Research Article

An Experimental Study on the Embedded Software Development Approach Using TI-RTOS

Aloysio Augusto Rabello de Carvalho and Luiz Eduardo Galvão Martins

Institute of Science and Technology, UNIFESP, São José dos Campos-SP, Brazil

Article history
Received: 23-05-2024
Revised: 22-08-2024
Accepted: 05-09-2024

Corresponding Author:
Aloysio Augusto Rabello de Carvalho
Institute of Science and Technology, UNIFESP, São José dos Campos-SP, Brazil
Email: aloysio.rabello@unifesp.br

Abstract: Embedded systems have experienced significant growth in recent years. These systems, which power a wide range of devices from smart appliances to industrial machinery have become integral to our daily lives and industries. However, this rapid expansion introduces new challenges for developers. A critical decision involves whether to employ a Real-Time Operating System (RTOS) or to forgo it, depending on the specific project requirements. Additionally, developers are focused on finding effective strategies to enhance code quality, promote code reuse, reduce complexity, and streamline the learning curve. These challenges underscore the evolving landscape of embedded systems, where maximizing potential while maintaining efficiency and ease of development has become a top priority. This study presents a comparative evaluation of an embedded system utilizing the TI-RTOS, which includes an RTOS kernel and a set of libraries, against an equivalent system using a minimalistic custom kernel without these libraries. The analysis examines various metrics, including source code size, complexity, architecture, learning curve, and development time. The results indicate that the learning curve associated with adopting TI-RTOS did not show a significant increase compared to the system without an operating system (bare metal). Furthermore, productivity remained largely unchanged when using TI-RTOS. Importantly, the implementation of TI-RTOS did not lead to a notable increase in source code complexity. This study offers valuable insights for embedded systems developers and engineers, demonstrating that integrating TI-RTOS can be a viable option without adding undue complexity to a project. These findings are particularly relevant for those seeking an efficient and user-friendly solution for embedded systems.

Keywords: Operational Systems, Embedded Software, Embedded System, Real-Time Operating System

Introduction

The presence of embedded systems has grown steadily in recent years, reaching near ubiquity in people’s lives. As a result, embedded software development projects have become larger and more complex, following the increasing complexity of electronic components that demand a high degree of reliability and security. (Martins and Oliveira, 2014; Joe and Kin, 2017; Carvalho and Martins, 2021)

Currently, there are numerous options for embedded hardware, offering different brands, features, capacities, and electronic components. These aspects pose new challenges in embedded software development, such as dealing with hardware diversity, meeting growing demand, and reducing development time. (Martins *et al.*, 2015; Lin *et al.*, 2015)

This study seeks to address questions regarding the impact of developing embedded software using a Real-Time Operating System (RTOS), investigating potential gains in performance, changes in code complexity, project development time, the learning curve, and the advantages and disadvantages of using an RTOS in an embedded system.

Embedded systems can be developed using an embedded Operating System (OS) or by programming only the necessary functions for the system to operate without an OS. Both approaches are validated, each offering advantages and disadvantages. The choice depends on factors such as hardware, the problem to be solved, and project requirements. During the project design process, developers may face uncertainty about which approach will bring more benefits and whether or not an OS or an RTOS would be advantageous. (Li and Yao, 2003; Stallings and Paul, 2012; Noergaard, 2012)

The focus of embedded system development in this context is on microcontrollers, which typically have limited memory resources and reduced processing power. (Barr and Massa, 2006; Ball, 2002)

Materials and Methods

In this section, we present the materials and methods employed in this study. We provide an overview of the Insulin Infusion Pump and TI-RTOS, which constitute the core subjects of this research. Additionally, we outline the Research Questions that guided our investigation, followed by a detailed discussion on the Study Definition, Planning, and Hypothesis Formulation. Furthermore, we describe the Variable Selection process and the Operational aspects of the study, ensuring methodological rigor and reproducibility.

Insulin Infusion Pump

The prototype used in this study was a low-cost insulin infusion pump designed for use in public healthcare. The insulin infusion pump is an Embedded System (ES) used in the treatment of Diabetes Mellitus. This device simulates the insulin release of a healthy pancreas, replacing manual insulin administration. It provides the user with precise doses and microinfusions throughout the day, allowing them to continue other activities. (Berget *et al.*, 2019; Casini *et al.*, 2020)

The microcontroller used in the insulin infusion pump is the MSP430, which features ultra-low power consumption, a 16-bit RISC architecture, 16-bit timers, up to 512KB of flash memory, and various analog and digital peripherals. This information is crucial for understanding the hardware constraints and capabilities.

As a medical device that administers medication to a patient, the insulin infusion pump requires extremely stringent deadline accuracy, as any failure could be catastrophic, leading to injury or even death. The system responsible for controlling insulin infusion is a periodic task that uses the insulin dose (previously configured by the user), and the minimum and maximum permissible infusion rates per hour, and triggers a stepper motor to administer the infusion while adhering to specific time intervals. (Borgioli *et al.*, 2022)

In light of the context presented, an experimental study was conducted on the development of a low-cost insulin infusion pump, incorporating an RTOS, specifically the TI-RTOS, into its control software.

TI-RTOS

TI-RTOS is an operating system developed by TI for use on its microcontrollers. For example, the MSP430 family facilitates system development by eliminating the need to create basic system functions from scratch. It provides a range of built-in features, such as device drivers, power management, a multitasking kernel, TCP/IP networking, and a File Allocation Table (FAT) file system, all licensed under the open Berkeley Software Distribution (BSD) code license.

