Intelligent Environments 2020
C.A. Iglesias et al. (Eds.)

doi:10.3233/AISE200025

HarSaRK-RS: Hard Safe Real-Time Kernel in Rust

Kanishkar JOTHIBASU [a], Gourinath BANDA [a,1]
[a] *Discipline of Computer Science and Engineering, Indian Institute of Technology Indore, India*

Abstract. With the growth of the use of embedded systems in safety-critical applications, the demand for predictable and reliable real-time systems has increased drastically. A large percentage of real-time systems developed today are still built using C due to the performance requirements, and hence inherently unsafe. The advent of Rust has made it possible to achieve safety and reliability without any compromise on performance. This paper presents HarSaRK-RS, a priority-based preemptive hard real-time kernel implemented in Rust. The proposed kernel design and architecture ensure safety at compile time keeping the data-structure and runtime overhead of the kernel minimal, thus enhancing the real-time guarantees of the system. It guarantees freedom from data races, deadlocks, and priority inversion at compile-time. The Kernel core is independent of any clock for its operation, making it power efficient and ideal for battery-operated environments.

Keywords. Real-time systems, real-time kernels, hard real-time systems, Rust, safety critical systems

1. Introduction

Embedded systems are being deployed in a variety of safety-critical applications. Failure in such systems can have serious consequences, e.g., aerospace industry, automobiles, power plants, etc. [1]. Thus, the safety requirements and reliability expectations from real-time systems have increased drastically over time. Due to the complications in writing applications directly over embedded platforms, researchers have developed Real-time Kernels which are middlewares that operate between the application and the hardware. It provides the application developers abstractions over the hardware that takes care of various intricacies and provide friendly APIs so that application developer can focus on building the application rather than handling the complexity of dealing with the hardware.

There are two aspects to the real-time guarantees made by kernels. One of them is the adherence to completion before a deadline, and the other one being the safety and reliability of the system and its comprising tasks. This includes guarantees against invalid memory accesses, invalid type casts, memory leaks, data races, deadlocks, etc. The presence of such bugs in the real-time systems' implementations directly questions

[1]Corresponding Author: Gourinath Banda; E-mail: gourinath@iiti.ac.in

the reliability of the system itself. Thus, it is of utmost importance that the software written for real-time systems is guaranteed to be safe and reliable.

The price of running unsafe code is high. For example, in 2017, the Common Vulnerabilities and Exposures database lists 217 vulnerabilities that enable privilege escalation, denial-of-service, and other exploits in the Linux kernel [2], two-thirds of which are a consequence of the usage of unsafe programming languages. These errors can be attributed to ignoring the intricate details of object lifetimes, synchronization, bounds checking, etc., in a complex, concurrent environment. Even worse, the pervasive use of pointer aliasing, pointer arithmetic, and unsafe typecasts which keeps modern systems beyond the reach of software verification tools [3]. This emphasizes the need for usage of safe programming languages in the development of safety-critical systems.

Rust is a modern multi-paradigm system programming language focused on safety and performance [4]. It uses strong types and static analysis of source code to guarantee various aspects of software safety at compile-time [5]. By safety, we mean type and memory safety, type safety, and freedom from data-races. This makes Rust an Ideal candidate for building real-time systems. This paper proposes a Real-Time Kernel developed in Rust, HarSaRK-RS. The major goals of the kernel are:

- Minimal memory footprint.
- Low runtime overhead (Kernel Data-Structures).
- Freedom from Data races, Priority Inversion, and Deadlocks.
- Safe and reliable memory management.
- Hard real-time guarantees.

The bold safety guarantees made by Rust has attracted various developers into developing operating systems and kernels in Rust. Redox OS is a UNIX-like operating system written in Rust. The motivation behind the development of Redox OS is the fact that operating systems are an integrated part of computing, and thus a very security-critical component [6]. There is an increasing interest in implementing an OS in Rust [7]. Levy et al. (2017) provide a detailed case study of writing a kernel in Rust and argue that Rust will enable the next generation of safe operating systems [8]. These efforts motivated us to explore Rust in the realm of real-time systems and develop HarSaRK-RS, a hard safe real-time kernel in Rust.

