Embedded and Ambient Systems

Topic of exercise 6

Runtime measurement, time measurement

During the exercise, we will learn about the following topics:

- the technique of measuring runtime,
- resource requirements of some typical instructions,
- time measurement in general.

1. Runtime measurement

Background knowledge

In the following, we will get to know with the method of measuring the runtime and also examine the runtime of some typical instructions.

Several methods of measuring the runtime are known (see lecture), during the exercise we will use a built-in special timer/counter of the microcontroller. This counter is contained in the unit called Data Watch point and Trace (DWT) within the Debug Interface in the processor. Of course, the task can be solved with other counter/timer units, but in this case this counter is used being the simpler one. The counter is 32 bits, so in the case of the default clock signal of 14 MHz, $T_{max} = \frac{1}{14\ MHz} 2^{32} = 306\ sec\ \approx 5\ min$ is the duration that can be measured as runtime. Here, we note that the measurement of the runtime is actually the measurement of the runtime cycle time, so we will determine how many clock cycles elapses as a given piece of code runs, which, in turn, can easily be converted into time if you know the clock frequency.

The name of the counter used to measure the runtime is: Cycle Count Register. This register starts at zero when the processor starts up, and its value increases by one every clock cycle elapses. The counter register can be accessed as follows:

```
DWT->CYCCNT
```

A possible solution for measuring:

```
runTime = DWT->CYCCNT;

      Here comes the program code whose runtime should be measured.

runTime = DWT->CYCCNT - runTime - COMP_CONST;
```

After running the above code, the runTime variable will contain the runtime measured in clock cycles. The value of COMP_CONST, is a constant with the help of which the above expression returns a zero value if no program code is specified. With this, we correct the time spent on calculation and register reading, which is included in the runtime, but is otherwise not part of the program. So runtime measurement has an overhead!

Tasks

1. Create an empty project and define the value of the COMP_CONST constant in the program code specified for measuring the runtime.
2. Measure the runtime of the following three operations for three types of data according to the following table:

-O0	int32_t	float	double
addition			
multiplication			
division			

Sample code:

```
int32_t a=300, b=20, c=0;
runTime = DWT->CYCCNT;
      c=a+b;
runTime = DWT->CYCCNT - runTime - COMP_CONST;
```

How can you tell that the processor has an instruction that supports division?

3. With the -O0 level optimization setting (i.e., optimization is turned off), measure how many clocks cycles the following conversion types take.

Sample code:

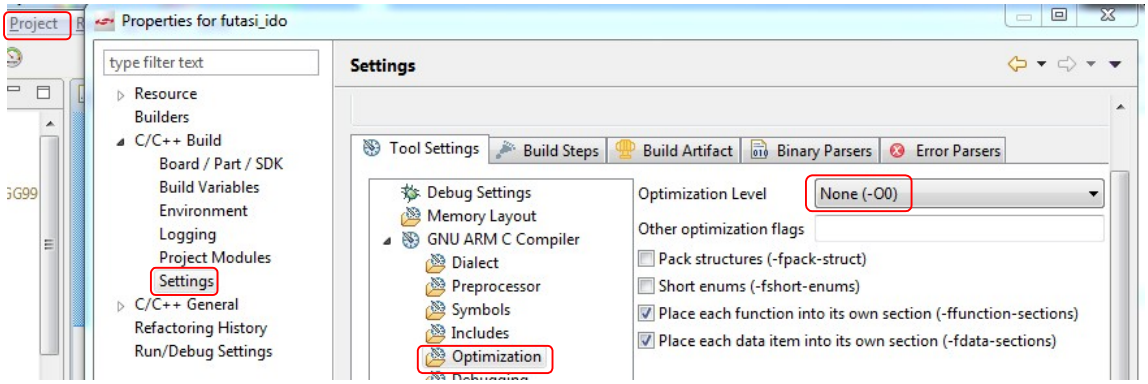
```
a_float = (float) a_int;
```

Note that the type conversion (casting) is carried out by the compiler even if it is not specified explicitly, so the next line also contains an implicit type conversion, the same as writing the (float) to_int conversion:

```
a_float = a_int;
```

source \ destination	int32_t	float	double
int32_t			
float			
double			

4. Let's measure again the runtime of the type conversions also with -O3 optimization level:
   Project → Properties → C/C++ Build → Settings → Optimization → Optimize now (-O3)



Notice the result we get for the runtime after re-compiling the project (e.g. in the case of division of double data type).

In this case, we can observe strange things because the compiler removes unused variables, so the measured program code can practically disappear. It is therefore worth declaring the output and input variables of the operations, as well as the variable used to measure the runtime, as **volatile** variables, so that the compiler does not optimize them.

5. With -O3 level optimization turned on, declare two int32_t arrays of size N (make them volatile) and calculate the sum of the products: $s = \sum_{i=0}^{N-1} A[i] * B[i]$. Make a note on the runtimes for blocks of size H=15...20.

