Examining and Supporting Multi-Tasking in EV3OSEK

Nils Hölscher, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen
Department of Informatics, TU Dortmund University, Germany
{nils.hoelscher, kuan-hsun.chen, georg.von-der-brueggen, jian-jia.chen}@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 serious errors. In this work we address task preemption, a common feature desired in every RTOS. We reveal the errors in the current version and propose corresponding solutions for EV3OSEK that fix the errors in the IRQ-Handler and the task dispatching properly, thus enabling real multi-tasking on EV3OSEK. Our verifications show that the current design flaws are solved. Along with this work, we suggest that researchers who performed experiments on nxtOSEK should carefully examine if the flaws presented in this paper affect their results.**

I. Introduction

Since 1998 Lego Inc. released a series of programmable robotics kits called Mindstorms [8], which have been extensively used in graduate level researches and college education. For the Lego Mindstorms robots of the NXT series, the OSEK standard [11] compatible real-time operating system (RTOS) nxtOSEK [4] has been widely adopted as an experimental platform [1, 3, 14]. However, EV3 as the latest generation of Mindstorms robots, released in 2013, is still not popularly used in the real-time community. One reason is that the only RTOSs for EV3 robots, namely EV3RT [9] and EV3OSEK [12], were release a few years after the EV3 robots, i.e., in 2016. In this paper we only focus on EV3OSEK, since it is the only RTOS for EV3 aiming at supporting the OSEK standard.

EV3OSEK is a porting of nxtOSEK to the EV3 platform, provided by a group at Westsächsische Hochschule Zwickau [5]. 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 [10] on a host machine. Moreover, unlike nxtOSEK that needs to flash the ROM on the brick, EV3OSEK can directly boot from a SD-Card.

During our experiments with EV3OSEK, we noticed that the task preemption mechanism did not function as expected. Gupta and Doshi [6] 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 us to investigate if the problems were related. In course of this investigation, we discovered that EV3OSEK was unable to correctly restart preempted jobs but instead reexecuted them completely. A more detailed description of the preemption behaviour of EV3OSEK as well as of nxtOSEK can be found in Section III. We encourage researchers who performed experiments on nxtOSEK to carefully examine if the flaws presented in this paper affect their results.

To narrow down the source of the problem, we examined the ARM specifications, the hardware dependent IRQ-Handler, and the task dispatching routines. In this work, we provide the corresponding solutions to the errors in the current EV3OSEK, which are released on [7]. After solving these problems, EV3OSEK is now able to provide preemptive scheduling, and therefore multi-tasking, with all the advantages inherited from nxtOSEK.

Our Contributions: This paper presents the errors that exist in the current version of EV3OSEK when task preemption takes place and provides a solution to tackle these problems.

- We detail a flawed behaviour regarding task preemption in EV3OSEK in Section III, and explain the origin of these problems in Section IV.
- The corresponding solutions for the IRQ-Handler and the task dispatching routine are provided in Section V, hence enabling multi-tasking under EV3OSEK.
- We evaluated our solutions, the results are displayed in in Section VI, showing that the provided solutions solve the problems and allow fully preemptive fixed-priority scheduling, and therefore multi-tasking, in EV3OSEK.

II. System Model

A. Application Model

We consider the scheduling of n independent periodic real-time tasks $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ in a uniprocessor system. 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. The deadlines is assumed to be implicit, i.e., if a task instance (job) is released at θ_a, it must be finished before $\theta_a + T_i\ \forall \tau_i$. We consider fully preemptive fixed-priority scheduling, i.e., each task τ_i is associated to a predefined priority $p(\tau_i)$[1], since the issues considered in this work only happens under a fully preemptive scheduling policy.

B. Lego Mindstorms EV3 and EV3OSEK

In this paper, we 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++

[1]Although EV3OSEK defines the lowest priority as 0, we use the more common notation that lower priority values indicate higher priorities.

compatible environment. EV3OSEK [12] is a real-time operating system which aims for compatibility to the OSEK standard [11]. It is a recent portation [5] of nxtOSEK [4], which is only available for the older LEGO Mindstorms NXT robots. EV3OSEK 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 [10] is used, which realizes an USB to UART bridge. 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 [13].

C. Preemption in the OSEK Standard

Here we briefly review the specifications for task preemption defined by the OSEK standard [11]. 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 at the point in time a higher priority task enters the system and that higher priority task is scheduled instead. The context of the preempted task is stored accordingly so that it can resume back later on.

