Lab #9 – Delayed event

Leonardo Fusser (1946995)

Objectives:

- Understand a real time system program.
- Find the differences between blocking and non-blocking functions in a real time system.
- Use of timer interrupt.
- Measure the super loop overhead.
- Profile an embedded program.

Hardware: simulation on MPLABX version 5.1 or less. No target needed.

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 on GitHub.
- The main.c file must include a prologue header: Name, lab number, a short description, authors, date, version.

MPLAB X:

Install MPLAB X version 5.1 on your computer. DO NOT install the latest MPLAB X version.

The logic analyzer does not work with MPLAB X version 5.35 or higher.

https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-ecosystem-downloads-archive

Introduction:

The goals of this lab are to:

1. **Print a message periodically on the console - every 3000 sysclk cycles.**
2. **Keep the super loop tight – short overhead.**

The period will be determined by a blocking or a non-blocking method.

We will compare and profile three different paradigms:

Part 1: Blocking delay.

Part 2: No-blocking delay using system clock polling.

Part 3: No-blocking delay using timer interrupt.

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/QWjX6E1r

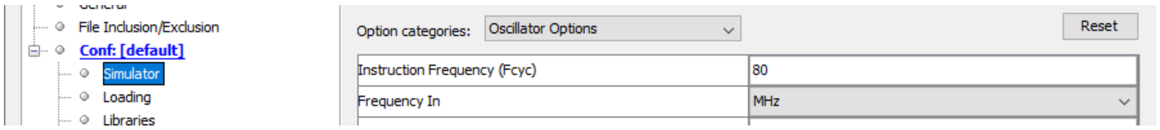
You will get a private new repository for this lab. You will push your project to this repo at the end of this lab.
https://github.com/System-Programming-Vanier/lab9_405-yourName

Lab work:

SYSCLK

All along this lab, delays are in sysclk cycles (AKA sysclk ticks).

By setting sysclk to 80 MHz, one tick corresponds to 1/80MHz or 12.5 nS



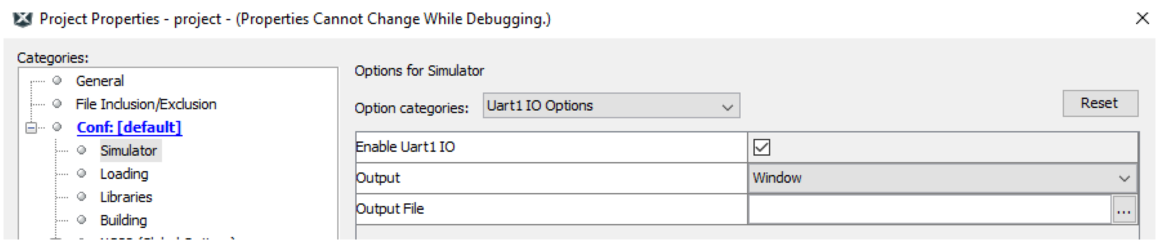
Option categories: Oscillator Options

Instruction Frequency (Fcyc)	80
Frequency In	MHz

System setup

Before writing any code, build the project to make sure no pieces are missing.

Also, make sure Uart1 IO Options is enabled:

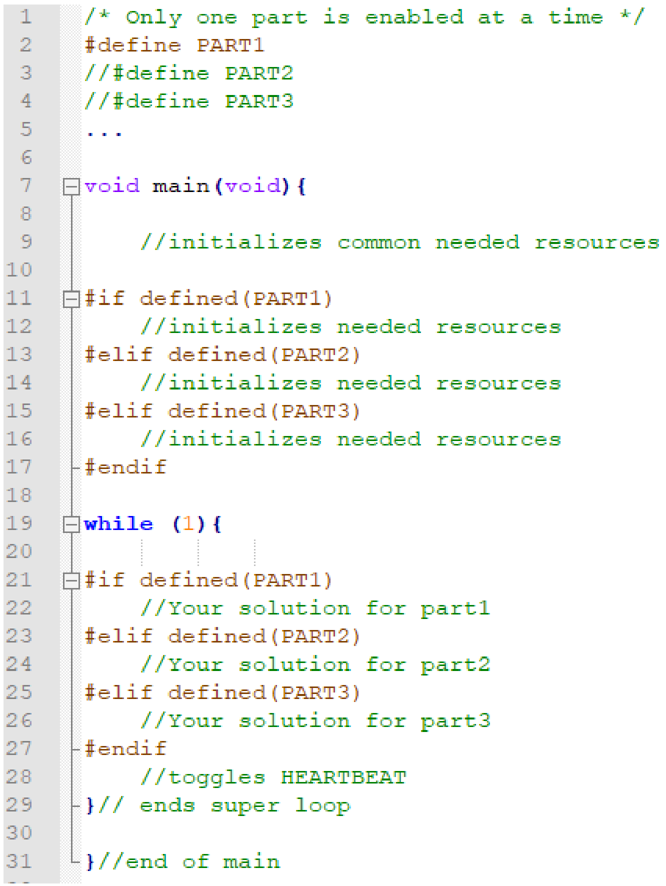

Project Properties - project - (Properties Cannot Change While Debugging.)

Options for Simulator

Option categories: Uart1 IO Options

Enable Uart1 IO	✓
Output	Window
Output File	

The skeleton of your program should be as follows:



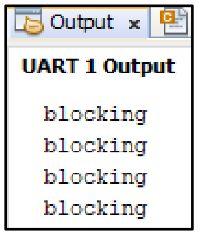
```c
/* Only one part is enabled at a time */
#define PART1
//#define PART2
//#define PART3
...

void main(void){

    //initializes common needed resources

#if defined(PART1)
    //initializes needed resources
#elif defined(PART2)
    //initializes needed resources
#elif defined(PART3)
    //initializes needed resources
#endif

while (1){

#if defined(PART1)
    //Your solution for part1
#elif defined(PART2)
    //Your solution for part2
#elif defined(PART3)
    //Your solution for part3
#endif
    //toggles HEARTBEAT
}// ends super loop

}//end of main
```

This way, you can easily enable/disable a part.

Part 1: *Blocking delay*

Requirements

In this part you must print a small message periodically:



```
blocking
blocking
blocking
blocking
```

The message must be printed from the main loop and the required delay must be blocking for 3000 sysclk cycles.

Also, to profile the loop, the heart beating output must toggle at the end of the loop.

```
while(1){
   blocking delay of 3000 sysclk ticks
   print a message "blocking" to the console
   toggles port RA7 (AKA HEARTBEAT)at the end of the loop
}
```

Use `delay_ticks()` blocking delay. See annex for all API definitions.

Note: The core timer increments once for every two ticks of SYSCLK.

Profiling

Execute the program and find out the overhead of the loop.

To measure, you must display the toggling heartbeat on the logic analyser: open the logic analyser window to visualize RA7.

Looking at the logic analyser:

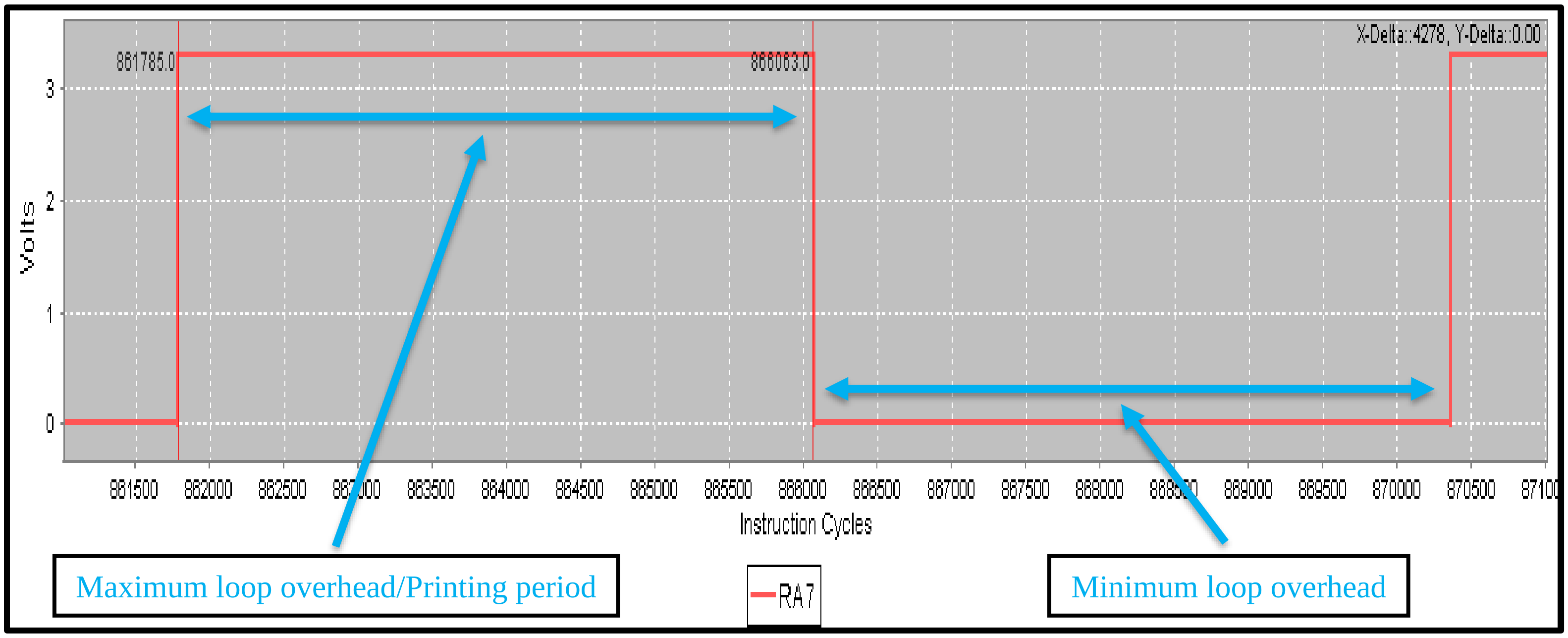
Is the main loop overhead constant? Explain.

For this blocking delay type program, the main loop overhead is constant. The measured minimum loop overhead is around 4'270 cycles, and the measured maximum loop overhead is around 4'290 cycles. The main loop overhead is not tight at all because if it were to be considered tight, then the minimum loop overhead would need to be very small (a few cycles – close to 0 seconds). It will be shown in the next parts, using different methods to print the text to the console, how the main loop overhead can be diminished. See "Measures" section for more details.

Measure the following by filling in table 1 at the end of this document:

- Loop overhead– minimum and maximum.
- Printing overhead.
- Printing period.

Take a screenshot of the logic analyser.
In the waveform identify the printing overhead and printing period. You can also use the stopwatch.



Screenshot of logic analyzer in MPLAB X IDE. Pin RA7 (LED heartbeat) shown on logic analyzer above. As seen in the screenshot above, the maximum loop overhead is around 4'280 cycles, and the minimum loop overhead is around 4'270 cycles. The maximum loop overhead is due to the printing and the minimum loop overhead is when the processor skips that if statement. As mentioned above, the main loop overhead is constant, since the maximum and minimum loop overhead are more or less the same. Also, the main loop overhead is not tight.

Part2: *Non-blocking by polling the system clock*

In the previous blocking method, the loop overhead is quite large.

To make the loop tight, the blocking delay method will be replaced by a clock polling method.

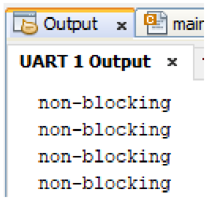
In this part, `delay_ticks()` is NOT allowed in order to reduce the overhead and get a more responsive system.

Also, the program must NOT contain any blocking function like while loop or for loop.

```
while(1){
      clock polling method to print "non-blocking" every 3000 sysclk ticks
      toggles port RA7 (AKA HEARTBEAT)at the end of the loop
}
```

Any time-related blocking functions must be replaced by 'if' statements along with clock polling - time stamping.

Also, print a different message every 3000 sysclk cycles:



```
non-blocking
non-blocking
non-blocking
non-blocking
```

The `Tick_core.c` library provides `TickGet()` and `TickDiff()` functions. See Annex.

Profiling

Execute the program and find out the overhead of the loop.

Also, you must display the toggling heartbeat on the logic analyser.

