Basic Real-Time Operating System

targeting the ARMv7-M architecture

ROS01, Rotterdam December 16, 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.	

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.

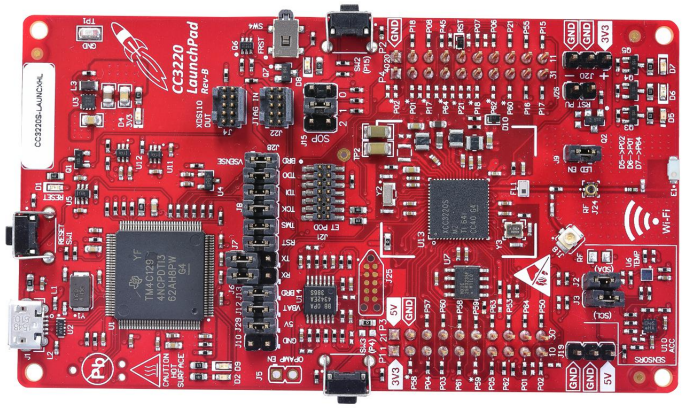

Figure 1: The CC3220s development board used during labs

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) [2]. 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.

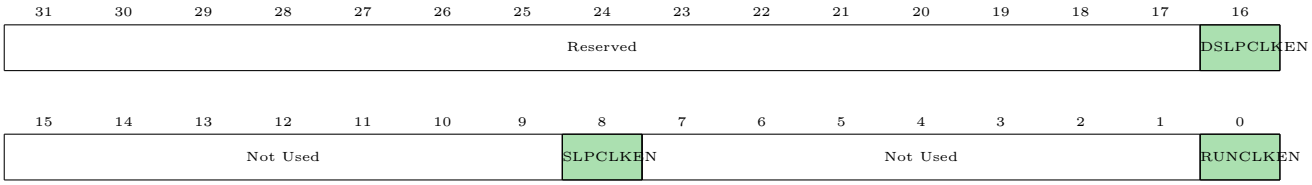

Figure 2: `GPIOOCLKEN` register for the CC3220s

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.

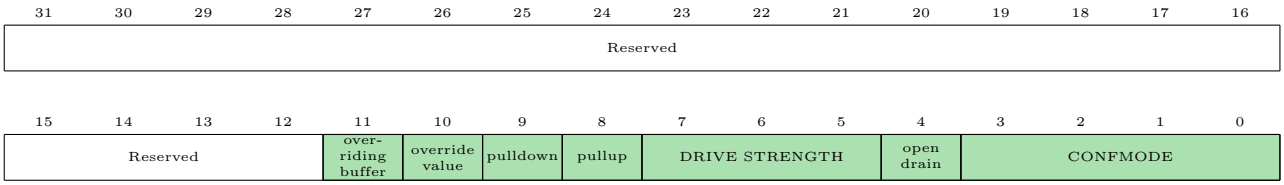

Figure 3: `GPIO_PAD_CONFIG_x` register for the CC3220s

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.

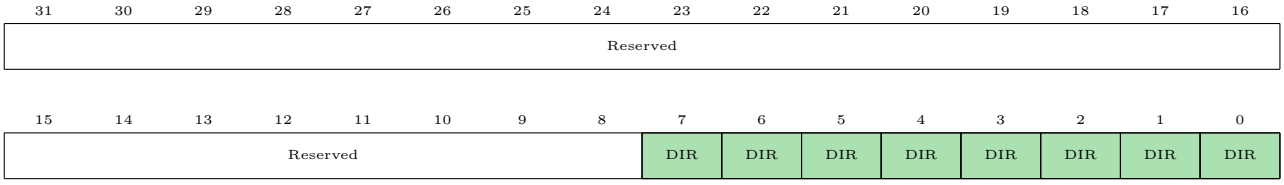

Figure 4: `GPIO_DIR` register for the CC3220s

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.

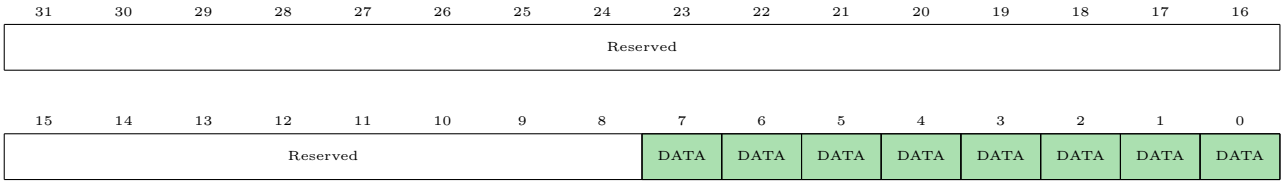

Figure 5: `GPIO_DATA` register for the CC3220s

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.

