Usable RTOS-APIs?

Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, Wolfgang Schröder-Preikschat
Chair of Distributed Systems and Operating Systems
{klaus,franzmann,engelhard,scheler,wosch}@cs.fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

***Abstract*—We believe that the *Application Programming Interfaces* (APIs) is a commonly ignored but very important property of a *real-time operating system* (RTOS). It should not only be complete i. e., offer all mechanisms needed to implement common real-time systems, but also be easy to use in order to prevent programming errors and make real-time systems more reliable. Sadly there exists only little information about how a usable RTOS API should look like. Therefore this paper aims to give assistance in assessing and designing RTOS APIs. First we give an overview of concepts we expect an RTOS API to offer and introduce criteria we think must be fulfilled for an API to be called 'usable'. Then we examine the widely known API specifications POSIX and OSEK OS as well as the APIs of the RTOSes *FreeRTOS* and *Windows Embedded Compact 7* w. r. t. to these criteria. Finally we discuss possible reasons for the outcome of our examination and we deduce some advice on how to design RTOS APIs.**

Keywords*—*Operating Systems, System software, Real-Time Systems, Application programming interfaces, Ergonomics, Human factors

I. Introduction

In our everyday world, real-time systems are pervasive. Often, laymen may not realize that they are surrounded by real-time systems but without these systems the most mundane things of our industrialized world would not be possible. Industry needs real-time systems for production, many modern aircraft cannot fly without real-time-critical control systems, and not even something as mundane as a modern car's engine would function without an underlying real-time system. People may take something like a mobile telephone for granted, however, even this is a real-time system.

This enumeration reveals how large parts of our civilization depend on real-time systems acting at the right time in the right way. However, what does 'right' mean in this context? Though in general this is a very tough question often demanding intricate domain knowledge about the physical processes and machines involved, for the underlying RTOS it is simple: Do what the application programmer told you to do!

Therefore it is crucial for an RTOS to know precisely what the programmer wants it to do. The 'language' the programmer uses for this purpose is the API of the RTOS. Since misunderstandings between RTOS and programmer can lead to serious damages like plane crashes, we believe it is crucial that RTOS APIs are well designed and straight forward to use. This makes programming real-time systems less error-prone and provides more time for actually testing the application instead of wondering how the API might be used correctly. Moreover an understandable and easily usable RTOS API can seduce programmers to make their decisions explicit in the code instead of expressing them in terms of the implicit temporal order of instructions. This makes the job of analysis or transformation tools much easier, helping to increase the reliability of the real-time system even more. Nevertheless there appears to be little guidance as to what a well designed API looks like and what to take care of when implementing an RTOS.

In this paper we will first establish what the term real-time means, and what semantics an OS has to offer in order to be called 'real-time capable'. From this we will present our criteria for assesing the appropriateness of an API for programming real-time systems. After that we will take a look at some OSes and OS standards to find out how these perform w. r. t. our criteria. We will then discuss the appropriateness of the APIs we examined and explain where we think these succeed and fail at being usable RTOS APIs. From this discussion we try to deduce some short remarks on how to design an RTOS API.

II. Requirements for an RTOS API

In order to assess RTOS APIs w. r. t. their semantics, we first want to present an overview of important properties of real-time systems. Throughout this paper we will be focusing on embedded real-time systems running on microcontrollers. We will derive concepts and semantics we believe an RTOS for this environment should provide to the application programmer.

Most real-time systems interact with their environment using one or more sensors and actuators. The real-time system reacts to external stimuli signified by a sensor value. These stimuli are called *physical events*. The reaction to such an event is called a *task* and produces some kind of result like e. g., a setpoint for an actuator.[1] More complex tasks may be triggered by a set of events combined by *AND-* and *OR-relations*. In contrast to non-real-time critical computer systems, a real-time system has to react to an ever-changing external environment in a timely manner. Events and tasks are therefore attributed with timing information. Events may be *periodic* or *non-periodic*. Periodic events are described by a *period* and a *phase* while non-periodic events are described by their *minimum inter-arrival time*.