The main objective of this study was to explore different implementation approaches for developing the control software of a low-cost insulin infusion pump. Specific objectives included developing the control software with and without using the TI-RTOS, comparing these different approaches, measuring the learning curve, development time, and code complexity, and identifying the pros and cons of using TI-RTOS in the development of an embedded system.

Experimental Study Protocol

This study was based on the framework proposed by Wohlin *et al.* (2012).

The goal of this experiment was to evaluate different approaches in the development of embedded systems. The focus was on comparing software development using TI- RTOS versus development without it, in order to address our research questions, as follows.

Research Questions

- RQ1: Does the use of TI-RTOS impact the learning curve in embedded software development? This question aims to assess the difficulty of learning how to start using TI-RTOS in an embedded software project.
- RQ2: Can the use of TI-RTOS increase productivity in embedded software development? This question investigates changes in productivity and time spent when developing embedded systems with TI-RTOS.
- RQ3: Does the use of TI-RTOS reduce the complexity of the code produced? This question seeks to determine whether code developed using TI-RTOS in embedded software is simpler or more complex.

Study Definition

To analyze: The development strategies of embedded systems with TI-RTOS and bare metal.

With the purpose: Of evaluating these strategies and identifying the advantages and disadvantages of using the TI-RTOS.

With regard to: Software architecture, learning curve, development time, and the complexity of the code produced.

From the point of view: Of the development team.

In the context: Of the control software for a low-cost insulin infusion pump.

Planning

This study involved developing the control software for a low-cost insulin infusion pump Bare Metal with TI-

RTOS implementation. The system remained the same, with modifications pertinent to the operation of the RTOS. The results demonstrated the benefits and challenges that RTOS brought to the project.

Formulation of Hypotheses

Null hypothesis (H0.1): The use of TI-RTOS in development does not significantly impact the learning curve.

Alternative hypothesis (H1.1): The use of TI-RTOS in development significantly impacts the learning curve.

Null hypothesis (H0.2): The use of TI-RTOS does not increase development productivity.

Alternative hypothesis (H1.2): The use of TI-RTOS increases development productivity.

Null hypothesis (H0.3): The use of TI-RTOS in development does not reduce the complexity of the produced code.

Alternative hypothesis (H1.3): The use of TI-RTOS in development reduces the complexity of the produced code. Hypotheses H0.1 and H1.1 were formulated to investigate our first research question, hypotheses H0.2, and H1.2 were developed to address the second research question, and finally, hypotheses H0.3 and H1.3 were formulated to help answer the third research question.

Variable Selection

This study aimed to evaluate the development process both without and with the use of TI-RTOS. Therefore, the independent variable was the implementation approach of the control software for the low-cost insulin infusion pump, while the dependent variables were the learning curve, development time, and code complexity.

Operation

Preparation: The development of the low-cost insulin infusion pump control software bare metal was carried out by a group of four developers. This project was completed before the research on implementing TI-RTOS in the insulin infusion pump began and they are treated as separate projects. The bare metal software served as the basis for the experimental study, with some parts of the code being reused.

Execution: The study was conducted through the following steps:

1. Utilize the pre-existing low-cost insulin pump control software bare metal as a baseline
2. Select the features to be implemented in the new software version
3. Implement the selected functionalities in the low-cost insulin pump control software using an RTOS
4. Compare the two approaches by selecting specific characteristics for analysis
5. Analyze and interpret the obtained data
6. Validate the analysis
7. Produce the conclusion report

Results

Executable Size

To evaluate the size of the insulin pump control software, we analyzed the total size of the output file (the .out file, which is sent to the development board and contains the executable along with symbolic debug information). The analysis considered the total size and the disk size of the .out file with TI-RTOS and Bare Metal, as shown in Table (1).

Since this is embedded software, the system size is crucial. Many microcontrollers have limited storage capacity and may not support larger code sizes. In this case, we observed an increase in size when using TI-RTOS. The bare metal code had a total file size (on disk) of 258,048 bytes and when TI-RTOS was added, this value increased to 290,816 bytes. This represents a 12.69.

Table 1: Total and disk size of the output file

TI-RTOS code	Bare metal code
284k	258k
290,816 bytes	258,048 bytes

Code Complexity Analysis

This section reports the results of the code complexity analysis when using TI-RTOS and compares them to the bare metal source code. The analysis was divided into two parts: The total number of code lines and the cyclomatic complexity.

Code Lines

For this analysis, we counted the total number of lines for each function in the insulin pump control software. We first counted the lines in the bare metal source code and then counted again after implementing TI-RTOS.

To begin, we listed the functions with the highest and lowest number of lines: The function configuring the maximum basal doses had the highest number of lines.

(141) in both systems, while the put CPU to sleep function had the lowest number of lines (3) in both systems. The average number of lines across all functions was 34 in the bare metal version and 33 with TI-RTOS. Based on these results, we classified the functions into three complexity categories: Functions with up to 33 lines were considered low complexity, those with 34-68 lines were classified as medium complexity, and functions with more than 69 lines were categorized as high complexity. These thresholds were determined by analyzing the entire system, considering maximum, minimum, and average values.

