Basic Real-Time Operating System

targeting the ARMv7-M architecture

ROS01, Rotterdam December 22, 2019

Student:
Nick van Endhoven
0998831hr.nl
Breda

Student:
Youri Klaassens
0996211@hr.nl
Zwaag

Contents

Version history

Version	Date	Change(s)	Note
0.1	11-30-2019	Initial document	Created version history, introduction, acknowledgement and appendix.
0.2	12-07-2019	Minor changes in Appendix A.1	Documented the first assignment (toggling LEDs)
0.3	12-08-2019	A figure of the CC3220s has been added in the introduction. The compiler version is added aswell	
0.4	12-14-2019	Added first part of assignment 2 and documented it.	
0.5	12-16-2019	Finished the second part of assignment 2 and started on assignment 3	Two minor changes in a code Listing detected while documenting them.
0.6	12-21-2019	Finished describing assignment 3	
0.7	12-22-2019		Added logic analyzer verification for assignment two and three. Fixed bug in Listing 7 which was found when using the logic analyzer

Table 1: Overview of the different versions

1 Introduction

For the Real-time Operating Systems course (ROS01) taught at Rotterdam University of Applied Science, the authors had to implement a scheduler for a Real-time Operating System developed by one lecturers. Because these types of programming issues like implementing a scheduler require the programmer to be able to program at a low level and it cannot be assumed that every student following this course is familiar with low level progamming (both in the C programming language and assembler), this course contains multiple assingments to bridge this gap. The code has been flashed and tested on the CC3220s development board (Figure 1). The compiler used is TI v18.12.2.LTS.
What's worth mentioning is that some code snippets in this document make a function call to `delay_1sec()`. Because this is used quite a few times and redundant to have multiple definitions in this document its implementation can be seen in Appendix A.1.

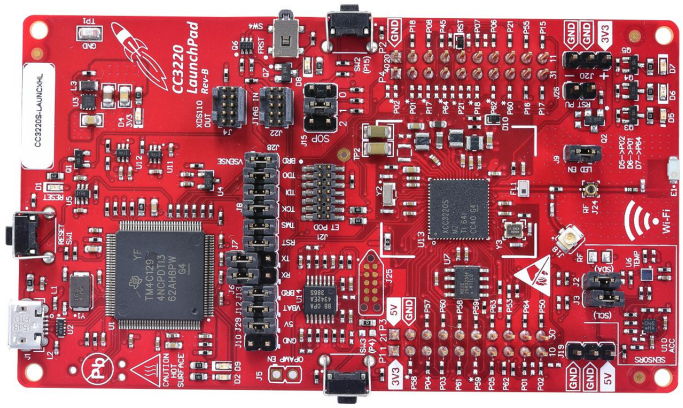

Figure 1: The CC3220s development board used during labs

2 Toggling LEDs bottom-up

The purpose of the first assignment is to become familiar with low level programming. This is done by toggling LEDs using different levels of abstraction. The sequence of this "LED show" can be seen in Table 2. Between every sequence should be a delay of approximately 1 second.

Green	*Yellow*	*Red*
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Table 2: Order of visualisation of different LEDs

As said earlier, these "LED show" should be programmed on three different levels of abstraction. The first one is no abstraction at all using a programming technique called Direct Register Manipulation (DRM) [4]. The second implementation should make use Driverlib. The third implementation should make use of the Texas Instruments (TI) Driver.

2.1 Direct Register Manipulation

Listing 1 shows the source code to toggle LEDs according to Table 2. The reader may wonder why the implementation of `delay_1sec()` is missing. This is because it is a common function used in lots of code snippets throughout this document. For the implementation details see Appendix A.1. Now follows an explanation about interesting lines of code. Line 15 enables the GPIOA peripheral during run mode (see Figure 2). This program does not enter sleep mode or deep sleep mode. Setting bit 8 and bit 16 (Figure 2) does not make sense.

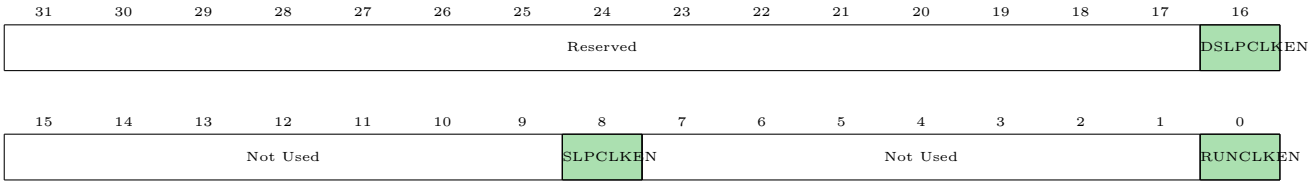

Figure 2: `GPIOOCLKEN` register for the CC3220s

The behavior of the pins being used must be configured. The assignment only requires the use of GPIO9, GPIO10 and GPIO11 because the built-in LEDs are routed to these pins. Configuration for these pins are done in Line 17 up to and including Line 19. The value $0x60_{16}$ is written to the `GPIO_PAD_CONFIG_x` register where x is 9, 10 and 11. It affects the second nibble of the register. Since only bit 1 and bit 2 of the nibble are affected it does not set the bit in the `open drain` field (Figure 3). Writing 001_2 to the `DRIVE STRENGTH` field means that the related GPIO pin will drive 6 mA.

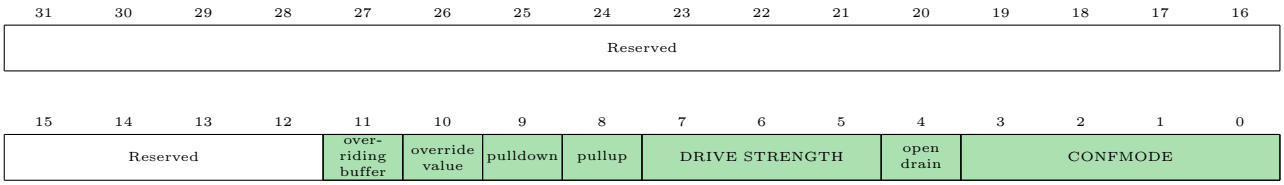

Figure 3: `GPIO_PAD_CONFIG_x` register for the CC3220s

A GPIO pin is either input or output. Writing a 1 to the `GPIO_DIR` register (Figure 4) configures a GPIO pin as output pin. Writing a 0 to the `GPIO_DIR` register configures a GPIO pin as input pin. Because this program only needs GPIO9, GPIO10 and GPIO11 the program needs to write a 1 to bit 1, bit 2 and bit 3 respectively. 00001110_2 is $0x0E_{16}$ in hexadecimal representation. That is why the program writes $0x0E_{16}$ to the `GPIO_DIR` register at Line 21.

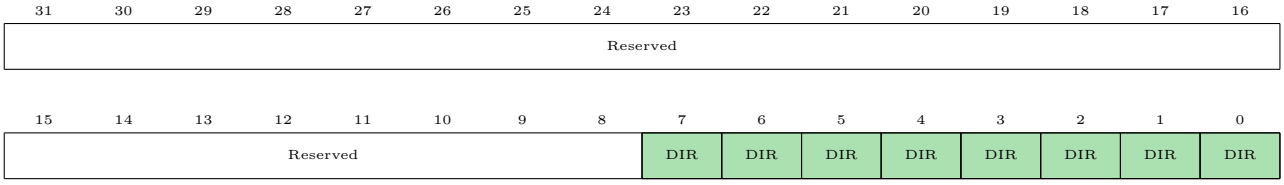

Figure 4: `GPIO_DIR` register for the CC3220s

Setting the output pin to logic high or logic low requires a little extra explanation. There is a mask which should be added to the base address plus the address of the `GPIO_DATA` register. This prevents software from a read-modify-write operation. A change in logic level of an GPIO output pin is done in a single cycle. The bits one would like to change should be shifted 2 positions to the right and added to base address + the `GPIO_DATA` offset. Line 22 turns the three LEDs off by writing 00000000_2 or $0x00_{16}$ to the `GPIO_DATA` register. However, because of the mask added to the address only bit 1, bit 2 and bit 3 are affected.