The paper is organized into six major sections. A brief discussion on Rust programming language and its features that make it safe is presented in Section 2. Section 3 explains the Kernel architecture and details regarding sub-systems. It also explores how Rust has been leveraged to make kernel and application-level safety guarantees. Section 4 discusses the development process, experimentation details, and comparison with existing real-time kernels. It also explores the accompanying results. Section 5 concludes giving the summary of contributions of this work.

2. Significance of Rust in Embedded Systems Programming

The Rust programming language exists at the intersection of low-level systems programming and high-level application programming. It aims to empower the programmer with both fine-grained control over memory and performance and high-level abstractions that make the software more reliable and quicker to produce [5]. The design of Rust exploits the state of the art research in linear types, affine types, alias types, and region-based memory management [9]. The ownership-based type system has been formulated into formal statements and proven for the safety guarantees they make [10]. These features allow Rust to guarantee memory, type, and concurrency safety at compile-time. Ownership[10] and Borrowing[5] form the crux of the safety mechanisms of Rust. These mechanisms allow Rust to guarantee compile-time memory safety and freedom from data races boldly.

Generics[11] and traits[12] are the primary mechanism to achieve polymorphism and code-reuse in Rust. The ownership type system of Rust allows expressing and enforcing typestates at compile-time, which is useful in safety-critical applications.[13]. Because C does not support generics, kernels written in C tend to use pointers and unsafe typecasts to emulate type polymorphism. But this is a highly unsafe approach and opens doors to severe vulnerabilities [3]. HarSaRK leverages Rust generics to provide a type-safe Inter-process messaging and resource management primitives.

Null pointer de-referencing and flawed exception handling have been the cause for various notorious bugs in the history of software development. Rust replaces these with strong types defined using generics and enums, which makes compile-time error handling and other static checks in Rust very robust. The Rust library defines two Enums: `Option<T>` and `Result<T,E>`. The `Option` type encodes the very common scenario in which a value could be something or nothing. The `Result` enum helps represent cases where a function might not succeed, so the return value will either contain the value or the error [11].

Though the kernel does not use dynamic memory, the application tasks might need support for it, and thus HarSaRK supports it. This is where Rust shines ahead of most other compiled languages. The ownership and borrowing mechanism statically manage memory, thus wiping away a large group of bugs like memory leaks, double free, use after free, etc [14].

The resource primitive provided by HarSaRK encapsulates the actual resource and ensures that the only way tasks can access the resource is via the methods provided by the primitive, which are safe by design. The static checks are done by Rust to ensure that only concurrency safe variables are shared across tasks [15]. In this way, HarSaRK is able to guarantee concurrency safety at compile time.

2.1. Unsafe Rust

In low-level programming, there are various scenarios, for which full control over the function stack layout is necessary. A typical example is the implementation of a context switch. Typically, this is addressed by building the required code with an assembler and linking it to the rest of the Kernel code. However, this breaks the development workflow, and the Rust compiler cannot check these parts [16]. Thus, these code regions have to be explicitly marked unsafe.

It is the responsibility of the developer to ensure well-defined behavior inside the unsafe blocks. If the developer can guarantee that code inside the unsafe blocks does not lead to undefined behavior, then the compiler can proceed to verify the rest of the codebase, assuming that this unsafe block works fine [17].

In the Kernel implemented, the only routine that uses unsafe blocks is the context switch routine, as it requires access to CPU registers and modification of process stacks.

3. Kernel Architecture

The proposed Kernel is organized into various sub-systems based on their functionalities. Each module encloses the associated data-structures and routines in it. The sub-systems interact with each other via appropriate interfaces. They also provide certain public calls that can be invoked by the end-application.

The Kernel Architecture has been inspired by the Hartex Micro-kernel design [18]. Boolean Vectors have been used internally in the sub-systems instead of other complex data structures which reduces the kernel overhead considerably. Optional modules can be chosen not to be included in the compiled binary. A Boolean Vector internally translates to a 32-bit integer. Each bit represents the state of the task with TaskID as the position of the bit. This implies that the kernel supports a maximum of 32 tasks. In the current work, we have restricted ourselves to primary level scheduling. However, if the need for more tasks arises, then one of the tasks in the tasks can itself act as a scheduler and perform secondary level scheduling of additional tasks. Figure 1 depicts the block diagram of HarSaRK-RS.