III. Motivational Example

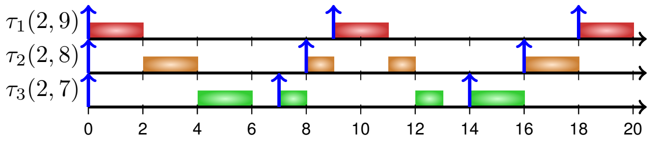
To demonstrate the flaws in the current EV3OSEK, Figure 1 provides and example that detail the EV3OSEK preemption behaviour. We consider three tasks: $\tau_1 = (2, 9)$, $\tau_2 = (2, 8)$, and $\tau_3 = (2, 7)$, indexed according to their priority, i.e., $p(\tau_1) > p(\tau_2) > p(\tau_3)$.

Figure 1a 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. 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 our code, we ensured a start at 0, hence the first job of τ_1 is released as well.

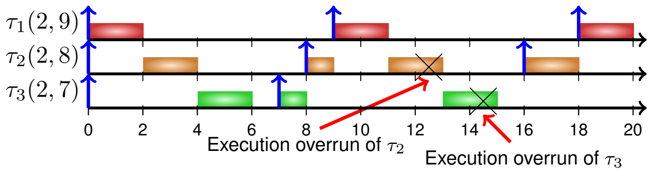
In contrast, Figure 1b shows the execution behaviour of EV3OSEK.[2] Both the second job of τ_2 and the second job of τ_3 are not resumed correctly but either resumed wrongly or completely restarted which leads to one additional unit of execution time for both jobs, called overrun in Figure 1b. Note that, due to the deadline miss at 14, the third release of τ_3 at 14 is skipped and the next job of τ_3 will be released at 21.

Since EV3OSEK is a portation from nxtOSEK, this behaviour could directly be inherited. However, the flawed behaviour in the original nxtOSEK was different and only

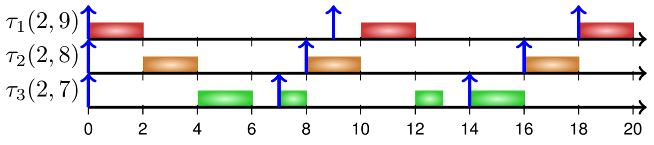
[2]The related source code is released on [2] as `NestPreemption`.



(a) **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.



(b) **Observed behaviour in EV3OSEK**: the jobs of τ_2 and τ_3 are restarted instead of resumed after a preemption.



(c) **Observed behaviour in nxtOSEK:** τ_3 is preempted by τ_2, but τ_2 cannot be preempted by τ_1.

Fig. 1: Expected behaviour compared to the actual behaviour of EV3OSEK and nxtOSEK.

effected nested task preemption as displayed in Figure 1c. 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 at time 9 although $p(\tau_1) > p(\tau_2)$. Only when τ_2 finishes at time 10, τ_1 is allocated to the processor. However, when Gupta and Doshi [6] tried to fix this problem, their efforts resulted in an identical behaviour as in Figure 1b due to the already existing problems with the IRQ-Handler and the task dispatching.

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 us to check the functions responsible for the IRQ-Handler and the task dispatching. It turned out that the IRQ handler, expended from TexasInstruments [13], has critical errors that could have lead to complete corruption of the program counter.

IV. Original Task Preemption in EV3OSEK

In this section, we first review the current design of the functions that are responsible for IRQ-Handler[3] and task dispatching in EV3OSEK[4]. Afterwards we point out the source of the aforementioned errors.

A. IRQ-Handler

To follow the OSEK standard, EV3OSEK has a hook routine named `user_1ms_isr_type2()`, which is invoked

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

[4]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.

