technische universität
dortmund

Bachelor thesis

Nested Preemption Fixed-Priority Scheduler for EV3OSEK

Nils Hölscher
nils.hoelscher@udo.edu
May 2018

Supervisors:
Prof. Dr. Jian-Jia Chen
M.Sc. Kuan-Hsun Chen

Technische Universität Dortmund
Fakultät für Informatik
Technische Informatik und Eingebettete Systeme
(LS-12)
http://ls12-www.cs.tu-dortmund.de

Abstract

Lego Mindstorms Robots are a popular platform for graduate level researches and college education purposes. As a portation of nxtOSEK, an OSEK standard compatible real-time operation system, EV3OSEK inherits the advantages of nxtOSEK for experiments on EV3, the latest generation of Mindstorms robots. Unfortunately, the current version of EV3OSEK still has some critical issues. In this work I address, task preemption, a common feature desired in every RTOS. Before relevant ARM specifications, parts of the OSEK standard and the historical background of EV3OSEK with nxtOSEK are explained. With this I reveal the related issues in the current design and propose corresponding solutions for EV3OSEK that fix the issues in the IRQ-Handler and the task dispatching properly, thus enabling real multi-tasking on EV3OSEK. My evaluations show that the current design flaws are solved. Along with this work, I checked the current usability of EV3OSEK and suggest some future work to enhance EV3OSEK further, to make it a competitive alternative to other OS's for the EV3 devices.

Contents

Chapter 1

Introduction

1.1 Motivation

Since 1998 Lego Inc. released a series of programmable robotics kits called Mindstorms [10], which have been extensively used in graduate level researches and college education. For the Lego Mindstorms robots of the NXT series, the OSEK standard [14] compatible real-time operating system (RTOS) nxtOSEK [5] has been widely adopted as an experimental platform [4, 19, 2]. Nevertheless, EV3, the third generation of Mindstorms robots which was released in 2013, is still not popularly used in the real-time community. The only RTOSs for EV3 robots, namely EV3RT [11] and EV3OSEK [15], were released a few years after the EV3 robots, i.e., in 2016. In this thesis I focus on EV3OSEK, the only EV3 RTOS trying to fulfil the OSEK standard.

EV3OSEK is a porting of nxtOSEK to the EV3 platform, provided by a group at Westsächsische Hochschule Zwickau [6]. Hence, it is generally compatible to applications for nxtOSEK. Instead of using the limited-sized display to capture the results, the output of EV3OSEK can be obtained directly via the EV3 Console [12] on a host machine. Moreover, unlike nxtOSEK that needs to flash the ROM on the brick, EV3OSEK can boot from a SD-Card.

During experiments with EV3OSEK, I noticed that the task preemption mechanism did not function as expected, see Figure 1.1. Gupta and Doshi [7] described similar problems after implementing nested task preemption in nxtOSEK and abandoned the project due to problems with the IRQ-Handler and dispatch routines. This motivated me to investigate if the problems were related. In course of this investigation, I discovered that EV3OSEK was unable to correctly resume preempted jobs but instead re-executed them completely. A more detailed description of the preemption behaviour of EV3OSEK as well nxtOSEK can be found in Section 1.1.1. Researchers who performed experiments on nxtOSEK should carefully examine if the flaws presented in this thesis affect their results.

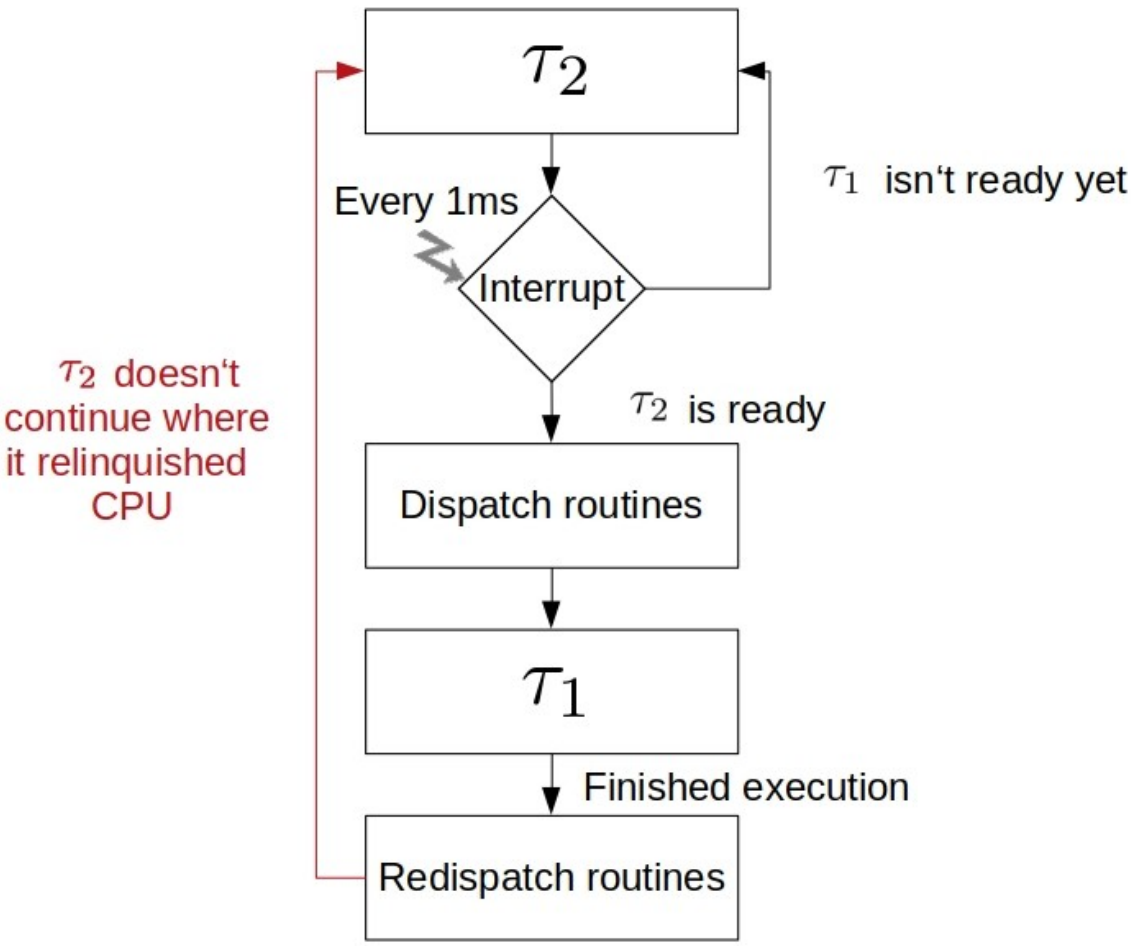


Figure 1.1: Misbehavior of context-switch at task preemption.

To narrow down the source of the problem, I examined the ARM specifications, the hardware dependent IRQ-Handler, and the task dispatching routines. In this work, I provide the corresponding solutions to the design issues in the current EV3OSEK, which are released on [8]. After solving these problems, EV3OSEK is now able to provide preemptive scheduling, and therefore multi-tasking, with all the advantages inherited from nxtOSEK.

1.1.1 Motivational Example

If the reader is not familiar with the used task/application model, read Section 2.3 before continuing.

To demonstrate the flaws in the current EV3OSEK, Figures 1.2, 1.3 and 1.4 provide examples that show the EV3OSEK preemption behaviour. Three tasks are considered: $\tau_1 = (2, 9)$, $\tau_2 = (2, 8)$, and $\tau_3 = (2, 7)$, indexed according to their priority, i.e., $p(\tau_1) > p(\tau_2) > p(\tau_3)$.

Figure 1.2 shows the expected behaviour. The second job of τ_3 released at time 7 is preempted by the second job of τ_2 released at time 8, which afterwards is preempted by the release of τ_1 at time 9, and both τ_2 and τ_3 have one unit of execution time left. After τ_1 finishes its execution, the remaining portions of τ_2 and τ_3 are executed.

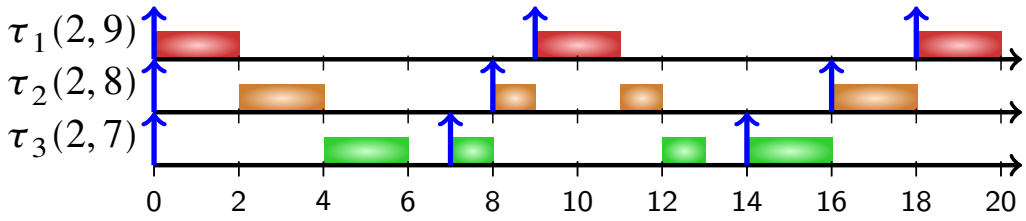


Figure 1.2: Expected behaviour: τ_2 preempts τ_3 and is afterwards preempted by τ_1. The jobs of τ_2 and τ_3 are resumed where they were preempted.

Note that in the original EV3OSEK also the problem occurs that not all tasks are activated at time 0, i.e., the first release of τ_1 was missing due to an index error. The array containing the tasks/alarms was read starting at index 1. In the code, we ensured a start at 0, hence the first job of τ_1 is released as well. For more information about this issue please refer to section 4.1.

In contrast, Figure 1.3 shows the execution behaviour of EV3OSEK.[1] Both the second job of τ_2 and the second job of τ_3 are not resumed correctly but either resumed wrongly or completely restarted which leads to one additional unit of execution time for both jobs, called overrun in Figure 1.3. Note that, due to the deadline miss at 14, the second release of τ_3 at 14 is skipped and the next job of τ_3 will be released at 21.

