Basic Real-Time Operating System
targeting the ARMv7-M architecture

ROS01, Rotterdam December 7, 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	12-06-2019	Minor changes in Appendix A.1	Documented the first assignment (toggling LEDs)

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.

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.

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) [1]. 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 1). This program does not enter sleep mode or deep sleep mode. Setting bit 8 and bit 16 (Figure 1) does not make sense.
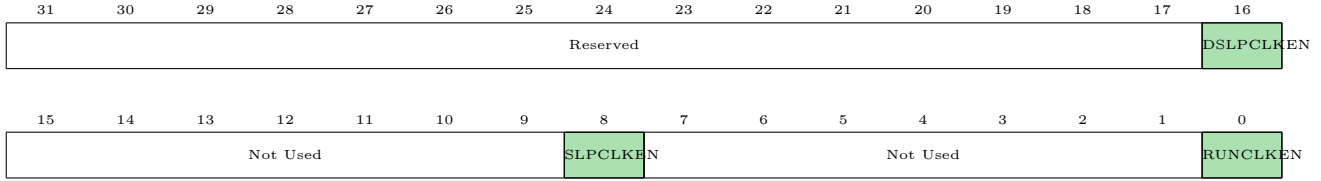
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							Reserved								DSLPCLKEN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			Not Used				SLPCLKEN				Not Used				RUNCLKEN

Figure 1: `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 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 2). Writing 001_2 to the `DRIVE STRENGTH` field means that the related GPIO pin will drive 6 mA.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							Reserved								

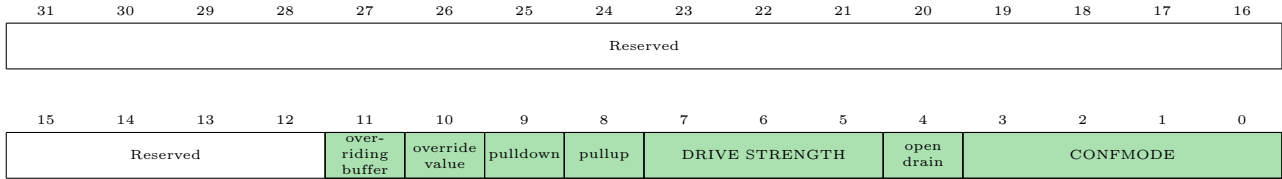| 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		Reserved		over-riding buffer	override value	pulldown	pullup		DRIVE STRENGTH		open drain		CONFMODE		

Figure 2: `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 3) 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 in hexadecimal representation. That is why the program writes 0x0E to the `GPIO_DIR` register at Line 21.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							Reserved								

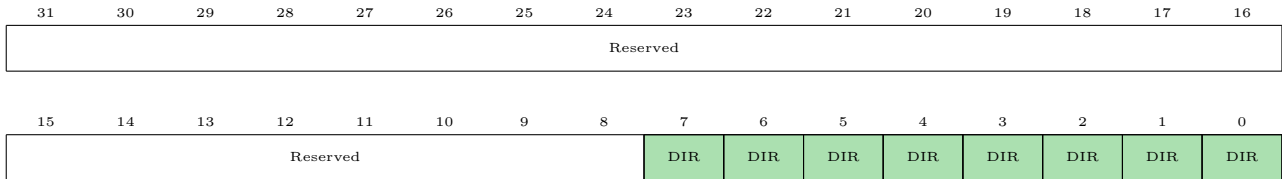| 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				Reserved				DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR

Figure 3: `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 to the `GPIO_DATA` register. However, because of the mask added to the address only bit 1, bit 2 and bit 3 are affected.
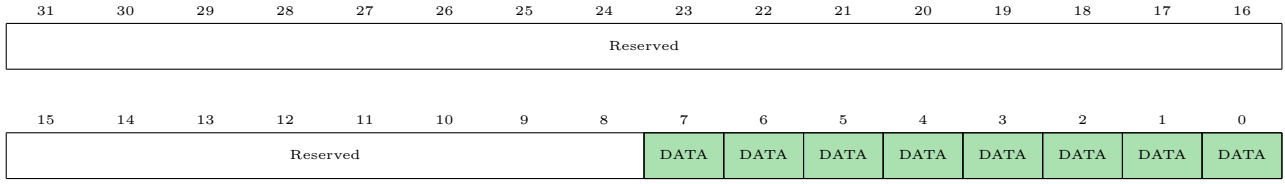
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA

Figure 4: `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 $0 \leq x < 16$ (although $0 \leq x < 8$ 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

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

/* Function delay_1sec() used to be here */

int main(void)
{

    HWREG(ARCM_BASE + APPS_RCM_O_GPIO_A_CLK_GATING) = 0x01;

    HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_9) = 0x60;
    HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_10) = 0x60;
    HWREG(OCP_SHARED_BASE + OCP_SHARED_O_GPIO_PAD_CONFIG_11) = 0x60;

    HWREG(GPIOA1_BASE + GPIO_O_GPIO_DIR) = 0x0E;
    HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = 0x00;

    int index = 0;

    while(1)
    {
        HWREG(GPIOA1_BASE + GPIO_O_GPIO_DATA + (0x0E << 2)) = (index << 1);
        index = (index + 1) % 16;
        delay_1sec();
    }

    return 0;
}
```

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

2.2 Driverlib

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

#include "gpio.h"
#include "pin.h"
#include "prcm.h"

/* GPIO 9 is PIN_64 is (red) */
/* GPIO 10 is PIN_2 is (green) */
/* GPIO 11 is PIN_1 is (yellow) */

/* Function delay_1sec() used to be here */

int main(void)
{
    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 */

    int switcher = 0;

    while(1)
    {
        GPIOPinWrite(GPIOA1_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, switcher);
        delay_1sec();

        switcher = (switcher + 1) % 16;
    }
}
```

Listing 2: as

2.3 TI Driver

```c
void *mainThread(void *arg0)
{
    /* Call driver init functions */
    GPIO_init();

    /* Configure the LED */
    GPIO_setConfig(Board_GPIO_LED0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(Board_GPIO_LED1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(Board_GPIO_LED2, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

    /* Turn on user LED */
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);    /* Turn red LED off */
    GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_OFF);    /* Turn yellow LED off */
    GPIO_write(Board_GPIO_LED2, Board_GPIO_LED_OFF);    /* Turn green LED off */

    unsigned int switcher = 0;

    while(1)
    {
        GPIO_write(Board_GPIO_LED0, switcher & 1);
        GPIO_write(Board_GPIO_LED1, (switcher & 2) >> 1);
        GPIO_write(Board_GPIO_LED2, (switcher & 4) >> 2);
        delay_1sec();



        switcher = (switcher + 1) % 8;
    }


    return (NULL);
}
```

Listing 3: as

3 Acknowledgement

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 4 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 4: 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) [1]. 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 4 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 [2]. 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.

References

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

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