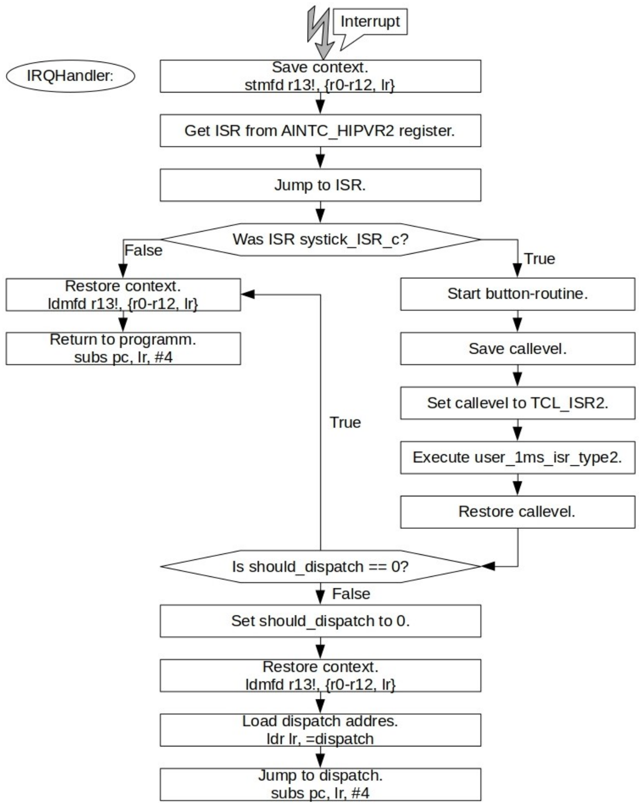


Fig. 2: Flowchart of the current IRQ-Handler in EV3OSEK.

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 progress of EV3OSEK. However this design partially violates the OSEK standard.

Once an ISR occurs, the CPU loads the IRQ-Handler shown in Figure 2. 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.

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, we noticed that there are five errors in the current implementation as shown in Listing 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}

LDR       lr, =dispatch
SUBS      pc, lr, #4
```

Listing 1: Assembler code fragment responsible for the five errors related to the IRQ-Handler.

B. Task Dispatching

Before introducing the current design of task dispatching in EV3OSEK, we 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, we 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, 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 `dispatch_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

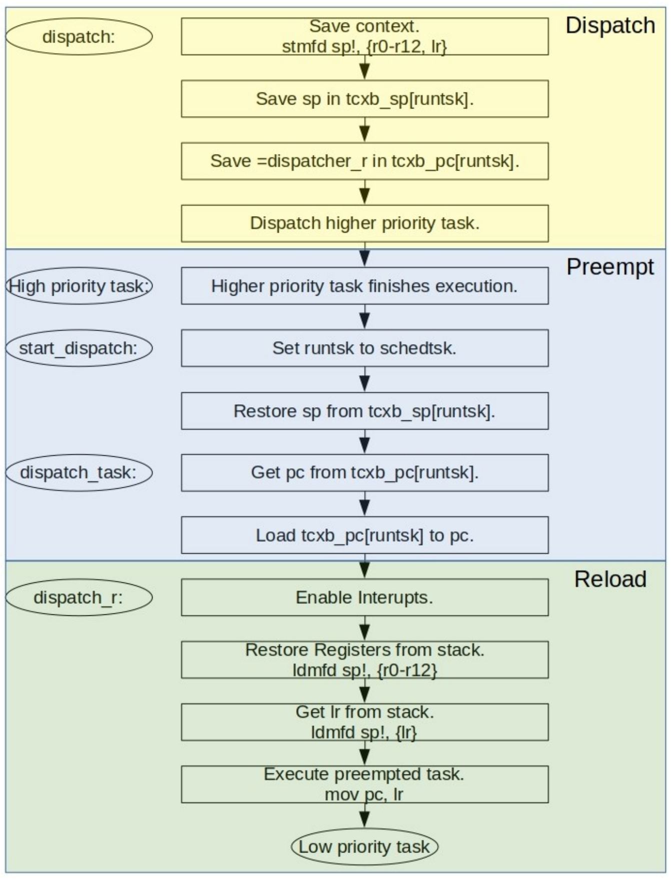


Fig. 3: Task dispatching and re-dispatching.

`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 loading the tasks program counter, the preempted task executes `dispatch_r` to restore its context from the stack and enable interrupts, which were disabled by `TerminateTask()`.

There are two errors in the current implementation:

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