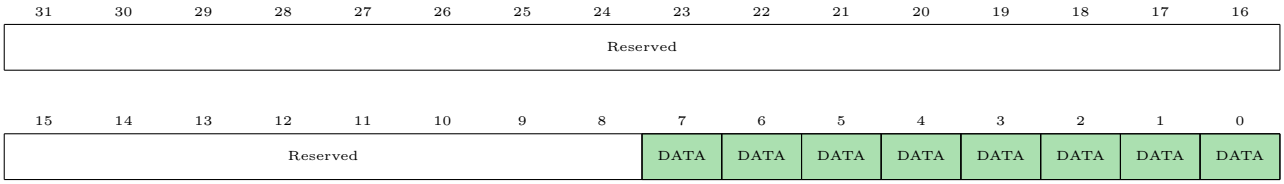

Figure 5: `GPIO_DATA` register for the CC3220s

The following things happen in an infinite loop. Line 28 in Listing 1 write a variable `index` to `GPIO_DATA` register. This value is shifted 1 position to the left because the first LED is positioned at GPIO9 and not at GPIO8. Line 29 increments `index`. Variable `index` can hold $0 \leq index < 16$ (although $0 \leq index < 8$ would be sufficient). The second operand of the modulo operator does not really matter as long as it is a multiple of 2^3. Line 30 makes a call to `delay_1sec()`. The function returns after 1 second since it is a busy-wait like implementation where the CPU just burns CPU cycles.

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include "register_def.h"
4
5  #include "inc\hw_memmap.h"
6  #include "inc\hw_gpio.h"
7  #include "inc\hw_apps_rcm.h"
8  #include "inc\hw_ocp_shared.h"
9
10 /* Function delay_1sec() used to be here */
11
12 int main(void)
13 {
14
15     HWREG(ARCM_BASE + APPS_RCM_O_GPIO_A_CLK_GATING) = 0x01;
16
17     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_9) = 0x60;
18     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_10) = 0x60;
19     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_11) = 0x60;
20
21     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DIR) = 0x0E;
22     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = 0x00;
23
24     int index = 0;
25
26     while(1)
27     {
28         HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = (index << 1);
29         index = (index + 1) % 16;
30         delay_1sec();
31     }
32
33     return 0;
34 }
```

Listing 1: Toggling LEDs according to Table 2 using DRM programming technique

2.2 Driverlib

Driverlib is a library which provides access to peripherals. The advantage of this is that the programmer does not need to know base addresses and offsets in order to access special function registers. Line 17 in Listing 2 enables the `GPIOA1` peripheral by enabling a clock signal to the peripheral. Line 19 up to and including Line 21 configures the GPIO9, GPIO10 and GPIO11 pin characteristics. The pins can use now a maximum of 2 mA per pin. Note that the function accepts a macro which contains the physical pin number instead of the logical pin number. Line 23 up to and including Line 25 configures GPIO9, GPIO10 and GPIO11 to an output pin. Line 27 up to and including Line 29 turns the three LEDs off by default.
Line 35 writes the value `switcher` to the `GPIO_DATA` register but only `GPIO_PIN_1`, `GPIO_PIN_2` and `GPIO_PIN_3` (which are GPIO9, GPIO10 and GPIO11) are sensitive for this value change. Line 36 makes a function call to `delay_1sec()` which will return to the caller after 1 second.

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include "register_def.h"
4
5  #include "gpio.h"
6  #include "pin.h"
7  #include "prcm.h"
8
9  /* GPIO 9 is PIN_64 is (red) */
10 /* GPIO 10 is PIN_2 is (green) */
11 /* GPIO 11 is PIN_1 is (yellow) */
12
13 /* Function delay_1sec() used to be here */
14
15 int main(void)
16 {
17     PRCMPeripheralClkEnable(PRCM_GPIOA1  ,PRCM_RUN_MODE_CLK);
18
19     PinTypeGPIO(PIN_64, PIN_STRENGTH_2MA, false);    /* Red LED is push-pull */
20     PinTypeGPIO(PIN_01, PIN_STRENGTH_2MA, false);    /* Yellow LED is push-pull */
21     PinTypeGPIO(PIN_02, PIN_STRENGTH_2MA, false);    /* Green LED is push-pull */
22
23     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_2, GPIO_DIR_MODE_OUT);
24     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_3, GPIO_DIR_MODE_OUT);
25     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_1, GPIO_DIR_MODE_OUT);
26
27     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_2, ~GPIO_PIN_2);  /* Turn yellow LED off */
28     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_3, ~GPIO_PIN_3);  /* Turn green LED off */
29     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1, ~GPIO_PIN_1);  /* Turn red LED off */
30
31     int switcher = 0;
32
33     while(1)
34     {
35         GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, switcher);
36         delay_1sec();
37
38         switcher = (switcher + 1) % 16;
39     }
40 }
```

Listing 2: Toggling LEDs according to Table 2 using Driverlib library

2.3 TI Driver

TI Driver is a library which provides access to peripherals. The advantage of this is that the programmer does not need to know base addresses and offsets in order to access special function registers. Line 4 in Listing 3 enables the GPIOA peripheral. Line 7 up to and including Line 9 configures the GPIO pins wired to the LEDs as output pins. Line 12 up to and including Line 14 turns the LEDs off by default.
Line 20 up to and including Line 22 set GPIO9, GPIO10 and GPIO11 respectively. The function `GPIO_write()` its second parameter accepts either a 1 or 0 according to the documentation generated by Doxygen. That is why there are masks and bit shifts. Line 23 calls the `delay_1sec()` function which waits 1 second before returning to the caller. Variable `switcher` is incremented (Line 25) modulo 8 to make sure no overflow will happen.

```
1  void *mainThread(void *arg0)
2  {
3      /* Call driver init functions */
4      GPIO_init();
5
6      /* Configure the LED */
7      GPIO_setConfig(Board_GPIO_LED0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
8      GPIO_setConfig(Board_GPIO_LED1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
9      GPIO_setConfig(Board_GPIO_LED2, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
10
11     /* Turn off user LED */
12     GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);     /* Turn red LED off */
13     GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_OFF);     /* Turn yellow LED off */
14     GPIO_write(Board_GPIO_LED2, Board_GPIO_LED_OFF);     /* Turn green LED off */
15
16     unsigned int switcher = 0;
17
18     while(1)
19     {
20         GPIO_write(Board_GPIO_LED0, switcher & 1);
21         GPIO_write(Board_GPIO_LED1, (switcher & 2) >> 1);
22         GPIO_write(Board_GPIO_LED2, (switcher & 4) >> 2);
23         delay_1sec();
24
25         switcher = (switcher + 1) % 8;
26     }
27
28
29     return (NULL);
30 }
```

Listing 3: Toggling LEDs according to Table 2 using TI Driver

2.4 Verification of the behaviour using a logic analyzer

The behavior of the programs must be verified to be able to determine with certainty whether the requirements of the assignment has been met. This verification is done using a logic analyzer. The logic analyzer has been used for all three programs, but the results are almost the same. That is why there are only three figures instead of nine. Figure 6 up to and including Figure 8 are the same snapshot but the mouse hovers different channels to pop up extra information. Channel 0 is the red LED, channel 1 is the yellow LED and channel 2 is the green LED.