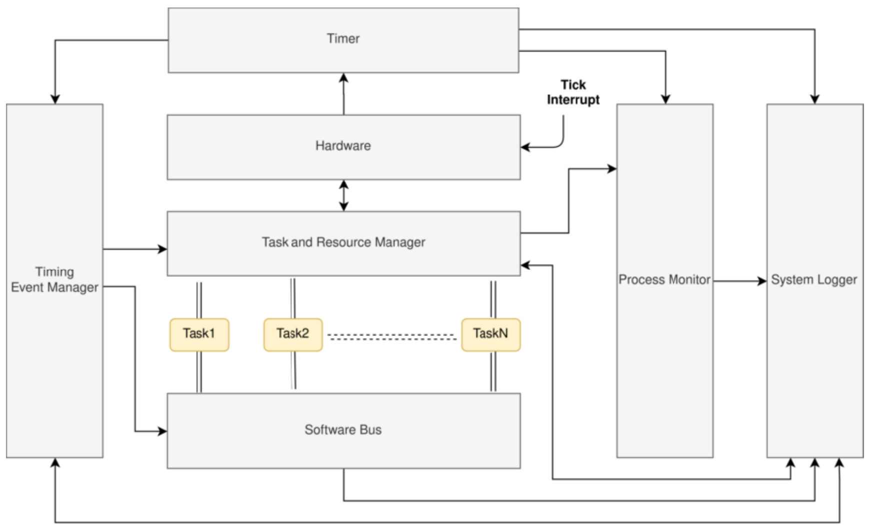

Figure 1. HarSaRK-RS block diagram

3.1. *Hardware Adaptation Layer (HAL)*

This is the closest layer to the hardware. It provides safe abstractions to the other Kernel modules to access and modify the hardware peripherals. Due to the modular approach of HarSaRK-RS, all hardware-dependent routines in the Kernel comprise the HAL. Thus, the porting of Kernel onto other machine architectures would only require the rewriting of this module.

3.2. *Task Management*

The task-management module schedules tasks preemptively on the CPU based on priority. On activation of a higher priority task, this module handles storing the context of the current task and loads the context of the highest priority task. Multi-tasking in single-core processors is done by scheduling multiple tasks one-by-one on the CPU. In case the Scheduler does not have any tasks to schedule on the CPU, then it puts the CPU to sleep. The CPU is woken up again on the occurrence of an interrupt. In this manner, the Kernel optimizes the power consumption of the system as a whole.

After the creation of all tasks, all operations on it are on two Boolean vectors `active_tasks` and `blocked_tasks`. Each task has its own separate task stack. A few examples of how the use of Boolean Vectors reduces the kernel overhead drastically are:

- Releasing tasks is as simple as enabling the bits in `active_tasks`. Let's say multiple tasks have to be released, which are encoded into a boolean vector `task_mask`. Now all that's needed to be done is `active_tasks = active_tasks & task_mask` , which is a `O(1)` operation. Hence improves the execution time of the whole system drastically. The same follows for blocking and unblocking tasks.
- Improvements in the release, block, and unblock of tasks directly influence the performance of the other kernel primitives like Semaphore, Message, and Resource, as they also internally use Boolean Vectors and are highly dependent on these routines.
- The Scheduler can compute the new highest priority task to be scheduled by just finding the Most significant bit (MSB) of `active_tasks & !blocked_tasks`. In HarSaRK-RS, MSB is calculated using a CPU instruction CLZ(Count leading Zeroes)[19], which is a constant time operation. As this is an integral part of a context switch, improvements enhance the responsiveness and hence real-time guarantees of the system.

The tasks in HarSaRK-rs are user (application) tasks, which could be composed to realize an end control system. These tasks are basic tasks meaning they are of non-blocking style. HarSaRK is a library kernel, thus tasks are defined as functions and the kernel allocates the stack space for it and takes care of scheduling it. However, if the deployment demands additional hardware support (e.g. device drivers), such support could be implemented as a task with certain higher priority. The task priorities are static and fixed, such a scheme prevents deadlocks by design. The tasks are not infinitely looping, rather they are released by events, and once they finish execution they yields. As of our current implementation, it is based on stack-based priority ceiling protocol that overcomes deadlock and bounded priority inversion. But in extreme cases where a high priority task is being released, again and again, there are chances of task starvation. This can be addressed in application design by assigning frequently released tasks a lower priority.