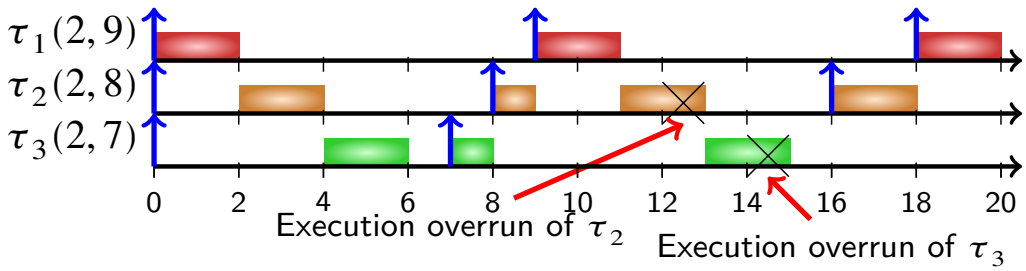


Figure 1.3: Observed behaviour in EV3OSEK: the jobs of τ_2 and τ_3 are restarted instead of resumed after a preemption.

Since EV3OSEK is a portation from nxtOSEK, this behaviour could directly be inherited. However, the flawed behaviour in the original nxtOSEK was a different one that only effected nested task preemption and is displayed in Figure 1.4. Once τ_3 is preempted by τ_2 at time 8, the interrupt from the scheduler is deactivated and hence τ_1 cannot preempt τ_2 although $p(\tau_1) > p(\tau_2)$. Only when τ_2 finishes at time 10, τ_1 is allocated

[1] The related source code is released on [3] as `NestPreemption`.

to the processor. However, when Gupta and Doshi [7] tried to fix this issue, their efforts resulted in an identical behaviour as in Figure 1.3 due to the already existing problems with the task dispatching.

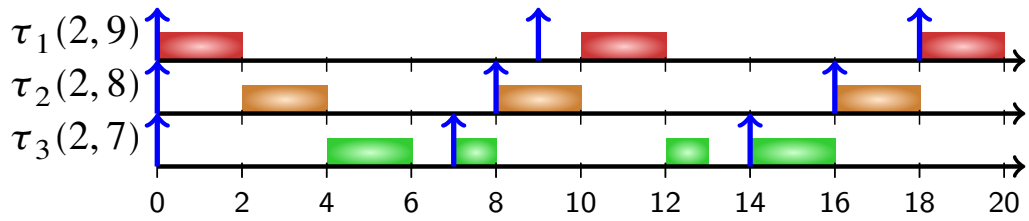


Figure 1.4: Observed behaviour in nxtOSEK: τ_3 is preempted by τ_2, but τ_2 cannot be preempted by τ_1.

Overall, the current EV3OSEK does not match the expectation when resuming previously preempted tasks. Since the misbehavior is observed right after the preempting task finishes, e.g., τ_1, this motivated me to check the routines responsible for the IRQ-Handler and the task dispatching. It turned out that the IRQ handler, expended from TexasInstruments [16], has serious issues that should have lead to complete corruption of the program counter. However, the main flaw of the IRQ handler, being unable to save the lookup register, was covered by the dispatch routines, inherited from nxtOSEK, which were unable to restore the lookup register.

1.2 The Goals of this Thesis

This thesis focuses on introducing and analysing the problems in the current design of EV3OSEK when task preemption takes place. And provides a solution to tackle these problems.

- Flawed behaviour, described regarding task preemption in EV3OSEK in Section 1.1.1, and the problems are further explained in Section 3.

- The corresponding solutions for the IRQ-Handler and the task dispatching routine are provided in Section 4, hence enabling multi-tasking under EV3OSEK.

- I evaluated my solutions, by running the example used in Figure 1.3 and predict the behavior. When the example, behaves as shown in Figure 1.2 the issue should be fixed. To tighten evaluation, an even more complex example is run and compared to, expected behavior. The results are displayed in in Section 5, showing that the provided solutions solve the problems and allow fully preemptive fixed-priority scheduling, and therefore multi-tasking, in EV3OSEK.

1.3 Structure of this Thesis

This thesis is about fixing an issue of an operating system aiming at supporting the OSEK standard. To understand the issues the underlying hardware is introduced in Section 2.1. The relevant parts of the OSEK standard are provided in Section 2.2. For improving readability and simplifying the description of tasks, the application model is described in 2.3. Moreover EV3OSEK has been ported from nxtOSEK. Therefore historical and mechanical differences are described in Section 2.4.

After the background knowledge is provided, this thesis focuses on fixing the described issues. The source code, responsible for task dispatching and interrupt handling, are described in Chapter 3. Afterwards, fixes based on the analysis, are proposed in Chapter 4. To evaluate the proposed changes, more examples are presented in Chapter 5. In addition to enabling multitasking in EV3OSEK, I also test the usability of EV3OSEK. Please read Chapter 6 to see the current state of EV3OSEK.

Finally this thesis concludes EV3OSEK in its current state and suggests some future work to further enhance EV3OSEK in Chapter 7.

Chapter 2

Background

2.1 Lego Mindstorms EV3 and EV3OSEK

In this thesis, I focus on the third generation of Lego Mindstorms robots (EV3), which are equipped with a uniprocessor ARM926EJ-S 300MHz and 64MB RAM on a TexasInstruments AM1808, running EV3OSEK with a C/C++ compatible environment. EV3OSEK [15] is a real-time operating system which aims to be compatible to the OSEK standard [14]. The brick itself is shown in Figure 2.1a.

(a) The EV3 brick [10]

(b) The EV3 Console [12]

Figure 2.1: EV3 brick and Ev3 Console.

Ev3OSEK is a recent portation [6] from nxtOSEK [5], which is only available for the older LEGO Mindstorms NXT robots. And consists mainly of three parts:

1. Drivers for sensors and actors (leJOS)

2. API for development (ECRobot)

3. OSEK-OS for the EV3 robot

This work focuses on the OSEK-OS. To obtain the output from the EV3 robots with our host machines, the EV3 Console [12] is used. Which realizes an USB to UART bridge see Figure 2.1b. It connects with one of the Lego sensor cables and a micro-USB cable. The suggested driver to access the device are provided by Texas Instruments [16].

Many RTOS's have been written for this device but this paper focuses on EV3OSEK, since it is the only RTOS that implements the OSEK standard. The interrupt handlers and the functions realising task dispatching are written in assembler files, so that the compiler does not optimize the code and breaks it.

2.1.1 Relevant CPU specifications

As mentioned before, the EV3 uses an ARM926EJ-S Processor. Since the work in this thesis is done in ARM assembly and uses mode switching, it is important to know about the CPU's specifications. The ARMv5 architecture realises these seven following processing modes:

1. User mode.

2. System mode.

3. FIQ (Fast Interrupt reQuest) mode.

4. Supervisor mode.

5. Abort mode.

6. IRQ (Interrupt ReQuest) mode.

7. Undefined mode.

In this thesis I mainly focus on User/System mode and the IRQ mode, since these are relevant for task preemption.

All of these modes use the registers `r0` to `r15`, where `r13` is the stack pointer (`sp`), `r14` is the lookup register (`lr`) and `r15` is the program counter (`pc`). Additional the CPU uses a program status register to save run-time relevant settings and information. Every mode has its own status register except User and System mode, they use the CPSR, Current Program Status Register. The other status registers are called SPSR, Special Program Status Register.

Some of these modes have banked registers. Banked registers are registers that change from mode to mode, i.e. FIQ mode addresses another register with `r8` than User Mode.

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Figure 2.2: The ARM registers. [1]

All banked registers are shown in Figure 2.2, the banked registers are marked with a grey triangle in the bottom left corner.

It is important to mention, that all modes except User and System mode have their own stack and lookup register.

2.2 Preemption in the OSEK Standard

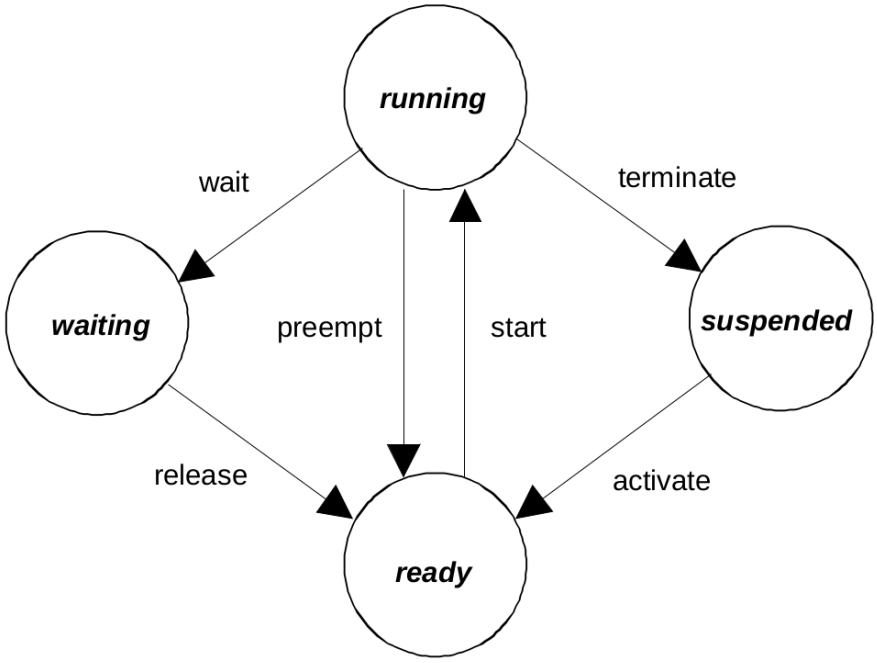


Figure 2.3: Extended task state model in the OSEK standard [14].