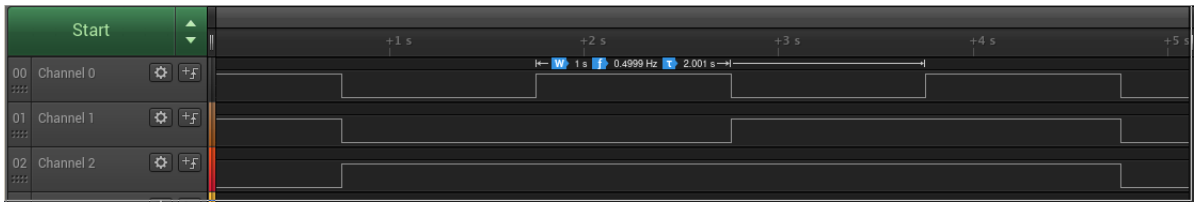

Figure 6: The red LED is toggled every second

Figure 7: The yellow LED is toggled every two seconds (twice as much as the red LED)

Figure 8: The green LED is toggled every four seconds (twice as much as the yellow LED)

3 Replace busy-wait techniques using the SysTick timer

Using a busy-wait technique to delay for a certain amount of time is not efficient. Wasting "expensive" CPU cycles should be avoided whenever possible. This is where hardware timers comes in. Hardware timers decrement or increment a given value at a certain frequency and generates a signal if the value reached a treshold (this could be in the form of underflow or overflow aswell as reaching a certain value where 0 is the most common one). This chapter describes the SysTick hardware timer.

3.1 SysTick timer and interrupt using DRM

As mentioned in the previous section, when one uses the DRM programming technique it is important that the programmer knows the microcontroller really well. Understanding the different Special Function Registers (SFR) related to the hardware module one wants to use is a consequence. The three SFR related to SysTick are described below.

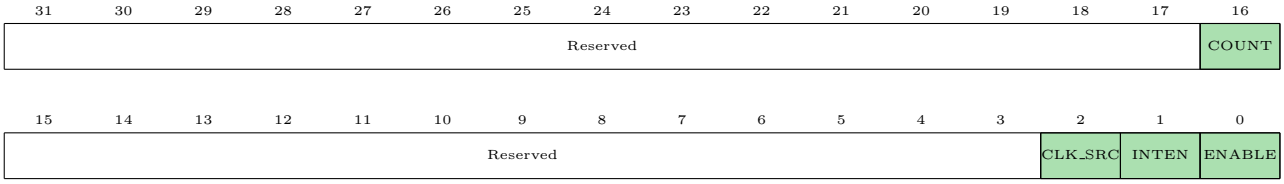

Figure 9: STCTRL register for the CC3220s

First, there is STCTRL (SysTick Control Register) which enables the SysTick features [1]. Bit 0 (Figure 9) enables the SysTick module if this bit is set or disables the SysTick module if this bit is cleared. The SysTick module will generate an interrupt only if bit 1 is set. The clock source fed into the SysTick module can be the system clock if bit 3 is set or a precision internal oscillator if bit 3 is cleared [1].

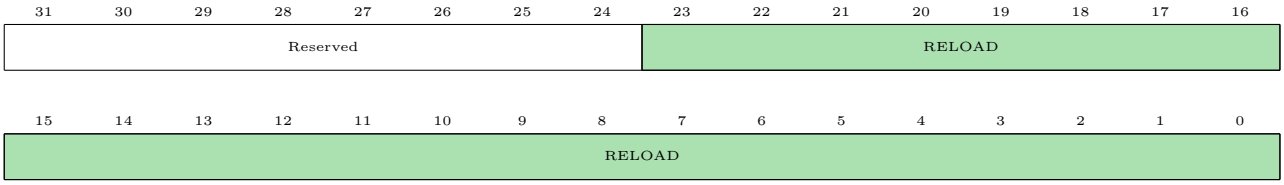

Figure 10: STRELOAD register for the CC3220s

Second, there is STRELOAD register (Figure 10) which stores the constant that should be loaded to the STCURRENT register if the previous value reached value 0. One should keep in mind that if the desired behaviour is an interrupt or a flag every x ticks, one should store $x - 1$ ticks in this register. This is because the RELOAD value is copied to STCURRENT current register if 0 is reached (so not one tick after zero).

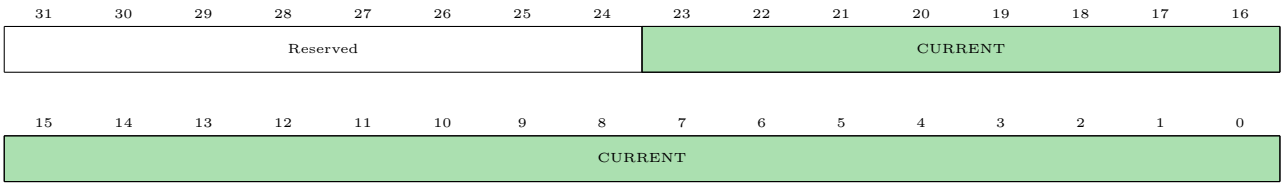

Figure 11: STCURRENT register for the CC3220s

The last register is the STCURRENT register. This register holds the current value being decremented. An important detail is that the CURRENT field is write-clear behaviour. This means that writing any value to this field clears the register and the COUNT bit of the STCTRL register [1].

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include "register_def.h"
4
5  #include "inc\hw_memmap.h"
6  #include "inc\hw_gpio.h"
7  #include "inc\hw_apps_rcm.h"
8  #include "inc\hw_ocp_shared.h"
9
10 static volatile _Bool flag_led_update;
11
12 void SysTickHandler()
13 {
14     static int tick_count = 0;
15     flag_led_update = tick_count == 1000 ? 1 : 0;
16     tick_count = (tick_count+1) % (1000 + 1);
17 }
18
19 int main(void)
20 {
21
22     /* Init SysTick */
23     HWREG(NVIC_ST_CTRL) = 0x00;          // Disable SysTick during setup
24     HWREG(NVIC_ST_RELOAD) = 79999;       // Get every millisecond an interrupt (80 000 - 1)
25     HWREG(NVIC_ST_CURRENT) = 0x00;       // Clear any flags and set current value to 0
26     HWREG(NVIC_ST_CTRL) = 0x07;          // Enable SysTick, Enable interrupt, CLK_SRC = System
       clock
27
28     /* Init LEDS */
29     HWREG(ARCM_BASE + APPS_RCM_O_GPIO_A_CLK_GATING) = 0x01;
30
31     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_9) = 0x60;
32     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_10) = 0x60;
33     HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_11) = 0x60;
34
35     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DIR) = 0x0E;
36     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = 0x00;
37
38     unsigned int index = 0;
39
40     while(1)
41     {
42         if(!flag_led_update)
43             continue;
44
45         HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = index;
46         index = (index + 1) % 16;
47         flag_led_update = 0;
48     }
49
50     return 0;
51 }
```

Listing 4: Toggling LEDs according to Table 2 using DRM programming technique

Listing 4 shows the programming source code to toggle the LEDs using the `SysTick` timer module to delay the correct amount of time. Line 10 contains a shared variable which is used as a flag. If this variable is set the main loop should toggle the LEDs in the next row in Table 2. If this variable is not set then the main loop has to do nothing. Line 12 is the entry point for the `SysTick` event handler. Every time a `SysTick` interrupt occurs this piece of event handler code is executed. There should be a delay of 1 second between every row in Table 2. Since the `SysTick` interrupt is executed every 1 millisecond the code should execute the `SysTick` handler 1000 times before updating the shared variable. `tick_count` takes care of that. This static variable does not lose its content between two execution runs of the event handler. Every interrupt this variable is incremented (Line 16) and when its incremented 1000 times the shared variable is set (Line 15).

After booting, the first thing done is disabling the `SysTick` module (Line 23) by setting all bits in the `STCTRL` register to 0 (including the `ENABLE` bit). Then it loads the reset value $80000 - 1$ into the `STRELOAD` register (Line 24) and clears the current value and any flags already set (Line 25). `ENABLE`, `INTEN` and `CLK_SRC` bits in the `STCTRL` register (Figure 9) are set by writing 00000111_2 or $0x07_{16}$ to this register. From this point in the code the `SysTick` module is enabled and decrementing.

What now follows is a description of the main code. This is almost identical to the code described in Section 2.1. The registers related to GPIO can be found in that section aswell. Line 29 in Listing 4 enables the GPIOA peripheral. Line 31 up to and including Line 33 configures the behaviour of GPIO pin 9, GPIO pin 10 and GPIO pin 11. Those pins are configured to have a maximum current usage of 6 mA. Line 35 configures GPIO pin 9, GPIO pin 10 and GPIO pin 11 as an output pin. Line 36 turns those GPIO pins off according to first row of Table 2.

The following code is executed in an endless loop. Line 42 and Line 43 were not present in Section 2.1. Line 42 checks if the shared variable `flag_led_update` is set by the `SysTick` event handler. It it is not Line 43 is executed and the check will be executed again. If it is set then Line 45 up to and including Line 47 will be executed. Line 45 writes the new sequence of LEDs to the GPIO pins. Line 46 increments the sequence so the next row in Table 2 will be set next time the shared variable is set. Line 47 clears the flag so the next iteration of the loop won't update the LEDs again.
The reader may wonder how the microcontroller knows which function should be executed on a `SysTick` interrupt. This is done using the vector table. This vector table contains different function names for different kind of interrupts (Listing 5). Because the function definition `SysTickHandler` is in `main.c` we declare it as extern so the compiler will not fail and the linker will search for the external references in a later stage of the compilation process.