The functions were then divided into three groups based on their complexity (simple, medium, and complex). This classification is illustrated in Figure (1), where the total number of functions in each category is shown for both systems.

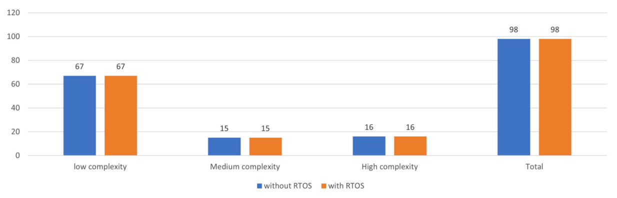

Fig. 1: Comparison between functions based on number of lines of code

In Figure (2), all functions (from 1-98) in the software are listed, along with the number of code lines in each function, both with TI-RTOS and bare metal.

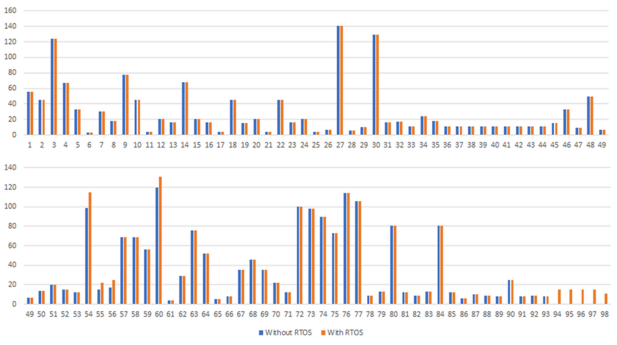

Fig. 2: Comparison of the individual number of lines of functions

Figure (3) shows the functions belonging to each complexity group and the total number of functions for both systems. The expected values for each group are presented in Table (2). A chi-square test was performed using MS Excel, resulting in a p-value of 1, indicating that there was no significant statistical difference when using TI-RTOS.

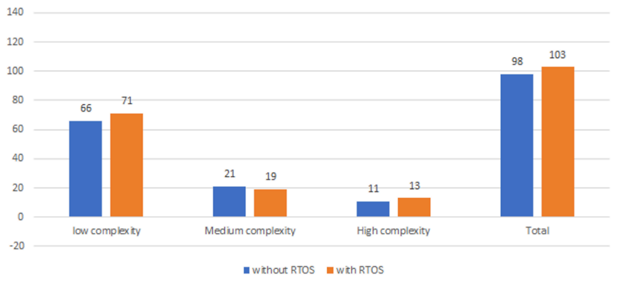

Fig. 3: Comparison of total functions by complexity

Cyclomatic Complexity

The cyclomatic complexity of the management software for the insulin infusion pump, bare metal, and with the use of TI-RTOS. It can be observed that there are five additional low-complexity functions when using TI-RTOS. This increase is due to the fact that these functions are not necessary for the system to work on bare metal versions. Additionally, there was a decrease of two medium-complexity functions and an increase of two high-complexity functions. These changes occurred because these two functions required modifications to work with TI-RTOS, resulting in their classification being elevated to high complexity.

Table 2: Obtained values, expected values, and p-value in code line analysis

Complexity Level	Bare Metal	TI-RTOS	Total
Low Complexity	67	67	134
Medium Complexity	15	15	30
High Complexity	16	16	32
Total	98	98	196
Expected	Bare Metal	TI-RTOS	Total
Low Complexity	67	67	134
Medium Complexity	15	15	30
High Complexity	16	16	32
Total	98	98	196
p-value	1		

All the functions developed in both systems and their complexity levels are shown in Figure (4). The functions are numbered from 1-98, with the minimum and maximum complexity values being 0 and 20, respectively.

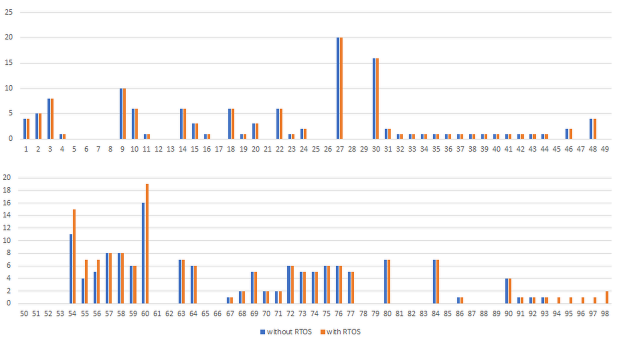

Fig. 4: Comparison of the individual complexity of functions

To validate the values obtained from the cyclomatic complexity analysis of the system with TI-RTOS and a bare metal version, a chi-square test was applied Table (3) presents the functions categorized into each complexity group, along with the total number of functions for both systems and the expected values for each group. The chi-square test was conducted using MS Excel, with a significance level of 0.05. The resulting p-value was 0.850028. Since the p-value was greater than 0.05, the chi-square test indicates that there were no results outside the expected pattern (i.e., there is no significant statistical difference when using TI-RTOS).

The chi-square test results for our table of functions (classified as low, medium, and high complexity) indicated that adding TI-RTOS to the software did not introduce significant variance among the three classes. In other words, this metric suggests that cyclomatic complexity is not significantly increased. Given the benefits that TI-RTOS can provide to a project—such as deterministic task execution timing, semaphore usage, mutual exclusion, network connectivity modules, drivers, energy management, and file systems—this is a positive indication, as it implies that these advantages can be realized without increasing the overall complexity of the project.

