Time triggered scheduler for TMS570LS3137

Generated by Doxygen 1.8.12

Contents

Chapter 1

Introduction

The goal of this work was to develope time triggered (TT) scheduler of tasks for TMS570LS3137, compatible with existing rpp-library and FreeRTOS operating system. Time triggered scheduler organise TT tasks in a way that it's deterministic when a task is started to be executing. These times are defined offline before the launch of the scheduler.

Chapter 2

Functionality and problems

2.1 Project installation

To be able to run software properly, two projects need to be set up: a library `rpp-lib` and project `helloworld` where the TT scheduler is implemented.

2.1.1 Getting sources from git repository

To get the source codes from Git reposiotry, execute the following command:

```
git clone git@rtime.felk.cvut.cz:pes-rpp/rpp-lib -branch personal/langrfil/ttcpu
```

2.1.2 Launching the project

Project `rpp-lib` is located at:

`rpp-lib/build/tms570_hdk`

Project `helloworld` is located at:

`rpp-lib/build/tms570_hdk/apps`

Open and compile both projects in Code Composer Studio (developed with version 5.5), plug in the TMS570LSHDK and launch `helloworld` project by clicking the Debug button.

2.2 Functionality

This section describes functionality of the TT scheduler while providing references to code documentation where implementation can be reviewed.

2.2.1 TT scheduler data type

TT scheduler is implemented with data structure `tt_scheduler_t`. An array of TCBs `TCB_storage` is saved there. TCB is a structure for storing task status and information. Next, scheduler has a table of pointers on TT tasks – `TCB_table`,the scheduler table defining the order of task exectuion. One task can appear multiple times in the table. When the end of the table is reached the first task is started again.

Next, scheduler has its table of times – `task_computing_time` – where is specified when exactly every TT task in `TCB_table` has to be started. Having such times, it's also determined how maximally long can the TT task be computed. If TT task finishes before the next one should start then a time window when no task runs occurs. To utilize this time, it's filled with FreeRTOS tasks, which are handled by FreeRTOS scheduler.

2.2.2 Real-time interrupt module

To measure the time, the real-time interrupt (RTI) module is used. RTI provides timer functionality, it incorporates counters that define the timebases needed for scheduling. The TT scheduler uses Free Running Counter 0 (FRC0) with Compare Register 2 (CR2). When the FRC0 matches the value in CR2, interrupt is generated. CR2 is then incremented by value stored in Update Register 2 (UR2). This allows to generate periodic interrupts. See `prvSetupTimerInterrupt()` in port.c or `RTI_init()` to view the initialization and setting of the RTI.

2.2.3 Switching tasks

RTI CR2 generated interrupts are handled with `ttInInterrupt()`. The function checks current time and the status of the TT scheduler. Several situations can occur:

1. When a TT task runs longer that it can, i.e. next TT task should run but the previous hasn't finished yet, an error message is printed and the program freezes (ends in infinite loop).
2. It's time for another TT task to be started and currently running task is a FreeRTOS one. In such case FreeRTOS scheduler is disabled and it's switched from FreeRTOS task to the next TT task.
3. The time for the next TT task hasn't come yet, so just keep running current task.

2.2.4 TT task implementation

While TT tasks are started over and over again, the functions they call need to be implemented as an infinite loop. At the end of the loop `ttWait()` is called. It enables FreeRTOS scheduler and generates software interrupt handled with `ttYieldFreeRTOSWithinAPI()` which save context of the TT task and switches from it to FreeRTOS task. When the time comes and TT task should run again, its context is restored and it continues from the beginning of the loop. For a simple example of TT task implementation, please review `tt_task01`.

2.2.5 TT scheduler initialization

To initialize the scheduler, the size of `TCB_storage`, `task_computing_time` and `TCB_table` have to be defined with macros `NUM_TASKS` and `NUM_TABLE`. Tasks are created with `ttCreateTask()`, a pointer to an element of `TCB_storage` needs to be passed as an parameter. Scheduler table `TCB_table` and corresponding table of times `task_computing_time` have to be filled manually. Also `deadline` of the first task and `next_interrupt` have to be set. To be able to run FreeRTOS scheduler, it's necessary to initialize it too. This is done by calling edited `vTaskStartScheduler()`, see 3.1. At the end of the initialization, the first TT task in the scheduler table is started with `ttStartFirstTask()`. Please review the `scheduler_init()` where the inicialization is implemented.

2.3 Problems

In this section known problems and necesarry future work are listed.

1. There is no API, initializing the scheduler, creating the scheduler table and the definition of computing times is not user friendly.
2. It would be more suitable to implement a structure describing one record in the schedule including pointer to TCB and computing time and use array of these structures. Currently there are two separate arrays of primitives, `task_computing_time` and `TCB_table`.
3. Scheduler hasn't been tested for higher interrupt frequencies, more tasks with various periods in scheduler table, various computing times etc. In fact it hasn't been tested at all, the developement ended at the moment when it started to work somehow.
4. Using `rpp_sci_printf()` for printing cause freezing. This probably has very serious cause and should be solved ASAP. Try using `rpp_sci_printk()` until it's solved.
5. Scheduler has no support for mutexes and semaphores.
6. The initial value of compare 0 unit in RTI module, generating interrupts for FreeRTOS scheduler, is set to cause delay by 50ms. This is done because rest of the initialization of the TT scheduler which is done afterwards has to be finished before interrupt occures, otherwise it freezes. This is probably the sign of bad design.
7. Badly used `#include` and generally badly "connected" files. For example, in main.c, there is `extern PRIVILEGED_DATA TCB_t * volatile pxCurrentTCB;` which generally is not recommended but still is used in order to make the variable available in the source file.