Here I briefly review the specifications for task preemption defined by the OSEK standard [14]. The OSEK standard defines two types of tasks, Basic task and Extended Task: 1) Basic tasks releases the CPU only on following three conditions:

- The task terminates.
- A higher priority task was activated and gets scheduled.
- An interrupt occurs

2) Besides the above three conditions, a Extended task is additionally allowed to use the system call `WaitEvent()` to enter the waiting state and relinquish its execution on the CPU. EV3OSEK supports both types of tasks. To configure an OSEK application, the OIL language [20] is also introduced in EV3OSEK containing CPU-specific configuration items. OIL files are hand-written and then generated by a system configuration tool addressing for a specific CPU platform. The OIL language is introduced in Section 2.2.1 The OSEK standard defines two different scheduling policies: non-preemptive scheduling and fully preemptive scheduling. In a non-preemptive scheduling policy, a job cannot be preempted once its execution has been started. In fully preemptive scheduling, any task is preempted, when a higher priority task enters the system and that higher priority task gets scheduled instead. The context of the preempted task is stored accordingly, so that it can resume back

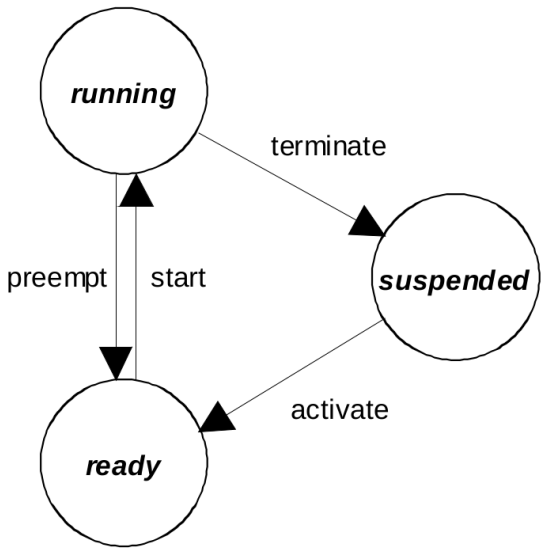


Figure 2.4: Basic task state model in the OSEK standard [14].

later on. As motivated in the introduction previously, I only focus on the full preemptive scheduling policy in this thesis. Here is a list when the scheduling in the OSEK competitive operating system should take place:

1. Successful termination of a task by calling the system routine `TerminateTask()`.
2. Successful termination of a task by calling the system routine `ChainTask()` to define a successor task.
3. Activation of a task at task level by calling the system routine `ActivateTask()`.
4. When the task calls the system routine `WaitEvent()` (only extended tasks).
5. When releasing a waiting task.
6. When a task releases a resource by calling the system routine `ReleaseResource()`.
7. At the return from interrupt level to task level.

With my enhancements, all the above features work well and EV3OSEK is still competitive to the OSEK standard. To specify all task relevant information for an OSEK operating system, the OIL language is used.

2.2.1 The OIL language

Whenever a task is compiled for an OSEK compatible operating system, the operating system needs information about the compiled task. With this information the operating system is able to manage the tasks as specified by the programmer. The OIL language is

used to convey all this relevant information to the operating system. This Section covers parts of the OIL language relevant for this thesis, rephrasing the OIL-manual [13]. These parts are the specifications for a task and an alarm.

```
1  #include "kernel.h"
2  #include "kernel_id.h"
3  #include "ecrobot_interface.h"
4  #include "stdio.h"
5
6  #define LOOP1(ID,End) if(ID==1){a++;}
7    else if(ID==2){b++;}else if(ID==3){c++;}
8    for(int k=0; k<End; k++){j|=j<<1;}
9  #define LOOP2(ID,End) if(ID==1){a++;}
10   else if(ID==2){b++;}else if(ID==3){c++;}
11   for(int l=0; l<End; l++){j|=j<<1;}
12 #define getTime() current_time = systick_get_ms()-first_start;
13
14 DeclareTask(Task1);
15 DeclareTask(Task2);
16 DeclareCounter(SysTimerCnt);
17
18 void user_1ms_isr_type2(void){
19   (void)SignalCounter(SysTimerCnt);
20 }
21
22 static U32 current_time;
23 static U32 first_start = 0;
24 static int j = 100;
25 static int End;
26 static int a,b,c = 0;
27
28 TASK(Task1)
29 {
30   getTime();
31   printf("Task 1(%i, %i) start at %i.\n\r",a,b,current_time);
32   LOOP1(1,562*500);
33   LOOP2(1,562*1500);
34   getTime();
35   printf("Task 1(%i, %i) end at %i.\n\r",a,b,current_time);
36   TerminateTask();
37 }
```

Listing 2.1: First 38 lines of an example used in this thesis.

At first a task or a counter has to be declared in its c code, as in Listing 2.1 lines 15 and 17. A task has to been declared, otherwise can not compile, see Listing 2.1 lines 29 to 38. With this the c dependent part is covered.

Now the task has to be specified in the OIL language. In Listing 2.2 a OIL implementation for Task1 and Task1_Alarm and its counter SysTimerCnt is shown. At first in lines 3 to 16 the operating system behavior is specified, for example line 7 allows the use of extended tasks. When ever a task is executed, it has to be applied to an appmode, which is defined in listing 2.2 line 19. The appmode has no standard attributes. Appmode is the object used to define OSEK properties for an OSEK application mode. In a CPU, at least one appmode has to be defined, to run a tasks in an OSEK operating system.

Task1 from the example in listing 2.1 is defined in listing 2.2 lines 23 to 30. A task can be specified with the following objects:

1. Priority: OSEK defines the lowest priority as 0. Larger values mean higher task priority. (Listing 2.2 line 26)
2. Schedule: Defines wether a task is allowed to be preempted (FULL) or not (NON). (Listing 2.2 line 28)
3. Activation: Defines the number of active tasks of its type to be active at once. (Listing 2.2 line 27)
4. Resource: Defines resource, that are used by the task.
5. Event: Define events, that extended task might react to.
6. Message: Define messages accessed by the task.
7. Stacksize: Defines the tasks stack size. (Listing 2.2 line 29)

In this example an alarm is used to activate Task1 every 8 seconds, it can be found at listing 2.2 lines 32 to 45. An alarm can be defined with three attributes, counter, action and autostart. The counter specifies the counter that is used to call the alarm. Action defines, what happens when the alarm gets triggered. In my example this would be the activation of Task1.

To fully define an alarm, a counter has to be defined. The counter is defined in listing 2.2 lines 48 to 53, setting its constraints and tick base.

```
1  #include "implementation.oil"
2
3  CPU ARM926EJ_S
4  {
5    OS LEJOS_OSEK
6    {
7      STATUS = EXTENDED;
8      STARTUPHOOK = FALSE;
9      ERRORHOOK = FALSE;
10     SHUTDOWNHOOK = FALSE;
11     PRETASKHOOK = FALSE;
```

```
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = FALSE;
  };

  /* Definition of application mode */
  APPMODE appmode1{};

  /* Definition of tasks */

  TASK Task1
  {
    AUTOSTART = FALSE;
    PRIORITY = 7;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 4096;
  };

  ALARM Task1_Alarm
  {
    COUNTER = SysTimerCnt;
    ACTION = ACTIVATETASK
    {
      TASK = Task1;
    };
    AUTOSTART = TRUE
    {
      ALARMTIME = 1;
      CYCLETIME = 8000;
      APPMODE = appmode1;
    };
  };

  /* Defintion of counter and alarm */
  COUNTER SysTimerCnt
  {
    MINCYCLE = 1;
    MAXALLOWEDVALUE = 10000;
    TICKSPERBASE = 1;
  };

};
```

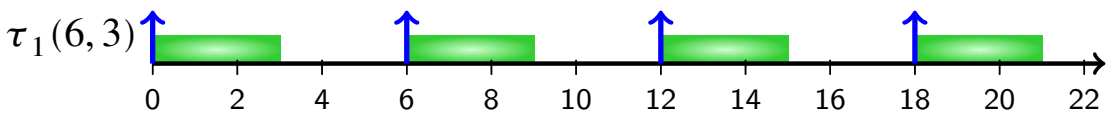
Listing 2.2: OIL implementation of a task and an alarm.

2.3 Application Model

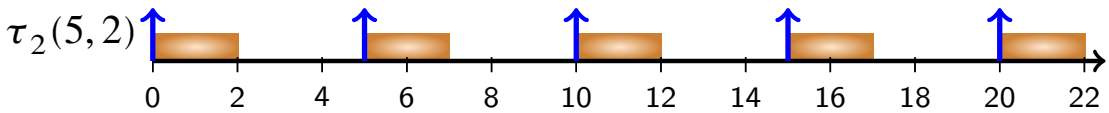
The scheduling of n independent periodic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in a uniprocessor system, is defined as follows. Each task is defined by a tuple $\tau_i = (C_i, T_i)$ where T_i is an interarrival time constraint (or period) and C_i the tasks worst-case execution time.