Table 3: Obtained values, expected values, and p-value in the cyclomatic complexity analysis

Complexity Level	Bare Metal	TI-RTOS	Total
Low	66	71	137
Medium	21	19	40
High	11	13	24
Total	98	103	201
Expected	Bare Metal	TI-RTOS	Total
Low	66.79	70.2	137
Medium	19.5	20.49	40
High	11.7	12.29	24
Total	98	103	201
p-value	0.850028		

Development Time

For the analysis of development time, six functions of the control software were selected and redeveloped using TI-RTOS. Each function was tested 10 times and the time for each attempt was measured using a stopwatch on a cell phone to individually evaluate the development time.

This test aimed to show the evolution of development time when using TI-RTOS. The results revealed a decrease in development time, with a rapid reduction observed until the time stabilized. To avoid bias in the collected data, all these functions were developed by the same software engineer, with a two-day interval between each attempt. This interval was used to prevent the development process from becoming too mechanical and to minimize bias in comparing the time spent. After each development attempt, the code was compiled and executed on the Printed Circuit Board (PCB) to validate the attempt and ensure it was error-free. Each function showed a reduction in development time, which can be observed in the learning curve of the functions. The percentage of time saved is shown in Table (4), where each row indicates the percentage reduction in time compared to the first attempt.

It was observed that the time measured for each attempt showed a significant reduction, particularly between the first and second attempts. This pattern was consistent across all functions of the control software. However, after a few attempts, the reduction became smaller with each new attempt.

To analyze these time curves, the Cox survival analysis statistical process was used. The curve was analyzed until the occurrence of an event of interest, defined as the number of attempts required for the development time of each function using TI-RTOS to match the bare metal time.

Table 4: Time reduction in percentage with each new development attempt

Task	Semaphore	Config	Basal Infusion	Bolus Infusion	Standard Bolus
50%	79.32%	50.45%	96.34%	90.32%	90.62%
46.04%	59.58%	33.33%	77.07%	77.06%	84.59%
40.70%	47.84%	24.62%	73.56%	67.62%	80.80%
35.58%	37.96%	19.52%	68.78%	63.80%	76%
33.02%	36.73%	21.32%	61.18%	62.84%	64.40%
32.55%	35.49%	20.42%	59.91%	62.48%	59.37%
31.39%	37.04%	19.82%	58.37%	61.53%	54.57%
30%	36.42%	24.02%	57.24%	60.69%	51.11%
29.53%	35.80%	18.91%	56.54%	60.09%	48.77%

The time curve for implementing the task function in the insulin pump software is shown in Figure (5). This curve showed the greatest reduction in time during the first attempt and reached stability after the seventh attempt.

The development time curve for the implementation of the semaphore function is shown in Figure (6). This curve exhibited a significant reduction in time during the first two attempts, with a slight increase in the eighth attempt, after which it reached stability.

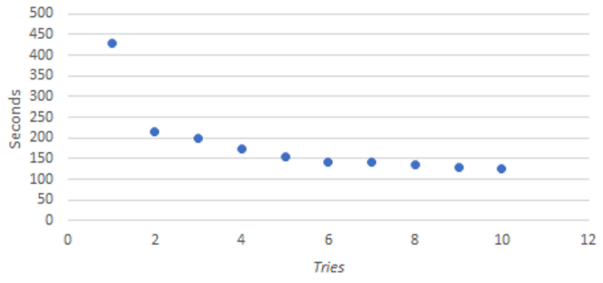

Fig. 5: Time spent in seconds per number of retries of the task function

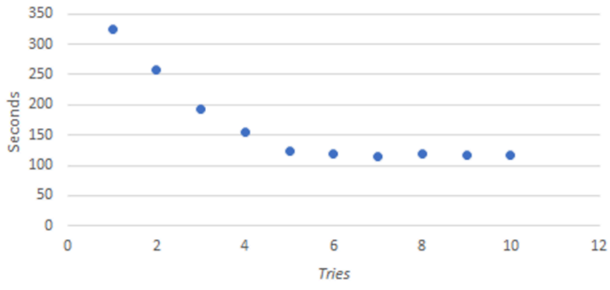

Fig. 6: Time in seconds per number of development attempts for the semaphore function

The system configuration curve was the only one that did not indicate stability, including two attempts with increased time. It was observed that the first attempt presented the greatest drop-in time, which continued on a smaller scale until the fifth attempt. After that, there was a small increase in time, and on the ninth attempt, a new rise in development time was observed, as shown in Figure (7).

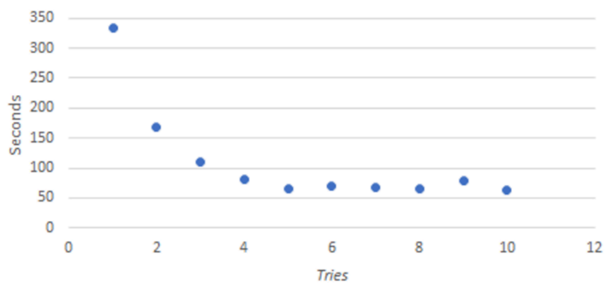

Fig. 7: Time in seconds per number of attempts of TI-RTOS configuration

The basal infusion function curve was the only one in the insulin infusion pump control software that did not show a considerable time reduction in the second attempt, as seen in Figure (8). The third attempt showed the largest decrease in time, which continued at a smaller scale until the sixth attempt when it began to stabilize.