```
dispatcher_r:
        BL      IntMasterIRQEnable
        BL      IntMasterFIQEnable
        ldmfd   sp!, {r0-r12}
        ldmfd   sp!, {lr}
        MOV     pc, lr
```

Listing 2: The lookup register is loaded without ˆ flag, the status bits are not loaded at all.

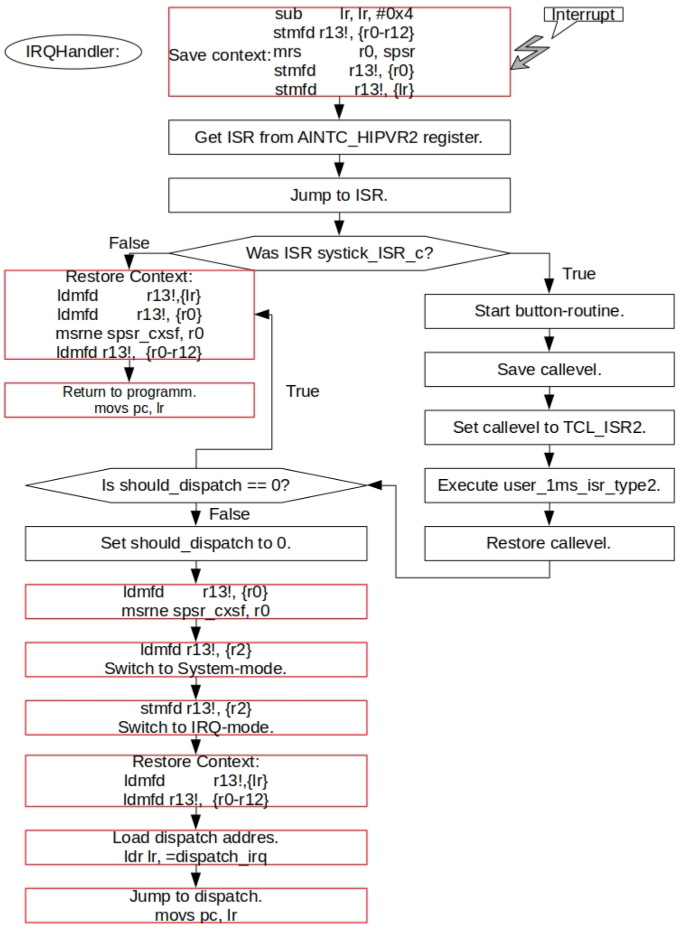


Fig. 4: Enhanced version of the IRQ-Handler.

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

V. Fixing Task Preemption in EV3OSEK

After discussing the flaws in the current EV3OSEK, we here present how we 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.

Based on the observations in Section IV, the proposed solutions can be summarized as follows:

- correcting the register operations in the IRQ-Handler,
- 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 and Figure 5, respectively, where the red blocks are added or changed due to our solutions. In the rest of this section, we explain more details about our 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 modes have their own registers called banked registers which are not shared with other modes.

The errors 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 error 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.

Correcting the errors in `dispatch_r`: As shown in Figure 5, 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 5.

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 informations in $r0$ to $r12$ are 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, we moved the code setting the flag `should_dispatch` to the function `SetDispatch()` and call it after `user_1ms_isr_type2()` has finished.

VI. Evaluation of the Proposed Solution

As illustrated in Section III, the current EV3OSEK is not able to provide task preemption correctly. With the enhancement mentioned in the previous section, task preemption, and hence multi-tasking, now should work properly. We present an additional example with three tasks to evaluate our proposed

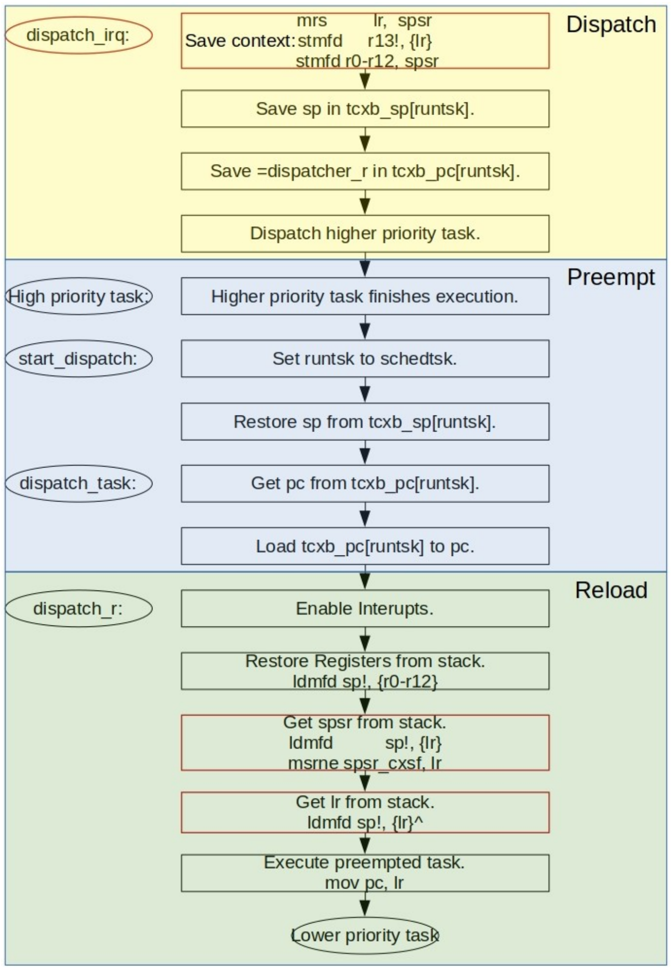


Fig. 5: Enhanced version of task dispatching.

solution in EV3OSEK[5].

In the following experiment, we 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 our enhanced version of EV3OSEK is identical, we can determine whether our enhancement solved the discovered problem. The related source code can be found at [7].

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 $2000\,\text{ms}$ and priority's are $p(\tau_1) > p(\tau_2) > p(\tau_3)$. The tasks are released as follows:

- τ_1 releases at $0\,\text{s}$ with a period $5\,\text{s}$.
- τ_2 releases at $0\,\text{s}$ with a period $8\,\text{s}$.

[5]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_3 releases at $0\,\mathrm{s}$ with a period $10\,\mathrm{s}$.

We verified that all the task preemptions behave as we expect over a certain amount of time, checking the resulting log file, and if the number of jobs for each task is exactly as we predict 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. We also intend to show that the program counter does not get corrupted any more, even after long run times, i.e., $10\,\mathrm{min}$.

We 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 us the number of jobs from τ_i in a $10 \times t$ second long interval:

$$\begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} = \begin{pmatrix} \frac{8}{4} \times t \\ \frac{5}{4} \times t \\ \frac{4}{4} \times t \end{pmatrix} = \begin{pmatrix} 2t \\ 1.25t \\ t \end{pmatrix} \tag{1}$$

The equation is detailed as follows:

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

We can now predict l_1, l_2 and l_3 after an interval of $10\,\mathrm{min}$.

$$t = 600001(ms) \approx 60 \times 10sec \Rightarrow \begin{pmatrix} l_1(60) = 120 \\ l_2(60) = 75 \\ l_3(60) = 60 \end{pmatrix} \tag{2}$$

In the current version of EV3OSEK the example hangs after $7000\,\mathrm{ms}$, because the program counter is set to a random address. With our enhancement, the aforementioned problem does not exist anymore in the enhanced version of EV3OSEK. The output can be found at listing 3.