Looking at the logic analyser:

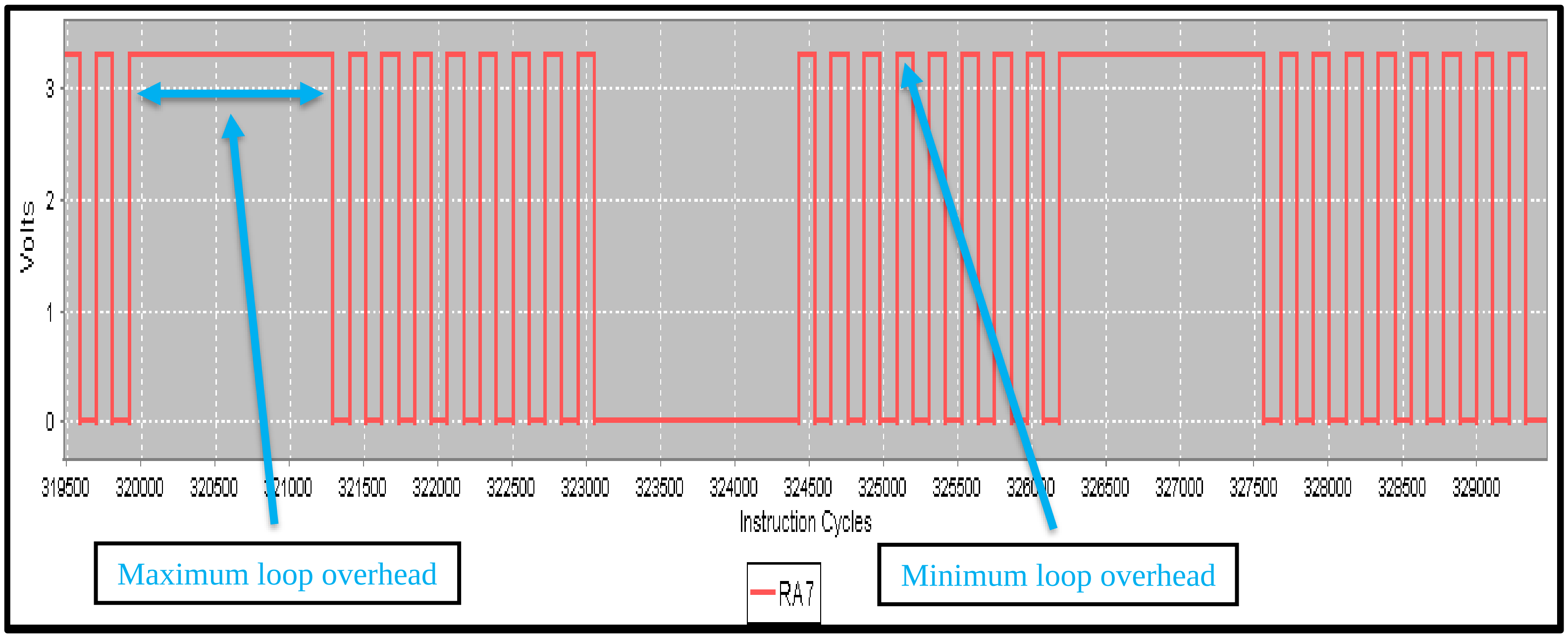
Is the main loop overhead constant? Explain.

For this non-blocking delay type program, the main loop overhead is no longer constant. Unlike with Part 1 from above, the measured minimum loop overhead is around 110 cycles, and the measured maximum loop overhead is around 1'380 cycles; these are much better results than from before, yet we are still able to print a message to the console like from before. However, the main loop overhead is still not tight at all because the minimum loop overhead is still not very small (needs to be a few cycles – close to 0 seconds). It will be shown in the next part, using a different method to print the text to the console, how the main loop overhead will be diminished drastically. See "Measures" section for more details.

Measure the following by filling in table 1 at the end of this document:

- Loop overhead– minimum and maximum.
- Printing overhead.
- Printing period.

Take a screenshot of the logic analyser.
In the waveform identify the printing overhead and printing period.



Screenshot of logic analyzer in MPLAB X IDE. Pin RA7 (LED heartbeat) shown on logic analyzer above. As seen in the screenshot above, the maximum loop overhead is around 1’380 cycles, and the minimum loop overhead is around 110 cycles. As mentioned above, the main loop overhead is not constant, since the maximum and minimum loop overhead are not the same. Also, the main loop overhead is not tight.