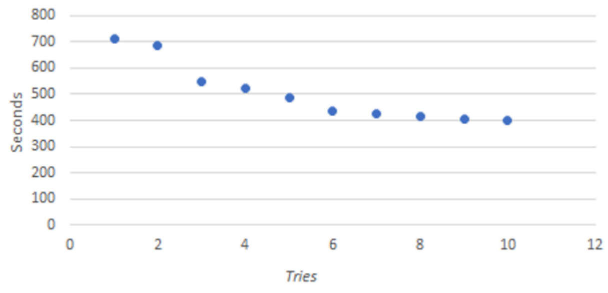

Fig. 8: Time in seconds per number of attempts of the basal infusion function

During the development of the bolus infusion function, a significant time reduction was observed until the fourth attempt. Figure (9) shows that stability was achieved after the sixth attempt.

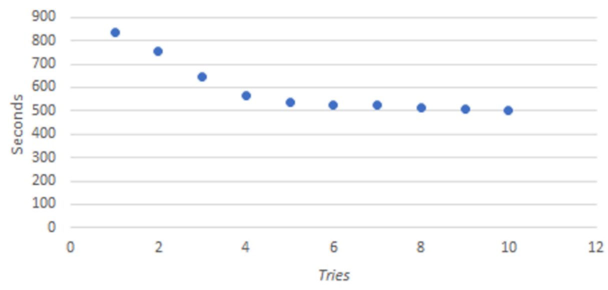

Fig. 9: Time in seconds per number of attempts of the bolus infusion function

The implementation of the standard bolus function did not reach stability and showed continuous time reductions. However, this decrease became less significant after the seventh attempt, as shown in Figure (10). The second and sixth attempts showed the greatest time reductions.

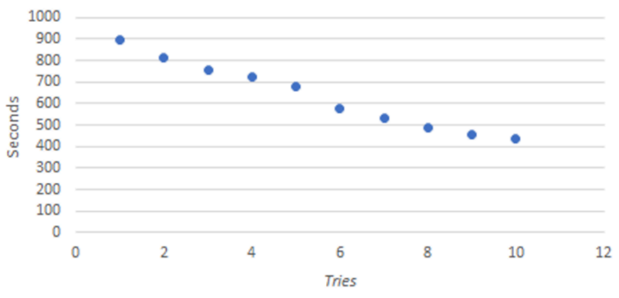

Fig. 10: Time in seconds per number of attempts of the standard bolus function

Complexity Assessment

The evaluation of the control software was conducted using 10 functions, which were extracted and analyzed individually to highlight differences in complexity (cyclomatic complexity, number of code lines, and time spent in development).

Function Selection

To evaluate the development of the insulin infusion pump control software, 10 functions were purposely chosen based on the following criteria: 5 high-complexity functions and 5 medium-complexity functions, focusing on those with the highest complexity present in both systems. The analyses included cyclomatic complexity, number of code lines, and time spent in the development of each function—both in the version with TI-RTOS and the bare metal version.

Comparison

After selecting and redeveloping each of the functions, Tables (5-7) were created to facilitate the visualization and analysis of these functions.

Table 5: Functions developed and their complexities with TI-RTOS and bare metal

Function	Bare metal Complexity	Complexity with RTOS
Adjust qnt steps	8	8
configure hour	10	10
configure dose maxima basal	20	20
configure dose maxima bolus	16	16
calculator bolus	16	19
infusion bolus standard	4	7
infusion bolus extended	5	7
duration bolus	6	6
menu infusion basal	5	5
menu infusion bolus	5	5

Cyclomatic complexity: In the cyclomatic complexity analysis, 7 functions remained unchanged, while 3 functions experienced an increase in complexity (an increase of 3 points for the functions calculator bolus and infusion bolus standard and an increase of 2 points for the function infusion bolus extended). However, despite these increases, none of the functions moved from the medium to the high complexity class. In other words, even with the increase in complexity, all the selected functions remained in the same complexity group, as shown in Table (5).

Number of Code Lines: The analysis of the number of lines for each function closely mirrors the findings from the cyclomatic complexity analysis. Seven functions remained unchanged in terms of line count, while three functions saw an increase: The calculator bolus function gained 11 lines, the infusion bolus standard function increased by 7 lines and the infusion bolus extended function added 8 lines. These changes were necessary to adapt the code to work with TI-RTOS. Similar to the cyclomatic complexity analysis, none of the evaluated functions experienced a change in their complexity class, as shown in Table (6).

Table 6: Functions developed and total lines of code

Function	Bare Meal Number of Lines	Number of Lines with TI- RTOS
adjust qnt steps	65	65
configure hour	78	78
configure dose maxima basal	141	141
configure dose maxima bolus	129	129
calculator bolus	120	131
infusion bolus standard	15	22
infusion bolus extended	17	25
duration bolus	56	56
menu infusion basal	98	98
menu infusion bolus	90	90

Development Time: Each of the selected functions was developed once and the development time was recorded to compare the time spent using TI-RTOS versus not using TI-RTOS. The results indicated that development took longer with TI-RTOS in 7 functions: Adjusting the number of steps, configuring time, configuring the maximum basal dose, configuring the maximum bolus dose, bolus calculator, standard bolus infusion, extended bolus infusion, bolus duration, basal infusion menu, and bolus infusion menu. Conversely, shorter development times were observed when using TI-RTOS in 3 functions: Setting the time, configuring the maximum basal dose, and basal infusion menu. These results suggest that development with TI-RTOS generally takes slightly longer. The most significant difference was observed in the bolus calculator function, where the development time was approximately 1 minute and 4 seconds longer, representing about a 12% increase. However, this additional time is not excessively high for the infusion pump software, as shown in Table (7).