Additionally, all real-time critical events have a *deadline* which denotes the latest point in time at which the result of the associated task has to be available. Missed deadlines may make the result of a task unusable or even destroy the real-time system, leading to damage to people and loss of life. A real-time system's tasks are often split into multiple *jobs*, which represent the basic unit of work. In addition to physical events, jobs may have to react to a change in state of the real-time

This work was supported by the German Research Foundation (DFG) under grant no. SCHR 603/9-1.

[1] Note that we distinguish between the abstract notion of a task and its concrete implementation in the form of an OS process, thread, coroutine etc.

software. These state changes are triggered by the real-time application's jobs and are called *logical events*.

A. Mapping of Real-Time Concepts to APIs

Now that we know which abstract model an OS has to conform to in order to be suitable to a real-time environment, we will take a look at the particular semantics an RTOS API has to offer and the concepts an RTOSes may use to implement them. Most RTOSes do not use the concepts of jobs and tasks directly. In principle, jobs and tasks are passive entities that have to be executed by the real-time system. Most RTOS APIs, however, provide active entities like *threads* [1], *coroutines* [2] or *continuations* [3] to allow the real-time application programmers to execute the abstract jobs and tasks they envision. For performance reasons, multiple jobs that share temporal parameters may be executed by the same active entity. In the rest of this section we will first show how RTOS APIs can handle events. Next we will introduce options for dealing with data dependencies in real-time applications and finally we will detail how an RTOS API may manage shared resources.

1) Event Handling: There are two fundamental ways a real-time system can model event handlers. The first option is to create the active entity when the event occurs and start it. With respect to memory consumption this may be preferable since an active entity that does not exist will not consume any memory for its stack. The second option is to use an *event flag*. An already existing and running active entity blocks itself on the flag and waits for the flag to change state. When the event occurs, the detecting entity toggles the flag and the formerly blocked active entity resumes execution. As event flags represent the most fundamental synchronization mechanism, more complex concepts are built on top of them. These include counting mechanisms like *semaphores* [4] and *barriers* [5], which can be used to implement AND-semantics, as well as the more complex *condition variables* [6]. To support complex tasks it is desirable that the API supports waiting on combinations of events e. g., by specifying event masks. In a non-real-time application these mechanisms would already be sufficient, however, in a real-time system we also require an option for deferring the execution of an event handler to a later point in time. Moreover there need to be concepts like reoccurring *timers* to also support periodic events.

The aforementioned mechanisms are already sufficient to handle logical events. Physical events, however, are usually signalled by interrupts, which by their very nature occur *asynchronously*. An *interrupt service routine* (ISR) that is activated by the occurrence of an interrupt will run immediately, and therefore it will violate the priorities assigned to other active entities. The RTOS API must thus provide some way of requesting the execution of a *synchronous* active entity, from the interrupt handler, which will then obey its assigned priority. If the real-time application programmer keeps the ISR as short as possible, a priority inversion that would otherwise lead to missed deadlines may be avoided. For an RTOS API to be appropriate for a real-time system with firm deadlines, where a silently missed deadline might lead to injury or loss of life, the API should also provide some way of finding out if a deadline has been missed and of reacting accordingly.

2) Data Flow: According to Wolfe and Blaza [7], approximately one third of all real-time systems execute some kind of control law to prevent a physical system from leaving its operating point. The most natural way to model these control systems is to represent them by a data flow from sensors through filters and controllers to actuators. Therefore it would be convenient to have some mapping of this data flow to the real-time system. Although in principle it is possible to establish producer-consumer patterns using the mechanisms we presented in the previous section, this may be awkward. A more natural way of modelling data flow is through mechanisms that combine synchronization with copying of data, like message queues, mailboxes, blackboards etc. This approach has the additional advantage that it may also be used for communication between remote processing nodes or to conveniently migrate jobs from one processing node to another.

