2021 IEEE XXVIII International Conference on Electronics, Electrical Engineering and Computing (INTERCON) | 978-1-6654-1221-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/INTERCON52678.2021.9532670

A Methodology for Customization of a Real-Time Operating System for Embedded Systems

Luiz Rubens Lencioni
Department of Computer Systems
Aeronautics Institute of Technology, ITA
São José dos Campos, SP, Brazil
lencioni@ita.br

Denis S. Loubach
Department of Computer Systems
Aeronautics Institute of Technology, ITA
São José dos Campos, SP, Brazil
dloubach@ita.br

Osamu Saotome
Electronics Engineering Division
Aeronautics Institute of Technology, ITA
São José dos Campos, SP, Brazil
osaotome@ita.br

***Abstract*—There is a considerable availability of real-time operating systems (RTOS) for embedded systems in the market, from commercial to free and open source types of software licenses. Nevertheless, not all of them accomplish application-specific requirements as they are provided. In that case, a customization effort is often required. In this paper, we propose a generic methodology to guide embedded systems designers to customize a chosen RTOS to meet design constraints, such as reduced memory footprint layout and static memory allocation of kernel objects. An illustrative reduced example is provided, applying different candidate RTOS through the methodology, and the results are analyzed.**

***Index Terms*—Real-Time Operaring Systems (RTOS), Embedded Systems, Application-Specific RTOS.**

I. Introduction

Real-time operating systems (RTOS) design has evolved since the '70s. In recent years, state-of-the-art architectures have been developed to explore advances in memory innovations, energy efficiency, and multicore processing of embedded systems [1]. Besides providing resources for both time and space task scheduling, they also offer a variety of optional services, such as a communications protocol stack and file management systems. Nowadays, there are dozens of RTOS available, prepared to be ported onto different hardware platforms. They range from commercial solutions, qualifiable towards safety-critical standards, as DO-178 (*e.g.* VxWorks, QNX Neutrino, and Integrity), to no-fees, open source packages (*e.g.* FreeRTOS, Linux with real-time patches [2]).

At the same time, trends in computer architecture such as domain-specific architectures (DSA) [3] have brought new challenges like heterogeneous computing nodes dedicated to particular functions. In the automotive industry, the automotive software architecture (AUTOSAR) standard [4] requires a statically configured RTOS. There, tasks configuration should be performed during compile time and remain unchanged during runtime, for predictability and safety reasons [5].

Moreover, in competitive sectors as consumer electronics and internet of things (IoT), the need to fulfill stringent embedded systems constraints, as cost and energy savings, can lead to a design solution employing a microcontroller with a limited memory footprint and a single-core processor [6]. Although in well-established RTOS configuration options are available, there are boundaries in these parameters usage that prevent their full employment by the applications previously mentioned, and then a customization effort might be needed.

Given this scenario, we propose a *methodology to guide application-specific RTOS customization*. At first, project requirements are regarded, and a candidate RTOS from a list is selected. Next, predefined steps must be followed and the customization feasibility is assessed. At the end of the proposed workflow, the customized RTOS is presented together with a set of configuration options and source code files. Our customization methodology does not require special techniques nor modeling languages throughout the entire process.

II. Related Work

There have been several studies to implement application-specific RTOS over the past decades. In [7], a highly configurable RTOS based on aspect-oriented programming (AOP) is presented. An approach using RTOS modeling is introduced in [8] using a finite state machine (FSM) for an OSEK/VDX-based RTOS. A newer model-based approach is proposed in [9]. In recent years the automotive industry has leading RTOS customizations studies, due to the AUTOSAR-OS requirements [10]. A similar proposal to our work can be found in [11], however, only open source RTOS were analyzed.

III. Proposed methodology

A. Overview

An overview of the proposed methodology is illustrated in Fig. 1. The main steps are illustrated inside the dashed line.