Table 7: Functions and total time in seconds for development

Function	Bare Metal development time	Development time usingRTOS
adjust qnt steps	288 sec	294 sec
configure hour	243 sec	237 sec
configure dose maxima basal	498 sec	495 sec
configure dose maxima bolus	458 sec	462 sec
calculator bolus	532 sec	596 sec
infusion bolus default	73 sec	111 sec
infusion bolus extended	104 sec	146 sec
duration bolus	231 sec	245 sec
menu infusion basal	392 sec	391 sec
menu infusion bolus	359 sec	378 sec

Learning Curves

This section presents the selection criteria for the functions analyzed, the development time for each function, and the learning curves of five functions within the low-cost insulin infusion pump control software. The purpose of this analysis is to examine the learning curves when using TI-RTOS, to understand their behavior, and to determine whether the learning curve is steep or gradual.

Function Selection

Functions were selected based on their need to utilize TI-RTOS functions, allowing for a meaningful comparison, as some functions in the source code did not require modification for TI-RTOS implementation. The chosen functions include: The task function, which is a small piece of code that creates and initiates a task used within the system; the semaphore function, which creates and initiates a semaphore used for task management; system configuration, which involves the TI-RTOS setup process (although not directly related to coding, it is essential for TI-RTOS operation); the infusion basal function, responsible for basal insulin infusion by running in the background, calculating intervals and insulin doses and triggering the stepper motor for insulin delivery; and the infusion bolus function, responsible for bolus insulin infusion, which receives insulin units and activates the infusion motor, adhering to the restriction of 1.5 units of insulin per minute.

Comparison

The learning curves for both systems were plotted and presented individually for each function. Each function was developed ten times with a consistent time interval between attempts and the time spent (in seconds) was recorded. The times for each attempt are shown in Table (8), with functions listed in rows and attempts in columns. Based on this data, a learning curve was generated for each function.

The curves illustrate the speed of learning convergence. The closer the curve approaches a vertical line (between two consecutive attempts), the faster the learning process for that function using TI-RTOS.

After measuring the execution time of each chosen function, the Wright model equation y = C1 · xB was applied, where: y represents the time required to perform the xth repetition of the observed task; C1 is the execution time of the first iteration; and B represents the learning rate, with values ranging from 0 to -1. The closer B is to -1, the faster the learning process. The calculated B values for each function are listed in Table (9).

Table 8: Development time in seconds of the control software functions

Code	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Task	430s	215s	198s	175s	153s	142s	140s	135s	129s	127s
Semaphore	324s	257s	193s	155s	123s	119s	115s	120s	118s	116s
Configuration	333s	168s	111s	82s	65s	71s	68s	66s	80s	63s
Basal	711s	685s	548s	523s	489s	435s	426s	415s	407s	402s
Bolus	837s	756s	645s	566s	534s	526s	523s	515s	508s	503s

Table 9: Values of B for each curve

Function	Value of B
Task	-0,755343419
Semaphore	-0,840911127
Configuration	-0.71831459
Basal Infusion	-0.924426041
Bolus Infusion	-0.865067849

The curve for the Task function, shown in Figure (11), exhibits an exponential decrease. The most significant reduction occurs between the first and second attempts, while the later attempts (6 through 10) show minimal time improvement, with a decrease of just two seconds from the ninth to the tenth attempt.

The curve for the Semaphore function, shown in Figure (12), demonstrates a sharp decline during the initial attempts, particularly between the first and second attempts. After the sixth attempt, the curve begins to stabilize, with increasingly smaller differences in time.

The curve for the Configuration function, shown in Figure (13), exhibits a significant reduction in time during the initial attempts, reaching stability after the fifth attempt. There was a slight increase in time during some attempts, particularly from the fifth to the sixth and from the eighth to the ninth.

The curve for the Basal Infusion function, shown in Figure (14), demonstrates a slightly slower learning process compared to the previous curves, with a steady reduction in time at each attempt, albeit small. The most significant time decrease occurred during the initial attempts, with stability beginning to emerge after the seventh repetition.

The curve of the Bolus Infusion function is illustrated in Figure (15). This curve exhibited a slight decrease across attempts, with a consistent downward trend. The most significant reduction occurred during the initial repetitions and the curve did not stabilize over the course of the ten attempts.