```
1  ......
2  ......
3  extern void SysTickHandler();
4  ......
5  ......
6
7  void (* const resetVectors[43])(void) =
8  {
9      (void (*)(void))((unsigned long)&_STACK_END),
10                                          // The initial stack pointer
11     resetISR,                            // The reset handler
12     nmiISR,                              // The NMI handler
13     faultISR,                            // The hard fault handler
14     defaultHandler,                      // The MPU fault handler
15     busFaultHandler,                     // The bus fault handler
16     defaultHandler,                      // The usage fault handler
17     0,                                   // Reserved
18     0,                                   // Reserved
19     0,                                   // Reserved
20     0,                                   // Reserved
21     defaultHandler,                     // SVCall handler
22     defaultHandler,                      // Debug monitor handler
23     0,                                   // Reserved
24     defaultHandler,                      // The PendSV handler
25     SysTickHandler,                      // The SysTick handler
26     defaultHandler,                      // GPIO Port A0
27     defaultHandler,                      // GPIO Port A1
28     defaultHandler,                      // GPIO Port A2
29     defaultHandler,                      // GPIO Port A3
30     ......
31     ......
32 }
```

Listing 5: Part of `cc3220_startup_ccs.c` which contains a part of the vector table

For more information about exception, interrupts and the vector table, see Appendix A.2.

3.2 `SysTick` timer and interrupt using Driverlib

The code listings presented in the previous subsection can also be converted using Driverlib. This converted code using Driverlib can be seen in Listing 6 Quite some code in Listing 6 has not been changed compared to Listing 4. The shared variable `flag_led_update` in Line 10 is still there. The `SysTick` event handler in Line 12 is identical to the `SysTick` event handler in Line 12 of Listing 4. In fact the behaviour of the code is exactly the same as the one generated by Listing 4. However, because Driverlib is used some code changed.

Line 21 disables the `SysTick` module. Line 22 sets the period (every millisecond an interrupt is generated). Please note that using DRM the value 79 999 has to be loaded in the `STRELOAD` register. By using an abstraction layer such as Driverlib we do not have to take this into account. Line 23 will register the function `SysTickHandler` to be called once a SysTick event occurs. Line 25 enables the `SysTick` module. Line 27 enables the GPIO A1 module (where GPIO 9, GPIO 10 and GPIO 11 are connected to). Line 29 up to and including Line 31 configures the characteristics of GPIO 9, GPIO 10 and GPIO 11. Those pins can not use more than 2 mA. Line 33 up to and including Line 35 configures GPIO 9, GPIO 10 and GPIO 11 as output pins.
The shared varible `flag_led_update` is polled in a loop and when it is set by the `SysTickHandler` it will write the next LED sequence of Table 2 to the three GPIO pins (line 47), update the variable `switcher` so the next sequence will be written once `flag_led_update` is set again (Line 48) and the shared variable `flag_led_update` is cleared so the next loop iteration will not enter Line 47 again.

Currently the main program polls the shared variable `flag_led_update` to see whether if it has to update the LEDs. Using this technique does not solve our issue when we started using the `SysTick` module. Busy-wait or polling should be avoided as much as possible and by using the `SysTick` module the current implementation polls on the shared variable `flag_led_update`. A really simple solution is to turn the microprocessor in a "sleep" mode. During such mode no instructions are fetched and executed thus less power is consumed. There is an ARM assembly instruction `WFI` (Wait For Interrupt) which turns the microprocessor in a sleep mode untill an interrupt occurs. This is ideal for such a simple program as Listing 6 because no other useful work can be done untill the `SysTickHandler` eventually sets the shared variable. Line 52 uses the DRM (like) technique by executing this Assembly instruction. One could also use Driverlib to enter the wait for interrupt state (Line 53).

```c
#include <stdint.h>
#include <stddef.h>
#include "register_def.h"

#include <gpio.h>
#include <pin.h>
#include <prcm.h>
#include <systick.h>

static volatile _Bool flag_led_update;

void SysTickHandler()
{
    static int tick_count = 0;
    flag_led_update = tick_count == 1000 ? 1 : 0;
    tick_count = (tick_count+1) % (1000 + 1);
}

int main(void)
{
    SysTickDisable();
    SysTickPeriodSet(80000);
    SysTickIntRegister(SysTickHandler);
    SysTickIntEnable();
    SysTickEnable();

    PRCMPeripheralClkEnable(PRCM_GPIOA1 ,PRCM_RUN_MODE_CLK);

    PinTypeGPIO(PIN_64, PIN_STRENGTH_2MA, false);   /* Red LED is push-pull */
    PinTypeGPIO(PIN_01, PIN_STRENGTH_2MA, false);   /* Yellow LED is push-pull */
    PinTypeGPIO(PIN_02, PIN_STRENGTH_2MA, false);   /* Green LED is push-pull */

    GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_2, GPIO_DIR_MODE_OUT);
    GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_3, GPIO_DIR_MODE_OUT);
    GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_1, GPIO_DIR_MODE_OUT);

    GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_2, ~GPIO_PIN_2);  /* Turn yellow LED off */
    GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_3, ~GPIO_PIN_3);  /* Turn green LED off */
    GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1, ~GPIO_PIN_1);  /* Turn red LED off */

    unsigned int switcher = 0;

    while(1)
    {
        if(flag_led_update)
        {
            GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, switcher);
            switcher = (switcher + 1) % 16;
            flag_led_update = 0;
        }