Chapter 3

Changes in rpp-lib files

There are several files in `rpp-lib` project that had to be slightly changed to provide functionality of TT scheduler. The changes are described in this chapter. In source codes, nearby the affected lines or functions, commentaries are written. They specifies the changed behaviour and contains "TODO", so that these lines could be more easily found and reviewed. This chapter provides the description of changes in particular files.

3.1 port.c

In port.c source file the end of `xPortStartScheduler()` function is changed, calling function `vPortStartFirstTask()` is commented out. As a result, calling `vTaskStartScheduler()` in `scheduler_init()` will initialize the FreeRTOS scheduler but won't launch any task.

Changes were also made in `prvSetupTimerInterrupt()`. This function originally sets RTI module for purpose of FreeRTOS scheduler. Current implementation adds settings of the RTI module also for TT scheduler – compare unit 2 and its update register are set and interrupts it generates are enabled.

To make changes functional, tt_scheduler.h is included using relative path at the beginning of the file.

3.2 portASM.asm

Several assembly macros and functions were added. At first, `ttRESTORE_CONTEXT` and `ttSAVE_CONTEXT` macros were added. Their functionality is the same as `portRESTORE_CONTEXT` and `portSAVE_CONTEXT`, while the pointer on the top of the stack is saved to `tempTCB`. This is necesarry because whether the task is TT or FreeRTOS can be checked after saving task context. The check is done in functions called within ISR, `ttSwitchToFreeRTOS()` or `ttSwitch()`. Based on whose task's context was saved, functions assign pointer on the top of stack from `tempTCB` to either `ttCurrentTCB` or `pxCurrentTCB`.

Next there are three new functions, `ttStartFirstTask()`, `ttYieldFreeRTOSWithinAPI()` and `ttInInterrupt()`.

3.3 portmacro.h

In portmacro.h header file a macro `portYIELD_WITHIN_API()` is defined, which generates software interrupt. Please note that the interrupt is handled by function `ttYieldFreeRTOSWithinAPI()` and not by original FreeRTOS function `vPortYeildWithinAPI()`. While both schedulers use software generated interrupt, the function has to check which scheduler generated it. If it's the TT scheduler, it switches from TT task to FreeRTOS task. In case FreeRTOS task generated interrupt, it calls `vTaskSwitchContext()`, i.e. it behaves just like `vPortYeildWithinAPI()`.

3.4 sys_startup.c

In sys_startup.c source file an interrupt table `s_vim_init` is defined.

Interrupt source for the 4th channel is a compare unit 2 of RTI module, which is used to provide TT scheduler functionality. The function `ttInInterrupt()` is used as ISR.

Interrupt source for the 21st channel is a software interrupt, `ttYieldFreeRTOSWithinAPI()` is used as ISR.

To make changes functional, tt_scheduler.h is included using relative path at the beginning of the file.

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Data Structure Documentation

6.1 tt_scheduler_t Struct Reference

Time triggered scheduler data structure.

`#include <tt_scheduler.h>`

Data Fields

- uint32_t task_computing_time [NUM_TABLE]
- TCB_t * TCB_table [NUM_TABLE]
- uint32_t current_task_index
- uint32_t next_interrupt
- uint32_t deadline
- unsigned char tt_running
- TCB_t TCB_storage [NUM_TASKS]

6.1.1 Detailed Description

Time triggered scheduler data structure.

Data structure contains array of tasks (their TCBs), scheduler table, table of tasks' computing times, deadline for currently running task and time when the next interrupt will come.

6.1.2 Field Documentation

6.1.2.1 current_task_index

`uint32_t tt_scheduler_t::current_task_index`

Index to a TCB_table indexing currently running task.

6.1.2.2 deadline

```
uint32_t tt_scheduler_t::deadline
```

Time when current task has to be already finished (so at that time, a FreeRTOS task has to run). In other words it says how long time after the execution of the current task the next task has to be started.

6.1.2.3 next_interrupt

```
uint32_t tt_scheduler_t::next_interrupt
```

Time when the next interrupt will come.

6.1.2.4 task_computing_time

```
uint32_t tt_scheduler_t::task_computing_time[NUM_TABLE]
```

Array of times, each indicates how maximally long can the computation of the task at the same index in TCB_table takes.

6.1.2.5 TCB_storage

```
TCB_t tt_scheduler_t::TCB_storage[NUM_TASKS]
```

Array of TCBs.

6.1.2.6 TCB_table

```
TCB_t* tt_scheduler_t::TCB_table[NUM_TABLE]
```

Scheduler table, array of pointers on TCBs.

6.1.2.7 tt_running

```
unsigned char tt_scheduler_t::tt_running
```

Boolean indicating whether TT task is currently running.

The documentation for this struct was generated from the following file:

- tt_scheduler.h

Chapter 7

File Documentation

7.1 tt_scheduler.h File Reference

Header file for time triggered scheduler.

```
#include "rpp/rpp.h"
#include "TCB_t2.h"
```

Data Structures

- struct tt_scheduler_t

  Time triggered scheduler data structure.