Part3: *Triggering an event using timer interrupt*

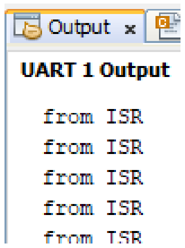
In the previous blocking method, the loop overhead is reduced because the blocking delay was removed.

There exists another way to delay an event: timer interrupt.

While the main loop still works in the foreground, a timer interrupt executes periodically in background.

Again, any blocking delay or blocking function are NOT allowed.

Print the following message every 3000 sysclk cycles:



```
Output ×
UART 1 Output

  from ISR
  from ISR
  from ISR
  from ISR
  from ISR
```

The main loop must still blink the heartbeat.

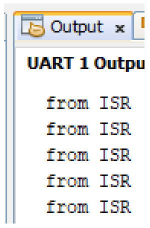
```
while(1){
          toggles port RA7 (AKA HEARTBEAT)
}
```

Enable timer3 in the main:

```
    initT3();
```

Setup Timer3 to timeout every 3000 sysclk ticks (No pre-scale).

Print the message "from ISR" inside T3 ISR:



```
Output ×
UART 1 Outpu

  from ISR
  from ISR
  from ISR
  from ISR
  from ISR
```

Profiling:

Execute the program and find out the overhead of the loop.

Looking at the logic analyser:

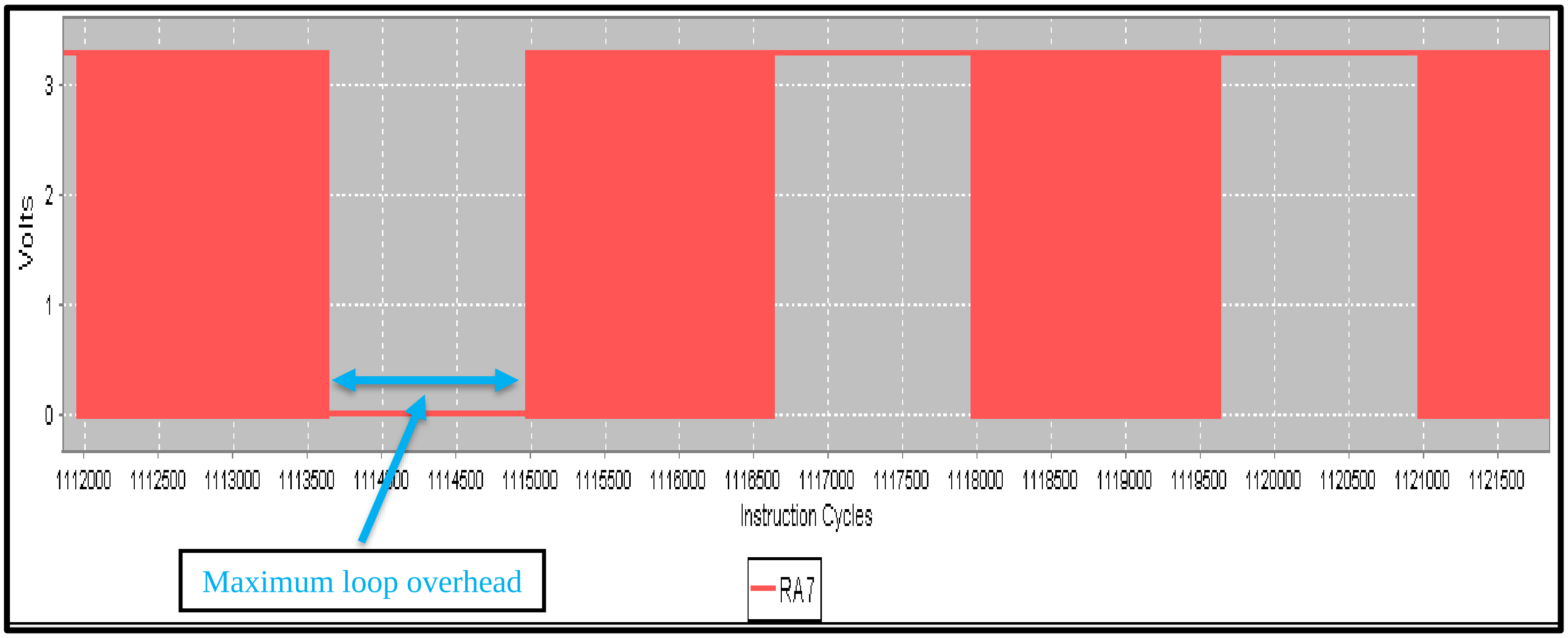
Is the main loop overhead constant? Explain.