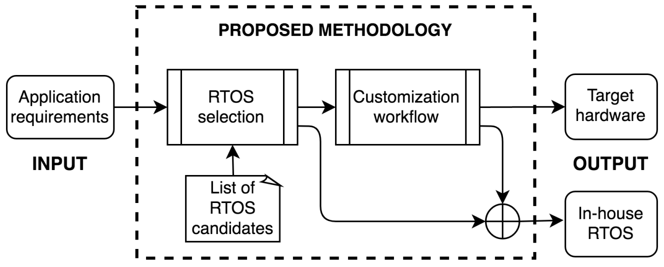


Fig. 1. Proposed methodology overview.

Application requirements or hardware resource constraints are used as inputs, such as the target processor, maximum code memory size, specific set of inter-process communication (IPC), and compatibility to legacy software versions. There can be two different outputs, depending on whether the RTOS customization can or cannot meet the input *application requirements*. In case of positive confirmation, the resulting application-specific RTOS can be cross-compiled and implemented into the *target hardware*. Otherwise, a negative response from the workflow means that no candidate RTOS (from the *list of candidates*) was able to pass through the *RTOS selection phase*, or could not be customized. In that case, a review of the input requirements might be necessary, or an alternative *in-house RTOS solution* should be designed [12].

B. List of Candidate RTOS

An internal list of candidate RTOS needs to be created, maintained, and constantly updated. It should be organized in an orderly fashion. For instance, this order can be based on RTOS availability or popularity in different industry sectors, *e.g.* avionics, medical, automotive. There are specialized embedded systems publications that can be used as a source of information, as they conduct market surveys and report RTOS currently in use by several applications [13].

C. RTOS Selection Step

This initial step in the methodology has the objective to select the most adequate RTOS candidate. This evaluation is made under guidance criteria given by a set of *application requirements*, such as:

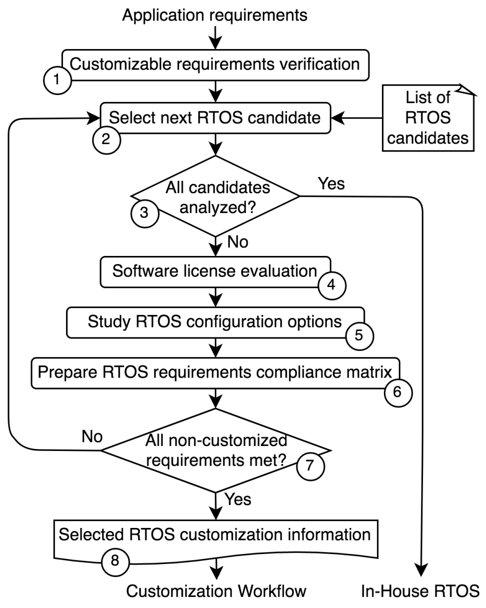
- Budget availability: if cost is a critical constraint, a no-fees RTOS should be chosen;
- Target processor: it can narrow candidate RTOS , as some have limited board support package (BSP) options;
- Backwards compatibility: legacy code developed in previous RTOS, by the development team, can be taken into account;
- Memory footprint: also due to cost, memory layout boundaries can be an important factor to consider;
- Kernel objects allocation: for predictability reasons, some safety-related standards can require kernel components to be statically allocated (instead of dynamically allocated during runtime); and
- Technical support: not only from the original developer that released the RTOS, but also from the open source knowledge shared through community forums.

The sub-steps related to *RTOS selection* are illustrated in Fig. 2, and detailed as follows.

1) Customizable Requirements Verification: For each input requirement, there will be a Boolean classification of whether it can be customizable according to this methodology.

2) Select Next Candidate RTOS: It picks up the next candidate in the already ordered list of candidates.

3) Decision on All Candidates Analyzed: This decision point avoids an infinite loop, in the case none candidate in the list passes through the selection.



