Timing Analyzer

Learning focus:
State machines, interrupts, timing, structs

In complex applications, different tasks are processed with different priorities and within certain timing constraints. It is important, that each interrupt is executed with correct timing, and that we also check the execution time, in order to keep an eye on the CPU load.

For simple applications the debugger works well – but as you already have learned is, there is no perfect synchronization of the CPU and the hardware when a breakpoint is reached. That means, that e.g. a hardware counter might go on counting a little bit while the CPU is already stopped. That means, that debugging in the presence of interrupts does not make sense.

But, how can we check, which interrupt preempts another interrupt and how long? Therefore, we mustn't stop the system. We're simply measuring the timing behavior. We have two options for that:

- Measure internally, by counting CPU cycles or using a timer.
- Measure externally, by attaching a logic analyzer.

Therefore, your task is to develop an easy to use API, which we can use for time measurement of any code regions in your code. Multiple analyzers must be working in parallel. This API will be used in the next lab experiments as well.

Marking

Implementation	6 points
Proper Design	4 points

You need 40% = 12 points of all three lab assignments in order to pass the lab!

1.1 Requirements

We want to be able to configure the analyzer based on the intended use. Three techniques should be selectable:

- SysTick timer at 1ms for long-term measurements
- CPU cycle counter from the DWT trace unit for precision measurements
- Hardware pins for external measurement with a logic analyzer (which can be used in addition to the time measurements mentioned above)

Req-Id	Description
NFR1	The system will be developed as "bare metal," without an operating system.
NFR2	All status information and IDs of the analyzers will be stored in self-explanatory structs and described by self-explanatory enums.
NFR3	Use the concept of object oriented C as presented in the presemester class.
NFR4	For configuring access to the output pins, use function pointers to the respective API call.

FR5	We want to **initalize** the API. This includes the initialization of the necessary peripherals.
FR6	We want to **create** an analyzer with one of the following configurations: • DWT Cycle Counter • DWT Cycle Counter + Output Pin • SYSTICK • SYSTICK + Output Pin • Output Pin only Moreover we want to give the instance of the analyzer a string name, which will be printed out in the print function. Hint: Use the red, green and yellow LEDs as output pins. Recommended API call: `TimingAnalyzer_create(me, mode, pin, name);`
FR7	We want to **print the status** of a single timer: • For SysTick timers, the elapsed time will be given in milliseconds. • For the DWT counter, the elapsed time will be given in milliseconds with six decimal places, and the elapsed CPU cycles will also be displayed. The string to be printed must be fully assembled before being sent to the UART (i.e., UART_LOG_PutString() should only be called once per print). Hint: For concatenation of strings, keep in mind, that the return value of `sprintf()` is the number of bytes written by `sprintf()`.
FR8	We want to **print all** existing analyzers.
FR9	We want to **start** an analyzer. This is only possible when it is configured and not currently running. If the analyzer is paused, it should resume.
FR10	We want to **stop** an analyzer and calculate the elapsed time. This is only possible when the analyzer is running or paused. The elapsed time will be stored in the analyzer's struct.
FR11	We want to **pause** an analyzer. The elapsed time since the last start will be added to the analyzer's total duration. This is only possible when the analyzer is running.
FR12	We want to **resume** an analyzer. This is only possible when the analyzer is paused.
FR13	We want to use the **Systick Interrupt** with an interval of 1ms.

Some more non-functional requirements

Req-Id	Description
NFR14	Fetching the cycle counter or SysTick counter must be the first action within the start, stop, pause, or resume functions for precise measurement.

1.2 Analysis

The time measurement using the cycle counter highly depends on the CPU frequency. The current CPU frequency can be accessed via the define `BCLK__BUS_CLK__HZ`. Check this value and compare it to your clock settings. Which CPU frequency do you use in your project?

BCLK_CLK_HZ is 24MHZ Frequency

Which will be the longest intervals that can be measured with the SysTick and with the DWT Cycle Counter without generating an overflow?

For Systick counter the maximum ticks are 16,777,215 with 24MHZ frequency we get 42.67 *16,777,215 which gives the value nearly 699 milli seconds
For DWT : Maximum value 178.96 seconds

How do we calculate the time in ns from the CPU cycle counter without loss of precision and without overflows?

with the below mentioned formula we calculate the time in nan second as

time in nansec = (cycleCount/CPU frequency)*10°9

Perform an object oriented analysis for the timing analyzer objects. Which elements should be included in the data structure for a timing analyzer?