Similar to the previous part, for this program implementing interrupts, the main loop overhead is no longer constant. Unlike with Part 2 from above, the measured minimum loop overhead is around 14 cycles, and the measured maximum loop overhead is around 1'340 cycles; these are much better results than from before, yet we are still able to print a message to the console like from before. This time, the main loop overhead is tight because the minimum loop overhead is very small. Therefore, in order to print a message to the console efficiently and to keep the main loop overhead tight, interrupts should be used. See "Measures" section for more details.

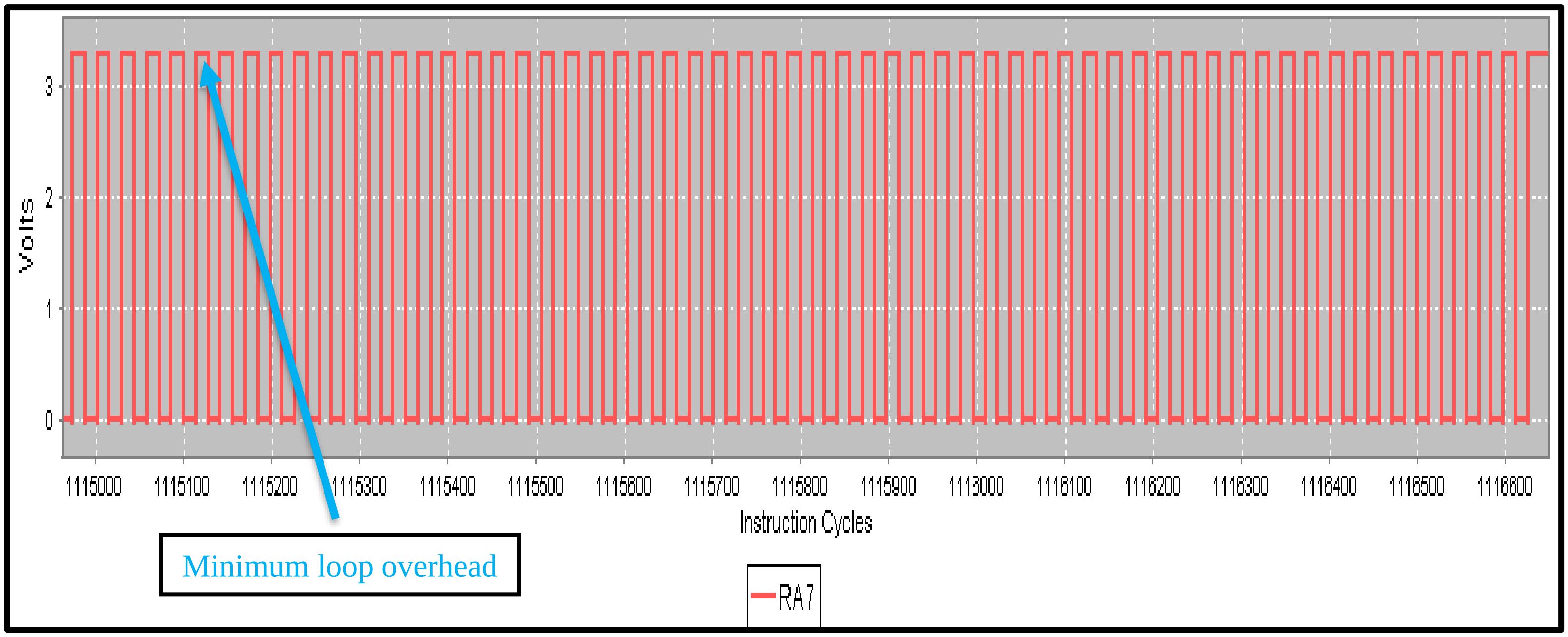
Measure the following by filling in table 1 at the end of this document:

- Loop overhead– minimum and maximum.
- Printing overhead.
- Printing period.