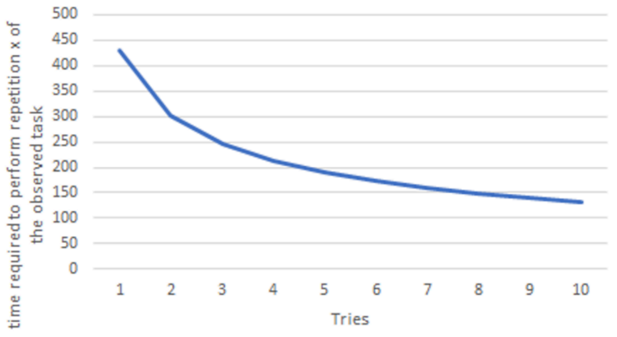

Fig. 11: Task learning curve

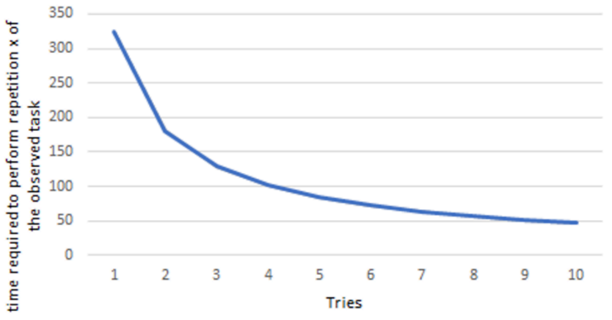

Fig. 12: Semaphore learning curve

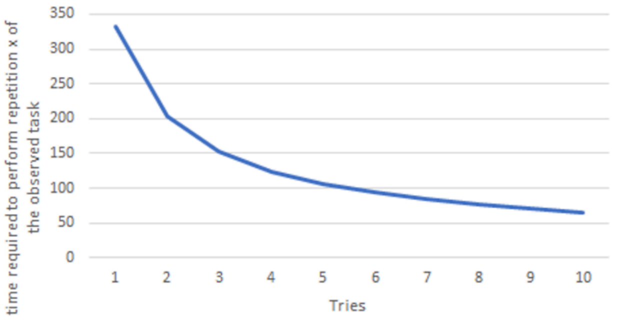

Fig. 13: Configuration learning curve

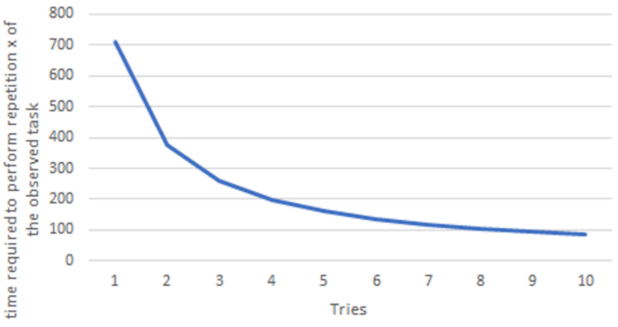

Fig. 14: Basal infusion learning curve

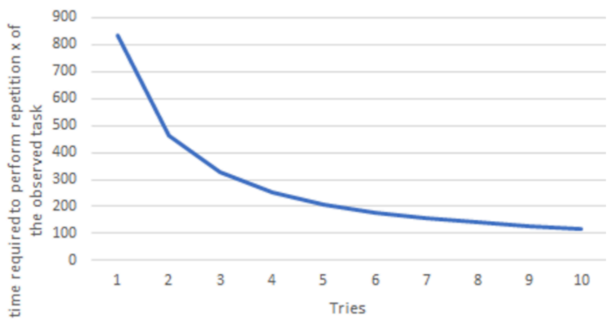

Fig. 15: Bolus infusion learning curve

Discussions

The results of this experimental study were obtained directly from the code complexity analysis through the number of code lines, cyclomatic complexity, .out file size, development time, and learning curve. The results showed us that the complexity variance between the low-cost insulin infusion pump management software was not significant in all complexity tests.

The development of the system using TI-RTOS ran into problems mainly at the beginning of development, given the lack of previous experience with any RTOS and even with the development of embedded systems. Most of the time spent implementing the code with TI-RTOS was spent studying its operation and consulting its documentation. Writing the code was as simple and easy as the bare metal development without using a TI-RTOS, which contradicts the learning curve because it was created taking into account the time taken to develop, and write each code of a given function of the system and did not take into consideration all the time dedicated to understanding TI-RTOS its particularities and system configuration.

TI-RTOS was essential for the development of the insulin infusion pump, as its development brought convenience to the project. It allowed for a higher level of abstraction for the most basic functionalities of the embedded system, as these system functions were already incorporated into the OS, along with access to some hardware resources that were also abstracted by the operating system.

Many of the modifications needed to run TI-RTOS did not occur by coding. During the execution of this experimental study, a lot of time was dedicated to system configuration using IDE CCS. As this configuration was not made for any specific system function or IDE configuration, this learning time is not accounted for in the system complexity analysis.

TI-RTOS brought several benefits to our system. However, not all of its features were utilized. According to the analysis performed, evidence was found that RTOS did not significantly impact all complexity tests, development time, and the learning curve. The only notable impact was observed in the analysis of the output file size sent to the microcontroller, which showed an increase of 12.69%. This rise could be substantial for certain microcontrollers.

The lack of experience with RTOS was the greatest challenge faced in this work. However, this inexperience contributed to achieving the learning curve, which might not have been possible with experienced programmers. It is essential to consider the time required for training and study before starting development with TI-RTOS. While the benefits are considerable, from the perspective of this research, the use of TI-RTOS becomes particularly advantageous after some time, as the development team gains experience, making the development process more fluid and straightforward.