Macros

- #define TT_PRINT 0
- #define REPEAT_FIRST_TASK 1
- #define ttSYS_SSIR2_REG (* ((volatile uint32_t *) 0xFFFFFFB4))
- #define ttSYS_SSIR2_SSKEY (0x8400UL)
- #define ttMACRO_YIELD() { ttSYS_SSIR2_REG = ttSYS_SSIR2_SSKEY; asm(" DSB "); asm(" ISB "); }
- #define SSIVEC (* ((volatile uint32_t *) 0xFFFFFFF4))
- #define **N2HET1_direction** (* ((volatile uint32_t *) 0xFFF7B84C))
- #define **N2HET1_dout** (* ((volatile uint32_t *) 0xFFF7B854))
- #define **PINMMR0** (* ((volatile uint32_t *) 0xFFFFEB10))
- #define **portRTI_GCTRL_REG** (* ((volatile uint32_t *) 0xFFFFFC00))
- #define **portRTI_TBCTRL_REG** (* ((volatile uint32_t *) 0xFFFFFC04))
- #define **portRTI_COMPCTRL_REG** (* ((volatile uint32_t *) 0xFFFFFC0C))
- #define **portRTI_CNT0_FRC0_REG** (* ((volatile uint32_t *) 0xFFFFFC10))
- #define **portRTI_CNT0_UC0_REG** (* ((volatile uint32_t *) 0xFFFFFC14))
- #define **portRTI_CNT0_CPUC0_REG** (* ((volatile uint32_t *) 0xFFFFFC18))
- #define **portRTI_CNT0_COMP0_REG** (* ((volatile uint32_t *) 0xFFFFFC50))
- #define **portRTI_CNT0_UDCP0_REG** (* ((volatile uint32_t *) 0xFFFFFC54))
- #define **portRTI_SETINTENA_REG** (* ((volatile uint32_t *) 0xFFFFFC80))
- #define **portRTI_CLEARINTENA_REG** (* ((volatile uint32_t *) 0xFFFFFC84))
- #define **portRTI_INTFLAG_REG** (* ((volatile uint32_t *) 0xFFFFFC88))

- #define **portRTI_CNT0_COMP1_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC58))
- #define **portRTI_CNT0_UDCP1_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC5C))
- #define **portRTI_CNT0_UC1_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC34))
- #define **portRTI_CNT0_FRC1_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC30))
- #define **portRTI_CNT0_CPUC1_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC38))
- #define **portRTI_CNT0_COMP2_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC60))
- #define **portRTI_CNT0_UDCP2_REG** (∗ ((volatile uint32_t ∗) 0xFFFFFC64))
- #define **REQENASET** (∗ ((volatile uint32_t ∗) 0xFFFFFE30))
- #define RTI_INT_MS 1
- #define RTI_TO_TICK(x) ((uint32_t) (x) ∗ ((configCPU_CLOCK_HZ / 2) / configTICK_RATE_HZ))
- #define RTI_NEXT_INTERR (RTI_TO_TICK(RTI_INT_MS))
- #define COMMON_TIME (RTI_INT_MS ∗ 1000)
- #define **TT_FALSE** 0
- #define **TT_TRUE** 1
- #define NUM_TASKS 4
- #define NUM_TABLE 5

Functions

- void tt_task01 (void ∗p)

  Function for time triggered task01.
- void tt_task02 (void ∗p)

  Function for time triggered task02.
- void tt_task03 (void ∗p)

  Function for time triggered task03.
- void tt_task04 (void ∗p)

  Function for time triggered task04.
- void task_freertos1 (void ∗p)

  Function for FreeRTOS task01.
- void task_freertos2 (void ∗p)

  Function for FreeRTOS task02.
- void task_freertos3 (void ∗p)

  Function for FreeRTOS task03.
- void scheduler_init ()

  Initialize TT scheduler. This function is called at the beginning of the program.
- void ttCreateTask (TaskFunction_t func, TCB_t ∗task_TCB, const char ∗const name, void ∗const params, const uint16_t stack_depth)

  Create time triggered task. Tasks are created in scheduler_init(). Inspired with code in tasks.c.
- void TCB_save_name (TCB_t ∗task_TCB, const char ∗const name)

  Store `name` *in* `task_TCB` *correctly. This function is called within ttCreateTask(). Copied from tasks.c.*
- void ttSwitch (void)

  Called within interrupt (generated by RTI module), decide which task should run next.
- void ttSwitchToFreeRTOS (void)

  Called within software interrupt, sets a FreeRTOS task to run.
- void ttWait ()

  Let TT task wait by calling software interrupt.
- void ttStartFirstTask (void)

  Runs the first TT task. Assembly function.
- void ttYieldFreeRTOSWithinAPI (void)

  Switch to FreeRTOS task. Assembly function.
- void ttInInterrupt (void)

Switch to task (TT or FreeRTOS). Assembly function.

- void LED_init ()

  Initialize LEDs.
- void LED_on ()

  Turn LED on.
- void LED_off ()

  Turn LED off.
- void LED_switch (void)

  Switch LED.
- void LED_error_on ()

  Turn LED_error on.
- void LED_error_off ()

  Turn LED_error on.
- void LED_error_switch ()

  Switch LED_error.
- void RTI_init ()

  DEPRECATED, initialize RTI module, currently is done in port.c.

7.1.1 Detailed Description

Header file for time triggered scheduler.

Author

langrfil@fel.cvut.cz

Date

September 2016

7.1.2 Macro Definition Documentation

7.1.2.1 COMMON_TIME

```
#define COMMON_TIME ( RTI_INT_MS * 1000 )
```

Common time for defining computing time for TT tasks, see scheduler_init().

7.1.2.2 NUM_TABLE

```
#define NUM_TABLE 5
```

The size of scheduler table `TCB_table`.

7.1.2.3 NUM_TASKS

```
#define NUM_TASKS 4
```

Number of tasks.