3.3. *Resource primitive*

In multitasking systems, the shared resources are to be used in an atomic fashion, which can be easily achieved by the use of Mutex. But Atomic allocation of resources introduces Deadlocks. Deadlocks cannot be accepted in real-time systems as they put the whole system on a halt. HarSaRK-RS employs the Stack-based priority ceiling protocol (SBPC)[20], which ensures the atomic use of resources without running into deadlocks and priority inversion. The Resource manager ensures atomicity by execution locking, i.e., when a resource is being locked, all competing tasks are blocked right away. These tasks are unblocked when the resource is unlocked.

The concept of ownership allows the resource primitive to take ownership of the actual resource i.e., after a resource primitive is initialized with a variable, the variable becomes invalid. The only way to access the variable is to call lock on the resource primitive. This ensures at compile time that all accesses to resources go via the primitive. The Resource primitive supports safe polymorphism via Traits and Generics.

3.4. Synchronization primitive

Some tasks execute certain set of instructions only if another task has completed execution till a certain point (called a synchronization point). This is known as task synchronization. It is achieved by notifying the receiver tasks on reaching the synchronization point, from where it resumes execution. Note that the Synchronization primitive provided by HarSaRK-RS is asynchronous, i.e., it just checks if the notification was received and if yes, it executes those instructions; if not, it skips those set of instructions and proceeds with the rest of the instructions.

3.5. Communication primitive

At the core, communication is just synchronization, but in addition to notification at synchronization points, messages can be passed to the tasks. The message primitive ensures that the details of the sender, receivers, and message buffer need not be specified every time to send a message; instead, only the name of the message is required. These details are provided while initializing the message primitive. The Communication primitive also supports safe polymorphism via Traits and Generics.

3.6. System Timer (Optional)

For the operation of a few sub-systems, system time is necessary. The system timer module uses the timer interrupt to update its time. The developer can choose the interruption frequency by choosing an appropriate `timer_interval`. This `timer_interval` defines the granularity and precision of time and event management in the system.

3.7. Timing event manager (Optional)

The timing event manager uses the System timer to detect timing events. Every event created has an event descriptor that holds a threshold. This defines the time period of the event as a multiple of the `timer_interval` of the system timer. It also holds a function pointer, which will be executed when its threshold expires. Once the threshold of an event expires, then the threshold is reset.

3.8. System Event Logger (Optional)

For monitoring and debugging purposes, some systems would like to keep track of all the internal kernel events that are occurring in the system with the timestamp. This log data can be used to modify the tasks. Alternatively, they can be sent to an operator station via ports like UART/Serial. Some microcontrollers support external storage, which can be used to store these logs. System Event Logger collects events like the release of tasks, tasks exiting, locking of resources, etc. Consequently, this data can be collected and processed by the developers. The logs collected during runtime can prove to be extremely useful in debugging, analyzing, and performance tuning of the system.

3.9. Process Monitor (Optional)

Some hard real-time systems would want to monitor the tasks and keep track of the tasks and their deadlines. This module allows the developer to specify the relative deadline by which a task should have finished once the task has been released. It must be noted that the deadline is specified as a multiple of `timer_interval`. This module stores the deadline and checks with the system time to see if any task has expired. Whenever a task exceeds its deadline, it creates a log entry in the System event logger.

3.10. Privileges and Memory Protection

The Kernel operates at two operation modes and two privilege levels, which are discussed in this section [19].

- The operation modes:
    - Thread mode: The processor is running a normal program.
    - Handler mode: The processor is running an exception handler like an interrupt handler or system exception handler.
- Kernel level and User level: It provides a mechanism for safeguarding memory accesses to important memory regions while providing a basic security model.

In thread mode, the CPU can either be in a Kernel state or in the User state. But while handling exceptions/interrupts, the processor switches to the Kernel state [19]. Tasks in User state are restricted from executing some operations.