Take a screenshot of the logic analyser.
In the waveform identify the printing overhead and printing period.



Screenshot of logic analyzer in MPLAB X IDE. Pin RA7 (LED heartbeat) shown on logic analyzer above. As seen in the screenshot above, the maximum loop overhead is around 1'340 cycles, and the minimum loop overhead is in the shaded red; it can be viewed by looking at the screenshot below. As mentioned above, the main loop overhead is not constant, since the maximum and minimum loop overhead are not the same. Also, unlike with the two previous parts, the main loop overhead is tight.



Zoomed in view of the shaded red from the above screenshot. As seen with this screenshot, the minimum loop overhead is around 14 cycles (very tight).

Measures

Fill the following measure table for all parts:

Measure/Part#	Part1	Part2	Part3
Loop overhead (toggling LED) (sysclk cycles)	Min: 4'271 cycles (53.3875uS)	Min: 110 cycles (1.375uS)	Min: 14 cycles(175nS)
	Max: 4'288 cycles (53.6uS)	Max: 1'378 cycles (17.225uS)	Max: 1'340 cycles (16.75uS)
Printing overhead (sysclk cycles)	1'227 cycles (15.3375uS)	1'244 cycles (15.55uS)	1'244 cycles (15.55uS)
Printing period (sysclk cycles)	4'288 cycles (53.6uS)	3'138 cycles (39.225uS)	3'001 cycles (37.5125uS)

Check list

- ☐ Indented code
- ☐ Commented code – main file prolog header- function prolog header
- ☐ All commented screenshots

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

You must provide a prolog header with a version number.

Extra questions

Q1- In part1, does it print every 3000 sysclk cycles or so? Explain.

For the blocking type program in Part 1, the message does not get printed every 3'000 sysclk cycles. Instead, the message gets printed around every 4'290 sysclk cycles; a lot more than what we had programmed it to do and what we had expected! The reason why the message does not get printed every 3'000 sysclk cycles is due to various reasons that are beyond the scope of this lab. However, it will be shown in the next two questions how we can obtain a result close to every 3'000 sysclk cycles to print the message by using different methods in doing so.

Q2- In part2, does it print every 3000 sysclk cycles or so? Explain.

For the non-blocking type program in Part 2, the message still does not quite print every 3'000 sysclk cycles as what was expected. Instead, the message gets printed around every 3'138 sysclk cycles; a lot better than what was seen in Part 1, but we can do better! As mentioned before, it is beyond the scope of this lab to explain why the message does not get printed every 3'000 sysclk cycles as what was thought. However, in the question below, you will see how precise we can get the message to print to our expected value (very close to every 3'000 sysclk cycles).

Q3- In part3, does it print every 3000 sysclk cycles or so? Explain.

For the timer interrupt type program in Part 3, the message prints very closely to every 3'000 sysclk cycles. Using this method with interrupts, we see that the message gets printed every 3'001 sysclk cycles; much better and the most precise compared to what was seen in Part 1 and Part 2! After looking at these three questions, we can conclude then that in order to get the most precise timing (very close what the theoretical value would be), interrupts and timers should be used in place of blocking and non-blocking type program.

Q4- Is there a difference in term of loop overhead between part1 and part2? Explain.

There is a noticeable difference in terms of loop overhead between Part 1 and Part 2. As was explained in the different parts above, the loop overhead is much tighter in Part 2 than Part 1 because the minimum loop overhead is much less in Part 2 than in Part 1. This is because the program is using a non-blocking delay approach in Part 2 and in Part 1, a blocking delay approach was used. Non-blocking delay approach reduces loop overhead whereas blocking delay increases it significantly.

Q5- Explain the maximum loop overhead for part2 and part3.

In Part 2, the maximum loop overhead is around 1'380 sysclk cycles and the maximum loop overhead in Part 3 is around 1'340 sysclk cycles. For Part 3, the maximum loop overhead is smaller than the maximum loop overhead in Part 2. This is due to the printing of the message to the console.

Q6- Explain the minimum loop overhead for part2 and part3.

In Part 2, the minimum loop overhead is around 110 cycles and the minimum loop overhead in Part 3 is around 14 cycles. For Part 3, the minimum loop overhead is smaller than then minimum loop overhead in Part 2. This is due to when the processor skips the if statement.