```
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 at 6995.
...
Task 1(120, 75, 60) start at 600001.
```

Listing 3: Output generated with the evaluation example using the enhanced of EV3OSEK.

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

VII. Conclusion

EV3OSEK as an OSEK inspired real-time operating system for the third generation of LEGO Mindstorms robots (EV3) has many benefits in graduate level researches and college education. In this work, we explain how we have fixed the IRQ handler and the task-dispatcher for the current version of EV3OSEK to achieve a generally expected task preemption feature. Consequently, the proposed solution fixes multi-tasking in EV3OSEK. The release source code of our enhancement can be found in [7].

Acknowledgments

This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject A1.

References

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

[2] K.-H. Chen. Motivational Examples for the flaws in EV3OSEK. https://github.com/kuanhsunchen/ev3osek/tree/master/example, 2017.

[3] K.-H. Chen, B. Bönninghoff, J.-J. Chen, and P. Marwedel. 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, pages 82–91, New York, NY, USA. ACM.

[4] T. Chikamasa. nxtOSEK. http://lejos-osek.sourceforge.net/, 2013.

[5] F. Grimm. Portierung des nxtOSEK-Frameworks auf die Lego EV3 Plattform, February 2016.

[6] S. Gupta and J. Doshi. Support for Nested Preemption in nxtOSEK. http://moss.csc.ncsu.edu/~mueller/rt/rt14/projects/p1/report4.pdf, 2014.

[7] N. Hölscher and K.-H. Chen. Enhanced ev3osek. https://github.com/kuanhsunchen/ev3osek, 2018.

[8] Lego Inc. Lego mindstorms. http://www.lego.com/en-us/mindstorms/.

[9] Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada. A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support. In *Operating Systems Platforms for Embedded Real-Time Applications, OSPERT*, 2014.

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

[11] OSEK. OSEK/VDX Operating System Manual. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf, February 2005.

[12] A. Stuy. EV3OSEK. https://github.com/ev3osek/ev3osek, 2017.

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

[14] 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)*, pages 1599–1604, May 2015.