The Kernel uses two stack pointers: a Main Stack Pointer (MSP) and a Process Stack Pointer (PSP). By default, all Kernel state (Kernel code and interrupt handlers) code runs on the MSP while the User state code (threads/tasks) runs on PSP [19]. The fact that the operating system and exception handlers use a different stack from the application means that the OS can protect its stack and prevent applications from accessing or corrupting it [19]. Also, it ensures that the OS does not run out of stack if the application consumes all the available stack space.

4. Experimentation

This section discusses the details of kernel implementation and testing.

Table 1. The execution time of various Kernel routines.

Description	Clock Cycles	Execution Time (10^{-6} Sec)
Load PendSV Interrupt Handler	20	0.120
Return from PendSV Interrupt Handler	23	0.137
Task Restore	20	0.120
Task Save	23	0.137
Context Switch	129	0.768
Effective Context Switch	43	0.256
Resource Lock	120	0.714
Resource Lock	84	0.500
Semaphore Signal	64	0.381
Semaphore Test and Reset	31	0.185
Message Broadcast	64	0.381
Message Receive	47	0.280

- Development machine: Linux
- Target platform: STM32F407 Microcontroller (Cortex M4 - ARM based) [21].
- Compiler toolchain: Standard Rust compiler, with cargo for package management.
- Debugger: `arm-none-eabi-gdb` for debugging.

The overhead of a real-time kernel can be measured by parameters: the binary size, the kernel data structure size, and the task switch time. The size of kernel after compilation is 3.8 Kb, which will fit into most microcontrollers without an issue. For measuring the execution time of different routines, we used the Data Watchpoint and Trace (DWT) peripheral [19] which allows us to measure the clock cycles elapsed between two points in the code [22].

Two very popular industrial real-time kernels Freescale MQX[23] and Quadros RTXC[24] have been compared with HarSaRK-RS for their task switch times. Both of these kernels are coded in C. The performance benchmarks of these kernels on Cortex M4 based microcontroller clocked at 96Mhz state the following about their task switch time [25]:

- Quadros RTXC takes 5.2μs, which translates to roughly 500 clock cycles and
- Freescale MQX takes 12.8μs, which translates to roughly 1171 clock cycles.

Task switch in HarSaRK-RS takes about 129 clock cycles, which is much smaller than the task switch time of Freescale MQX and Quadros RTXC. Though the microcontroller used for the benchmarks is different, comparisons of clock cycles is very much valid as they are using the same CPU. This is despite the safety guarantees made by HarSaRK.

5. Conclusion

The Kernel has been developed in a modular fashion, and with the help of cargo feature flags, it supports the optional compilation of Kernel modules. We have done an overhead analysis of the Kernel to get an idea of the execution times of various kernel routines and the final binary size. All the modules of the Kernel have been developed as per specification and tested manually on ST32F407 microcontroller. The Kernel has been published as a Rust library on the cargo package directory. The source code has been open-sourced under MIT license accessible via `https://github.com/Autonomous-Cyber-Physical-Systems/harsark.rs`.

In summary, HarSaRK-RS brings novelty in design and implementation as listed below:

- The use of Boolean Vectors reduces the overhead of all kernel sub modules. All core kernel routines are O(1) operations, thus minimizing jitter across the system.
- The Kernel is built over strong types and robust error handling, thus reducing chances of erroneous behavior. The use of Rust has helped guarantee compile-time concurrency safety and much better memory safety.
- The modular approach of the Kernel allows extensibility to various platforms and supports even support smaller microcontrollers.
- Separation of stack for kernel and user tasks and different privileges guarantees the safer operation of the system on the whole.
- Use of SBPC(subsection 3.3) protects the system against Data Races, Deadlocks, and Priority Inversion.
- Process monitor(subsection 3.9) and System logger (subsection 3.8) can prove to be extremely useful in debugging, analysis and performance tuning of the system.
- The Kernel core itself is independent of any clock; thus the timing module can be disabled to achieve better performance and lower power consumption without loss of any real-time guarantees. This makes HarSaRK-RS ideal for battery operated environments like IoT.

References

[1] Armoush A. Design patterns for safety-critical embedded systems; 2010. .