3) Resource Management: A real-time application always needs some way to interact with its environment. As a consequence, it needs access to physical devices, which can usually only be used by one active entity at a time. An RTOS API must therefore have some way of providing mutually exclusive access to *shared resources*. However, exclusive access may incur uncontrolled priority inversion [8], risking deadline violations. Therefore an RTOS API must supply mechanisms like the *Priority Ceiling Protocol* (PCP), *Deadline Inheritance* (DI) or *Non-Preemptive Critical Sections* (NPCSes) to avoid such behaviour. More complex resource access schemes like the reader-writer-lock may be provided by the RTOS API to coordinate complex resources like system memory.

B. Criteria for Assessing Real-Time Operating System APIs

Although we now know which semantics are necessary to represent real-time systems, we still have to determine how to judge the implementations the RTOS API offers. In this paper we will employ two criteria to this end.

1) Completeness: Henning [9] identifies *completeness* as the most important criterion of API design. In this paper, we will consider an RTOS API complete if it provides at least a representation of real-time properties like periods, phases, deadlines etc., control flow and data flow dependencies, with and without specifiable delay, and resource management that avoids uncontrolled priority inversion.

2) Usability: Completeness of an RTOS API may be enough to build a real-time system, however, we think that this is not enough to build a reliable real-time system. Many real-time systems are used in a firm or hard real-time environment where missed deadlines have serious consequences. An RTOS API is a man-machine interface since it is used to express a human's wishes in a form that can be processed by a machine. We therefore propose to also consider the psychological aspects of an RTOS API.

Raskin [10] identifies two properties of a good man-machine interface. *Modelessness*, which means that the user does not have to remember which state the machine is in to discern what effect an action will have; and *monotony*, which means that there should be exactly one way for the user to achieve some effect. Both requirements aim to reduce the cost of training personnel, developing the OS and application, and to reduce the cost of maintenance. The *locus of control* of users that are

provided with multiple ways of achieving some goal will be drawn away from what they are trying to achieve. Instead of solving the problem at hand, they will spend time choosing the right mechanism for doing so. Since average application programmers are no experts in interface design, they will not necessarily choose the best mechanism.

Note, however, that modelessness and monotony do not mean that the user interface may not offer composite mechanisms. Such a requirement, however – which might be the guiding principle of a naive approach to user interface design – is counterproductive. It would encourage programmers to implement mechanisms they frequently use themselves, leading to a myriad of implementations of the same concepts.

Calculating feasible schedules is an NP-hard problem, while determining events and event handlers is a precondition for creating any schedule at all. Therefore it is much easier for a human to just find the necessary events and event handlers instead of calculating the required schedule. In this paper we will therefore focus on RTOS APIs following the event-triggered paradigm. We do not deny that hard real-time systems, whose failure would endanger human lives, should be executed in a time-triggered fashion. However, in our opinion, real-time system designers should develop even hard real-time systems in the event-triggered way and then use supporting tools to transform the event-triggered design into a time-triggered system. See [11] for an example of such a tool.

III. Choice of RTOSes and RTOS API Standards

One goal of this paper is to give an overview of the state of the art in RTOS APIs. We will therefore present two OS API standards and two OSes, all of which are in widespread use in academia and industry [7]. One fundamental decision an OS designer has to make is whether the system should be configured statically or dynamically w. r. t. active entities and resources. The static approach has numerous advantages: A statically configured RTOS will usually require less run-time resources like RAM and processing power, which – even in our times, where cheap 32 bit microcontrollers are on the way of becoming the norm rather than the exception – is an important issue in the design of real-time systems. Also, it is much easier to analyze the real-time properties of a static system, which allows the designer to guarantee firm and hard deadlines. A dynamically configured OS, on the other hand, may be much more flexible at run-time when reacting to seldom circumstances. The rest of this section will give a short overview of each RTOS API and introduce the job execution abstraction of each approach.