To illustrate this definition and its behavior, two examples are provided, see Figure 2.5a and Figure 2.5b. Task τ_1 has a cycle time/interval of $T_1 = 6$ time units and a worst case execution time of $C_1 = 3$ time units, written as $\tau_1 = (6, 3)$. Task τ_2 executes more rapid with $T_2 = 5$ time units but executes shorter with $C_i = 2$ time units, written as $\tau_2 = (5, 2)$

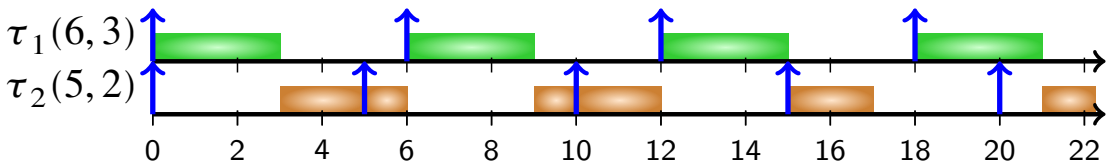
The deadlines are assumed to be implicit, i.e., if a task instance (job) is released at θ_a, it must be finished before $\theta_a + T_i \ \forall \tau_i$. I consider fully preemptive fixed-priority scheduling, i.e., each task τ_i is associated to a predefined priority $p(\tau_i)$, since the issue considered in this thesis only happens under a fully preemptive scheduling policy. Fully preemptive scheduling is illustrated in Figure 2.5c. In this figure are both example tasks run in a fully preemptive schedule with $p(\tau_1) > p(\tau_2)$.



(a) First example τ_1: $\Gamma = \{\tau_1\}$, $\tau_1 = (6, 3)$.



(b) Second example τ_2: $\Gamma = \{\tau_2\}$, $\tau_2 = (5, 2)$.



(c) Combined example τ_1 and τ_2: $p(\tau_1) > p(\tau_2)$.

Figure 2.5: Some examples of the used application model.

2.4 Comparison of nxtOSEK and EV3OSEK

This section starts with introducing nxtOSEK and then compare it with EV3OSEK.

NxtOSEK is a portation of TOPPERS/ATK (Automotive Kernel, formerly known as TOPPERS/OSEK) and TOPPERS/JSP [18]. To use the Sensors and actors of the nxt robot the leJOS NXJ [9] device drivers are also used. A basic setup of these components and the underlying hardware can be seen in Figure 2.6

When the group at Westfälische Hochschule Zwickau started porting, they first evaluated which component of nxtOSEK they needed for the EV3. These components are marked red in Figure 2.6. Because they were lacking time the TOPPERS/JSP part was not ported to EV3OSEK, therefore only the OSEK part of nxtOSEK was ported to EV3 and the device drivers. The group also decided not to port the Enhanced NXT firmware/NXT BIOS, cause the EV3 is able to directly load programs from the SD card without running its original operating system. This is also the reason that EV3OSEK needs no flashing to the device ROM.

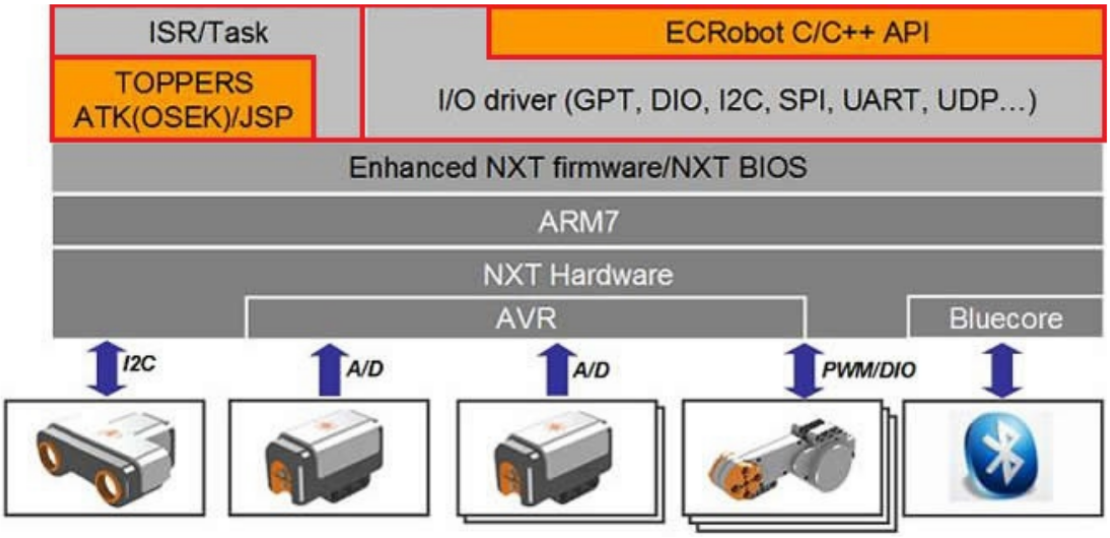


Figure 2.6: Parts ported from nxtOSEK to EV3OSEK are marked red. [6]

Chapter 3

Original Task Preemption in EV3OSEK

In this section, I first review the current design of the sources that are responsible for IRQ-Handler[1] and task dispatching in EV3OSEK[2]. Afterwards I point out the source of the aforementioned issue.

3.1 IRQ-Handler

As specified in the OSEK standard, EV3OSEK has a hook routine named `user_1ms_isr_type2()`, which is invoked from a periodic interrupt service routine in category 2 (ISR2) every 1 ms. This hook routine can be redefined by the programmer but it should always execute the system routine `SignalCounter()` to maintain the process of EV3OSEK. However this design partially violates the OSEK standard.

Once an ISR occurs, the CPU loads the IRQ-Handler shown in Figure 3.1. It first saves the context of the interrupted task. Now it can handle the ISR without overriding registers of the interrupted task. The address of the ISR that called the interrupt is saved in the `AINTC_HIPVR2` register by the hardware interrupt handler. When the ISR has finished its execution, it returns back to the IRQ-Handler.

If the ISR was not `systick_ISR_c`, i.e., the function that handles the 1ms timer, the task context is restored and the IRQ-Handler returns to the interrupted task. But if the ISR was `systick_ISR_c`, the button-routine and `user_1ms_isr_type2()` are executed. In the hook function `user_1ms_isr_type2()`, `SignalCounter()` will set the Boolean `addr_should_dispatch` to TRUE if the current running task is not the highest priority task anymore.

[1] IRQ stands for Interrupt ReQuest from the underlying hardware.

[2] The reviewed files are downloaded from https://github.com/ev3osek/ev3osek/tree/master/OSEK_EV3. The latest update for exceptionhandler.S and cpu_support.S was on 18 Sep 2016.

In case that `should_dispatch` is false, the task context is restored and the IRQ-Handler returns to the interrupted task. In the other case, when `should_dispatch` is set to true, the task context is restored, i.e., all registers $r0$ to $r12$ and the lookup register. Afterwards the IRQ-Handler loads the dispatch routine address in the lookup register and loads it with an offset of -4.

Within the analyses, I noticed that there are five issues in the current implementation as shown in Listing 3.1:

1. The lookup register contains the return address of the preempted task and is always overwritten.

2. The lookup register has to be saved in the stack for the CPU User-/System-mode before jumping to the dispatch routine, since different CPU modes may have their own lookup registers.

3. The lookup register, which already contains the address of the dispatch routine, loads with an offset of -4. This is not necessary, since the address is loaded from the memory instead of the decoder.

4. The status register also has to be saved/restored, when interrupting a task, since it also contains information about the interrupted task.

5. `SignalCounter()` in ISR2 determines whether the task dispatching should take place or not. However, the OSEK standard defines that scheduling should be bound to ISR2 rather than `SignalCounter()`.

```
LDMFD    r13!, {r0-r12, lr} @ Restore registers from IRQ stack

LDR      lr, =dispatch
SUBS     pc, lr, #4         @ SUBS = Return from exception
```

Listing 3.1: Assembler code fragment responsible for the five issues related to the IRQ-Handler.