Application requirements
Customizable requirements verification
1
Select next RTOS candidate
2
List of RTOS candidates
All candidates analyzed?
3
Yes
No
Software license evaluation
4
Study RTOS configuration options
5
Prepare RTOS requirements compliance matrix
6
No
All non-customized requirements met?
7
Yes
Selected RTOS customization information
8
Customization Workflow
In-House RTOS

Fig. 2. RTOS selection sub-steps.

4) Software License Evaluation: The software license type must be analyzed in detail, as often the same RTOS offers different levels of licensing arrangements. For the sake of simplicity, this work classifies RTOS licenses into only two different groups: commercial and open source.

a) Commercial: It refers to proprietary and non-free software license types, where no rights are granted for source code customization (*i.e.* even if the source code is provided by the software vendor, it cannot be redistributed).

b) Open source: It relates to open source and free software license types [14], where the source code is entirely available, can be modified, and further redistributed. Some licenses in this group are known as permissive BSD-style, allowing for customizations and relicensing with minimum restrictions (*e.g.* copyright and permission notice to be included in the code or documentation). Others, as the GNU General Public License (GPL), also known as copyleft, forbids proprietization, allow customizations, but imposes requirements for developers to provide their application source code, as the original GPL license.

Examples of candidate RTOS classified into different license groups, based on their main license types, are shown in Table I.

5) Study RTOS Configuration Options: This is an important sub-step especially for the commercial license group, as the

TABLE I
EXAMPLES OF CANDIDATE RTOS AND THEIR LICENCES TYPES.

	RTOS name	License type	License group
1	QNX Neutrino	Copyrighted	Commercial
2	FreeRTOS	MIT	Open Source
3	μC/OS-III	Apache 2.0	Open Source

source code is mostly not available. Software vendors offer vast documentation for tailoring RTOS for a specific use.

6) Prepare RTOS Requirements Compliance Matrix: Based on the output of the previous sub-steps, a compliance matrix for each selected RTOS should be prepared. It must state if each requirement can or cannot be met.

7) RTOS Complies With All Non-Customized Requirements Decision: The next sub-step can be triggered only if there are requirements able to be customized. Otherwise, the next RTOS in the list of candidates should be evaluated.

8) Customization Information: The requirements compliance matrix and the configuration options for the chosen RTOS must be gathered to be used in the next step.

D. Customization Workflow Step

The customization workflow step is illustrated in Fig. 3. Each sub-step is detailed next.

1) Identification of Modules and Objects: Each RTOS is based upon elementary building blocks [15]. The core module is known as the kernel, responsible for scheduling, shared resources management and task communication, for instance. Some common elements in the kernel are the scheduler, tasks, semaphores, message queues, mailboxes, event handlers, and the task control block (TCB). Accordingly, these kernel objects need a memory space to be stored and an application programming interface (API) that provides access to operations to be performed (*e.g.* task creation and messaging passing). Support modules to high-level functionalities can be ready to use, such as a file system, device drivers, and networking protocols. To be scalable, generally, these non-essential modules can be switched off during compile time, especially in a RTOS based on microkernel architectures. A list of these modules and objects must be built, based on manuals or even looking directly at the source code.

2) High-Level Modeling: A mathematical model of the scheduling algorithm to be employed is necessary to perform an offline schedulability analysis. For instance, the extended

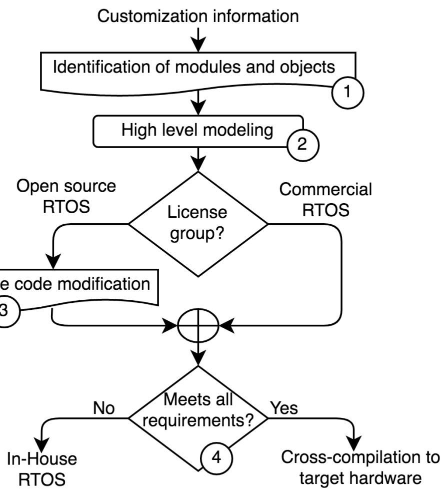


Fig. 3. Customization workflow step.