N	**15**	**16**	**17**	**18**	**19**	**20**
clock cycle (–O3)						
clock cycle (–O0)						

Check how smooth the change is as the cycle length increases. Notice the function of the compiler loop unroll [1] in the disassembled code.
In the case of N=15, see what the runtime is when optimization is turned off.

[1] loop unroll: unfolding the for loops so that the code to be executed N times is repeated N times by the compiler. It increases the size of the code, but is generally faster because there is no need to handle the loop variable and jump instruction to execute. Another problem may be that the instruction cache cannot be used as in the case of a cycle.

Sample code:

sum=0;

```
for (ii=0; ii<N; ii++) {

      sum += A[ii]*B[ii];

}
```

2. Time measurement, timer configuration

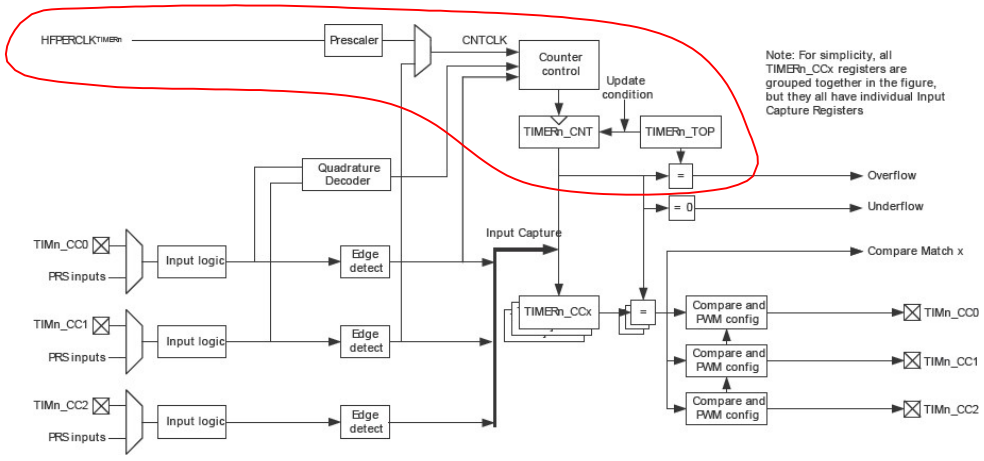
In the following, we will learn about the general method of time measurement, the use of timers.

The timer units of microcontrollers usually have two main operating modes:

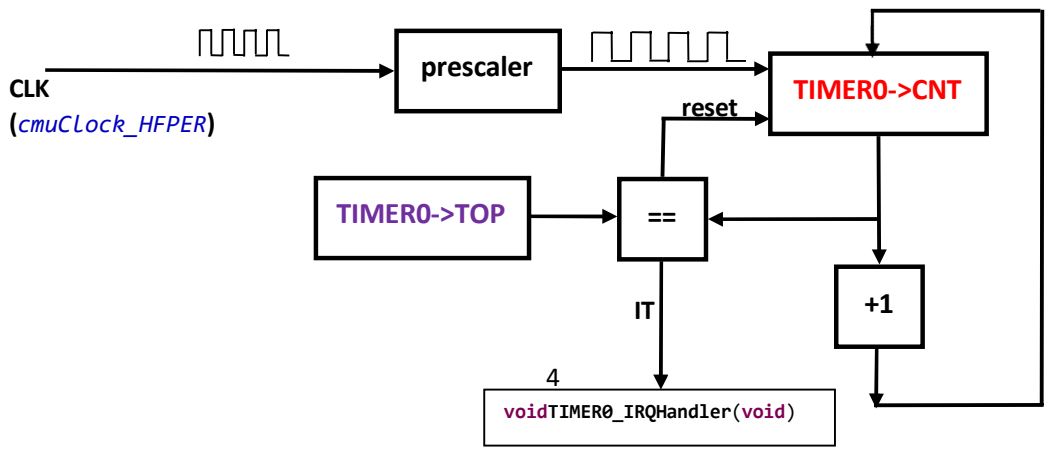
- Timer: with the help of some internal clock signal source, we measure time and generate interrupts.
- Counter: we usually count incoming pulses from some external source.

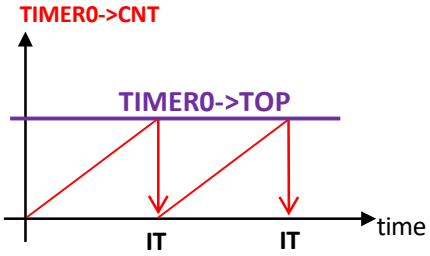
Within these, even more sub-cases are possible.