        // __asm("     WFI");    /* Uncomment to meet assignment 2.4 */
        // PRCMSleepEnter();     /* Uncomment to meet assignment 2.5 */
    }

    return 0;
}
```

Listing 6: Toggling LEDs according to Table 2 using Driverlib

3.3 Traffic light using Driverlib

A very simple static scheduler can be implemented to control a traffic light. The characteristics of the different coloured lights can be seen in Table 3.

Colour	*Time bright (s)*
Red	5
Yellow	1
Green	4

Table 3: Timing characteristics for the different colours of the traffic light

The code is split up into two parts to keep the overview. The first part defines a few datatypes, global variables and two functions (where one is an event handler). Line 14 in Listing 7 defines a datatype `state_t` which is a datatype to hold the current LED colour to lit. Line 16 makes one global variable (`state`) of the just created datatype `state_t` to hold the current LED to lit. Line 17 makes a global variable `ticks` which is used in the `SysTickHandler` to increment (Line 21). Those ticks are used to determine if the amount of time a certain LED has lit is expired. If it has not expired then the microprocessor enters a sleep mode (Line 27). However, if the time a certain LED has expired the `tick` variable is reset (Line 29) and the next state is set (Line 30). The different states are explained in part two of the static scheduler.

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include "register_def.h"
4
5  #include <gpio.h>
6  #include <pin.h>
7  #include <prcm.h>
8  #include <systick.h>
9
10 typedef enum {
11     STATE_RED,
12     STATE_YELLOW,
13     STATE_GREEN
14 } state_t;
15
16 static volatile state_t state;
17 static volatile unsigned int tick;
18
19 void SysTickHandler()
20 {
21     tick++;
22 }
23
24 void next_state(state_t newState, unsigned int delayInSeconds)
25 {
26     while((tick/1000) < delayInSeconds)
27         PRCMSleepEnter();
28
29     tick = 0;
30     state = newState;
31 }
```

Listing 7: First part of the static scheduler which defines some data types and states

The second part of the static scheduler can be found in Listing 8. Line 3 up to and including Line 21 will not be explained again because those lines have been explained in the previous subsection. Explaining them again would be redundant. The defaultk state is that the red LED is lit (Line 23). There is no particular reason why the initial behaviour is that the red LED is lit. The reader is free to change the initial LED colour. A kind of state machine has been implemented to switch between LEDs (Line 27 up to and including Line 41). In a certain state the LEDs are written with a certain pattern for that particular state using the `GPIOPinWrite` function. The interesting part comes after setting the LEDs. A function call to `next_state` is made with the first parameter being the next state that should be entered and the second parameter howmuch seconds this current state should last. `next_state` uses the `ticks` global variable to keep track if the current state should be changed to the next state. If the current state has not been up for the desired amount of seconds the microprocessor enters the sleep mode. After an interrupt and the `SysTickHandler` increments `ticks` the function `next_state` will test again if the current state has been up for the desired amount. This goes on and on untill the desired amount of seconds has been reached and the next state is set.

```
1  int main(void)
2  {
3      SysTickDisable();
4      SysTickPeriodSet(80000);
5      SysTickIntRegister(SysTickHandler);
6      SysTickIntEnable();
7      SysTickEnable();
8
9      PRCMPeripheralClkEnable(PRCM_GPIOA1  ,PRCM_RUN_MODE_CLK);
10
11     PinTypeGPIO(PIN_64, PIN_STRENGTH_2MA, false);    /* Red LED is push-pull */
12     PinTypeGPIO(PIN_01, PIN_STRENGTH_2MA, false);    /* Yellow LED is push-pull */
13     PinTypeGPIO(PIN_02, PIN_STRENGTH_2MA, false);    /* Green LED is push-pull */
14
15     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_2, GPIO_DIR_MODE_OUT);
16     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_3, GPIO_DIR_MODE_OUT);
17     GPIODirModeSet(GPIOA1_BASE, GPIO_PIN_1, GPIO_DIR_MODE_OUT);
18
19     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_2, ~GPIO_PIN_2);  /* Turn yellow LED off */
20     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_3, ~GPIO_PIN_3);  /* Turn green LED off */
21     GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1, ~GPIO_PIN_1);  /* Turn red LED off */
22
23     state = STATE_RED;
24
25     while(1)
26     {
27         switch(state)
28         {
29         case STATE_RED:
30             GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x02);
31             next_state(STATE_YELLOW, 5);
32             break;
33         case STATE_YELLOW:
34             GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x04);
35             next_state(STATE_GREEN, 1);
36             break;
37         case STATE_GREEN:
38             GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x08);
39             next_state(STATE_RED, 4);
40             break;
41         }
42
43     }
44
45     return 0;
46 }
```

Listing 8: Second part of the static scheduler

3.4 Verification of the timing specifications

According to Table 3 the yellow LED should be lit for 1 second. Channel 1 is connected to the yellow LED. Figure 12 shows that the yellow LED is lit for 1.001 seconds.

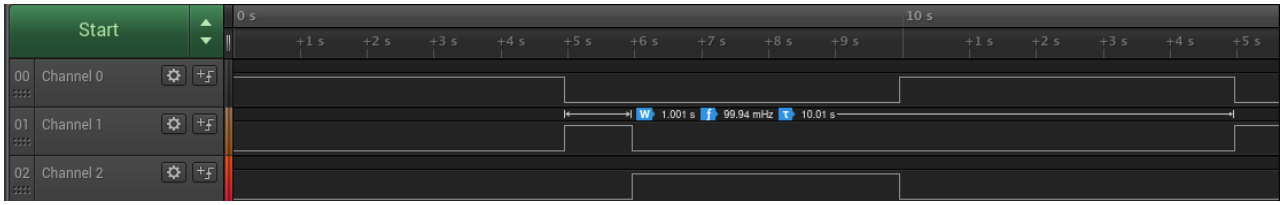

Figure 12: The yellow LED is lit for 1.001 seconds

According to Table 3 the green LED should be lit for 4 seconds. Channel 2 is connected to the green LED. Figure 13 shows that the green LED is lit for 4.002 seconds.

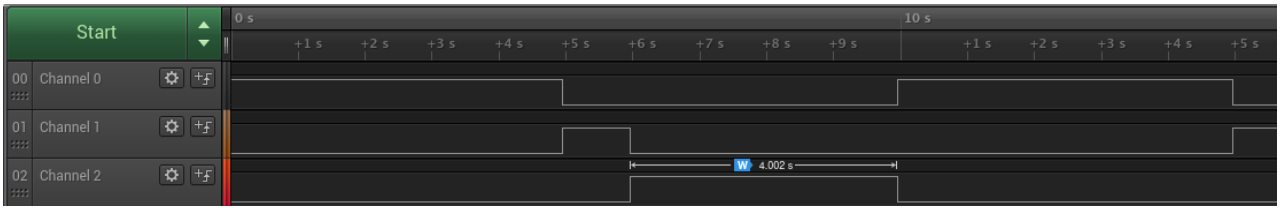

Figure 13: The green LED is lit for 4.002 seconds

According to Table 3 the red LED should be lit for 5 seconds. Channel 0 is connected to the red LED. Figure 14 shows that the red LED is lit for 5.002 seconds.

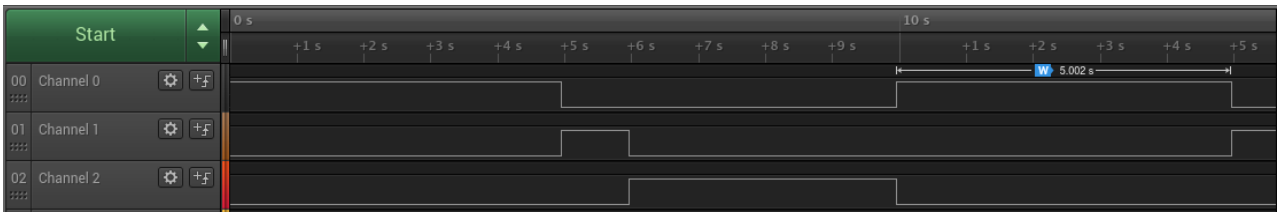

Figure 14: The red LED is lit for 5.002 seconds

The desired amount of time a certain LED should be lit and the actual time the LED is lit is nearly the same. One may conclude that the software results in the correct behaviour both from a functional and time requirement perspective.