POSIX: From a historical perspective, the *Portable Operating System Interface* (POSIX) standard is the most influential standard in the world of dynamically configured OSes. First adopted by the IEEE in 1988 [12], all Unix-like OSes implement at least part of it. Even the Windows NT series of OSes has had POSIX support in the past, and many embedded non-Unix OSes provide a POSIX compatibility layer. The standard has since grown considerably and by now encompasses a profile that targets real-time systems even for microcontrollers without *memory managment unit* (MMU) and filesystem support [13]. POSIX offers two abstractions for active entities, *processes* and *threads*. These differ insofar as each process has its own address space and therefore cannot access another process's memory, while multiple threads may share the same address space. In this paper we will limit ourselves to the POSIX facilities appropriate for real-time systems. We think that it is appropriate to include POSIX in this paper since many embedded operating systems like eCos or RTEMS implement at least part of the standard. POSIX does not provide direct access to the interrupt handling mechanisms of the hardware. Instead, interrupt handling may be mapped to POSIX's *real-time signals* mechanism. These signals are guaranteed to be delivered in the order they are generated, and if a signal is triggered multiple times, it will be delivered exactly as often as it was triggered.

OSEK OS: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) OS is an API standard for statically configurable OSes. The standard consists of parts that are mandatory as well as optional parts. In this paper we are going to take a look at the mandatory OS standard [14], which provides an abstraction for active entities, and the optional communication standard [15], which specifies a message passing interface. OSEK OS's active entities are called *basic* and *extended tasks*. Basic tasks have run-to-completion semantics and map directly to our concept of abstract tasks. Extended tasks, on the other hand, may yield the processor voluntarily and therefore are more like other operating system's threads. OSEK OS is in widespread use throughout the automotive industry. Although OSEK has been superseded by the newer *AUTomotive Open System ARchitecture* (AUTOSAR) standard, it is still relevant since it is part of this new standard.

FreeRTOS: The free and open source FreeRTOS[2] is a dynamically configured OS with support for a wide range of different processors. FreeRTOS's API is limited to the facilities required by real-time applications. It provides two implementations of active entities. Similar to the OSEK standard, FreeRTOS names its preemptively scheduled threads *tasks*. Additionally, jobs may also be mapped to *coroutines*, which are scheduled cooperatively.

Windows Embedded Compact 7: This OS is a component-based embedded RTOS that has compile-time configurable support for diverse hardware, a *Graphical User Interface* (GUI), touch screen and playback of digital media. In contrast to all other examined RTOSes it demands an MMU but was included nevertheless, since according to [7] it is used quite frequently in industrial applications. Windows Embedded Compact 7[3] configures resources and active entities dynamically and provides three abstractions for the later ones: *Processes* and *threads* are very similar to their POSIX namesakes, while *fibres* are thread-local coroutines.

Unfortunately none of these RTOSes include an API to detect deadline misses as mentioned in Section II-A1. Therefore in this paper we will not examine this requirement, though we think such an API should at least be offered in firm RTOSes. Moreover we will only take the thread abstraction offered by the four presented approaches into account. The reasons for this decision are twofold: 1. Processes guarantee separate memory address spaces and therefore rely on the presence of an MMU. Since the kind of RTOS we have in mind usually runs on a microcontroller and most microcontrollers do not

[2] http://www.freertos.org/

[3] http://www.microsoft.com/windowsembedded/en-us/windows-embedded-compact-7.aspx

come equipped with an MMU, full-blown processes simply are not an option. 2. Coroutines are non-preemptive and therefore increase the chance of an active entity violating the priority of some other active entity. In principle it is possible to build a real-time system with coroutines, however, we think that this makes the programmer's task much harder and is in conflict with our goal of providing a usable API.