7.1.2.4 REPEAT_FIRST_TASK

```
#define REPEAT_FIRST_TASK 1
```

First task is once repeated in scheduler table yes(1) or no(0).

7.1.2.5 RTI_INT_MS

```
#define RTI_INT_MS 1
```

Every RTI_INT_MS milliseconds interrupt occurs.

7.1.2.6 RTI_NEXT_INTERR

```
#define RTI_NEXT_INTERR ( RTI_TO_TICK( RTI_INT_MS ) )
```

Get how many RTI ticks it takes between two interrupts.

7.1.2.7 RTI_TO_TICK

```
#define RTI_TO_TICK(
              x ) ( ( uint32_t ) ( x ) * (( configCPU_CLOCK_HZ / 2 ) / configTICK_RATE_HZ) )
```

Transform from milliseconds to RTI ticks.

7.1.2.8 SSIVEC

```
#define SSIVEC ( * ( ( volatile uint32_t * ) 0xFFFFFFF4 ) )
```

Contains information about software interrupts (not used).

7.1.2.9 TT_PRINT

```
#define TT_PRINT 0
```

Status prints during interrupts enabled(1) or disabled(0).

7.1.2.10 ttMACRO_YIELD

```
#define ttMACRO_YIELD( ) { ttSYS_SSIR2_REG = ttSYS_SSIR2_SSKEY; asm( " DSB " ); asm( " ISB "
); }
```

Generates software interrupt2.

7.1.2.11 ttSYS_SSIR2_REG

```
#define ttSYS_SSIR2_REG ( * ( ( volatile uint32_t * ) 0xFFFFFFB4 ) )
```

System software interrupt request 2 register.

7.1.2.12 ttSYS_SSIR2_SSKEY

```
#define ttSYS_SSIR2_SSKEY ( 0x8400UL )
```

A specific value generating software interrupt2 when written to `ttSYS_SSIR2_REG`.

7.1.3 Function Documentation

7.1.3.1 LED_error_off()

```
void LED_error_off ( )
```

Turn LED_error on.

LED on N2HET1[29] is turned off.

```
499 {
500     /* Turn LED_error off. */
501
502     /* Turn off LED on N2HET1[29] */
503     N2HET1_dout &= 0xDFFFFFFF;
504 }
```

7.1.3.2 LED_error_on()

```
void LED_error_on ( )
```

Turn LED_error on.

LED on N2HET1[29] is turned on.

```
491 {
492     /* Turn LED_error on. */
493
494     /* Turn on LED on N2HET1[29] */
495     N2HET1_dout |= 0x20000000;
496 }
```

7.1.3.3 LED_error_switch()

```
void LED_error_switch ( )
```

Switch LED_error.

LED on N2HET1[29] is switched (turned off when it's currently turned on and vice versa).

```
507 {
508     /* Switch LED_error. */
509
510     /* Turn on or off LED on N2HET1[29] */
511     if( (N2HET1_dout & 0x20000000) > 0)
512         N2HET1_dout &= 0xDFFFFFFF;
513     else
514         N2HET1_dout |= 0x20000000;
515 }
```

7.1.3.4 LED_init()

```
void LED_init ( )
```

Initialize LEDs.

Initialize LEDs on N2HET1[0] and N2HET1[29].

```
454 {
455     /* Initialize LEDs. */
456
457     /* IOMM settings set by default to N2HET1[0] */
458
459     /* Set control register address to control LED on N2HET1[29], it's on PINMMR0[18] */
460     PINMMR0 = 0x00040000;
461     /* Set direction of N2HET1[0] and [29] to output. */
462     N2HET1_direction |= 0x20000001;
463 }
```

7.1.3.5 LED_off()

```
void LED_off ( )
```

Turn LED off.

LED on N2HET1[0] is turned off.

```
473 {
474     /* Turn LED off. */
475
476     N2HET1_dout &= 0xFFFFFFFE;
477 }
```

7.1.3.6 LED_on()

```
void LED_on ( )
```

Turn LED on.

LED on N2HET1[0] is turned on.

```
466 {
467     /* Turn LED on. */
468
469     N2HET1_dout |= 0x00000001;
470 }
```

7.1.3.7 LED_switch()

```
void LED_switch (
              void  )
```

Switch LED.

LED on N2HET1[0] is switched (turned off when it's currently turned on and vice versa).

```
480 {
481     /* Switch LED. */
482
483     /* Turn on or off LED on N2HET1[0] */
484     if( (N2HET1_dout & 0x00000001) > 0)
485         N2HET1_dout &= 0xFFFFFFFE;
486     else
487         N2HET1_dout |= 0x00000001;
488 }
```

7.1.3.8 RTI_init()

```
void RTI_init ( )
```

DEPRECATED, initialize RTI module, currently is done in port.c.