4 Creating a scheduler using SysTick

The next step is creating a simple scheduler that can run periodic tasks. There are three different tasks the scheduler should execute; blink a red, a yellow and a green LED for a given period after a given delay in ticks. Table 4 lists the requirements for these tasks.

	Toggle Period	*Initial Delay*
Red	200	100
Yellow	500	200
Green	750	300

Table 4: Requirements blinking LEDs in Ticks

The code that will be presented in Subsection 4.1 is actually all located in the same file, namely `main.c`. However, for the sake of clarity to the reader the code is split up into different listings which makes it easier to understand what piece of code is for what reason present.

4.1 Scheduling Tasks

Different tasks have different characteristics. Some housekeeping is required to map those characteristics to different functions (the behaviour a task is executing). Those housekeeping goes into what's called a Task Control Block (TCB) [3], [2]. This TCB is defined in Listing 9 at Line 25. Characteristics of a task is the function it executes (usually a pointer pointing to the function definition), the period time this function should be called, the amount of time it waits untill it is ready to be woken up and execute the function again as the new period arose and a state. This state is defined in Line 17. Usually there are three states which is `STATE_READY`, `STATE_WAITING` and `STATE_RUNNING` [3], [2]. Because this scheduler is almost too simple only `STATE_READY` and `STATE_WAITING` are necessary.

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include <stdlib.h>
4  #include "register_def.h"
5
6  #include "inc\hw_memmap.h"
7  #include "inc\hw_gpio.h"
8  #include "inc\hw_apps_rcm.h"
9  #include "inc\hw_ocp_shared.h"
10
11 #include "blinky_tasks.h"
12
13 typedef enum
14 {
15     STATE_READY = 0,
16     STATE_WAITING
17 } OS_state;
18
19 typedef struct
20 {
21     void (*task_function)(void);
22     uint32_t tick_period;
23     uint32_t ticks_used;
24     OS_state state;
25 } OS_tcb;
26
27 /* Global variables*/
28 OS_tcb tasks[8];
29 size_t amount_tasks = 0;
```

Listing 9: Definition code where a state datatype and housekeeping datatype for tasks is made

There are two global variables used in other parts within the same file. `tasks` (Line 28) is an array that has space for 8 `OS_tcb` objects which means a maxmimum of 8 tasks are schedulable by this scheduler unless a programer changes this number. `amount_tasks` keeps track howmuch stores `OS_tcb` objects are in the `tasks` array.

Tasks should be able to be added to the system and being executed via the scheduler. That is exactly what the functions `OS_add_task()` and `OS_run_ready_tasks()` do respectively (Listing 10). The parameters related to a task are set in the global array `tasks`. After the last parameter (which is state) is set the `amount_tasks` global variable is incremented so the next time the function `OS_add_task()` is called it will set the parameters in the correct index.
`OS_run_ready_tasks()` loops through the list of `tasks` starting at index 0 untill index `amount_tasks`. If a task has the state `STATE_READY` then its function pointer is being called and once the function returns its state is set to `STATE_WAITING`. The `SysTickHandler` explained in a moment will make sure the task its state is set to `STATE_READY` if needed.

```
void OS_add_task(void (*task_function)(void), uint32_t tick_period, uint32_t ticks_init_delay)
{
    tasks[amount_tasks].task_function = task_function;
    tasks[amount_tasks].tick_period = tick_period;
    tasks[amount_tasks].ticks_used = ticks_init_delay;
    tasks[amount_tasks++].state = STATE_WAITING;
}

void OS_run_ready_tasks()
{
    for(size_t idx = 0; idx != amount_tasks; idx++)
    {
        if(tasks[idx].state == STATE_READY)
        {
            (*tasks[idx].task_function)();
            tasks[idx].state = STATE_WAITING;
        }
    }
}
```

Listing 10: Task controllers

The `SysTickHandler()` decrements the `ticks_used` member variable of a task (Listing 11). If `ticks_used` is equal to 0 and the current state is `STATE_WAITING` then the task has been waiting enough and the state is set to `STATE_READY`. The `ticks_used` variable is set to its period so underflow is prevented (and so is invoking UB).

```
void SystickHandler()
{
    for(size_t idx = 0; idx != amount_tasks; idx++)
    {
        tasks[idx].ticks_used -= 1;
        if(tasks[idx].ticks_used == 0)
        {
            if(tasks[idx].state == STATE_WAITING)
                tasks[idx].state = STATE_READY;
            tasks[idx].ticks_used = tasks[idx].tick_period;
        }
    }
}
```

Listing 11: Scheduler SysTick

The reader should now be able to see how the `state` member variable of `OS_tcb` is used both in Listing 10 and in Listing 11 in different contexts. This `state` member variable is used as Inter Process Communication (IPC) between the event handler and the foreground program. It makes sure the scheduler executes the correct tasks and the event handler wakes up the correct tasks if neccesary.

The entry point for the main application is Line 1 in Listing 12. The first part of the code in Listing 12 has already been explained. It won't be repeated in this subsection. For an explanation about the GPIO initialisation which is Line 11 up to and including Line 11 see Subsection 2.1. For an explanation about the `SysTick` module initialisation see Subsection 3.1. If one looks at the given specification for the red LED, yellow LED and green LED in Table 4 the creation of the three tasks in Line 18, Line 19 and Line 20 should make sense. The initial delay column of Table 4 are passed as constants in the third argument to `OS_add_task()`. The function pointers are passed via the first argument to `OS_add_task()`. Those functions are defined in `blinky_tasks.c` and will be described in Subsection 4.2. The second parameter is the period of the to be created task. Those are defined using the `#define` preprocessor directive in `blinky_tasks.h`. The foreground process always calls `OS_run_ready_tasks()`. As long as there are tasks ready to be executed they will be called. If there are no tasks ready at a certain point in time then the the loop will be infinitely executed searching infinitely long for a task that is ready to be executed.

```
1  int main(void)
2  {
3
4      HWREG(ARCM_BASE + APPS_RCM_O_GPIO_A_CLK_GATING) = 0x01;
5
6      HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_9) = 0x60;
7      HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_10) = 0x60;
8      HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_11) = 0x60;
9
10     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DIR) = 0x0E;
11     HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = 0x00;
12
13     HWREG(NVIC_ST_CTRL) = 0x00;          /* Disable SysTick during setup */
14     HWREG(NVIC_ST_RELOAD) = 79999;       /* 80 000 reload value (1000p/s interrupt) */
15     HWREG(NVIC_ST_CURRENT) = 0x00;       /* Clear any flags and set current value to 0 */
16     HWREG(NVIC_ST_CTRL) = 0x07;          /* Enable SysTick, Enable interrupt, CLK_SRC = System
       clock */
17
18     OS_add_task(blinky_red, RED_PERIOD, 100);
19     OS_add_task(blinky_yellow, YELLOW_PERIOD, 200);
20     OS_add_task(blinky_green, GREEN_PERIOD, 300);
21
22     while(1)
23         OS_run_ready_tasks();
24
25     return 0;
26 }
```

Listing 12: Entry point for the scheduler foreground process

4.2 Blinky LED

The second part of the compiled code is related to the task. Creating tasks and executing tasks is explained in the previous subsection. This subsection describes the three functions of the three tasks that are called each time a task is executed. The code is not complex at all but for the sake of completeness it is briefly described. `blinky_tasks.h` contains the function prototyping for the functions that a task is executing with the period for the tasks. This can be seen in Listing 13. Using the preprocessor `#define` directive makes it easy to change the period on a single place.

```
1  #ifndef BLINKY_TASKS_H_
2  #define BLINKY_TASKS_H_
3
4  #define RED_PERIOD      200
5  #define YELLOW_PERIOD   500
6  #define GREEN_PERIOD    750
7
8  void blinky_red(void);
9  void blinky_yellow(void);
10 void blinky_green(void);
11
12 #endif /* BLINKY_TASKS_H_ */
```

Listing 13: `blinky_tasks.h` prototyping the functions tasks should be called and their period

The implementation of the functions is not exciting. In fact, all the functions are actually one-liners (see Listing 14). They set another bit pattern to the red LED, yellow LED and the green LED.