IV. Methodology

The goal of our evaluation is to find out in how far the RTOS APIs we haven chosen succeed w. r. t. the criteria of *completeness* and *usability*. To show if an RTOS's API is complete we first grouped the calls provided by the APIs w. r. t. the criteria we established in Section II. After that we determined how many functions have to be called in order to use these concepts correctly. In this assessment we ignored functions that are only required for setup and configuration.

Next we tried to quantify the monotony of the APIs by exploring all choices an application programmer has to make when implementing one of the basic semantics described in Section II-A. To do so we tried to implement each requirement in all ways permitted by the API and created a test case for each option to verify its semantics. The experience we gathered while implementing these proofs of concepts now allows us to also judge the RTOS APIs w. r. t. the criterion of modelessness.

In Section II-A1 we presented two ways of handling events – either by creating an active entity or by waking a preexisting one up. As part of our evaluation we implemented both methods. First we focused on handling events by directly starting active entities. We implemented the activation of jobs from within other jobs to support logical events and splitting tasks into multiple jobs. To support asynchronous i. e., physical events we also considered all ways of activating entities from within an ISR. Second we used synchronization mechanisms like semaphores to inform waiting active entities of events. For both approaches we also implemented snippets that trigger delayed events. A special case of delayed events may be implemented using the *sleep* mechanism since the event is handled by the same control flow that triggers it. In addition to using the sleep mechanism of most APIs, we also implemented delayed activations by other means.

In order to test the *data flow* mechanisms of the APIs, we implemented simple code snippets using the basic building block of data flow modelling, which is transferring one byte from one job to another.

To execute these test cases conveniently and quickly we did not deploy them on embedded hardware. Instead we used x86-64 based desktop computers. For POSIX snippets this was straight forward since the Linux operating system implements the relevant parts of the standard. As the set of mechanisms we used is basic and timing is not crucial for their semantics, we used Linux-based simulators for the other RTOSes: Trampoline [16] for OSEK OS and the FreeRTOS Linux simulator[4] for FreeRTOS. Unfortunately the FreeRTOS simulator does not support all features of the most recent FreeRTOS release, so we had to forgo execution of some snippets. Microsoft provides a virtual machine for testing Windows Embedded Compact 7 applications. As this virtual machine does not support ISRs we could not run the corresponding test cases. Instead we drew our conclusions from the API documentation. Our code snippets and the complementing test framework are provided for reference[5].

Table I. Number of Functions Associated with each API Mechanism

mechanism	OSEK OS	POSIX	FreeRTOS	Windows EC 7
event handling				
thread administration	✔ (2)	✔ (1)	✔ (8)	✔ (3)
interrupt administration	✔ (6)	✔ (11)	✔ (2)	✔ (5)
synchronization				
event flags	✔ (3)	✔ (7)	✔ (5)	✔ (4)
semaphore	✖	✔ (4)	✔ (6)	✔ (1)
condition variable	✖	✔ (4)	✖	✖
barrier	✖	✔ (1)	✖	✖
timer	✔ (2)	✔ (3)	✔ (4)	✔ (2)
OR-combination	✔ (1)	✔ (5)	✔ (2)	✔ (1)
AND-combination	✖	✔ (1)	✖	✖
data flow				
message queue	✔ (3)	✔ (5)	✔ (12)	✔ (3)
Read-write-lock	✖	✔ (7)	✖	✔ (2)
resource management	✔ (2)	✔ (4)	✔ (2)	✔ (3)
relevant functions	18	40	39	23

V. Results

In this section we will present the results of the experiments we introduced in the previous section.

Table I summarizes the concepts present in each RTOS API. The number of related functions is given in brackets if available. The POSIX standard describes all examined mechanisms and therefor is the most featureful API. Next in line is Windows Embedded Compact 7, followed by FreeRTOS and finally OSEK OS.

The number of options the user has when implementing a required real-time concept serves as an indicator for an APIs's monotony. Table II shows this number for all test cases described in Section IV. For RTOSes supporting ISRs we differentiated between triggering events from within an ISR and from regular active entities. In Table II the first number in each column refers to physical events while the second number represents the number of implementations for the logical event mechanism. Although POSIX signals can be interpreted as an abstraction for hardware interrupts, we did not differentiate them from threads since they are an OS service and not a property of the hardware.