Define frequency of generating interrupts by setting FRC0, compare blocks 0 (uses FreeRTOS) and 2 (uses TT scheduler) and their update registers.

```
518 {
519     /* DEPRECATED, initialize RTI module, currently is done in port.c. */
520     /* Set interrupt on match FRC0 with compare unit 2, needed by TT scheduler. */
521
522     /* Disable timer 0. */
523     portRTI_GCTRL_REG &= 0xFFFFFFFEUL;
524
525     /* Use the internal counter. */
526     portRTI_TBCTRL_REG = 0x00000000U;
527
528     /* All compare registers will be compared with RTIFRC0 counter. */
529     portRTI_COMPCTRL_REG = 0x00000000U;
530
531     /* Initialise the counter and the prescale counter registers. */
532     portRTI_CNT0_UC0_REG =  0x00000000U;
533     portRTI_CNT0_FRC0_REG =  0x00000000U;
534
535     /* Set Prescalar for RTI clock. */
536     portRTI_CNT0_CPUC0_REG = 0x00000001U;
537     portRTI_CNT0_COMP2_REG = RTI_NEXT_INTERR;
538     portRTI_CNT0_UDCP2_REG = RTI_NEXT_INTERR;
539
540     /* Clear interrupts. */
541     portRTI_INTFLAG_REG =  0x0007000FU;
542     portRTI_CLEARINTENA_REG = 0x00070F0FU;
543
544     /* Enable also interrupt channel 4 in VIM */
545     REQENASET |= 0x00000010;
546
547     /* Enable the compare 2 interrupt. */
548     portRTI_SETINTENA_REG = 0x00000004U;
549
550     /* Enable timer 0. */
551     portRTI_GCTRL_REG |= 0x00000001U;
552 }
```

7.1.3.9 scheduler_init()

```
void scheduler_init ( )
```

Initialize TT scheduler. This function is called at the beginning of the program.

TT tasks are created, scheduler table and time table are set, current TT TCB and temporary TCB are initialized. Also FreeRTOS tasks are created and scheduler initialized. In the end the first TT task in scheduler table is started.

```
150 {
151     /* Initialize TT scheduler. This function is called at the beginning of the program. */
152 
153     /* Index to the storage of TCBs (an array of TCBs). */
154     int storage_index = 0;
155     /* Index to the scheduler table (array of pointers to TCBs stored in storage). */
156     int table_index = 0;
157 
158     /* Set pointer on temporary TCB. */
159     tempTCB_p = &tempTCB;
160 
161     /* Initialize LEDs. */
162     LED_init();
163 
164     /************************************** CREATE TT TASKS ****************************************
    ****/
165 
166     /* Memory for TCBs is allocated in scheduler. */
167     ttCreateTask(tt_task01, &(scheduler.TCB_storage[storage_index]), "
    tt_task01", (void*) NULL, (uint16_t) 512);
168     /* Initialize scheduler table and computing time for the task. */
169     scheduler.task_computing_time[table_index] = RTI_TO_TICK(
    COMMON_TIME );
170     scheduler.TCB_table[table_index] = &(scheduler.TCB_storage[storage_index]);
171     storage_index++;
172     table_index++;
173 
174     /* Memory for TCBs is allocated in scheduler. */
175     ttCreateTask(tt_task02, &(scheduler.TCB_storage[storage_index]), "
    tt_task02", (void*) NULL, (uint16_t) 512);
176     /* Initialize scheduler table and computing time for the task. */
177     scheduler.task_computing_time[table_index] = RTI_TO_TICK(
    COMMON_TIME * 2 );
178     scheduler.TCB_table[table_index] = &(scheduler.TCB_storage[storage_index]);
179     storage_index++;
180     table_index++;
181 
182     /* Put the first task on the third place in scheduler table again. */
183     #if REPEAT_FIRST_TASK == 1
184     scheduler.TCB_table[table_index] = &(scheduler.TCB_storage[0]);
185     scheduler.task_computing_time[table_index] = scheduler.
    task_computing_time[0];
186     table_index++;
187     #endif
188     /* Memory for TCBs is allocated in scheduler. */
189     ttCreateTask(tt_task03, &(scheduler.TCB_storage[storage_index]), "
    tt_task03", (void*) NULL, (uint16_t) 512);
190     /* Initialize scheduler table and computing time for the task. */
191     scheduler.task_computing_time[table_index] = RTI_TO_TICK(
    COMMON_TIME * 3 );
192     scheduler.TCB_table[table_index] = &(scheduler.TCB_storage[storage_index]);
193     storage_index++;
194     table_index++;
195 
196     /* Memory for TCBs is allocated in scheduler. */
197     ttCreateTask(tt_task04, &(scheduler.TCB_storage[storage_index]), "
    tt_task04", (void*) NULL, (uint16_t) 512);
198     /* Initialize scheduler table and computing time for the task. */
199     scheduler.task_computing_time[table_index] = RTI_TO_TICK(
    COMMON_TIME * 4 );
200     scheduler.TCB_table[table_index] = &(scheduler.TCB_storage[storage_index]);
201     storage_index++;
202     table_index++;
203 
204     /*************************************************************************************************
    ****/
205 
206     //TODO check that RTI tick is a common divisor of all computing times.
207 
208     /* Set the first TT task in TT scheduler as currently running. */
209     scheduler.tt_running = TT_TRUE;
210     scheduler.current_task_index = 0;
```

```
211     ttCurrentTCB = scheduler.TCB_table[0];
212
213     /* Set tempTCB TopOfStack (ToS) to TT current TCB, because tempTCB ToS is used for restoring context
        when running first TT task. */
214     tempTCB_p->pxTopOfStack = ttCurrentTCB->pxTopOfStack;
215
216     /* Set deadline of the first TT task. */
217     scheduler.deadline = scheduler.task_computing_time[0];
218
219     /* Set time when next interrupt will come. */
220     scheduler.next_interrupt = RTI_NEXT_INTERR;
221
222     /* Create tasks for FreeRTOS scheduler. */
223     xTaskCreate(task_freertos1, "task_freertos1", 512, ( void * ) NULL, 2, NULL);
224     xTaskCreate(task_freertos2, "task_freertos2", 512, ( void * ) NULL, 3, NULL);
225     xTaskCreate(task_freertos3, "task_freertos3", 512, ( void * ) NULL, 1, &FRT3_handle);
226
227     /* Start FreeRTOS scheduler WITHOUT launching any task - vPortStartFirstTask() has to be commented in
        port.c, in xPortStartScheduler(). */
228     //TODO also initializing RTI in prvSetupTimerInterrupt was changed.
229     vTaskStartScheduler();
230
231     /* Disable FreeRTOS context switching. */
232     vTaskSuspendAll();
233
234     /* No initialization of RTI is done here, it was already done when vTaskStartScheduler() was called (in
        port.c, in prvSetupTimerInterrupt()). */
235     //RTI_init();
236
237     rpp_sci_printk( ( const char * ) "Just about to run ttStartFirstTask().\r\n");
238
239     /* Assembler function in portASM.asm. restore context of TT task saved in tempTCB. */
240     ttStartFirstTask();
241
242     /* Should not get here. */
243 }
```