```
#include "register_def.h"

#include "inc\hw_memmap.h"
#include "inc\hw_gpio.h"
#include "inc\hw_apps_rcm.h"
#include "inc\hw_ocp_shared.h

void blinky_red(void)
{
    HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x02 << 2)) ^= 0x02;
}

void blinky_yellow(void)
{
    HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x04 << 2)) ^= 0x04;
}

void blinky_green(void)
{
    HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x08 << 2)) ^= 0x08;
}
```

Listing 14: Toggling LED tasks according to Table 4

4.3 Verification of the timing specification

Figure 15 contains the output of the logic analyzer after running the program free for a couple of seconds. Channel 0, channel 1 and channel 2 are connected to the red LED, yellow LED and green LED respectively. According to Table 4 the red LED should toggle every 200 ticks (where 1 tick is equal to 1 millisecond). The logic analyzer shows that the red LED contains a logic high level for 0.2001 seconds which is close enough to 200 milliseconds.

Figure 15: Saleae logic analyzer trace after the program ran free for a couple of seconds

The yellow LED is logic high for 0.5001 seconds and the green LED is logic high for 0.7502 seconds. One can conclude that the timing specifications in Table 4 are met.

5 A basic Real-Rime Operating System and “clever” scheduler

A basic Real-Time Operating System and schudler can now be written. In fact, the basic Real-Time Operating System has already been developed by Daniel Versluis. From now on in this document we call this basic Real-Time Operating System “VersdOS”. This RTOS has a global variable `taskList` which is an array of type `task`. This datatype `task` can be seen in Listing 15. Every task has its own stack and thus a pointer to the stack. A task has a function pointer pointing to the function it should be executing if the task is ready to be run. Also described in the previous section, every task has a state. VersdOS acknowledges three states. The first one is `RUNNING`, the second one is `READY` and the last one is `WAITING`. `uiPriority` is the priority of the task. The higher this number the higher the priority. `uiCounter` is a variable containing a number. If a task is in the `WAITING` state this number is decremented each SysTick interrupt.

```
typedef struct _task{
  int *       stack;        //pointer to the stack (on the heap?)

  void(*function)(void);        //function to execute
  enum taskState  state;        //Tasks have a state
  uint8_t     uiPriority;     //Tasks have a priority
  uint32_t    uiCounter;      //Tasks have a counter for delays

  char      bInitialized;   //After initialization the stack also contains R4-R11
} task;
```

Listing 15: Task control block for VersdOS

The current implementation of the core functionality of the scheduler, which is selecting the next task to run, can be seen in Listing 16. There are three tasks. One that toggles the red LED, one that toggles the green LED and one that toggles the yellow LED. The scheduler is aware of the amount of tasks but its hard coded. This is very bad practice and makes a scheduler only usable for a certain set of task jobs [5]. The current implementation is ignoring the `uiPriority` member variable of the task control block aswell.

```
/*
 * Our genuine scheduler. Currently very static and
 * not-advanced
 *
 * returns a pointer to the selected task
 */
task * getNextTask(void)
{
  static int i=0;

  while(taskList[++i].function == 0)
  {
    if(i==3)
    {
      i = -1;
    }
  }

  return &taskList[i];
}
```

Listing 16: A relatively stupid scheduler

This scheduler in Listing 16 is not real time as it is not taking the characteristics or deadlines of any task into account. A better implementation in terms of guarantees and taking real-time characteristics into account is proposed in Listing 17.

```
/*
 * Our genuine scheduler. Currently very static and
 * not-advanced
 *
 * returns a pointer to the selected task
 */
task * getNextTask(void)
{
    int i = 0;

    int idx_task = 0;
    int task_priority_temp = 0;

    for(i = 0; i < amount_of_tasks; i++)
    {
      if(taskList[i].state != WAITING)
      {
            if(taskList[i].uiPriority > task_priority_temp)
            {
                idx_task = i;
                task_priority_temp = taskList[i].uiPriority;
            }
        }
    }

    return &taskList[idx_task];
}
```

Listing 17: An improved scheduler taking task characteristics into account

One can argue "what happends if all tasks are `WAITING`?" `taskList[0]` always contains the idle task. That is why variable `idx_task` is set to 0 before the for-loop starts. If no other tasks wants to use the micrcoprocessor then the idle task will be returned in Line 26. This idle task can be seen in Listing 18. All it does is setting the microprocessor in a sleep mode untill an event occurs.

```
void idleTask(void)
{
    __asm("    wfi");
}
```

Listing 18: An idle task basically setting the microprocesor in a sleep mode

Another thing that the authors noticed is the current implementation of a delay function to temporarry halt a certain task. The old implementation can be seen in Listing 19. This delay function depends on the clock frequency and compiler generated instructions. In addition, a task is not sleeping. IT is doing a function call to `delay` and hopes when the function returns it slept long enough. It keeps sitting on the CPU executing dumb `SUBS` instructions. Other tasks in the `READY` state cannot claim the CPU as the `RUNNING` task keeps the CPU busy by doing meaningless calculations (if it calls the `delay` functions). The authors agree with Daniel Versluis that this is an unworthy `delay` function.

```
//unworthy delay function
void delay(unsigned int count)
{
    while(count--);
}
```

Listing 19: An unworthy delay function, according to Daniel Versluis

The authors propose a worthy `delay` function. The implementation details can be seen in Listing 20.

```
1  void delayTask(unsigned int ticks)
2  {
3      for(int idx = 0; idx < MAX_TASKS; idx++)
4      {
5          if(taskList[idx].state == RUNNING){
6              taskList[idx].uiCounter = ticks;
7              taskList[idx].state = WAITING;
8          }
9      }
10     schedule();
11 }
```

Listing 20: A worhy `delay` function, according to the authors

The proposed `delay` deserves some explanation as it may not seem trivial in the first place. This implementation makes use of a property of a single core microprocessor. Only one task at a certain point in time can have the `RUNNING` state. If the function `delay` is entered it must be called from the current `RUNNING` task. It assumes the function call is atomic and no scheduling happends when between the function is called and the state of the running task is set to `WAITING`. The loop in Line 3 searches through the list of registered tasks. If the condition in Line 5 is true it means that the `RUNNING` task is found. If it is found the `uiCounter` variable which is decremented each `SysTickHandler` interrupt is decremented is set to the desired amount of ticks passed via the function parameter. The `state` is set to `WAITING`. The housekeeping is set. The task is ready to be put in `WAITING` mode. 1 important detail remains. A new task should be selected to be run on the microprocessor. The function call in Line 10 will take care of this. This function `schedule` can be seen in Listing 21.

