whatever its doing and forces it to serve the CN9 interrupt. Upon leaving the CN9 interrupt, the CNIF flag is cleared to prevent any unnecessary trapping within the ISR and the microcontroller resumes normal operation. This process repeats after each change of state that is detected on RG7.

Using a stopwatch, give the period of the interrupt in cycles:

The simulator stops at around every 5mS (around every 400'000 ticks) when the stimulus is applied. This matches with the CN9 calculation done above. See screenshot below.



```
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)
Target halted. Stopwatch cycle count = 400000 (5 ms)
Target halted. Stopwatch cycle count = 400001 (5.000012 ms)

Cycle count = 400001 (5.000012 ms)
```

Screenshot taken from MPLAB X IDE. After the stimulus is applied, the simulator stops at around every 5mS (around every 400'000 ticks) at the user-set breakpoint (beginning of CN9 interrupt service in isr_driven.c).

Stop applying the stimulus.

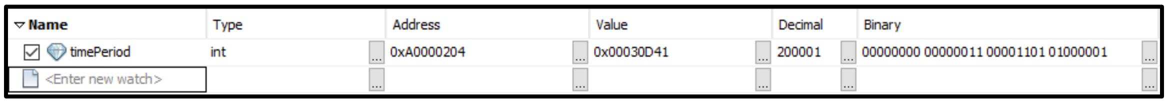
Comment on the result. Does it interrupt if you don't apply the stimulus?

When the stimulus is removed, the simulator no longer stops at unique time intervals like it did previously. This is because there are no longer any changes of state being detected on RG7, which doesn't cause the CNIF flag to be set anymore and in turn, does not force the microcontroller to execute the CN9 ISR (only gets executed when the CNIF flag is set). When the CN9 ISR is not executed, the microcontroller will be handling other interrupts or will stay stuck in the forever loop in main.c (super loop). In this case, the microcontroller is handling the Timer2 interrupts (configured previously) and once completed, stays looping forever in the super loop (while loop in main.c). This behaviour lasts forever when the stimulus is no longer applied.

Write some code inside the ISR to measure the period in ticks. That period must be stored in a variable.

Period value measured (in ticks)

The period value that is measured (in ticks) is 200'000. See screenshot and calculations below.



Name	Type	Address	Value	Decimal	Binary
timePeriod	int	0xA0000204	0x00030D41	200001	00000000 00000011 00001101 01000001
<Enter new watch>					

Screenshot taken from MPLAB X IDE. Using the "watches" feature in MPLAB X IDE simulator, we can see the value stored in "timePeriod" (variable that stores the measured value of the period in ticks). Value stored in "timePeriod" for 100Hz square wave stimulus is 200'001 (ticks) .

$$Square\ wave\ period = \frac{1}{100Hz} = 10mS$$

$$Clock\ tick\ frequency\ (PBCLK) = 20MHz$$

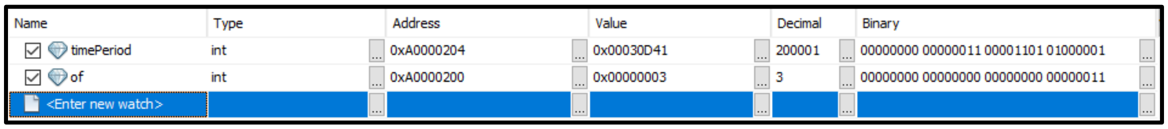
$$Clock\ tick\ time = \frac{1}{Clock\ tick\ frequency\ (PBCLK)} = \frac{1}{20MHz} = 50nS\ per\ tick\ (for\ 20MHz\ PBCLK)$$

$Calculated\ waveform\ period\ (in\ time) = Clock\ tick\ time * Measured\ period\ (in\ ticks) = 50nS * 200'000 = 10mS$

$Waveform\ period\ (in\ time) = 10mS\ (see\ other\ calculations\ above)$

Does Timer 2 overflow? If it does, say how many times it overflows for each external interrupt?

Timer2 overflows for a total of 3 times each time the CN9 interrupt is called. See screenshot and calculations below.



Name	Type	Address	Value	Decimal	Binary
☑ timePeriod	int	0xA0000204	0x00030D41	200001	00000000 00000011 00001101 01000001
☑ of	int	0xA0000200	0x00000003	3	00000000 00000000 00000000 00000011
<Enter new watch>					

Screenshot taken from MPLAB X IDE. Using the "watches" feature in MPLAB X IDE simulator, we can see the value stored in "timePeriod" (variable that stores the measured value of the period in ticks) and in "of" (variable that stores the counted value of Timer2 overflows). Value stored in "timePeriod" for 100Hz square wave is 200'001 (ticks) and value stored in "of" is 3.

$Number\ of\ ticks\ needed\ to\ trigger\ Timer2\ interrupt = 65'537\ ticks\ (configured\ previously)$
$Number\ of\ ticks\ needed\ to\ trigger\ CN9\ interrupt = 200'000\ ticks\ (configured\ previously)$

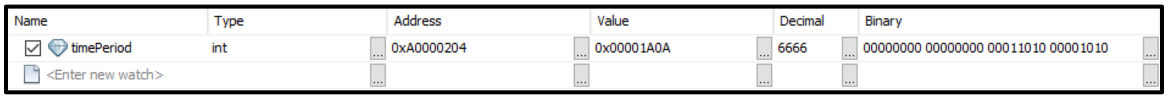
From these values, we can see that it takes longer before the CN9 interrupt will be called, so we know that the Timer2 interrupt will be called multiple times before the CN9 interrupt gets executed. We can find the exact number of times the Timer2 will be called before the CN9 interrupt gets called by doing the following calculation:

$$Number\ of\ times\ Timer2\ overflows\ before\ CN9\ interrupt\ gets\ called = \frac{Number\ of\ ticks\ needed\ to\ trigger\ CN9\ interrupt}{Number\ of\ ticks\ needed\ to\ trigger\ Timer2\ interrupt}$$
$$= \frac{200'000}{65'537} = \sim 3\ times$$

Change the square wave frequency to 3000Hz and do the previous step again:

Period value measured (in ticks)

The period value measured (in ticks) is 6'666. See screenshots and calculations below.



Name	Type	Address	Value	Decimal	Binary
☑ timePeriod	int	0xA0000204	0x00001A0A	6666	00000000 00000000 00011010 00001010
<Enter new watch>					

Screenshot taken from MPLAB X IDE. Using the "watches" feature in MPLAB X IDE simulator, we can see the value stored in "timePeriod" (variable that stores the measured value of the period in ticks). Value stored in "timePeriod" for 3kHz square wave is 6'666 (ticks) .

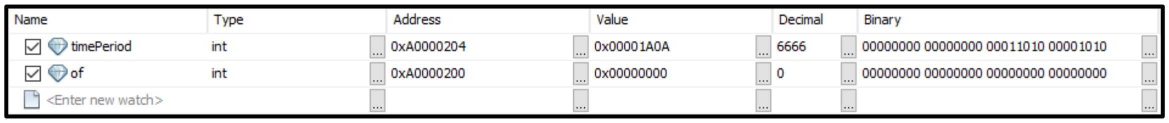
$$Square\ wave\ period = \frac{1}{3000Hz} = 333.33uS$$

$$Clock\ tick\ frequency\ (PBCLK) = 20MHz$$
$$Clock\ tick\ time = \frac{1}{Clock\ tick\ frequency\ (PBCLK)} = \frac{1}{20MHz} = 50nS\ per\ tick\ (for\ 20MHz\ PBCLK)$$

$$Calculated\ waveform\ period\ (in\ time) = Clock\ tick\ time * Measured\ period\ (in\ ticks) = 50nS * 6'666 = 333.33uS$$
$$Waveform\ period\ (in\ time) = 333.33uS\ (see\ other\ calculations\ above)$$

Does Timer 2 overflow? If it does, say how many times it overflows for each external interrupt?

Unlike when the stimulus was configured with a 100Hz square wave, Timer2 does not overflow anymore. See screenshot and calculations below.



Name	Type	Address	Value	Decimal	Binary
☑ timePeriod	int	0xA0000204	0x00001A0A	6666	00000000 00000000 00011010 00001010
☑ of	int	0xA0000200	0x00000000	0	00000000 00000000 00000000 00000000
<Enter new watch>					

Screenshot taken from MPLAB X IDE. Using the “watches” feature in MPLAB X IDE simulator, we can see the value stored in “timePeriod” (variable that stores the measured value of the period in ticks) and in “of” (variable that stores the counted value of Timer2 overflows). Value stored in “timePeriod” for 3kHz square wave is 6’666 (ticks) and value stored in “of” is 0.

$Number\ of\ ticks\ needed\ to\ trigger\ Timer2\ interrupt = 65'535\ ticks\ (configured\ above)$
$Number\ of\ ticks\ needed\ to\ trigger\ CN9\ interrupt = 6'666\ ticks\ (configured\ above)$

From these values, we can obviously see that the opposite is occurring from before when the stimulus was configured with a 100Hz square wave. Here, the CN9 interrupt will get called faster than the Timer2 interrupt. Therefore, Timer2 will not overflow before the CN9 interrupt anymore. We can also prove this using this calculation:

$$Number\ of\ times\ Timer2\ overflows\ before\ CN9\ interrupt\ gets\ called = \frac{Number\ of\ ticks\ needed\ to\ trigger\ CN9\ interrupt}{Number\ of\ ticks\ needed\ to\ trigger\ Timer2\ interrupt}$$
$$= \frac{6'666}{65'535} = 0\ times$$

Explain the result of the previous steps in term of measured period (in ticks) and overflow.

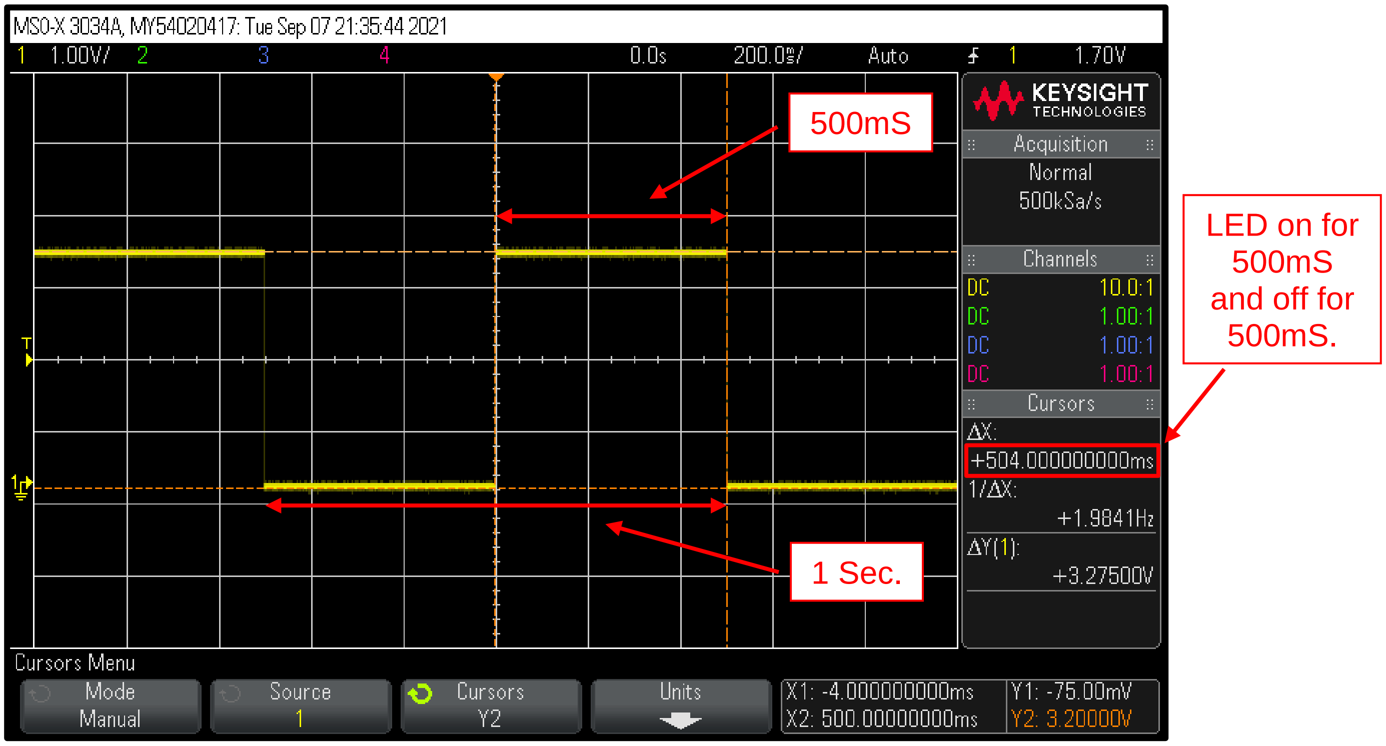
See above.

Any discrepancy with theory? Explain why?

The discrepancy with this theory is that we assume that Timer2 will always overflow faster than the rate of CN9 overflows. In reality, it’s possible that the opposite occurs, and this is what we see above. So, we cannot assume always that Timer2 will overflow faster than CN9 overflow rate.

In the main loop, write a snippet to calculate and display to the UART console the frequency in Hz. It must update every 500mS (use tick_core library).

Also, setup a LED heartbeat at the same 500mS period.



Screenshot taken from oscilloscope in lab. Probe is connected to LED (LED3 is used on Explorer 16/32 board. LED3 is connected to RA0 on PIC32 -> pin 17) and result shows that LED is blinking on and off at 1Hz frequency (500mS on and 500mS off).

Note: in simulator, all delays must be shorter (3 to 8 times shorter depending on your computer).

Hint: Use the SIMULATION macro mentioned above to control the delay.

Improvements

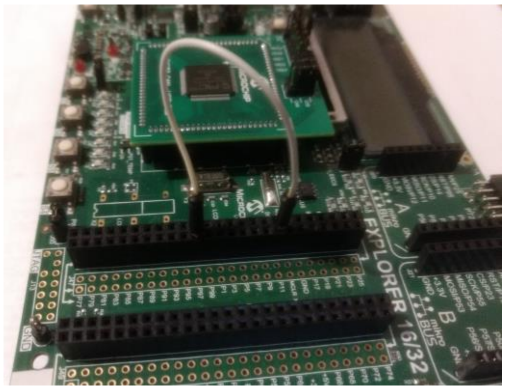
Add some code so that if the function generator is disconnected for more than 3 seconds, the frequency is set to 0 Hz and the LED stops blinking.

Hint: The ISR is not called if the motor is stopped.

Part 3: Motor speed measurement (processor generated square wave) – Target mode

Without even connecting the real motor, you will enable the internal generator module in order to generate a square wave on the target board.
The goal is to measure the square wave characteristics and display it on the LCD.

WARNING. DO NOT turn on power while wiring. The teacher must approve the wiring.



The internal generator output is at pin 97. Interconnect the generator output to the CN9 pin.

Display the motor speed in RPM on the LCD.

Hint: look at the motor characteristics.

Test the operation of your program by setting the speed to two different values: 3kHz and 100Hz. Measure and compare.

Generator at 100Hz

PIC measure value (Hz): measured value is 99Hz. Very close to configured value. Very accurate result.

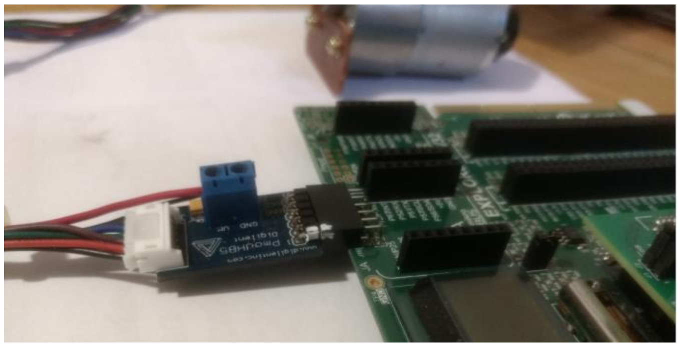
Generator at 3kHz

PIC measure value (Hz): measured value is 2’999Hz. Very close to configured value. Very accurate result.

Part 4: Motor speed measurement (external motor) – Target mode

In this last part, you are going to measure the speed of a real motor. You must factor in the gearbox and the encoder's count per revolution.

WARNING. DO NOT turn on power while connecting. The teacher must approve the connection.



You need to tweak your program to measure the motor speed in RPM.

1. Turn off the power.
2. Remove the jumper wire from the previous part.
3. Connect the pmod connector to JA top row (see picture above).
4. To enable the motor in your code:
   ```
   LATGbits.LATG8 =1;  //Enables the motor at full speed.
   ```

IMPORTANT: To avoid any risk of damage to the encoder, the teacher must approve before powering on.

Display the motor side speed in RPM and the gearbox side speed in RPM on two different lines of the LCD.

Hint: look at the motor characteristics.

Test the operation of your program by measuring and comparing with a Tachometer:

Motor speed

Tachometer measure value (RPM): measured value is ~9’250RPM.

PIC measure value (RPM): measured value is ~9’300RPM.

Gearbox speed

Tachometer measure value (RPM): measured value is ~460RPM.

PIC measure value (RPM): measured value is ~470RPM.

Test the operation of your program by gently slowing down the motor on the encoder side.

Does it display the new speed?

When the motor slows down, the LCD on the Explorer 16/32 board displays the new speed correctly (verified by using tachometer to check result).

Blocked motor

Modify your code so that if the rotor is blocked or not spinning for more than 3 seconds, the LCD should display “motor stopped“ and the LED must stop blinking.

To stop the motor, disconnect the motor’s power supply.
Does it work as expected?

When the motor is stopped for more than 3 seconds, the LED stops blinking and the LCD on the Explorer 16/32 board displays 0RPM for both motor RPM and gearbox RPM. The LED resumes blinking and LCD display shows both RPM readings when the motor is running again. This behaviour repeats forever.

Push your code to GitHub.

You must demo the last 3 parts (Part2 through part4).

After lab questions:

1- Explain Part 2 improvements section. Write proper comment about the improvement done in your code.

The addition in code for Part 2 allows for the LCD on the Explorer 16/32 board to display 0 Hz and stops a blinking LED if no measurement has been done within three seconds. In order to implement this, a few things have to be done. We know that if the CN9 interrupt does not get called (meaning that there is no motor running since pulses are not being generated from the encoder and triggering the CN9 interrupt), a variable called "of" will count up a finite number of times. Usually, if the motor was running, this counter would be cleared after a certain amount of time (done within the CN9 interrupt). Since we know that the Timer2 will overflow after a set amount of time regardless of if the motor is running or not, we can track the counter and determine after how many counts it will take to achieve a certain delay. We can find out by performing this basic calculation:

$$\frac{Desired\ delay\ (in\ seconds)}{(65'537 * Timer2\ tick\ time)} =$$

$$Value\ counter\ needs\ to\ reach\ for\ specific\ delay\ to\ occur$$

If we plug in our values for our wanted three second delay, we can see what value the counter needs to reach in order for the desired delay to be possible:

$$\frac{3\ seconds}{(65'537 * 50nS)} = \sim 916$$

$$* 65'537\ is\ Timer2\ maximum\ overflow\ loaded\ into\ PR2$$

$$* 50nS = \frac{1}{PBCLK\ (Timer2\ tick\ frequency)} = \frac{1}{20MHz}$$

We can see that around 916 counts are needed for the counter to show that a three second delay has elapsed.

In the main.c file, we can implement a basic if – else statement to display 0 Hz on the Explorer 16/32 LCD display and to stop the blinking LED if the motor has been stopped for more than three seconds:

```
if (of >= 916){
    …display 0Hz on LCD and stop blinking LED.
}
else{
    …perform normal frequency calculation and display it
     on LCD. Toggle LED.
}
```

2- Explain the equation of the frequency (Hz, RPM, Gearbox). Write proper comment in your code.

The way the LCD display on the Explorer 16/32 board is able to display both the motor RPM and gearbox RPM are thanks to a multi-step process that calculates the motor RPM and gearbox RPM. First, the motor frequency is calculated, using the "`frequency = (PBCLK/timePeriod)/MOTOR_PPR;`" line. Here, the value of the measured period (measured in ticks – performed in isr_driven.c file) is divided from the value of the timer tick frequency (PBCLK). The result is further divided by the motor's PPR, since in one turn there are multiple periodic cycles that are detected on the motor's encoder. With this calculated value, the motor's RPM and gearbox RPM can be calculated. The motor's RPM is calculated using the "`motor_rpm = frequency * 60;`" line. Here, the motor's calculated frequency gets multiplied by sixty, since in one hertz there are 60 revolutions (constant -> 1 Hz = 60 RPM). Lastly, the gearbox RPM is calculated using the "`gearbox_rpm = motor_rpm / GEARBOX_RATIO;`" line. Here, the motor's calculated RPM is divided by the motor's gearbox ratio since the gearbox usually always runs slower than the motor.

3- Give the modification to your code if the timer tick frequency is changed to 10MHz.

The following code modification resides in the motor RPM and gearbox RPM calculations:

```
#define PBCLK           10000000
#define MOTOR_PPR       //See motor datasheet.
#define GEARBOX_RATIO   //See motor datasheet.


...

frequency = (PBCLK/timePeriod)/MOTOR_PPR;   //Calculates motor frequency.*
motor_rpm = frequency * 60;                 //Calculates motor RPM.
gearbox_rpm = motor_rpm / GEARBOX_RATIO;    //Calculates motor gearbox RPM.

//*(Value of PBCLK depends on prescale and value of SYSCLK).


...
```