Completeness: As every examined API offers at least one way to implement each requirement, we consider all of them complete.

Monotony: None of the examined RTOS APIs is monotonous, since all of them offer multiple ways of achieving the same semantics. Nevertheless, OSEK OS comes quite close to meeting this criterion: the maximum amount of choices for developers to achieve their aim is four. This indicates a very straightforward API compared to the 45 choices imposed on users by POSIX.

[4] http://www.freertos.org/FreeRTOS-simulator-for-Linux.html

[5] https://www4.cs.fau.de/Research/AORTA/perfectRTOSAPI.tar.gz

Table II. NUMBER OF OPTIONS FOR IMPLEMENTING THE REQUIREMENTS

test case	OSEK OS	POSIX	FreeRTOS	Windows EC 7
event handling				
activation	3 / 4	1	1 / 1	1 / 1
delayed activation	2	1	1	1
periodic activation	3	1	1	1
suspend interrupt	3	1	1	3
synchronization				
control flow	3 / 3	45	19 / 19	18 / 17
delayed control flow	2 / 2	24	19 / 19	19 / 19
sleep	2	32	29	25
data dependency	1	4	3	2
resource management	1	2	2	3

Modelessness: Most APIs do quite well when it comes to modelessness. Even inside of ISRs most RTOSes permit the same API calls as in normal control flows. Nevertheless, dynamically configured systems like POSIX tend to violate this criterion since API mechanism often can be configured extensively. Each of these configurations represents a mode since each configuration introduces a different semantics for some API calls. In FreeRTOS different API calls have to be used for the same purpose, depending on the calling context. This serves to confuse the API user without need. Consequently it includes more relevant functions in Table I than POSIX, though it features less API concepts.

VI. DISCUSSION

Though all of the examined RTOSes can be called complete, none of them fulfil both criteria of *usability*. Nevertheless, some seem to do better than others. In this section we want to discuss particular problems of the APIs and if possible make a guess as to why they are designed the way they are.

The most obvious flaw that comes to mind when comparing Table I and Table II w. r. t. POSIX is that it is overloaded with functionality. During the nearly 30 years of its existence, it has been extended and improved again and again, adding new functionality to the API. The resulting extensive API leads to a huge amount of choices a user has to make in order to achieve some intended semantics. Especially the smorgasbord of synchronization mechanisms should be mentioned here. Although not all mechanisms are marked as 'mandatory' in the standard, and thus are not implemented in most embedded POSIX compatible OSes, this is a severe violation of the principle of monotony. Another problem is that POSIX is very configurable and flexible due to its design goal of being easily adaptable to arbitrary OSes. Normally these words bear a positive connotation but w. r. t. to usability of APIs this does not hold true. Configuring an API mechanism means changing its semantics, which introduces modes. From our point of view both problems can be attributed to the fact that POSIX has always been an integrating standard which has to fit many existing systems and provide even more mechanisms. This approach conflicts with designing a clear and usable API.

The FreeRTOS API requires the user to apply different functions for the same purpose depending on the execution context (thread, coroutine or ISR), which is in conflict with our requirement of modelessness. Whenever FreeRTOS application developers are trying to achieve some goal, they first have to make themselves aware which context they are programming in. Only after that can they decide which function to use. This mental overhead combined with the unnecessary amount of functions available for each task hinders efficient use of this API.