```
1  void schedule(void)
2  {
3    //This is our true scheduler function
4    //select a new task to run
5    taskToExecute = getNextTask();
6
7    //Only if the new task isn't equal to the current one,
8    //call the context switch
9    if(taskToExecute != currentTask || taskToExecute->bInitialized==0)
10   {
11     //States to help the scheduler decide
12     //currentTask->state = READY;
13     taskToExecute->state = RUNNING;
14
15     //call pendsv interrupt to perform the context switch
16     HWREG(NVIC_INT_CTRL)  |= (1<<28);
17
18   }else{
19     //Clearly no need to switch anything so we
20     //just restore things like they were before the Systick
21
22     currentTask->state   = RUNNING;
23   }
24 }
```

Listing 21: `schedule` will call the helper function `getNextTask` and makes sure that task is actually executed

Because of lack of time there was no time to show prove using the logic analyzer that this implementation works. Various instructors have seen during labs that it works and signed off the authors name on a list of assignments. If the reader still wants proof then he or she must send an e-mail to one of the authors e-mail addresses on the title page.

6 Acknowledgement

The authors want to thank Daniel Versluis for writing his Minimal Working Example (MWE) Real-time Operating Systems "VersdOS" and providing the authors access to the source code. The authors also want to thank Harry Broeders for his time and effort in solving the problem related to the `delay_1sec()` function and inline assembly instruction cycles mismatch.

A Appendix

The appendix contains subsections that support this report or its where its content goes too much off-topic with the purpose of this report, but are interesting for the reader to possibly read.

A.1 Delay *exactly* one second counting instruction cycles

Many assignments require a delay of 1 second to spot blinky LEDs by eye. One can use the Systick timer or hardware timers, but where is the fun in that? For the sake of some assignments, it is acceptable to burn clock cycles by wasting the CPU. Listing 22 contains a function which will delay the return moment by 1 second. Now each line containing inline assembly will be explained.

```
1  void delay_1sec(void)
2  {
3      __asm("    PUSH {r4-r11,lr}");
4
5      __asm("    LDR r4, [pc, #12]");
6
7      __asm("    MOV r5, pc");
8      __asm("    NOP");
9
10     __asm("    SUBS r4, #1");    /* 1 instruction cycle */
11     __asm("    ITE NEQ");        /* 1 instruction cycle */
12
13     __asm("    MOV pc, r5");     /* 1 + P instructions (where P is between 1 and 3 depending on
        pipeline refill) */
14
15
16     __asm("    POP {r4-r11,pc}");
17     __asm("    .word    10000000");
18 }
```

Listing 22: C function containing inline assembly to perform a delay of *exactly* one second

Line 3 pushes 8 registers onto the stack. This is part of the ARM Architecture Procedure Call Standard (AAPCS) which is part of the ARM Application Binary Interface (ABI) [4]. This standard describes that `R0` up to and including `R4` are used to pass input parameters into a C function. Functions should preserve the content of registers `R4` up to and including `R11`. Listing 22 does not use all of the registers a callee should save, but it is best practice to push them in case one does not know how many registers his or her piece of software will use. Line 7 stores the Program Counter (PC) into `R5`. Because the PC is two instruction (8 bytes) ahead in ARM mode it actually stores the address for Line 10. This is the first instruction that should be executed iterative. Line 8 makes sure the instruction located at Line 7 contains the correct address. The alternative is replacing this instruction with a `SUB` instruction and subtract 4 bytes from `R5`. Line 11 does a check whether the content of `R4` is equal to zero or not [6]. If `R4` is not equal to zero (which makes the statement true because we check for `NEQ` condition code) Line 13 is executed. If `R4` is equal to zero Line 16 is executed. Line 13 stores the PC we saved earlier in Line 7 to the PC. This results a branch to Line 10. Line 16 restores the saved registers and jumps back to the caller. It is not an option to leave out the restore to the PC because that means that the next instruction executed will be the one on Line 17. This is not an intentional instruction but just a location to store a number. If we let the PC execute this line we get undefined behaviour.

A.2 Events and the vector table in the ARM Cortex devices

The Nested Vector Interrupt Controller (NVIC) is a hardware module within the processor to prioritize events. A reader familiar with 8 bit microcontrollers may wonder why we use the term interrupts and events and if they are interchangeable. In ARM terminology, an interrupt is one type of exception. Other exceptions in Cortex-M processors include fault exception an other system exceptions to support the OS (e.g., SVC instruction) [6]. Other readers familiar with x86 can compare the NVIC to the Programmable Interrupt Controller (PIC).

```
#pragma RETAIN(resetVectors)
#pragma DATA_SECTION(resetVectors, ".resetVecs")
void (* const resetVectors[43])(void) =
{
    (void (*)(void))((unsigned long)&__STACK_END),
                                        // The initial stack pointer
    resetISR,                           // The reset handler
    nmiISR,                             // The NMI handler
    faultISR,                           // The hard fault handler
    defaultHandler,                     // The MPU fault handler
    busFaultHandler,                    // The bus fault handler
    defaultHandler,                     // The usage fault handler
    0,                                  // Reserved
    0,                                  // Reserved
    0,                                  // Reserved
    0,                                  // Reserved
    defaultHandler,                    // SVCall handler
    defaultHandler,                     // Debug monitor handler
    0,                                  // Reserved
    defaultHandler,                     // The PendSV handler
    SysTickHandler,                     // The SysTick handler
    defaultHandler,                     // GPIO Port A0
    defaultHandler,                     // GPIO Port A1
    defaultHandler,                     // GPIO Port A2
    defaultHandler,                     // GPIO Port A3
    0,                          // Reserved
    defaultHandler,                     // UART0 Rx and Tx
    defaultHandler,                     // UART1 Rx and Tx
    0,                          // Reserved
    defaultHandler,                     // I2C0 Master and Slave
    0,                          // Reserved
    0,                          // Reserved
    0,                          // Reserved
    0,                          // Reserved
    0,                          // Reserved
    defaultHandler,                     // ADC Sequence 0
    defaultHandler,                     // ADC Sequence 1
    defaultHandler,                     // ADC Sequence 2
    defaultHandler,                     // ADC Sequence 3
    defaultHandler,                     // Watchdog timer
    defaultHandler,                     // Timer 0 subtimer A
    defaultHandler,                     // Timer 0 subtimer B
    defaultHandler,                     // Timer 1 subtimer A
    defaultHandler,                     // Timer 1 subtimer B
    defaultHandler,                     // Timer 2 subtimer A
    defaultHandler,                     // Timer 2 subtimer B
    defaultHandler,                     // Timer 3 subtimer A
    defaultHandler                     // Timer 3 subtimer B
};
```

Listing 23: Vector table used in assignment two and three

The order of the entries in this table is important and predetermined. Exception 1-15 are system exceptions and exception 16 and above are interrupt inputs [6]. Exception 1 is always the Reset exception, exception 2 is always the NonMaskable Interrupt (NMI), exception 3 is always Hard Fault and so on [6]. From exception 16 it is manufacturer dependent. It depends if manufacturers put certain hardware modules in their core or not. It is therefore always recommended to check the datasheet of the microcontroller.

When an event occurs there is generally a sequence consisting of 5 steps that occurs [4].

1. The current instruction is finished
2. The execution of the currently running program is suspended, pushing eight registers on the stack (`R0`, `R1`, `R2`, `R3`, `R12`, `LR`, `PC` and `PSR` (assuming ARMv7-M architecture)
3. The `LR` is set to a specific value signifying an ISR is being run
4. The `IPSR` is set to the event number being processed
5. The `PC` is loaded with the address of the event handler (from the vector table)

The `IPSR` is the Interrupt Program Status Register and the least 9 significant bits represent the exception number being executed. The other Program Status Registers are the Application Program Status Register (`APSR`) and the Execution Program Status Register (`EPSR`). These will not be explained because they are too much off-topic regarding this report.

References

[1] *CC3220 SimpleLink^{TM} Wi-Fi® and Internet of Things Technical Reference Manual*. SWRU465. Texas Instruments. Feb. 2017.

[2] Jean J. Laboresse. *MicroC/OS-II The Real Time Kernel*. CmpBooks, 2002.

[3] Chowdary Venkateswara Penumuchu.
Simple Real-time Operating System. A kernel inside view for a beginner. Trafford, 2007.

[4] Jonathan W. Valvano. *Introduction to ARM Cortex-M Microcontrollers. Embedded Systems.* self-published, 2017.

[5] Jonathan W. Valvano.
Real-Time Operating Systems for ARM ortex-M Microcontrollers. Embedded Systems. self-published, 2019.

[6] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 processors*. Newnes, 2014.