7.1.3.10 task_freertos1()

```
void task_freertos1 (
            void * p )
```

Function for FreeRTOS task01.

This function runs as FreeRTOS task01. On start the function is delayed for 500ms, then it resumes task03. Next, in infinite loop it prints its name, switch off error LED and delays for 1000ms.

Parameters

p	Not used.

```
109 {
110     TickType_t xLastWakeTime = xTaskGetTickCount();
111     /* Wait 500ms. */
112     vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS(500) );
113
114     vTaskResume(FRT3_handle);
115     while(1)
116     {
117         rpp_sci_printk( ( const char * ) "\nFreeRTOS task1!\r\n");
118         LED_error_off();
119
120         /* Wait 1000ms. */
121         vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS(1000) );
122     }
123 }
```

7.1.3.11 task_freertos2()

```
void task_freertos2 (
            void * p )
```

Function for FreeRTOS task02.

This function runs as FreeRTOS task02. In infinite loop it prints its name, switch on error LED and delays for 1000ms.

Parameters

p	Not used.

```
126 {
127     TickType_t xLastWakeTime = xTaskGetTickCount();
128     while(1)
129     {
130         rpp_sci_printk( ( const char * ) "\nFreeRTOS task2!\r\n");
131         LED_error_on();
132 
133         /* Wait 1000ms. */
134         vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS(1000) );
135     }
136 }
```

7.1.3.12 task_freertos3()

```
void task_freertos3 (
            void * p )
```

Function for FreeRTOS task03.

This function runs as FreeRTOS task03. It prints its name + "before suspend", then waits. Upon resuming it prints its name + "after suspend" and goes into infinite loop.

Parameters

p	Not used.

```
139 {
140     rpp_sci_printk( ( const char * ) "\nFreeRTOS task3 before suspend!\r\n");
141     vTaskSuspend(NULL);
142     rpp_sci_printk( ( const char * ) "\nFreeRTOS task3 after suspend!\r\n");
143     while(1)
144     {
145 
146     }
147 }
```

7.1.3.13 TCB_save_name()

```
void TCB_save_name (
            TCB_t * task_TCB,
            const char *const name )
```

Store `name` in `task_TCB` correctly. This function is called within ttCreateTask(). Copied from tasks.c.

Parameters

task_TCB	Pointer to a `TCB_t` where the `name` will be stored.
name	Name of the task.

```
277 {
278     /* Store @p name in @task_TCB correctly. This function is called within ttCreateTask(). Copied from
        tasks.c. */
279
280     UBaseType_t x;
281     for( x = ( UBaseType_t ) 0; x < ( UBaseType_t ) configMAX_TASK_NAME_LEN; x++ )
282     {
283         task_TCB->pcTaskName[ x ] = name[ x ];
284         /* Don't copy all configMAX_TASK_NAME_LEN if the string is shorter than
285         configMAX_TASK_NAME_LEN characters just in case the memory after the
286         string is not accessible (extremely unlikely). */
287         if( name[ x ] == 0x00 ) break;
288         else mtCOVERAGE_TEST_MARKER();
289     }
290     /* Ensure the name string is terminated in the case that the string length
291     was greater or equal to configMAX_TASK_NAME_LEN. */
292     task_TCB->pcTaskName[ configMAX_TASK_NAME_LEN - 1 ] = '\0';
293 }
```

7.1.3.14 tt_task01()

```
void tt_task01 (
              void * p )
```

Function for time triggered task01.

This function runs as time triggered task01. In infinite loop it switches LED, prints its name and waits.

Parameters

p	Not used.

```
53 {
54     while(1)
55     {
56         /* Switch LED. */
57         LED_switch();
58
59         rpp_sci_printk( ( const char * ) "\nTASK_01\r\n");
60
61         /* Let task wait. */
62         ttWait();
63     }
64 }
```

7.1.3.15 tt_task02()

```
void tt_task02 (
              void * p )
```

Function for time triggered task02.

This function runs as time triggered task02. In infinite loop it switches LED, prints its name and waits.

Parameters

p	Not used.

```
67 {
68     while(1)
69     {
70         /* Switch LED. */
71         LED_switch();
72 
73         rpp_sci_printk( ( const char * ) "\nTASK_02\r\n");
74 
75         /* Let task wait */
76         ttWait();
77     }
78 }
```

7.1.3.16 tt_task03()

```
void tt_task03 (
            void * p )
```

Function for time triggered task03.

This function runs as time triggered task03. In infinite loop it switches LED, prints its name and waits.