In contrast, Windows Embedded Compact 7 seems to follow a rather good general development pattern. Its API may not be perfect but seems to be quite usable. All results in Table I and Table II show that Windows Embedded Compact 7 does better than the other dynamically configurable RTOS APIs (FreeRTOS and POSIX). Microsoft probably benefits from two factors: 1. The development team seems to have a structured and consistent vision of the API they want to offer. This is probably due to the tighter integration of developers and stricter leadership in the project. Although FreeRTOS is marketed by a commercial vendor too, its API seems to have been designed with much less regard for usability issues. This may be a result of feature growth over time and of designing an API matching the internals of the OS instead of the API user's requirements. 2. Since Microsoft's customers licence specific versions of embedded Windows, Microsoft has complete control over the degree of backward compatibility provided. This is very important since backwards compatibility is one of the major reasons APIs erode over time [9]. Not having to provide backward compatiblity enables the vendor to make drastic cuts where necessary. Nevertheless Windows Embedded Compact 7 suffers from the same issue other dynamically configured OSes have: configurable mechanisms imply different working modes for these mechanisms.

Most of the statically configured OSEK OS's API does not suffer from this flaw. It seems that great care has been taken in OSEK's design to avoid modes, and the standard's designers had the *application programmer* instead of the OS implementer in mind when they composed it. This seems surprising since standardization usually only achieves a perpetuation of the status quo. Another excellent decision in the design of OSEK was to target embedded hardware platforms exclusively. This approach has led to a very lean, straight forward and monotonous interface. A critical look at Table II, however, reveals an apparent exception. There are four different ways of activating threads where all other APIs only provide one function. This may seems excessive but the OSEK OS standard aims for a one-to-one mapping of jobs to active entities. Therefore, in order to support complex control flows, different ways of activating threads are necessary (e. g., one handy API call atomically ends the current thread and activates another one).

Regarding modelessness, OSEK fails only in one aspect, and it does so without a pressing need: An OSEK *event* is coupled with the existence of its associated thread, and therefore, if an event occurs while a thread does not exist, the event will be lost. This design decision is detrimental to the API's usability and OSEK's version of the event mechanism in general appears to be defective. It would have been a much better design decision to specify that an event flag exists even while its associated thread does not. Nevertheless, we think that of all APIs we examined in this paper, OSEK OS provides the best usability experience in an embedded context.

The discussion of the different APIs reveals one commonality: The design process is crucial for the usability of an API.

Since the API should be ergonomic for its end-users and their applications, the designers should have these in mind instead of the underlying implementation. Instead of trying to have one API to cover all applications and purposes w. r. t. to usability it seems to be profitable to aim at one core purpose i. e., real-time in our case. Additionally if planning ahead did not work out and already existing interfaces do not work the way they were planned, one should not shy away from breaking backwards compatibility in favour of better API design.

VII. Related Work

In the field of RTOSes only little scientific work seems to have been done w. r. t. the design of usable APIs. Most papers that have been published either take a look at the overall properties of the RTOS, or, they focus on microbenchmarks of individual properties. Almost no work has been done that assesses the RTOS API itself, and the few examples that do, like Timmerman and Perneel [17], use measures like the 'richness' of the API, which conflicts with its usability. Anh and Tan [18] present a relatively comprehensive collection of microbenchmarks for implementations of RTOS APIs but do not examine the usability of the APIs themselves. The preexisting literature thus does not give a usability assessement of RTOS APIs.

In those cases where new APIs for RTOSes are introduced, these are usually intended for model-driven real-time system development. Since one of the main goals of model-driven development is simulating the resulting real-time system, the invented APIs cater to the needs of simulators instead of those of programmers. In some cases this means that the API does not even offer resource management and therefore is incomplete w. r. t. our criteria. Examples of this approach include Hessel et al. [19], Shaout et al. [20] as well as Maeng et al. [21].

As far as we are aware, no published work targeting the real-time domain explores APIs w. r. t. ergonomic properties like modelessness and monotony. So far, the human factor does not seem to have been at the focus of real-time API design.