Rate Monotonic Analysis (RMA), taking into account blocking time due to shared resources [16], can be verified as in (1).

$$\sum_{j=1}^{m} \frac{C_j}{T_j} + \frac{B_m}{T_m} \leq U(m) = m(2^{\frac{1}{m}} - 1) \ , \ 1 \leq m \leq n \quad (1)$$

where C_j is the worst-case execution time associated with periodic task j; T_j is the period associated with task j; T_m is the period associated with task m; B_m is the longest duration of blocking that can be experienced by task m; and n refers to the number of tasks.

3) Source Code Modification: Should any requirement not be achieved and satisfied by simply applying a change in the available configuration options, source code modification must be addressed.

4) Meet Requirements: A final cross-check between the customized RTOS and the application requirements must be done. If all requirements are met, the modified source code with the suitable configuration options can be cross-compiled and able to be debugged into the target hardware. Otherwise, an alternative should be tried, such as developing an in-house RTOS.

IV. Illustrative Example

An illustrative example showing the applicability of our methodology for customization of a real-time operating system is presented as a proof-of-concept. As a starting point, a set of *application-specific requirements* for an optimum RTOS are shown in Table II. Also, we assume the list of candidate RTOS is already provided, as in Table I.

Following the first sub-step in the *RTOS selection step* showed in Fig. 2, customization feasibility for each requirement is performed. Since [Req.1] and [Req.2] deal with intrinsic costs and license grants, they cannot be customizable by this methodology. Next, QNX Neutrino is selected as it remains at the top of the list of candidates. Then, its license terms are evaluated, suitable configuration options are studied, and a requirements compliance matrix is prepared. Whereas QNX Neutrino does not comply with all non-customizable requirements (*e.g.* it has proprietary license policies), it must not be forwarded to the next step within the methodology.

Thus, the evaluation process restarts and FreeRTOS is the next candidate on the list. The requirements [Req.1] and [Req.2] can be satisfied since FreeRTOS adopts a free of

TABLE II
Example of application-specific requirements.

ID	Requirement description
[Req.1]	The RTOS must not have license nor royalties costs.
[Req.2]	The RTOS license must not hinder software proprietization.
[Req.3]	The RTOS must employ a fixed-priority preemptive scheduler.
[Req.4]	The RTOS must statically allocate task objects.
[Req.5]	The RTOS must deny task creation after the scheduler is first started.
[Req.6]	The RTOS must have a BSP compatible with ARM Cortex-M0 cores.

TABLE III
REQUIREMENTS COMPLIANCE MATRIX EXAMPLE FOR FREERTOS.

ID	Customizable?	FreeRTOS
[Req.1]	No	Compliant
[Req.2]	No	Compliant
[Req.3]	Yes	Compliant
[Req.4]	Yes	Compliant
[Req.5]	Yes	Not compliant
[Req.6]	Yes	Compliant

charge and permissive MIT license option. Also, [Req.3] and [Req.6] are met, as a fixed-priority scheduler is employed, and there are BSPs for ARM Cortex-M0 cores from different IC companies. Furthermore, with the configuration option `configSUPPORT_STATIC_ALLOCATION` enabled [17], static allocation of kernel objects are allowed through a different set of API functions, and consequently, [Req.4] can be met. However, FreeRTOS allows the creation of tasks after the scheduler is running, and then it cannot comply with [Req.5] provided "as is". The resulting requirements compliance matrix for FreeRTOS is presented in Table III.