Parameters

p	Not used.

```
81 {
82     while(1)
83     {
84         /* Switch LED. */
85         LED_switch();
86 
87         rpp_sci_printk( ( const char * ) "\nTASK_03\r\n");
88 
89         /* Let task wait. */
90         ttWait();
91     }
92 }
```

7.1.3.17 tt_task04()

```
void tt_task04 (
            void * p )
```

Function for time triggered task04.

This function runs as time triggered task04. In infinite loop it switches LED, prints its name and waits.

Parameters

p	Not used.

```
95 {
96     while(1)
97     {
98         /* Switch LED. */
99         LED_switch();
100
101         rpp_sci_printk( ( const char * ) "\nTASK_04\r\n");
102
103         /* Let task wait. */
104         ttWait();
105     }
106 }
```

7.1.3.18 ttCreateTask()

```
void ttCreateTask (
              TaskFunction_t func,
              TCB_t * task_TCB,
              const char *const name,
              void *const params,
              const uint16_t stack_depth )
```

Create time triggered task. Tasks are created in scheduler_init(). Inspired with code in tasks.c.

Function allocates memory for the stack and compute the top of the stack making it ready to be started. TCB is returned via the pointer obtained as parameter.

Parameters

func	Pointer to a function the task will do.
task_TCB	Pointer to a `TCB_t` of the task (usually points to an element of `TCB_t` storage in `tt_scheduler_t`).
name	Name of the task.
params	Pointer to parameters given to the function `func`.
stack_depth	Depth of the stack for the task.

```
246 {
247     /* Create time triggered task. Tasks are created in scheduler_init(). Inspired with code in tasks.c. */
248
249     StackType_t * topOfStack;
250
251     /* Stack used by task is allocated with malloc. */
252     task_TCB->pxStack = ( StackType_t * ) pvPortMallocAligned( ( ( ( size_t ) stack_depth ) * sizeof(
      StackType_t ) ), NULL ); /*lint !e961 MISRA exception as the casts are only redundant for some ports. */
253
254     /* Check that stack was allocated correctly. */
255     if ( task_TCB->pxStack == NULL)
256     {
257         rpp_sci_printk( ( const char * ) "ERROR: Stack == NULL\r\n");
258     }
259
260     /* Calculate the top of stack */
261     topOfStack = task_TCB->pxStack + ( stack_depth - ( uint16_t ) 1 );
262     topOfStack = ( StackType_t * ) ( ( ( portPOINTER_SIZE_TYPE ) topOfStack ) & ( ~( (
      portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK ) ) ); /*lint !e923 MISRA exception.  Avoiding casts between pointers an
       integers is not practical.  Size differences accounted for using portPOINTER_SIZE_TYPE type. */
263     /* Check the alignment of the calculated top of stack is correct. */
264     // TODO probably can be erased.
265     configASSERT( ( ( ( portPOINTER_SIZE_TYPE ) topOfStack & ( portPOINTER_SIZE_TYPE )
      portBYTE_ALIGNMENT_MASK ) == 0UL ) );
266
267     /* Save name of task to TCB. */
268     TCB_save_name(task_TCB, name);
269
270     /* Initialize TopOfStack like it was running and then suspended. Using FreeRTOS function. */
271     task_TCB->pxTopOfStack = pxPortInitialiseStack( topOfStack, func, params );
272
```

```
273     /* Task is not added to any list, no priority is set, no notifications. */
274 }
```

7.1.3.19 ttInInterrupt()

```
void ttInInterrupt (
             void  )
```

Switch to task (TT or FreeRTOS). Assembly function.

Called on RTI compare 2 interrupt. It saves context of the current task, clear interrupt flag, call `ttSwitch()` which either pick next task (TT or FreeRTOS) that should run or keep current task and runs it by restoring its context.

7.1.3.20 ttStartFirstTask()

```
void ttStartFirstTask (
             void  )
```

Runs the first TT task. Assembly function.

Assembly function defined in portASM.asm. It's used at the end of scheduler_init() and it runs the first TT task. It just restores context of the task `tempTCB` is pointing to.

7.1.3.21 ttSwitch()

```
void ttSwitch (
             void  )
```

Called within interrupt (generated by RTI module), decide which task should run next.

This function is called within `ttInInterrupt()` assembly function in portASM.asm, which is called on interrupt from RTI compare unit 2. It's called right after the context of currently running task is saved, it selects next task to run, which is then restored (see ttInInterrupt in portASM.asm).