VIII. Conclusion

In this paper we first presented a minimal set of semantic concepts that we think are necessary to express the needs of real-time applications. We then showed how these concepts can be mapped to RTOS APIs and introduced the criteria of *completeness*, *modelessness* and *monotony* for assessing the usability of RTOS APIs. After that we presented two RTOS API standards and two RTOSes that we had selected as candidates for a usable RTOS API. We examined these w. r. t. our criteria and are now convinced that the *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug* (OSEK) RTOS API comes quite close to an usable API. Furthermore we came to the conclusion that Windows Embedded Compact 7 is a pleasant API but suffers from the requirements the API of a dynamically configurable OS necessarily faces. We also discussed the shortcomings of FreeRTOS and the *Portable Operating System Interface* (POSIX) standard. The designers of FreeRTOS do not seem to have given much thought to usability while POSIX is trying to be adaptable to any OS regardless of the intended use case. It therefore implements many concepts in more than one way, burdening the application programmer with having to decide which part of the API to use.

In the future we intend to present an RTOS API that surpasses OSEK OS and expresses the abstract real-time concepts we introduced in Section II-B more directly.

References

[1] D. R. Cheriton, "Multi-process structuring and the Thoth operating system," Ph.D. dissertation, University of Waterloo, Ontario, Canada, 1978. [Online]. Available: https://cs.uwaterloo.ca/research/tr/1979/CS-79-19.pdf

[2] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, Jul. 1963. [Online]. Available: http://doi.acm.org/10.1145/366663.366704

[3] P. J. Landin, "A generalization of jumps and labels," in *Report, UNIVAC Systems Programming Research*, 1965.

[4] E. W. Dijkstra, "Cooperating sequential processes," Technische Universiteit Eindhoven, Eindhoven, The Netherlands, Tech. Rep., 1965, (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

[5] P. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," in *Parallel Processing, 1986. Proceedings. International Conference on*, August 1986, pp. 528–535.

[6] P. B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys (CSUR)*, vol. 5, no. 4, pp. 223–245, 1973.

[7] A. Wolfe and D. Blaza. (2013) *2013 Embedded Market Study*. online. UBM plc. [Online]. Available: http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE TC*, vol. 39, no. 9, pp. 1175–1185, 1990.

[9] M. Henning, "API design matters," *Queue*, vol. 5, no. 4, pp. 24–36, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1255421.1255422

[10] J. Raskin, *The Humane Interface – New Directions for Designing Interactive Systems*. ACM Press/Addison-Wesley Publishing Co., 2000.

[11] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. Washington, DC, USA: IEEE, May 2010, pp. 34–41.

[12] *Portable Operating System Interface (POSIX®) Base Specifications*, ISO Std. 9945, 1988.

[13] *Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*, IEEE Std. 1003.13-2003, 2004.

[14] *OSEK/VDX Operating System Specification 2.2.3*, OSEK/VDX, February 2005.

[15] *OSEK/VDX Communication Specification 3.0.3*, OSEK/VDX, July 2004.

[16] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline – an Open Source Implementation of the OSEK/VDX RTOS Specification," in *11th Int. Conf. on Emerging Technologies and Factory Automation (ETFA'06)*. Prague, Tchèque, République: IEEE, Sep. 2006.

[17] M. Timmerman and L. Perneel, "RTOS State of the Art," Dedicated Systems Experts, Tech. Rep., 2005.

[18] T. N. B. Anh and S.-L. Tan, "Survey and performance evaluation of real-time operating systems for small microcontrollers," *IEEE Micro*, 2009.

[19] F. Hessel, V. da Rosa, I. Reis, R. Planner, C. Marcon, and A. Susin, "Abstract RTOS modeling for embedded systems," in *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, June 2004, pp. 210–216.

[20] A. Shaout, K. Mattar, and A. Elkateeb, "An ideal API for RTOS modeling at the system abstraction level," in *Mechatronics and Its Applications, 2008. ISMA 2008. 5th International Symposium on*, May 2008, pp. 1–6.

[21] J. C. Maeng, J.-H. Kim, and M. Ryu, "An RTOS API Translator for Model-Driven Embedded Software Development," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 363–367.