The timer unit of the microcontroller used in the exercise is constructed as follows (we will use the Timer0 unit):



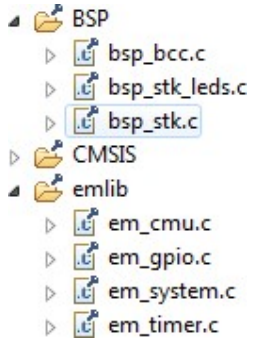
We will use it in the timer mode corresponding to the simplified diagram below (explanation of the red-framed part of the diagram above):





In simple timer mode of the Timer, the value of the CNT register increases by one for each cycle of the input clock until the value stored in the TOP register is reached. Then the CNT counter resets and generates an interrupt on the corresponding interrupt line (TIMER_IF_OF).

To manage the timer, add the files em_timer.c and em_cmu.c2 to the project and include the files em_cmu.h and em_timer.h. To handle the LEDs, add the files bsp_bcc.c, bsp_stk_leds.c, bsp_stk.c and em_gpio.c3 to the project and include the file bsp.h.




```
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_timer.h"
#include "bsp.h"
```

The timer is configured using its library functions as follows:

- setting the clock signal divider of the peripheral
- enable timer clock
- we create the parameter structure required for initialization
    - the prescaler is set to the appropriate value
- reset the timer
- we set the TOP value
- clear any pending interrupt
- We enable interrupt
    - Let's enable it at the timer peripheral
    - Enable the Timer interrupt (NVIC) at the central interrupt manager.

The above initialization process is implemented by the following program code.

2 path: SimplicityStudio\developer\sdks\gecko_sdk_suite\v1.1\platform\emlib\src\

3 path: SimplicityStudio\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\bsp\

```
  // set the clock divider of the peripheral
  CMU_ClockDivSet(cmuClock_HFPER, cmuClkDiv_1);

  // *******************************
  // *    TIMERinitialization      *
  // *******************************

  // enable timer clock
  CMU_ClockEnable(cmuClock_TIMER0, true);

  // create the parameter structure required for
  initializationTIMER_Init_TypeDefTIMER0_init = TIMER_INIT_DEFAULT;
  // reset the prescaler
  TIMER0_init.prescale=timerPrescale1;// timerPrescale1...timerPrescale1024
  // initialization with the parameter structure
  //void TIMER_Init(TIMER_TypeDef *timer, const TIMER_Init_TypeDef *init);
  TIMER_Init(TIMER0, &TIMER0_init);

  // reset the
  counterTIMER_CounterSet(TIMER0, 0);//

  // set the TOP value
  // STATIC_INLINE void TIMER_TopSet(TIMER_TypeDef *timer, uint32_t
  val)TIMER_TopSet(TIMER0, YOU MUST_WRITE_THE_TOP_VALUE HERE);// 14MHz/presc/TOP

  // clear any pending interrupts
  // STATIC_INLINE void TIMER_IntClear(TIMER_TypeDef *timer, uint32_t
  flags);TIMER_IntClear(TIMER0, TIMER_IF_OF);

  // Enable Timer IT
  //TIMER_IntEnable(TIMER_TypeDef *timer, uint32_t
  flags);TIMER_IntEnable(TIMER0, TIMER_IF_OF);

  // Enable Timer IT in
  NVICNVIC_EnableIRQ(TIMER0_IRQn);


  // *******************************
  // *     LEDsinitialization      *
  // *******************************
  BSP_LedsInit();
```

To handle the interrupt, the following function must be implemented in the program code:

```
voidTIMER0_IRQHandler(void){BSP_LedToggle(0);
      TIMER_IntClear(TIMER0, TIMER_IF_OF);//Delete TIMER flag
}
```

The function names are self-explanatory, we leave the interpretation of the above

program code to the students.

Default values used to initialize the timer.

```
#define TIMER_INIT_DEFAULT
{                                                                                \
    1,                  /* Enable timer when init complete. */                   \
    0,                 /* Stop counter during debug halt. */                     \
    timerPrescale1,         /* No prescaling. */                                 \
    timerClkSelHFPerClk,/* Select HFPER clock. */                                \
    0,                 /* Not 2x count mode. */                                  \
    0,                 /* Well ATI. */                                           \
    timerInputActionNone,/* No action on falling input edge.                    \
    */timerInputActionNone,/* No action on rising input edge. */                 \
    timerModeUp,            /* Up-counting. */                                   \
    0,                 /* Do not clear DMA requests when DMA channel is active. */\
    0,                 /* Select X2 quadrature decode mode (if used). */         \
    0,                 /* Disable one shot. */                                   \
    0                  /* Not started/stopped/reloaded by other timers. */       \
  }
```

Tasks

1. Based on what was described above, we initialize the timer0 peripheral.
2. Set the prescaler and the TOP value of the timer to generate interrupts every 1 second (change the state of one of the LEDs in the interrupt routine).
    a. What is actually the nominal value of the set frequency?
3. Set the prescaler to 1 and the timer TOP to 14000. In the interrupt routine, we solve it in software so that it changes the LED state every 1 second.