Function determines whether new TT task should run or current task should continue in running or current TT task runs too long.

```
296 {
297     /* Called within interrupt (generated by RTI module), decide which task should run next. */
298
299     /* Save time of current interrupt to local variable. */
300     uint32_t current_interrupt = scheduler.next_interrupt;
301
302     /* Compute time of the next interrupt. */
303     scheduler.next_interrupt += RTI_NEXT_INTERR;
304
305     if ( current_interrupt == scheduler.deadline )
306     {
307         /* New TT task should be started now. */
308
309         if ( scheduler.tt_running == TT_FALSE )
310         {
311             /* TT task is NOT currently running, so FreeRTOS task is. */
312
313             /* Suspend FreeRTOS context switching. */
314             vTaskSuspendAll();
315
316             /* Save tempTCB TopOfStack (ToS) pointer to FreeRTOS current TCB. */
317             pxCurrentTCB->pxTopOfStack = tempTCB_p->pxTopOfStack;
318
```

```
319             /* Compute new index to scheduler table. */
320             scheduler.current_task_index = ( scheduler.
    current_task_index + 1 ) % NUM_TABLE;
321
322             /* Set deadline for the task that is going to be started. */
323             scheduler.deadline = current_interrupt + scheduler.
    task_computing_time[ scheduler.current_task_index ];
324
325             /* Set pointer on new current task to TT current TCB. */
326             ttCurrentTCB = scheduler.TCB_table[ scheduler.
    current_task_index ];
327
328             /* Set tempTCB ToS, which context is going to be restored, to TT current TCB ToS. */
329             tempTCB_p->pxTopOfStack = ttCurrentTCB->pxTopOfStack;
330
331             /* Set TT running. */
332             scheduler.tt_running = TT_TRUE;
333
334             #if TT_PRINT == 1
335             rpp_sci_printk( ( const char * ) "\nSwitching from FreeRTOS to TT\r\n");
336             rpp_sci_printk( ( const char * ) "--------------------------------------------\r\n");
337             rpp_sci_printk( ( const char * ) "scheduler.current_task_index: %d\r\n", scheduler.
    current_task_index);
338             rpp_sci_printk( ( const char * ) "scheduler.deadline: %u\r\n", scheduler.
    deadline);
339             rpp_sci_printk( ( const char * ) "ttCurrentTCB->pcTaskName: %s\r\n", ttCurrentTCB->pcTaskName);
340             rpp_sci_printk( ( const char * ) "--------------------------------------------\r\n");
341             #endif
342         }
343         else
344         {
345             /* TT task is currently running. */
346
347             /* ERROR, old task is still running but should be finished already. */
348
349             rpp_sci_printk( ( const char * ) "\nERROR: TT task hasn't finished on time, freeze.\r\n");
350
351             while(1);
352         }
353     }
354     else
355     {
356         /* No TT task should be started now. */
357
358         if ( scheduler.tt_running == TT_FALSE )
359         {
360
361             /* TT task finished and new one should not run yet, so FreeRTOS scheduler is running, do
     nothing. */
362
363             #if TT_PRINT == 1
364             rpp_sci_printk( ( const char * ) "\nNo TT task should run now, keep FreeRTOS task.\r\n");
365             #endif
366         }
367         else
368         {
369
370             /* TT task is still running and has its time, let it go, do nothing. */
371
372             #if TT_PRINT == 1
373             rpp_sci_printk( ( const char * ) "\nLet TT task run.\r\n");
374             #endif
375         }
376     }
377 }
```

7.1.3.22 ttSwitchToFreeRTOS()

```
void ttSwitchToFreeRTOS (
             void  )
```

Called within software interrupt, sets a FreeRTOS task to run.

This function is called within `ttYieldFreeRTOSWithinAPI()` assembly function in portASM.asm, which is called on software interrupt (SI). Like the ttSwitch(), it's called between saving and restoring context. It checks if it should switch to next FreeRTOS task from either TT task or FreeRTOS task (while both schedulers use the one SI) i.e. whether the currently saved context to `tempTCB` is TT task or FreeRTOS task and then selects next FreeRTOS task to be restored.

```
380 {
381     /* Called within software interrupt, sets a FreeRTOS task to run. */
382 
383     if ( scheduler.tt_running == TT_FALSE )
384     {
385         /* FreeRTOS generated this interrupt. */
386 
387         /* Assign FreeRTOS current TCB the top of stack of tempTCB. */
388         pxCurrentTCB->pxTopOfStack = tempTCB_p->pxTopOfStack;
389 
390         /* Call FreeRTOS function for switching context. */
391         vTaskSwitchContext();
392     }
393     else
394     {
395         /* TT scheduler generated this interrupt. */
396 
397         /* Set that TT task is not currently running. */
398         scheduler.tt_running = TT_FALSE;
399 
400         /* Assign TT current TCB the top of stack of tempTCB. */
401         ttCurrentTCB->pxTopOfStack = tempTCB_p->pxTopOfStack;
402 
403         /* Enable FreeRTOS context switching. */
404         // TODO maybe a problem, can be called within ISR? So far it seems it can.
405         xTaskResumeAll();
406 
407         #if TT_PRINT == 1
408         rpp_sci_printk( ( const char * ) "\nSwitching from TT task to FreeRTOS task\r\n");
409         #endif
410     }
411 }
```

7.1.3.23 ttWait()

```
void ttWait ( )
```

Let TT task wait by calling software interrupt.

Service routine of SI `ttYieldFreeRTOSWithinAPI()` switches to FreeRTOS task.

```
414 {
415     /* Let TT task wait by calling software interrupt. */
416 
417     #if TT_PRINT == 1
418     rpp_sci_printk( ( const char * ) "TT task finished.\r\n");
419     #endif
420 
421     /* Generate software interrupt. */
422     ttMACRO_YIELD();
423 }
```

7.1.3.24 ttYieldFreeRTOSWithinAPI()

```
void ttYieldFreeRTOSWithinAPI (
             void  )
```

Switch to FreeRTOS task. Assembly function.

Called on software interrupt when TT task waits after `ttWait()` is called or when one FreeRTOS task should run instead of the other. It saves context of the current task, clear interrupt flag, call `ttSwitchToFreeRTOS()` which set a FreeRTOS task as currently running and runs it by restoring its context.

Chapter 8

Conclusion

This document described functionality and implementation of TT scheduler. Project implements data structure for the scheduler and functions for its initialization, TT task creation and interrupt-driven task switching. However, there is a lot of future work. No tests were done, so project has to be tested with various tasks, times and periods. Project is not ready for mutexes and semaphores. Also user-friendly API for scheduler initialisation should be done. Overall status of the project is very work-in-progress.