In timing analyzer the below elements can be included considering the time calculati on sunch as variables to store the current cycle counts like start time stop time pausetime and elapsed time, status and mode of the timing anayzer if its with DWT or with SYSTICK

Which enums do we need to describe the current config and status of the timer?

```
typedef enum
{
STATUS_START,
STATUS_PAUSED
STATUS_RESUME,
STATUS_STOPPED
}
```

```
typedef enum
{
DWT_Cycle_Counter,
DWT_Cycle_Counter_Output_Pin,
SYSTICK,
SYSTICK_Output_Pin,
Output_Pin_only
}
```

Which API calls do we need?

I have used the below API Calls

```
void timing_Analyzer_init();
voiud Systick_timing_Analyzer(Analyzer_t*me);
void Time_Analyzer_creat(Analyzer_t*me,uint8_id,AnalyzerMode,Char *Name);
void Analyzer_Start(Analyzer *me);
void Analyzer_Pause(Analyzer*me);
void Analyzer_resume(Analyzer *me);
void Analyzer_printstatus(Analyzer *me);
```

Which peripherals need to be started / initialized in the init function?

The below peripehrals are inititialed for DWT and SYSTICK Counters

```
void timing_Analyer_init();
```

Conditions for DWT and Systick Counter is available in this function.

UART peripheral is used to print the count values.

Now implement the API according to the given requirements!

2 Test

Now we need to write code snippets to test the functionality of the API.

2.1 Basic Functionality

Write code in the main.c with CyDelay() to prove that:

- start / stop and print works properly
- pause / resume works as expected
- multiple timers can be used in parallel without interfering

Paste the terminal output from your basic tests:

Start and Stop with Pauese resume and Stop states atre checked for DWT and Systic.

Images will be upladed for the results

2.2 Mathematical Functions

Measure the durations of the following calculations:

- 1000 integer additions
- 1000 integer multiplications /divisons
- 1000 float additions
- 1000 float multiplications /divisons
- 1000 Square root calculations / sin() / sinf()

Test	Time (ms)	CPU cycles	Time (Logic Analyzer)	Comment
integer additions	2010 ms	33932	1.91525 msec	for addition without delay time takes is verxý less about 1.41 msec hence checked with the delay. Incresing the range to 1000000 with red pin led is on for 421.11 mseccycle 418
integer multiplications	2010 with delay	33958	1.9125 milli sec	
integer divisons	3 milli secs	101416	3 milli secs with systick op	
float additions	5 mili secs	156001	7.36milisecs	
float multiplications	3 msecs	102877	3 milli secs	Unable to print the result of float data math functions with the linker settings
float divisons	0	29943	1.91 msec	Floating points are not working
sqrt()				These math functions are not supporting getting an error while compiling
sin()	0	29944	1.91 msec	It could be because of the code optimization tried to store the values with that undified reference to sin error is popping up
sinf()	0	25932	1.91 millisec	

2.3 Interrupts

Set up the following test scenario:

Implement two timer interrupts, both with same priority (7). The following actions within the Interrupt Service Routines (ISR) shall be performed:

- `ISR_1ms` (running @ 1 ms)
    - Start a time measurement (test the different timing sources)
    - clear the pending bits
    - Stop
    - return
- `ISR_2seconds` (running @ 2 seconds)
    - Start a time measurement (test the different timing sources)
    - Do some calculations or implement a delay for approx. 1 second
    - Stop or pause the time measurement
    - Print the elapsed time
    - return

Write a test table for the following scenarios (see next page)

1. Measure the duration of the code in both ISRs
2. Change the priority of `ISR_1ms` to prio4 (higher priority than `ISR_2seconds`)
3. Change the cycle time of Timer1 from 1ms to 0.1ms
4. Change the priority of ISR2 to prio0

Explain, where the differences in the time measurement come from when stepping though the above test cases.

Moreover, compare the time measurement from the logic analyzer to the internal measurement methods and describe the source for possible differences.

Test Nr.	Duration (SysTick)		Duration (cycle counter)		Duration (Logic Analyzer)		Observation	Justification
	ISR_1ms	ISR_2sec	ISR_1ms	ISR_2sec	ISR_1ms	ISR_2sec		
1								
2								
3								
4								

2.4 Extra Task

As we know, the debugger significantly changes timing behavior. However, consider the internal and external measurement methods mentioned above: does the timing behavior truly remain unaffected? Briefly discuss which situations might cause issues and how we can minimize their impact on the timing behavior of our tasks:

Check the disassembly and find out, how many cycles approximately get lost when calling start / stop / pause / resume to that point, where the counters are read.