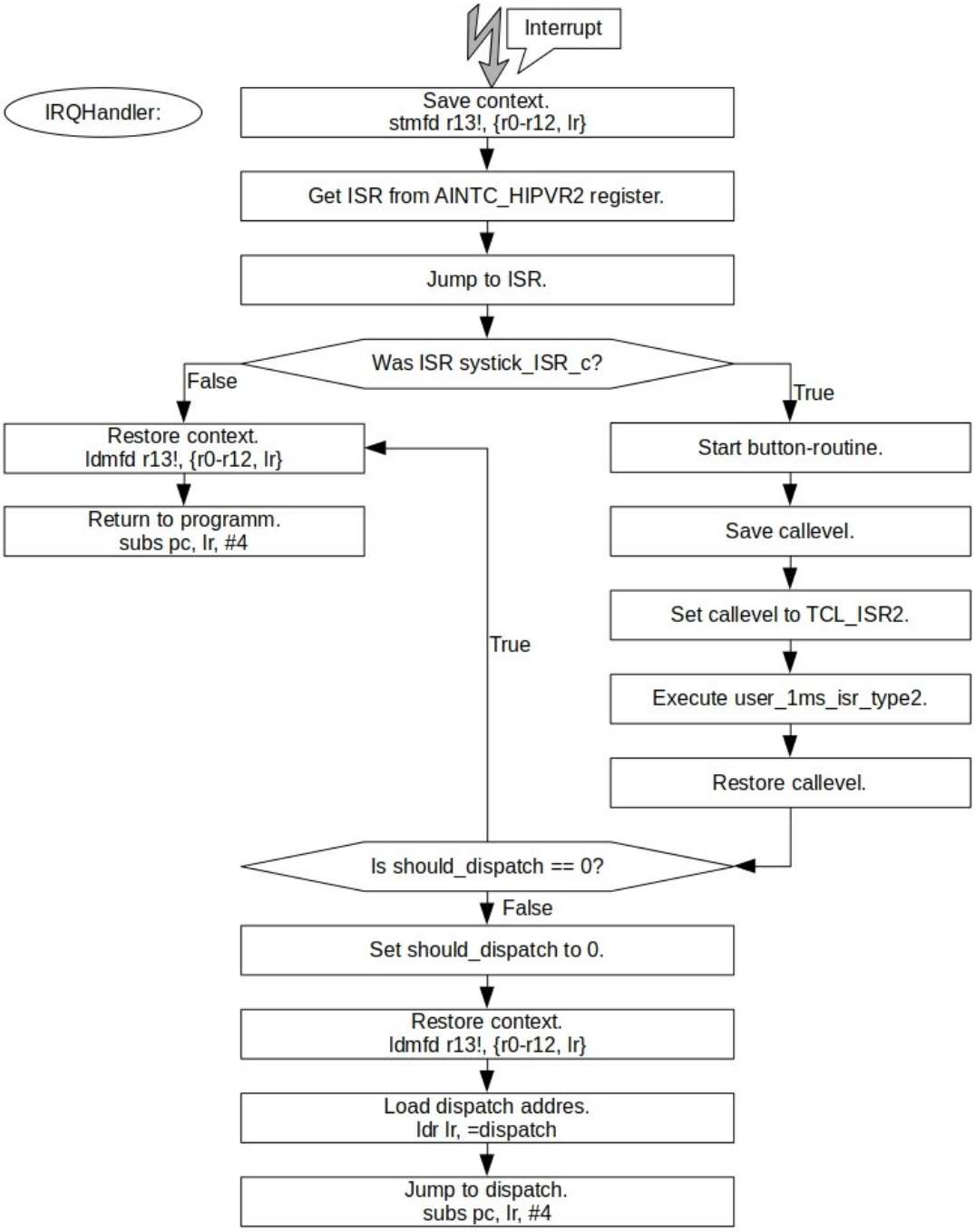


Figure 3.1: Flowchart of the current IRQ-Handler in EV3OSEK.

3.2 Task Dispatching

Before introducing the current design of task dispatching in EV3OSEK, I list some notations used in the implementation:

- `runtsk`: Address of the running task ID.
- `schedtsk`: Address of the highest priority task.
- `tcxb_pc[]`: Array for the program counters of tasks.
- `tcxb_sp[]`: Array for the stack addresses of tasks.

For the simplicity of the presentation, I further use τ_{low} and τ_{high} in the rest of the section to describe the scenario that there is an executing task τ_{low} which is going to be preempted by a ready task τ_{high} with higher priority.

When τ_{high} is ready in EV3OSEK, the currently running task τ_{low} has to relinquish its right on the CPU. As shown in Figure 3.2, the scheduler in EV3OSEK has three main steps: *Dispatch*, *Preempt* and *Reload*, detailed as follows:

- **Dispatch:** To preempt a task, the IRQ-Handler calls the dispatch routine, which saves the context of the preempted task on the tasks stack, and stores the stack pointer in `tcxb_sp[runtsk]`. The address of `disptach_r` is stored in `tcxb_pc[runtsk]`, allowing the task context to be restored when it is resumed.
- **Preempt:** After the dispatch step, the higher priority task is executed on the CPU. Once it finishes, it calls `TerminateTask()` to trigger the scheduler with `start_dispatch` to reload the lower priority task. In `start_dispatch`, at first `runtsk` is set to `schedtsk`, so that the scheduler knows that the current running task is the currently highest priority task in the system. Afterwards, the stack pointer is restored back from `tcxb_sp[runtsk]` and `dispatch_task` is called.
- **Reload:** In `dispatch_task` the program counter of the preempted task is restored from `tcxb_pc[runtsk]`. Instead of starting the task like normally, the preempted task executes `dispatch_r` to restore its context from the stack and enable interrupts, which were disabled by `TerminateTask()`.

There are three issues in the current implementation:

1. In `start_dispatch`, the array `tcxb_sp[runtsk]` is addressed with the address of `runtsk` and not with the value of `runtsk`. See Listing 3.2:

```
start_dispatch:
        ldr     r0, =schedtsk           // runtsk = schedtsk
        ldr     r0, [r0]
```

```
ldr     r1, =runtsk
str     r0, [r1]

cmp     r0, #INVALID_TASK       // if (runtsk == INVALID)
beq     pre_idle_2              // jump to "pre_idle"

ldr     r1, =tcxb_sp            // sp = tcxb_sp[runtsk]
ldr     sp, [r1,r0, asl #2]
```

Listing 3.2: `tcxb_sp[runtsk]` is addressed with the address of `runtsk` and not with the value of `runtsk`.

2. In `dispatch_r`, the lookup register is loaded from the stack without ^ flag, and the status bits are not loaded as well. See Listing 3.3:

```
dispatcher_r:
        BL      IntMasterIRQEnable      // enable interrupts
        BL      IntMasterFIQEnable
        ldmfd   sp!, {r0-r12}
        ldmfd   sp!, {lr}       // restore return address
        MOV     pc, lr  //. jump to the address
```

Listing 3.3: The lookup register is loaded without ^ flag, the statuts bits are not loaded at all.

3. The status register has to be part of the save context routine in `dispatch` and of the restore context routine in `dispatch_r`.

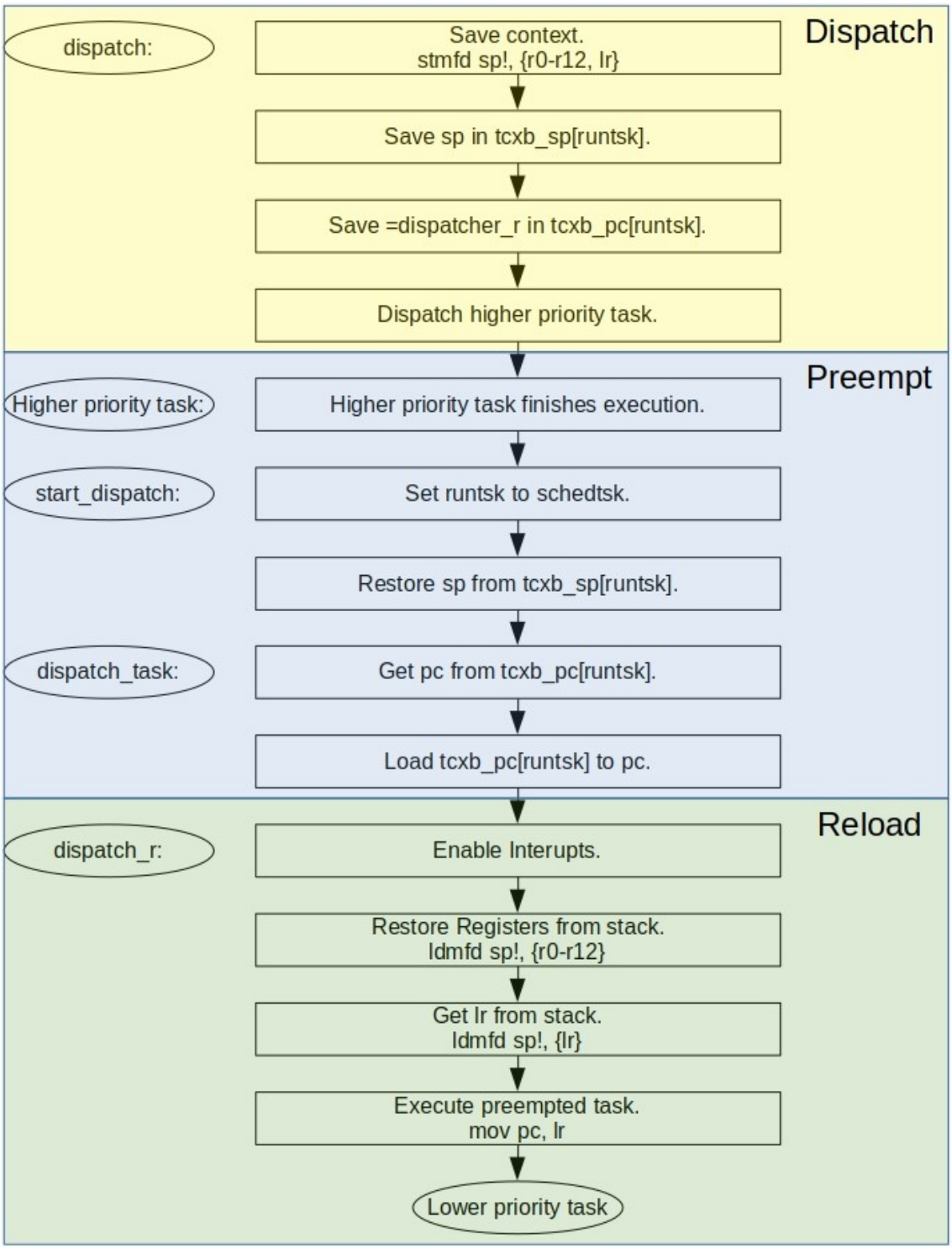


Figure 3.2: Task dispatching and re-dispatching.

Chapter 4

Fixing Task preemption in EV3OSEK

After discussing the flaws in the current EV3OSEK, I here present how to fix the task preemption accordingly. Please note that EV3OSEK's IRQ-Handler is not inherited from the portation of nxtOSEK and hence the nested task preemption problems in nxtOSEK are not inherited from the IRQ-Handler but the dispatch routines.