Conclusions

The results of the experimental study analyzing the impact of using TI-RTOS in the management software of a low-cost insulin infusion pump were based on the model published by Amaral (2003). To conduct this experimental study, several hypotheses were formulated, leading to the following results:

- The null hypothesis H0.1 was accepted: The use of TI-RTOS in development does not present a high learning curve, as the learning curve of TI-RTOS was indeed found to be low. Thus, the alternative hypothesis H1.1 The use of TI-RTOS in development presents a high learning curve was rejected in favor of H0.1. This result is discussed in Section 3.5
- The null hypothesis H0.2 was accepted: The use of TI-RTOS does not increase development productivity, as it was demonstrated that the time required for system implementation tends to be comparable to bare metal development after a few trials. The alternative hypothesis H1.2: The use of TI-RTOS increases development productivity was therefore rejected in favor of H0.2. This result is discussed in Section 3.3
- The null hypothesis H0.3 was accepted: The use of TI-RTOS in development does not reduce the complexity of the code produced, as demonstrated in Section 3.2. No significant variance was observed between systems using TI-RTOS and those without. Thus, the alternative hypothesis H1.3: The use of TI- RTOS in development reduces the complexity of the code produced was rejected in favor of H0.3. This result is discussed in Section 3.4

Work Limitations

The study utilized only one OS. The results obtained in this study might differ if other RTOS were involved.

The study considered only Texas Instruments microcontrollers (MSP430). Using different microcontrollers could yield results different from those observed here.

Future Work

The following future works are identified based on this article:

- Analyze whether the use of TI-RTOS affects the flexibility of the code, either increasing or decreasing it
- Investigate whether the results of this study hold true for other microcontrollers available on the market
- Examine whether the findings of this article remain consistent when using different RTOS
- Explore the role of rigorous testing and bug detection in safety-critical software, particularly in systems like insulin infusion pumps, as a potential direction for further research.

Acknowledgment

I am deeply grateful to my advisors for their invaluable guidance and to all the professors at UNIFESP for their continuous support and encouragement. I extend my special thanks to CAPES for providing financial assistance through a scholarship. Finally, I offer my heartfelt gratitude to my family for their unwavering support and understanding throughout this journey.

Funding Information

This study was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Funding Code 88887.809591/2023-00.

Author's Contributions

Aloysio Carvalho: Data collection, Conceptualization, methodology, software implementation, formal analysis, validation, writing—original draft & editing.

Luiz Martins: Conceptualization, formal analysis, validation, writing—review & editing.

Ethics

This research did not involve human or animal subjects and no ethical approval was required.

References

Amaral, E. A. G. (2003). *Empacotamento de experimentos em engenharia de software*.

Ball, S. R. (2002). Special Introduction to the Third Edition. *Embedded Microprocessor Systems: Real World Design*. https://doi.org/10.1016/b978-075067534-5/50022-9

Barr, M., & Massa, A. (2006). *Programming Embedded Systems: With C and GNU Development Tools*.

Berget, C., Messer, L. H., & Forlenza, G. P. (2019). A Clinical Overview of Insulin Pump Therapy for the Management of Diabetes: Past, Present, and Future of Intensive Therapy. *Diabetes Spectrum*, *32*(3), 194-204. https://doi.org/10.2337/ds18-0091

Borgioli, N., Zini, M., Casini, D., Cicero, G., Biondi, A., & Buttazzo, G. (2022). An I/O Virtualization Framework With I/O-Related Memory Contention Control for Real-Time Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *41*(11), 4469-4480. https://doi.org/10.1109/tcad.2022.3202434

Carvalho, A. A. R. de, & Martins, L. E. G. (2021). Sistemas Operacionais Para Software Embarcado: Um Mapeamento Sistemático Da Literatura. *Revista de Sistemas e Computação*, *11*(2), 26-34. https://doi.org/10.36558/rsc.v11i2.7267

Casini, D., Biondi, A., Nelissen, G., & Buttazzo, G. (2020). A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling. *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 239-252. https://doi.org/10.1109/rtas48715.2020.000-3

Joe, H., & Kim, H. (2017). Effects of Dynamic Isolation for full Virtualized RTOS and GPOS Guests. *Future Generation Computer Systems*, *70*, 26-41. https://doi.org/10.1016/j.future.2016.12.020

Li, Q., & Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. https://doi.org/https://doi.org/10.1201/9781482280821

Lin, J. (Denny), Cheng, A. M. K., Steel, D., Wu, M. Y.-C., & Sun, N. (2015). Scheduling Mixed-Criticality Real-Time Tasks in a Fault-Tolerant System. *International Journal of Embedded and Real-Time Communication Systems*, *6*(2), 65-86. https://doi.org/10.4018/ijertcs.2015040104

Martins, L. E. G., & de Oliveira, T. (2014). A Case Study Using a Protocol to Derive Safety Functional Requirements from Fault Tree Analysis. *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, 412-419. https://doi.org/10.1109/re.2014.6912292

Martins, L. E. G., Faria, H. de, Vecchete, L., Cunha, T., Oliveira, T. de, Casarini, D. E., & Colucci, J. A. (2015). Development of a Low-Cost Insulin Infusion Pump: Lessons Learned from an Industry Case. *2015 IEEE 28th International Symposium on Computer-Based Medical Systems*, 338-343. https://doi.org/10.1109/cbms.2015.14

Noergaard, T. (2012). *Embedded SystemsArchitecture: A Comprehensive Guide for Engineers and Programmers*.

Stallings, W., & Paul, G. (2012). *Operating Systems: Internals and Design Principles*. *9*.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. https://doi.org/10.1007/978-3-642-29044-2