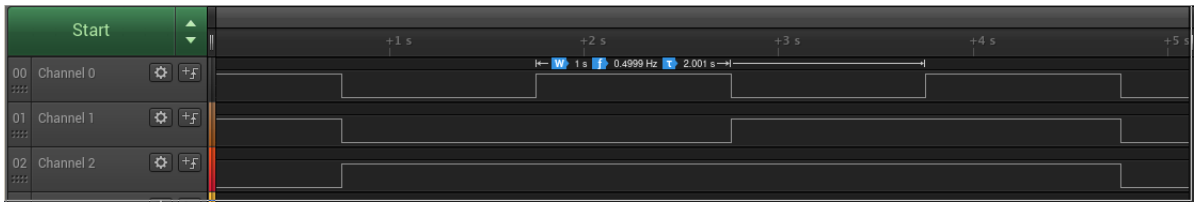

Figure 6: The red LED is toggled every second

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.

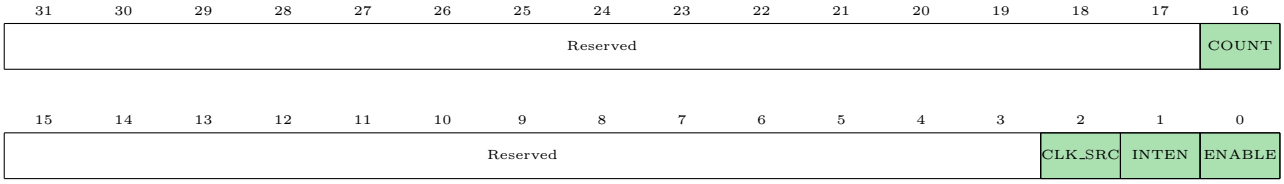

Figure 9: STCTRL register for the CC3220s

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].

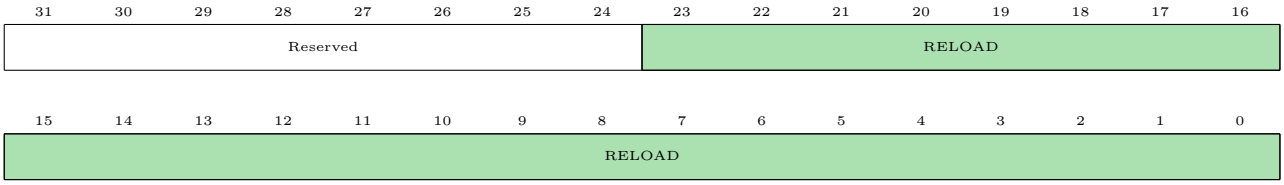

Figure 10: STRELOAD register for the CC3220s

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).

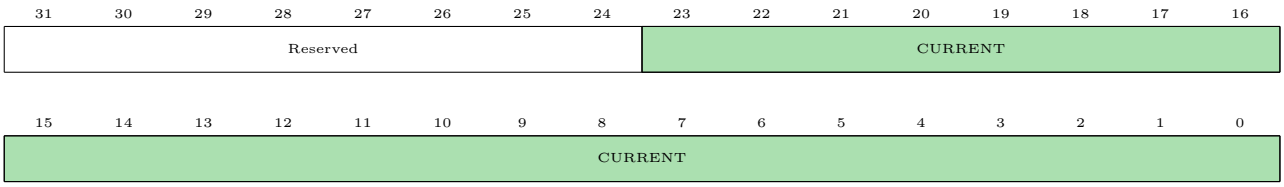

Figure 11: STCURRENT register for the CC3220s

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.

4 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 6 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    5000000");
18 }
```

Listing 6: 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) [2]. 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 6 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 [3]. 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) [3]. 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 7: 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 [3]. Exception 1 is always the Reset exception, exception 2 is always the NonMaskable Interrupt (NMI), exception 3 is always Hard Fault and so on [3]. 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 [2].

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

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