Summarizing the observations in Chapter 3, the proposed solutions can be summarized as follows:

- correcting the register operations in the IRQ-Handler,
- correcting the errors in `start_dispatch`,
- correcting the errors in `dispatch_r`,
- adding status register to context save/restore routines, and
- changing the trigger point of the task dispatching.

The flowcharts for the IRQ-Handler and the dispatching are shown in Figure 4.1 and Figure 4.2, respectively, where the red blocks are provided by my solutions. In the rest of this section, I explain more details about my solutions.

Correcting the register operations in the IRQ-Handler: In the current EV3OSEK, the lookup register in the IRQ-Handler contains the address of the preempted task and is overwritten. Moreover, the lookup register has to be saved in the User-/System-mode stack before jumping to dispatch, since IRQ- and User-/System-mode have their own lookup registers. Note that there are different execution modes in modern CPUs, where some modeshave their own registers called banked registers which are not shared with other modes, as described in section 2.1.1.

The issues can be solved by writing the lookup register in one of the registers r0-r12, switching to System-mode in the IRQ-Handler, pushing the register containing the lookup

register on the System-mode stack, and switching back. This solution requires to remove the instruction that stores the lookup register on the system stack in the dispatch routine. As a result, the dispatch routine can no longer be called from User-/System-mode. To resolve this, the branch `dispatch_irq` is introduced right after the dispatch routine stores the lookup register, as this is already done in the IRQ-Handler. Now the IRQ-Handler calls `dispatch_irq` and it is still possible to call the dispatch routine from User-/System-mode.

Another issue in the IRQ-Handler is that the lookup register contains the address of the dispatch routine, but it is loaded with an offset of -4. This can be easily fixed by removing the unnecessary offset from the branch instruction. The updated IRQ-Handler is displayed in Figure 4.1.

Correcting the errors in `start_dispatch`: Originally, the array `tcxb_sp[runtsk]` is addressed with the address of `runtsk` and not the value of `runtsk`. I solved this by loading the value from `runtsk` before accessing the array. This solution might fix the original issue of nxtOSEK described in [7] as well but needs implementation and verification.

Correcting the errors in `dispatch_r`: As shown in Figure 4.2, the lookup register is loaded from the stack without the ˆ flag in `dispatch_r`, so that the status bits are not loaded as well. This can be easily resolved by adding the ˆ flag to the load instruction. By doing so, the program status is loaded into the status register correctly. The enhanced dispatching is detailed in the flowchart in Figure 4.2.

Save/Restore status register with context: In the IRQ-Handler and dispatch routines, the status register is not part of saving/restoring context. However the status register contains information about comparing instructions for the interrupted/dispatched task. By saving and restoring the status register together with the context of registers, the information in $r0$ to $r12$ is not lost.

Changing the trigger point of the task dispatching: In the original implementation, `SignalCounter()` must be called by the hook routine `user_1ms_isr_type2()`, which is used to manage task scheduling. As defined in the OSEK standard, the task scheduling must be bound to ISR2. To fix this, I moved the code setting the flag `should_dispatch` to the function `SetDispatch()` and call it after `user_1ms_isr_type2()` has finished.

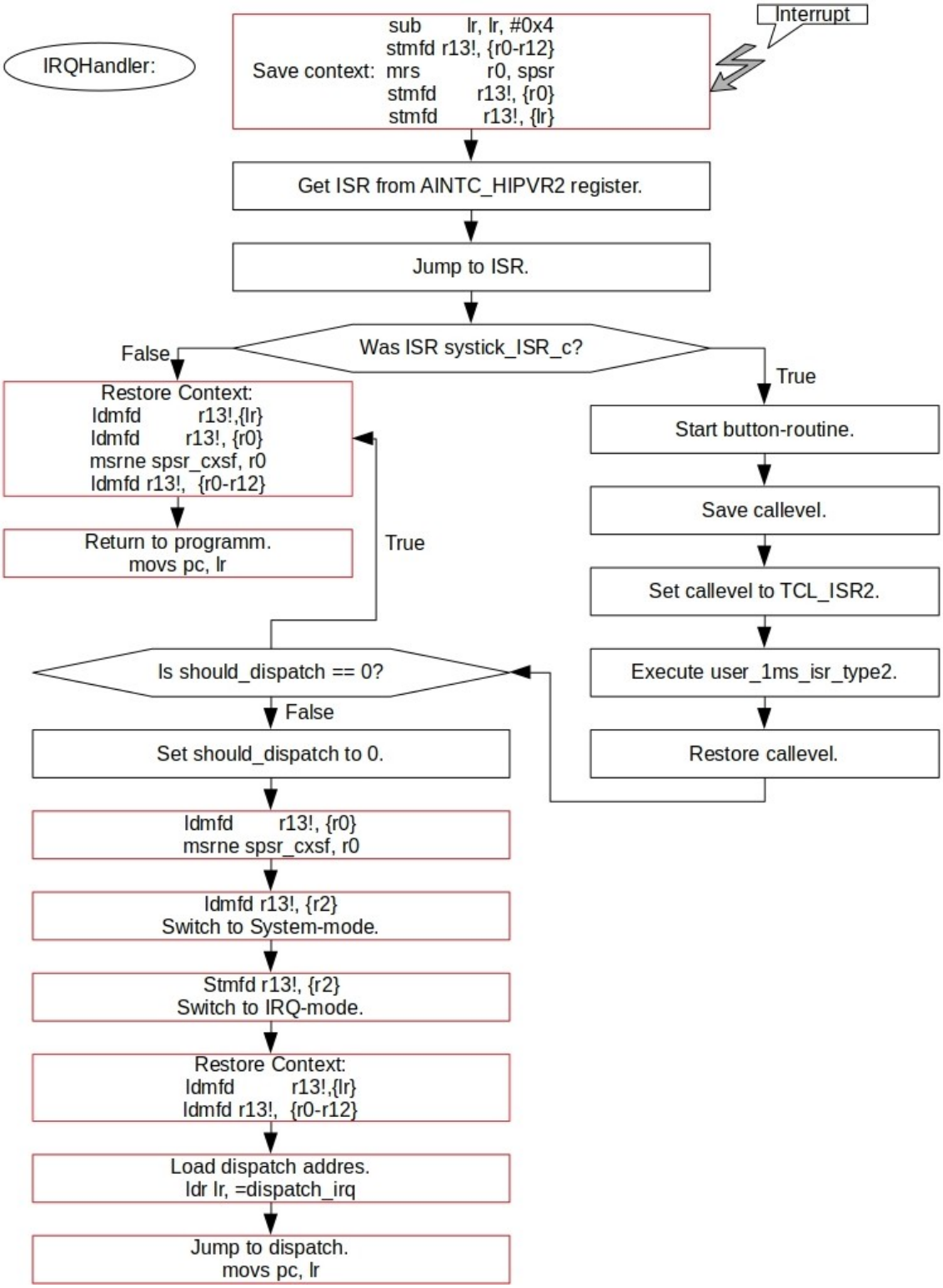


Figure 4.1: Enhanced version of the IRQ-Handler.

