Basic Real-Time Operating System

targeting the ARMv7-M architecture

ROS01, Rotterdam December 1, 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.

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 `delay1_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

3 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 1 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    0x5000000");
18 }
```

Listing 1: 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 1 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.