[2] Linux Linux Kernel : CVE security vulnerabilities, versions and detailed reports;. Available from: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[3] Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. New York, NY, USA: Association for Computing Machinery; 2011. Available from: https://doi.org/10.1145/2103799.2103805.

[4] Steve Klabnik CN. The Rust Programming Language - The Rust Programming Language;. Available from: https://doc.rust-lang.org/stable/book/.

[5] Weiss A, Patterson D, Matsakis ND, Ahmed A. Oxide: The Essence of Rust. arXiv:190300982 [cs]. 2019 Mar;ArXiv: 1903.00982. Available from: http://arxiv.org/abs/1903.00982.

[6] Why Rust? - The Redox Operating System; 2020. Available from: https://web.archive.org/web/20200510173103/https://doc.redox-os.org/book/ch01-07-why-rust.html.

[7] Oppermann P. phil-opp/blog_os; 2020. Available from: https://github.com/phil-opp/blog_os.

[8] Levy A, Campbell B, Ghena B, Pannuto P, Dutta P, Levis P. The Case for Writing a Kernel in Rust. In: Proceedings of the 8th Asia-Pacific Workshop on Systems. APSys '17. Mumbai, India: Association for Computing Machinery; 2017. p. 1–7. Available from: https://doi.org/10.1145/3124680.3124717.

[9] Balasubramanian A, Baranowski MS, Burtsev A, Panda A, Rakamarić Z, Ryzhyk L. System Programming in Rust: Beyond Safety. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems. HotOS '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 156–161. Available from: https://doi.org/10.1145/3102980.3103006.

[10] Jung R, Jourdan JH, Krebbers R, Dreyer D. RustBelt: Securing the Foundations of the Rust Programming Language. In: 45th ACM SIGPLAN Symposium on Principles of Programming Languages

(POPL 2018). vol. 2 of POPL. Los Angeles, CA, United States: ACM; 2018. p. 66. Available from: https://hal.archives-ouvertes.fr/hal-01633165.

[11] Steve Klabnik CN. Generic Data Types - The Rust Programming Language;. Available from: https://doc.rust-lang.org/book/ch10-01-syntax.html.

[12] Steve Klabnik CN. Traits: Defining Shared Behavior - The Rust Programming Language;. Available from: https://doc.rust-lang.org/book/ch10-02-traits.html.

[13] Erdin M, Astrauskas V, Poli F. Verification of Rust Generics, Typestates, and Traits. 2018;.

[14] Steve Klabnik CN. References and Borrowing - The Rust Programming Language;. Available from: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[15] Steve Klabnik CN. Send and Sync -;. Available from: https://doc.rust-lang.org/nomicon/send-and-sync.html.

[16] Lankes S, Breitbart J, Pickartz S. Exploring Rust for Unikernel Development. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. PLOS'19. New York, NY, USA: Association for Computing Machinery; 2019. p. 8–15. Available from: https://doi.org/10.1145/3365137.3365395.

[17] Heldring W. An RTOS for embedded systems in Rust. University of Amsterdam; 2018. Available from: https://esc.fnwi.uva.nl/thesis/centraal/files/f155044980.pdf.

[18] Banda G. HARTEX - Scalable Real-Time Kernel for Small Embedded Systems; 2003. Available from: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=84D11348847CDC13691DFAED09883FCB?doi=10.1.1.118.1909&rep=rep1&type=pdf.

[19] Yiu J. Definitive guide to the ARM Cortex-M3. Embedded technology series. Amsterdam ; Boston: Newnes; 2007. OCLC: ocn170955405.

[20] Liu JWS. Real-time systems. Prentice Hall; 2000.

[21] STM32F4 Series;. Available from: https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html.

[22] Aparicio J. Overhead analysis of the RTFM framework - Embedded in Rust;. Available from: https://blog.japaric.io/rtfm-overhead/.

[23] RTXC Quadros Real Time Operating System;. Available from: https://quadros.com/products/operating-systems/rtxc-quadros-rtos-advanced/.

[24] RTXC 3.2 RTOS;. Available from: https://quadros.com/products/operating-systems/rtxc-3-2-rtos/.

[25] Örnvall O. Benchmarking Real-time Operating Systems for use in Radio Base Station applications; 2012.