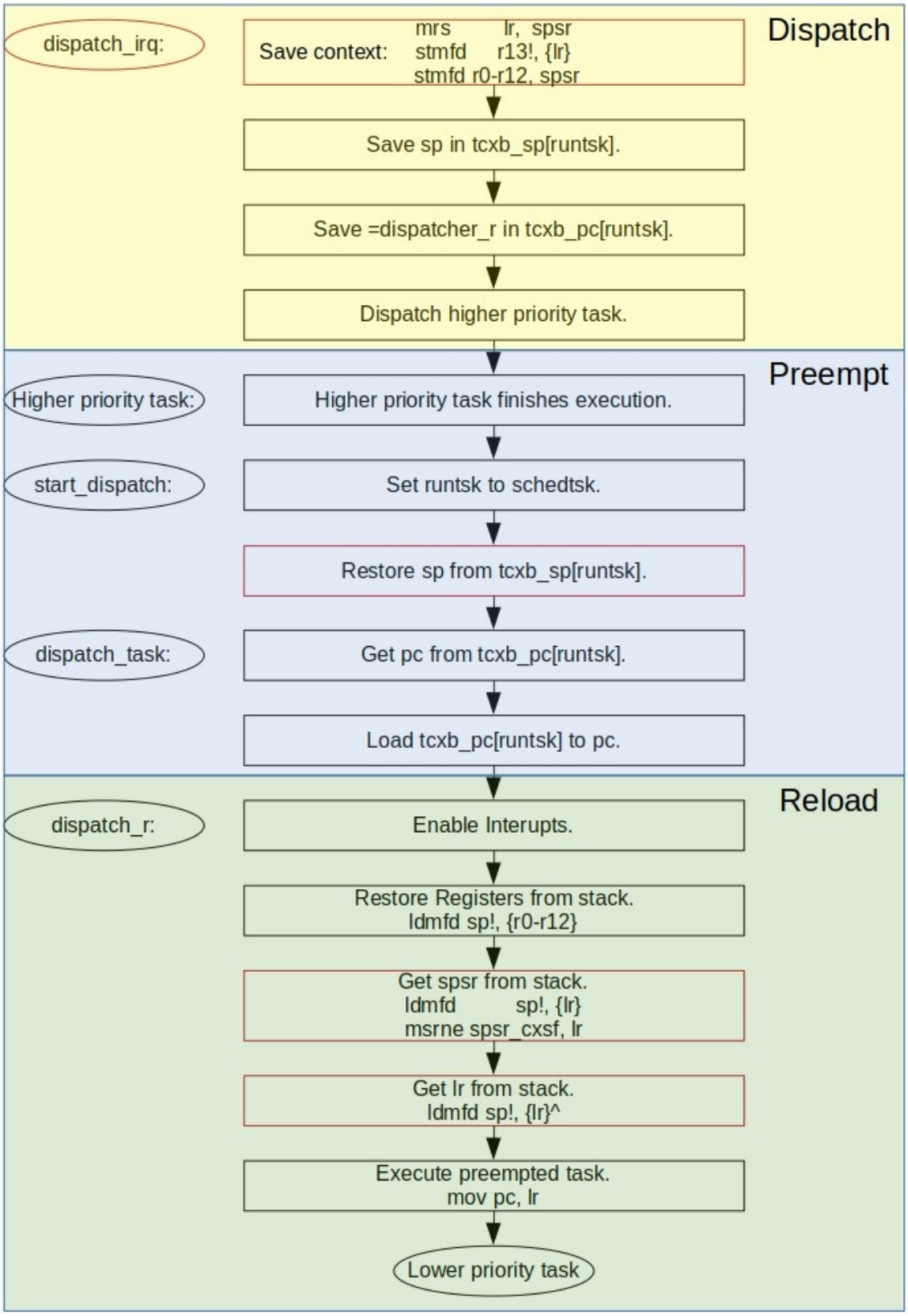


Figure 4.2: Enhanced version of task dispatching.

4.1 Alarm initialization in EV3OSEK

As mentioned, a minor issue about how alarms are activated in the operating system had to be fixed. To understand this issue I first introduce how alarms are managed in an OSEK compatible operating system. When an alarm expires, its call back routine is triggered. A call back routine is a short function, that provides the alarms service i.e. task activation and set events. Since every alarm is bound to a counter the call back routine gets called from `SignalCounter(CounterType cntid)`, when the alarm expires. This procedure is also part of the `user_1ms_isr_type2()`.

The `user_1ms_isr_type2()` calls `SignalCounter(CounterType cntid)` for every defined counter every 1 ms and when an alarm expires its call back routine gets called. This procedure is shown in FIgure 4.3.

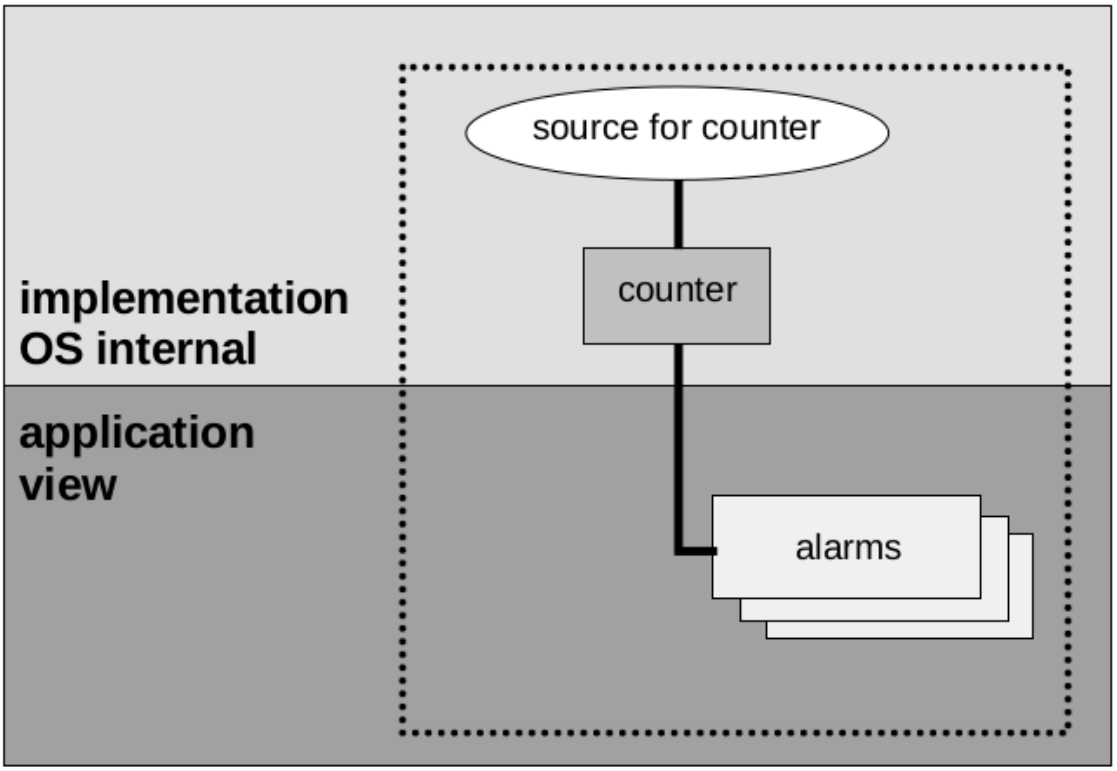


Figure 4.3: Alarm management in OSEK. [14]

What I observed was, that the first defined alarm was not triggered at time 0. So I ensured that all alarms are now initialized at time 0.

Chapter 5

Evaluation of the Proposed Solutions

In this section I evaluate my implementations by running the examples, used to show the faults in EV3OSEK, and a more complex one. This decision is based on the facts, that EV3OSEK has no debugger implemented in its tool chain. Formal proof management systems like Coq [17] are also not usable for evaluation, since my implementations are very dependant on the underlying hardware manufactured by Texas Instruments.

As illustrated in Section 1.1.1, the current EV3OSEK is not able to provide task preemption correctly. With the enhancement mentioned in the previous chapter, task preemption, and hence multi-tasking, now should work properly. I present an additional example with three tasks to evaluate my proposed solution in EV3OSEK[1].

In the following experiment, I considered a task set which is schedulable in a correct preemptive fixed-priority scheduling system while in the current EV3OSEK the unexpected additional workload due to task preemption leads to deadline misses. Once a job misses its deadline, the next job is only released after the current job is finished and hence the number of releases is reduced. Therefore, by checking if the number of jobs released in the current version of EV3OSEK and in my enhanced version of EV3OSEK is identical, I can determine whether my enhancement solved the discovered problem. The related source code can be found at [8].

Tasks τ_1, τ_2, and τ_3 all print out the following line right after it starts/finishes: "Task $\tau_i(l_1, l_2, l_3)$ starts/ends at t_{ms}". t_{ms} stands for the time point when a task starts or finishes its execution. τ_1, τ_2, and τ_3 all run roughly 2000 ms and priority's are $p(\tau_1) > p(\tau_2) > p(\tau_3)$. The tasks are released as follows:

- τ_1 releases at 0 s with a period 5 s.

[1] Please note that testing the nesting depth is not necessary. As the task stack for context-switch is managed in the OIL file, the management of the stack should be handled by the programmers.

- τ_2 releases at 0 s with a period 8 s.
- τ_3 releases at 0 s with a period 10 s.

I verified that all the task preemptions behave as I expect over a certain amount of time, checking the resulting log file, and if the number of jobs for each task is exactly as predicted in advance. If there is no additional execution time after preemptions (like in the current EV3OSEK), there should be no unexpected interference affecting the job releases. I also intend to show that the program counter does not get corrupted any more, even after long run times, i.e., 10 min.

I first derived an equation to predict the exact number of jobs l_i after a certain amount of time that is a multiple of 10 seconds. Since the least common multiple of three tasks' periods is 40 seconds, the so-called hyper-period, the following equation gives the number of jobs from τ_i in a $10 \times t$ second long interval:

$$\begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} = \begin{pmatrix} \frac{8}{4} \times t \\ \frac{5}{4} \times t \\ \frac{4}{4} \times t \end{pmatrix} = \begin{pmatrix} 2t \\ 1.25t \\ t \end{pmatrix} \tag{5.1}$$

The equation is detailed as follows:

- l_1 equals $2t$: τ_1 is released and finishes two times in 10 s.
- l_2 is $1.25t$: τ_2 releases and finishes 5 times in a hyper-period of 40, every 10 s it has on average $1.25t$ releases.
- l_3 is t: τ_3 has one release every 10 seconds.

I can now predict l_1, l_2 and l_3 after an interval of 10 min.

$$t = 600001(ms) \approx 60 \times 10sec \Rightarrow \begin{pmatrix} l_1(60) = 120 \\ l_2(60) = 75 \\ l_3(60) = 60 \end{pmatrix} \tag{5.2}$$

In the current version of EV3OSEK the example hangs after 7000 ms, because the program counter is set to a random address. Before it hangs, the outputs are as follows:

```
Task 1(0, 0, 0) start at 1.
Task 1(1, 0, 0) end at 2001.
Task 2(1, 0, 0) start at 2004.
Task 2(1, 1, 0) end at 3995.
Task 3(1, 1, 0) start at 3998.
Task 1(1, 1, 1) start at 5001.
Task 1(2, 1, 1) end at 6992.
```

Listing 5.1: Output generated with the evaluation example using the current EV3OSEK.