Although FreeRTOS cannot fulfill all the stated application requirements, it does comply with all non-customizable ones, and then can go on to the *customization workflow step*. As detailed in Fig. 3, an identification of modules, files, and objects is conducted. The high-level model for schedulability analysis presented in (1) is also valid for this case. As it is classified as an open source RTOS according to Section III-C4, the next sub-step in the workflow is to *modify its source code* to comply with [Req.5]. The static task creation function and information about the running state of the scheduler (an internal variable named `xSchedulerRunning`) are all contained in a single C-language file named *tasks.c*. A possibility to implement this modification is presented in Listing 1. If an application function tries to create a task while the scheduler is already running, a `NULL` pointer is returned, and the task is not created, as mandated by [Req.5]. Finally, the requirement compliance matrix for the customized FreeRTOS is reviewed. As it now *meets all requirements* the workflow is complete. Then, the output of the methodology consists of the modified source code and needed configuration options able to get *cross-compiled to the target hardware.*

Listing 1
CUSTOMIZED CODE SNIPPET OF FREERTOS `TASKS.C` FILE

```
TaskHandle_t xTaskCreateStatic( ... )
{
   TaskHandle_t xReturn;
   /* Code added to comply with [Req.5] */
   if(xSchedulerRunning == pdTRUE)
   {
      xReturn = NULL;
   }
   else
   {
      /* Legacy code */
   }
   return(xReturn);
}
```

In this example, FreeRTOS release 202012.00 was employed. Although not explicitly mentioned, a subsequent phase of debugging can take place. Ongoing maintenance is also an important point to consider, mainly to keep up with releases containing patches to kernel bug fixes.

V. CONCLUSION

In this paper, a new guideline to select and customize field-proven RTOS for embedded systems is proposed, requiring no special techniques nor modeling languages. As future work, the proposed methodology could be tested towards harsher application requirements, such as adherence to POSIX standards and modifications in real-time scheduling algorithms.

REFERENCES

[1] T. Kuo, J. Chen, Y. Chang, and P. Hsiu, "Real-time computing and the evolution of embedded system designs," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2018, pp. 1–12.

[2] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on PREEMPT_RT," *ACM Computing Surveys*, Feb. 2019.

[3] D. Loubach, J. Marques, and A. Cunha, "Considerations on domain-specific architectures applicability in future avionics systems," in *10th Aerospace Technology Congress*, Oct. 2019, pp. 156–161.

[4] AUTOSAR, *Requirements on Operating System*, 2018. [Online]. Available: https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/SystemServices.zip

[5] K. T. G. Tigori, J.-L. Béchennec, S. Faucou, and O. H. Roux, "Formal model-based synthesis of application-specific static RTOS," *ACM Trans. Embed. Comput. Syst.*, May 2017.

[6] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, S. Leonard, P. Pannuto, P. Dutta, and P. Levis, "The tock embedded operating system," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. Association for Computing Machinery, 2017.

[7] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk, "Ciao: An aspect-oriented operating-system family for resource-constrained embedded systems," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, 2009.

[8] C. Dietrich, M. Hoffmann, and D. Lohmann, "Back to the roots: implementing the RTOS as a specialized state machine," in *11th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT15)*, 2015, pp. 7–12.

[9] R. M. Gomes and M. Baunach, "A model-based concept for RTOS portability," in *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, Oct. 2018, pp. 1–6.

[10] T. Toussaint, J.-L. Béchennec, S. Faucou, and O. Roux, "Using formal methods for the development of safe application-specific RTOS for automotive systems," *CARS 2015 - Critical Automotive applications: Robustness & Safety*, 2015.

[11] P. S. Berntsson, L. Strandén, and F. Warg, "Evaluation of open source operating systems for safety-critical applications," *Software Engineering for Resilient Systems. SERENE 2017.*, pp. 117–132, 2017.

[12] T. D. Juhász, S. Pletl, and L. Molnar, "A method for designing and implementing a real-time operating system for industrial devices," in *2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2019, pp. 149–154.

[13] AspenCore. 2019 embedded markets study. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[14] M. Manteghi, "Understanding open source and free software licensing mechanism: A close review of the alternative approach to traditional notions of software licensing," *Social Science Research Network (SSRN)*, 2017. [Online]. Available: https://ssrn.com/abstract=3082313

[15] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*, 1st ed. USA: CRC Press, Inc., 2003.

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

[17] *FreeRTOS Reference Manual - API functions and configuration options*, Amazon Web Services, 2017, version 10.0.0 issue 1.