Q7- Is there a difference in term of loop overhead between part2 and part3? Explain

There is a noticeable difference in terms of loop overhead between Part 2 and Part 3. Similar to what was mentioned above in question 4, depending on the method used to print the message to the console, the loop overhead will differ. Here, in Part 3, the loop overhead is very small compared to the loop overhead in Part 2. Although the loop overhead in Part 2 is much better than what was seen in Part 1, the loop overhead in Part 3 is very tight and much better than the loop overhead in Part 1 and 2. This is because Part 3 uses interrupts instead of blocking or non-blocking type delay. With interrupts, there is nothing to execute in the super main loop, which causes the loop overhead to be very tight. The other parts have code to execute in the super main loop and therefore, there will be associated loop overhead with it.

Q8- Explain how the timer interrupt works. Do the main loop and the ISR run simultaneously? Explain

In Part 3, the main loop and the ISR do not run at the same time always. During the majority of the processor's instruction execution time, the processor will remain in the main loop. The only time the ISR is executed is when the Timer3 module times out and a status flag is set indicating to the processor that it should serve the interrupt. After the interrupt is complete, the processor will leave the interrupt and resume normal execution (be in the main loop). In our case, the majority of the processor execution time is indeed spent in the main loop and after every 3'000 sysclk cycles or so, the Timer3 module times out, an interrupt is triggered (Timer3 interrupt flag is set), and a message is printed to the user console (different for each part). Once the printing is done (interrupt finished), the processor will leave the interrupt and resume normal execution in the main loop.

Q9- What is (are) the best scenario(s) to execute an event periodically along with keeping the main loop tight (small overhead)?

Remember the goals of this lab:

1. Print a message periodically on the console - every 3000 sysclk cycles.
2. Keep the super loop tight – short overhead.

As mentioned briefly in some parts above, in order to obtain the two goals mentioned here, interrupts with timers are the best scenarios. This is because the interrupts used in this lab eliminate the bloating in the super loop (it has very tight overhead) that Part 1 and Part 2 cause. Furthermore, timing is better handled with interrupts and the outcome is very close to what would be expected (as seen in Part 3 above compared to Part 1 and Part 2). If interrupts were not an option, non-blocking delays (system clock polling method) should be used. Although the super loop is not as tight like with interrupts, it is not as bad as blocking delays. As seen in Part 1, blocking delays add massive overhead and printing timing is very uncontrollable. Therefore, this method should be avoided altogether.

Q10- Explain and compare the logic analyser for all 3 parts. Are they similar? Why do they differ?

Looking at the three logic analyzer screenshots for the three parts, it is evident that they are all different. Since each part has a different method to print the message to the console, each part will have different loop overheads associated with them. This can be evident by looking at the three screenshots, and for example, it can be seen that Part 1 has the worst loop overhead associated with it and that Part 3 has the best loop overhead (very tight) just by looking at the timings on the logic analyzer for each of the different parts. Basically, the only thing that differs across the three different logic analyzer screenshots is the loop overhead. See the screenshots above and the table under the “Measures” section.

Q11- If timer3 times out every 5000 cycles and the tick clock (PBCLK) is 40MHz, find its period in uS.

Annex

/* Prints a message to the specified console
 * mode must be C_UART1
 */
```
void fprintf2(int, char *)
```
Example : `fprintf2(C_UART1,"Hello\n"); // Prints Hello to UART1 tab`

/* Blocking delay function
 * Block for the amount sysclk cycles specified
*/
```
void delay_ticks(unsigned int cycles);

/* 10000 sysclk ticks blocking delay.  */
```
Example : `delay_ticks(10000);`

/* Gets the core clock timer current value in sysclk ticks */
```
int64_t TickGet(void);
```
Example : `stamp = TickGet();  // memorizes the core-timer’s current tick`

/* Returns the difference between the current core timer ticks and the latest stamp value */
```
int64_t TickDiff(int32_t)
```
Example :
```
int32_t stamp;
        …
/* Goes into the if every 1000 cycles */
/* Corresponds to 2000 sysclk ticks */
if(TickDiff(stamp) > 1000) {
        …
}
```