With my enhancement, the aforementioned problem does not exist anymore in the enhanced version of EV3OSEK. The output can be found at listing 5.2.

```
Task 1(0, 0, 0) start at 1.
Task 1(1, 0, 0) end at 2005.
Task 2(1, 0, 0) start at 2008.
Task 2(1, 1, 0) end at 4003.
Task 3(1, 1, 0) start at 4005.
Task 1(1, 1, 1) start at 5001.
Task 1(2, 1, 1) end atress 6995.
Task 3(2, 1, 1) end at 7998.
Task 2(2, 1, 1) start at 8001.
Task 2(2, 2, 1) end at 9995.
Task 1(2, 2, 1) start at 10001.
Task 1(3, 2, 1) end at 11996.
Task 3(3, 2, 1) start at 11998.
Task 3(3, 2, 2) end at 13993.
Task 1(3, 2, 2) start at 15001.
...
Task 3(117, 73, 59) end at 583995.
Task 2(117, 73, 59) start at 584001.
Task 1(117, 74, 59) start at 585001.
Task 1(118, 74, 59) end at 586996.
Task 2(118, 74, 59) end at 587995.
Task 1(118, 74, 59) start at 590001.
Task 1(119, 74, 59) end at 591996.
Task 3(119, 74, 59) start at 591999.
Task 2(119, 74, 59) start at 592003.
Task 2(119, 75, 59) end at 593998.
Task 1(119, 75, 60) start at 595001.
Task 1(120, 75, 60) end at 596996.
Task 3(120, 75, 60) end at 597992.
Task 1(120, 75, 60) start at 600001.
```

Listing 5.2: Output generated with the evaluation example using the enhanced EV3OSEK.

Hence, I conclude that my enhancement fixed the problems in EV3OSEK regarding task preemption which not only resulted in unexpected execution behaviour but also in system crashes.

Chapter 6

Current state of EV3OSEK

The main focus in this chapter lies with the peripheral sensors and actors of the Lego Mindstorms EV3, because they are the main benefit of EV3. Most of the presented information here is from the porting report of Westfälische Hochschule Zwickau. [6]. All tests regarding the mentioned sensors and actors have been recreated and are not taken from the report.

EV3OSEK uses the ECRobot-API to communicate with the peripherals. The API was already used in nxtOSEK and was ported to be fully compatible. With an compatible API and OS, applications written for nxtOSEK should work with EV3OSEK. When the group from Westfälische Hochschule Zwickau ported nxtOSEk to EV3, their main goal was to have a working line follower robot. Due to time limitations, only the peripherals needed for this goal were written and some more with easy portability. The peripherals can be divided in groups as follows:

1. Digital devices communicating with the I2C standard.

2. Analog devices communicating with an Analog-to-digital converter realised in the brick with the Serial-Peripheral-Interface (SPI) Bus.

3. Motors communicating with Pulse-Width-Modulation (PWM).

4. And others like Bluetooth and the display.

The standards, controllers and interfaces are not discussed here, since they are not part of the thesis. For the first two groups tables are provided, where ✓ stands for the device is working with the provided API, ~ device works but not as intended, *X* is the device is not implemented at all and *n.t.* the device was not tested by me but is implemented. All devices that are not mentioned in this section, are not implemented.

In Table 6.1 all analog sensors are presented. It is worth mentioning, that none of the analog sensors are implemented with interrupts. This means that all sensors have to be

Device	Status	Comment
Lego EV3 touch sensor	✓	Is not implemented with interrupt, therefore the state of the sensor has to be checked active.
Lego EV3 sound sensor	*n.t.*	Implemented but not tested.
Lego EV3 light sensor	~	Not working always responses with 0.
Lego NXT light sensor	✓	Works.

Table 6.1: List of tested analog devices.

checked by a task, running with very small cycle times. Which causes a massive overhead. The touch sensor is the only original EV3 device, that is supported.

Device	Status	Comment
Lego EV3 ultrasonic sensor	~	Every third API call returns 0.
Lego EV3 gyro sensor	*X*	Not implemented.
HiTechnic gyro sensor	✓	Works.
HiTechnic acceleration sensor	*n.t.*	Implemented but not tested.
HiTechnic compass	*n.t.*	Implemented but not tested.
HiTechnic color sensor V2	*n.t.*	Implemented but not tested.

Table 6.2: List of tested digital devices.

Table 6.2 lists all digital devices. HiTechnic Sensors are most present in the EV3OSEK ECRobot-API. Only the HiTechnic gyro sensor was tested by me, since it was the only HiTechnic sensor available to me. The digital sensors also have to be checked by task, causing the same problems as mentioned in the paragraph about analog sensors.

Motors are also one of the device groups as mentioned before. But since EV3OSEK only supports the NXT servo motor in its API, this group is the most lacking. In contrary to the other two device classes the motor driver is using FIQ interrupts. Other peripherals like Bluetooth, the LCD-display, WiFi and a memory controller for the SD-Card are not implemented at all.

Chapter 7

Conclusion

Within the scope of this bachelor thesis, EV3OSEK and its historical origin of nxtOSEK have been described. In addition this thesis introduced the OSEK standards task model and parts of the ARMv5 structure.

As initially stated, EV3OSEK had a major issue regarding task preemption, which has been analysed and described without a debugger. The responsible sources have been the IRQ handler and the dispatch routines. Fortunately all faults in the current implementation regarding task preemption have been found and fixed. Furthermore an issue regarding alarm releases at OS start up in EV3OSEK has been fixed, the source code of my enhancements can be found at [8].

Using a complex example to evaluate task preemption, I have been able to show, that the IRQ handler and dispatch routines are working as intended, and the program counter no longer gets corrupted.

I checked the EV3OSEK peripheral drivers and their sources, in regards to theirs usability. Most lacking are the motor drivers, since only the NXT servo motors are supported by EV3OSEK and nearly none of the original EV3 Lego hardware, which comes with the brick. Therefore I suggest future work to mainly address the device drivers in EV3OSEK, so that at least all the original EV3 sensors and actors can be used. This would make EV3OSEK a real choice as an RTOS for the EV3 device.

List of Figures

Listings

List of Tables

Bibliography

[1] *ARM9EJ-S Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0222b/DDI0222.pdf. ARM. URL: https://www.arm.com/.

[2] M. Canale and S. C. Brunet. "A Lego Mindstorms NXT experiment for Model Predictive Control education". In: *2013 European Control Conference (ECC)*. July 2013, pp. 2549–2554.

[3] Kuan-Hsun Chen. *Motivational Examples for the flaws in EV3OSEK*. https://github.com/kuanhsunchen/ev3osek/tree/master/example. 2017. URL: https://github.com/kuanhsunchen/ev3osek/tree/master/example.

[4] Kuan-Hsun Chen et al. "Compensate or Ignore? Meeting Control Robustness Requirements Through Adaptive Soft-error Handling". In: *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*. LCTES 2016. Santa Barbara, CA, USA: ACM, pp. 82–91.

[5] Takashi Chikamasa. *nxtOSEK*. http://lejos-osek.sourceforge.net/. 2013. URL: http://lejos-osek.sourceforge.net/.

[6] Frank Grimm. *Portierung des nxtOSEK-Frameworks auf die Lego® EV3 Plattform*. Feb. 2016.

[7] Saransh Gupta and Jimit Doshi. *Support for Nested Preemption in nxtOSEK*. http://moss.csc.ncsu.edu/~mueller/rt/rt14/projects/p1/report4.pdf. 2014. URL: http://moss.csc.ncsu.edu/~mueller/rt/rt14/projects/p1/report4.pdf.

[8] Nils Hölscher and Kuan-Hsun Chen. *Enhanced EV3OSEK*. https://github.com/kuanhsunchen/ev3osek. 2018. URL: https://github.com/kuanhsunchen/ev3osek.

[9] *Java for LEGO Mindstorms*. https://lejos.sourceforge.io/. LEJOS. URL: https://lejos.sourceforge.io/.

[10] Lego Inc. *Lego Mindstorms*. http://www.lego.com/en-us/mindstorms/. URL: http://www.lego.com/en-us/mindstorms/.

[11] Yixiao Li et al. "A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support". In: *Operating Systems Platforms for Embedded Real-Time Applications, OSPERT*. 2014. URL: http://ev3rt-git.github.io/.

[12] Mindsensors. *Console Adapter for EV3*. http://http://www.mindsensors.com/ev3-and-nxt/40-console-adapter-for-ev3. 2017. URL: http://http://www.mindsensors.com/ev3-and-nxt/40-console-adapter-for-ev3.

[13] OSEK. *OIL: OSEK Implementation Language*. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/oil25.pdf. Version 2.5. July 2004. URL: https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/oil25.pdf.

[14] OSEK. *OSEK/VDX Operating System Manual*. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf. Version 2.2.3. Feb. 2005. URL: https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf.

[15] Achim Stuy. *EV3OSEK*. https://github.com/ev3osek/ev3osek. 2017. URL: https://github.com/ev3osek/ev3osek.

[16] Texas Instruments Inc. *AM1808/AM1810 ARM Microprocessor Technical Reference Manual*. http://www.ti.com/product/AM1808/technicaldocuments. 2011. URL: http://www.ti.com/product/AM1808/technicaldocuments.

[17] *The Coq Proof Assistant*. https://coq.inria.fr/. Coq. URL: https://coq.inria.fr/.

[18] *Toyohashi OPen Platform for Embedded Real-time Systems*. http://www.toppers.jp/en/index.html. TOPPERS. URL: https://www.toppers.jp/.

[19] X. Weber, L. Cuvillon, and J. Gangloff. "Active Vibration Canceling of a Cable-Driven Parallel Robot in Modal Space". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 1599–1604. DOI: 10.1109/ICRA.2015.7139402.

[20] A. Zahir. "OIL-OSEK implementation language". In: *IEE Seminar on OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523)*. Nov. 1998, pp. 8/1–8/3.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift