ASEE CoED Journal
Published 1 March 2024

RIOS: A Cooperative Multitasking Scheduler in Source Code for Teaching and Implementing Concurrent or Real-Time Software

Frank Vahid[1*], Bailey Miller[2], and Tony Givargis[3]

[1]Dept. of Computer Science & Engineering Univ. of California, Riverside, Riverside, CA, USA
[2] zyBooks, Campbell, CA, USA
[3]Dept. of Computer Science, Univ. of California, Irvine, Irvine, CA, USA

ORIGINAL

Abstract

Embedded systems often implement multiple concurrent tasks of varying priority through the use of a real-time operating system (RTOS). However, an RTOS may introduce overhead, complexity, and maintenance issues. For embedded system applications whose tasks don't heavily compete with one another, an alternative approach is to write the tasks to be cooperative: For each call to the task, the task runs quickly, and then returns so other tasks can execute. For such cooperative tasks, a programmer may then write their own task scheduler in the application's course code like C. However, no common approach exists, and thus embedded programmers sift through myriad online articles and examples, many of which discuss but do not provide scheduler code, provide code only for same-period cooperative tasks, or provide code that is rather complex to learn. To remedy this situation, we introduce scheduler code designed to be ultra-simple to learn and use for the most common cases of cooperative multitask applications. The scheduler code, called RIOS (Riverside/Irvine OS), is written in C but that can be implemented in languages like C++, Java, Python, Javascript, etc. RIOS can be copy-pasted directly into a project's source code, and modified as desired. Via aggressive simplification over several years, the base scheduler code has fewer than 30 lines in C. We describe the core features of RIOS. We also summarize college class experiences with 70+ students showing most students could extend RIOS for various purposes, such as an extension to enable/disable any task (100% success among students), switch tasks between two periods (98% success), add a priority field and sort by priority (94% success), and calculate utilization and jitter (65-70% success). RIOS is used by dozens of universities to teach real-time software concepts to thousands of students, and is used by hundreds of embedded systems engineers in practice.

Keywords: RTOS, scheduler, cooperative tasks, multitasking, non-preemptive, education, source code, C

OPEN ACCESS
Volume
14

Issue
1

Corresponding author
vahid@cs.ucr.edu

Submitted *27 Aug 2022*

Accepted *3 March 2023*

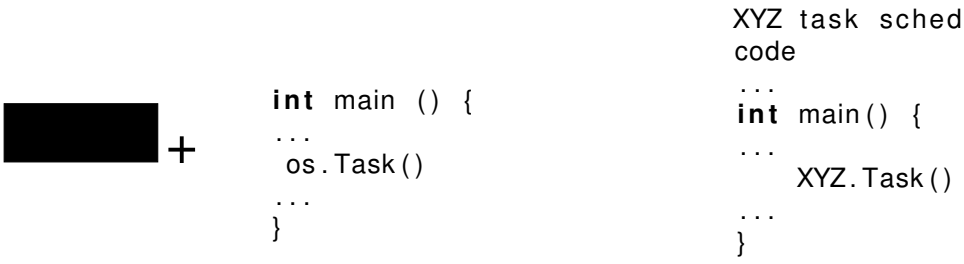
Citation
F. Vahid, B. Miller, and T. Givargis. RIOS: A Cooperative Multitasking Scheduler in Source Code for Teaching and Implementing Concurrent or Real-Time Software . Computers in Education Journal, vol. 14, no. 1, 2024.

1 Introduction

Concurrent tasks are needed in various software applications. Operating systems allow programmers to define processes or threads, which the operating system then manages by executing each task for some time, saving task state, switching to another task, and so on. Some languages like Java have thread concepts built-in. However, in many scenarios, a way is desired for a programmer to define multiple tasks without relying on an operating system or built-in language support.

One such scenario involves embedded systems. Although a trend is for many embedded systems to use increasingly powerful architectures that support a real-time operating system

(RTOS) like the popular FreeRTOS (FreeRTOS, 2021), the dominant architecture is small low-cost low-power microcontrollers, which continue to make their way into more devices like body-worn medical devices, sensors for security, toys, or even ingested pills. An operating system can incur execution overhead that leads to more power consumption and uses up extremely limited memory space. Other scenarios may not be so size or power constrained, but nevertheless benefit from avoiding reliance on OS mechanisms for supporting concurrent tasks.



(a) Task Scheduler (part of OS) + Programmer's Source

(b) Programmers' Source

Figure 1. (a) Traditional scheduler code is a black box, (b) RIOS puts scheduler code in the programmer's source.

In addition to overhead, another drawback of current multitasking approaches is the task scheduler is a "black box", as in Figure 1, meaning students/engineers can't see the code, nor extend the code. But both opportunities could be useful, for learning, and for real implementations.

To address this issue, RIOS was developed to enable programmers to easily program multiple tasks without any operating system, explicit language, or specialized library support. RIOS can be available as source code in the programmer's language, like Java, Python, C, C++, etc., that can be copy-pasted into the programmer's source code itself. RIOS then allows a user to define cooperative tasks each primarily as a normal function/method and a simple struct or class. RIOS can be easily extended to support task priorities, to perform various scheduling heuristics, and more. Due to this simple arrangement, RIOS is useful in many real application scenarios, and also helps college students learn the basics of a real-time scheduler, including extending RIOS. RIOS stands for Riverside/Irvine OS; though not actually an operating system, the name includes OS due to replacing an RTOS.

This paper introduces the RIOS code, describes various ways of extending the code, and summarizes usage results over the past decade. Although the paper focuses on embedded system programs on a microcontroller, the concepts extend to any software that has access to a timer, which includes mobile apps, web apps, and really nearly any computing environment today.

2 Task and microcontroller basics

RIOS is designed to support multiple cooperative periodic tasks on a microcontroller. A task is a program unit intended to run concurrently with other program units. A simple "Hello world"-like example in embedded systems is a "Blinking LEDs" application where Task1 toggles an LED B0 every 500 ms, while Task2 blinks three LEDs B5 B6 B7 in a round-robin sequence, sequencing once every 500 ms. Another simple example is an application where Task1 receives input data and pushes that data onto a queue, and Task2 pops data from the queue for processing.

A periodic task has a specified period at which the task should execute, such as every 500 ms. A periodic task has three possible states: Waiting (should not execute), Ready (should execute), and Running (executing). Consider a task Task1 with a 500 ms period. Suppose Task1 is in the Ready state at 0 ms, and that Task1 enters the Running state at 2 ms and finishes at 5 ms. Then Task1 will be in the Waiting state from 5 ms to 500 ms, becoming Ready again at 500 ms.

A cooperative task is a task that, upon entering the Running state, executes a piece of functionality as quickly as possible and then itself goes back to Waiting, so as to allow other tasks to execute. In contrast, a non-cooperative task is designed to always stay Running, and must be paused (typically by an OS) if any other task needs to run. Periodic cooperative tasks are very common in embedded systems. Figure 2 shows a simple example of a non-cooperative and a cooperative task.

```
void Task0() {
   while (1) {
      B0 = !B0;
      Delay(500);
   }
}
```

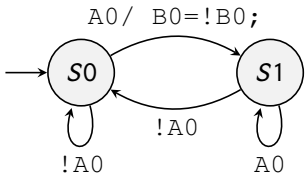
(a)

```
void Task0() {
   B0 = !B0;
}
```

(b)

Figure 2. (a) A non-cooperative task always executes, (b) A cooperative task executes then ends (or sleeps).

Writing cooperative tasks is straightforward. Students are taught not only to avoid putting an infinite loop in a task, but also to never to "wait" inside a task such as waiting for x seconds or waiting (via a loop) for an input value to equal some value. Rather, students are taught to convert such waiting into multiple states. For example, if a task should toggle `B0` whenever `A0` rises from 0 to 1, rather than writing "`while (A0 != 1);`" to wait for `A0` to rise, students can create a two-state state machine, as in Figure 3.




```
Task() {
    static int state = 0;
    switch (state) {
        case 0 : // S0
            if(A0) {
                B0 = !B0;
                state = 1;
            }
            break;
        case 1: // S1
            if(!A0) {
                state = 0;
            }
            break;
    }
}
```

Figure 3. Waiting outside a task, using a state machine. `Task()` executes and exits, then when called again resumes at the correct state (due to the static state variable retaining its previous value).

Note that in the two-state state machine, the task maintains an internal static state variable, to remember where to begin execution when the task is run again, thus avoiding waiting inside the task itself. The task's period must be fast enough to detect events on A0, so should be set to the minimum inter-event separation time that is specified for input A0 (which must be known for any embedded system input). Writing state machines for embedded systems is a discipline in itself and beyond this paper's scope, but is covered in nearly any modern embedded systems textbook Ganssle (2008); Lee and Seshia (2017); Marwedel (2021); Samek (2008); Vahid and Givargis (2001); Valvano (2013); Wang (2017).

To support periodic tasks, nearly all microcontrollers have a built-in timer. The timer can be

configured to automatically call an ISR (interrupt service routine) or otherwise wake a processor at a specified period, such as every 500 ms. In this paper, we assume that the microcontroller has a sleep function with a time parameter, as in `Sleep(500)`, putting the microcontroller into a low-power non-executing state for 500 ms, then automatically waking and resuming program execution. Of course, other assumptions are possible and easily supported.

3 Related work

Some engineers today implement source-level schedulers for cooperative tasks. However, materials to teach engineers how to do so are rare, and are often somewhat complicated. The proposed code in such materials is often more complex than necessary. Furthermore, most college-level embedded system educational content seems not to cover the topic.

Various academic and industry publications address the benefits of writing scheduler code without an RTOS, but many do not provide sample code (Aravindh et al., 2016). Koopman Koopman Jr (1989) described different cooperative techniques, but also without code. Some help programmers create cooperative tasks by using a state machine for each task (Keith Curtis, 2006), but again without scheduler code. Some provide sample code, but the code assumes all tasks have the same period, in which case the scheduler is a trivial loop that calls each task one at a time ("round robin") (CodeProject.org; Gi1, 2017). Others provide code specific to a particular application and not generalized to support different applications an embedded programmer might build (embedded.com, 2016). Github hosts several software projects for cooperative multitasking (Gi2, 2022) but many involve complex code such as using `setjmp/longjmp`, pointers, delay routines, and more. Various industry sites aim to help engineers create their own scheduler but have various complexities such as use of pointers, preemption, complex counting schemes, more functions, tracking of task state, hundreds of lines of code, and more (embedded.com, 2000; embedded.com; Kopják and Kovács, 2012; Binks; Brennan; Pereira; Gi2).

Most embedded systems textbooks do not describe how to write a scheduler in source coe, but may cover related topics like scheduling algorithms, various scheduling concepts, etc. (Vahid and Givargis, 2001; Ganssle, 2008; Lee and Seshia, 2017; Valvano, 2011; Marwedel, 2021). Some academic material can be found online that teaches writing a scheduler **(Patel)**, with that case being simple but differing from ours by using pointers and requiring tasks to put themselves to sleep, and also that work is not published and hard to find.

In short, the engineer seeking to create a cooperative multi-task scheduler currently has no straightforward way to learn how to create such a scheduler, and instead must sift through a myriad of information, often lacking code or having complex code. In fact, they will often find incorrect or conflicting information (Qu2). Likewise, the college instructor seeking to teach cooperative multi-task scheduling has limited educational materials to draw from and no simple template code. This paper aims to remedy those situations by providing complete simple code, and showing that learners are able to quickly understand and use that code to build working multi-task systems without an RTOS.

4 Basic cooperative multitask scheduling

We first consider a simple case as background, to lead up to introducing the RIOS code. In a simple case, all tasks are periodic and cooperative, and they all have a common period (such as 500 ms). In this case, a programmer can simply call the tasks in sequence, and then sleep, as in Figure 4. We refer to each task's function definition as the task's "tick function".

However, things get trickier if tasks have different periods, like 500 ms for Task1 and 750 ms for Task2. In this case, the programmer can set the timer to the greatest common divisor of the periods, then track elapsed time to determine which tasks should execute, as in Figure 5.

Note that the above code initializes a task's elapsed time to the task's period to ensure each task is run upon startup (i.e., all tasks are Waiting at startup). Also note that >= is used rather

```
void Task0() {...}
void Task1() {...}

int main(void) {
  while (1) {
    Task0();
    Task1();
    Sleep(500);
  }
}
```

Figure 4. C-like source code for a program with periodic cooperative tasks having a common period: Programmers can just wait for the common period (here 500 ms), then call each task.

than just ==, a safety measure in case a programmer uses a timer period that doesn't evenly divide a task's period.

The above approach is part of the RIOS approach, namely to create periodic cooperative tasks and then use the timer, variables, and branches to execute each task at the appropriate time. However, RIOS goes a step further to create cleaner, more extensible code.

5 The RIOS approach for executing periodic cooperative tasks

RIOS groups task items, using a struct in C or class in C++. In Figure 6, each task keeps a task's period, elapsed time, and pointer to the task's function. All tasks are in an array, and the scheduler code becomes a simple for loop. For more tasks, the array size and for loop bound are made larger than 2.

The below code is intentionally simple, yet quite powerful, enabling a wide variety of task-based implementations to be easily implemented without any OS or built-in language support, as long as tasks are written to be cooperative. The simplicity of the code evolved over several years of aggressively striving for the simplest-possible code for RIOS.

6 Extensions for education and implementation

In addition to supporting tasks implementation without an extra library or reliance on an OS, RIOS enables students to see that a scheduler's basic functionality is to determine Ready tasks and execute them, and furthermore allows for easy extension in various ways. Those extensions can be used to teach real-time concepts, and also are useful in real implementations.

6.1 Priority

One category of extensions involves scheduling based on priority. When two tasks are Ready at the same time, the scheduler must decide which to execute first; the chosen task is said to have "priority".

The above code assigns higher priority to tasks earlier in the array. Students can be given tasks where execution order might matter, and then see and learn the impact simply by placing the tasks in different orders in the array.

Having to arrange tasks in an array according to priority can be cumbersome, so a simple extension is to allow a programmer to specify a numeric priority for each task. Once all tasks are defined, the RIOS code can be extended to sort the tasks array by priority as in Figure 7, so if two tasks are ready at the same time, the higher-priority task gets executed first.

Another popular scheduling algorithm is "rate monotonic" scheduling, where tasks with smaller periods (faster rates) get higher priority. As such, a programmer could write and call a different sorting algorithm, like `SortByDescendingPeriod()`, to achieve rate monotonic scheduling.

```
...
int main(void) {
   int task0Period      = 500;
   int task0ElapsedTime = 500;

   int task1Period      = 750;
   int task1ElapsedTime = 750;

   int sleepPeriod      = 250; // GCD

   while (1) {
      if (task0ElapsedTime >= task0Period) {
         Task0();
         task0ElapsedTime = 0;
      }
      if (task1ElapsedTime >= task1Period) {
         Task1();
         task1ElapsedTime = 0;
      }

      Sleep(sleepPeriod);
      task0ElapsedTime += sleepPeriod;
      task1ElapsedTime += sleepPeriod;
   }
}
```

Figure 5. Source code for different-period tasks, executing any task whose period has elapsed.

Yet another feature of real time systems is that of a "deadline". A task may, upon becoming Ready, be required to execute within a certain time. For example, a task `Task0` with a period 500 ms and deadline 10 ms means that if the task becomes ready at say time 2000 ms, the task must begin executing by 2010 ms. Such deadlines ensure that an important task, like a task that updates the throttle in a cruise controller, does not experience extensive "jitter" – the delay between a task's Ready state and Running State. Thus, another possible RIOS extension is to add a "deadline" field to the task's struct, and then sort by deadline.

In fact, all three of the above sorting functions could be written, and then a single function `SortTasksForPriority(priorityType)` function could be written where the caller can just specify the desired type of priority as an argument – a common feature in real-time operating systems.

Further extensions are possible, such as extensions that treat priority as dynamic rather than static, which can be implemented by re-sorting the array immediately after the `Sleep()` function, or by searching the array for the next task to execute each time through the for loop.

6.2 Analyses

Another category of extensions involves writing code to analyze the execution of tasks.

For example, a common concern of programmers is microprocessor utilization: The percent of time the processor spends executing code. Assuming the timer's current value is available (as is the case for most microcontrollers), a programmer can extend the scheduler to keep track of how long each task executes, by reading the timer before and after a task executes. The programmer can read the timer in the main loop as well, and then use all those values to determine utilization.

Or, a programmer could read timer values to determine the jitter each task is experiencing,

```
// Task1(), Task2() functions omitted
typedef struct task {
  int period;
  int elapsedTime;
  void (*TickFct)(void);
} task;

int main(void) {
  task tasks[2];

  task[0].period      = 500;
  task[0].elapsedTime = 500;
  task[0].TickFct     = &Task0;

  task[1].period      = 750;
  task[1].elapsedTime = 750;
  task[1].TickFct     = &Task1;

  int sleepPeriod = 250;
  while (1) {
    // Scheduler code
    for (int i=0; i < 2; ++i) {
      if (tasks[i].elapsedTime >= tasks[i].period) {
        tasks[i].TickFct();
        tasks[i].elapsedTime = 0;
      }
      tasks[i].elapsedTime += sleepPeriod;
    }
    Sleep(sleepPeriod);
  }
}
```

Figure 6. RIOS task and scheduler code.

keeping track of average jitter, worst case jitter, etc. A programmer can analyze timer values to determine a task's average execution time, and the observed worst-case execution time, which could then be used to influence scheduling.

6.3 Other extensions

Numerous other extensions are possible. For example, particular tasks could be enabled or disabled based on the program's status or external input values, achieved by adding an enabled flag to each task struct and then updating those flags at appropriate times. Or, task priorities could be adjustable during runtime, such as if a system is in high-performance mode or low-power mode. As with the extensions above, such extensions can typically be implemented with small code adjustments to the RIOS scheduler code.

Related work involves the use of simulators to help students learn the impacts of different scheduling heuristics Diaz et al. (2007); Martinovic et al. (2003); Pillai and Isha (2013). Those approaches are complementary; RIOS allows students to code simple heuristics themselves in their own working programs, while those tools enable a more thorough analysis of the impacts of a wider range of heuristics on larger conceptual multi-task systems.

```
typedef struct task {
  ...
  int priority; // 0: lowest priority
} task;

...
int main(void) {
  task tasks[2];
  ... // Task definitions
  task[0].priority     = 3;
  task[1].priority     = 5;
  SortByDescendingPriority(tasks);
  ...
```

Figure 7. Defining priorities statically, and then sorting to put higher priority tasks earlier in the array.

7 Experiences and usage

RIOS was released for free use originally in 2012 via a public webpage: `https://www.cs.ucr.edu/~vahid/rios/`. Because RIOS is (intentionally) distributed simply via copy-paste, we cannot count the number of "downloads". However, since we began tracking webpage visits in 2016, the site has been visited 30,000 times by 20,000 users; in 2022 there were 4,200 visits by 3,100 users (with similar numbers in 2021 and 2020). In 2022, 82% of visitors came through search (mostly google searches). 13% were direct referrals from other sites, some of which were commercial microcontroller manufacturers (hence, apparently those manufacturers point users to RIOS), and others of which came from university websites (likely webpages of courses teaching embedded systems).

Furthermore, RIOS was integrated as a core chapter of a popular embedded systems textbook, Programming Embedded Systems by zyBooks/Wiley (Vahid et al., 2013), with growing use. That material has been used by 18,500 students at 90 universities (390 distinct class offerings), with usage increasing about 15% per year. Table 1 summarizes usage data.

Table 1. RIOS usage metrics from 2016 to 2022.

Metric	Value
Website visits (since 2016)	30,000
Website users (since 2016)	20,000
Subscribers to textbook (since 2014)	18,500
Distinct universities adopting textbook (since 2014)	90

At our own university, RIOS is a fundamental part of our introductory embedded systems course, taught to about 300 students per year. Prior to utilizing the RIOS approach for concurrent tasks (pre-2012), we observed about 10-20% of students failing to complete their final projects that required multiple tasks, usually due to bugs in the multi-tasking aspects of their programs. That situation was in fact a key motivator for developing RIOS. Since introducing the RIOS approach involving cooperative periodic tasks and the task structure and scheduler code, the failed projects rate has dropped to nearly 0%, even as the number of students has grown by about 5x, and the rare failed projects are usually due to a bad physical part, not due to failed multi-tasking software. Previously, like in many introductory embedded courses using basic microcontrollers, students were taught various concepts of multitasking, but not given actual multitasking code due to such code being too complex to understand in an intro course. Likewise, RTOS usage wasn't introduced either, due to being too advanced for the intro course. The result was that some students would try to create a single task for all their functionality whereas their application would have been more naturally captured as multiple tasks, or tried to

create their own scheduler. While some had success, both approaches have many pitfalls. A key point of this paper is that RIOS is simple enough to give to the students such that they can actually understand it, use it, and even extend it, even in an introductory course.

In our intermediate embedded and real-time systems course, in Spring 2021 we gave students several assignments that involved extending RIOS. The completion rates for those assignments are shown in Table 2. As can be seen, most students were able to successfully extend RIOS in various basic ways: Allowing a user to enable or disable specific tasks dynamically (via input pin settings), allowing a user to choose between two periods of each task (again via input pin settings), and extending the RIOS code to allow a priority to be set for each task and scheduling by that priority. However, as can also be seen, extending RIOS to calculate utilization or jitter was more challenging, since it involves more advanced programming (namely, creating and maintaining additional arrays, examining timer values, etc.). Our embedded systems class has a mix of computer science, computer engineering, and electrical engineering students – the latter often are not as adept at programming. We plan to improve instruction to help students with those extensions in the future.

Table 2. Scores for students attempting to extend RIOS, from our intermediate embedded systems course, Spring 2021.

Extension	# students attempted	Avg score (out of 10)
User can enable/disable per task	77	10.0
User can switch tasks between two periods	76	9.8
Add priority field to each task, schedule based on priority	74	9.4
Calculate utilizations	73	6.7
Calculate jitter	70	7.0

8 RIOS code without `Sleep()`

We also designed RIOS to operate even if a microcontroller does not have a sleep function. In those cases, the microcontroller surely still has an ISR that gets called automatically by a timer configured with a period. Thus, RIOS requires the programmer to first configure and activate that timer, and then uses a global flag in the ISR to notify the `main()` code of the passing of the timer's period, as shown in Figure 8.

9 Conclusion

Applications that execute multiple concurrent tasks are commonplace. In embedded systems, lightweight resource-constrained architectures continue to be ubiquitous, where installing an operating system is often undesirable due to program size, data size, power, and speed overhead. We therefore developed the RIOS approach to support concurrent tasks directly in a programmer's source code, and aggressively optimized the code over the years to consist of just a few dozen easy-to-understand lines of code (in fact, many may wonder what's the "big deal", as the final code looks quite simple – but that simplicity did not come easily and such code has not been developed widely by others). Though written in C, RIOS can also be implemented in other languages. The code's simplicity allows students to easily extend the code to implement various features common in a real-time OS, giving students a deep understanding of real-time scheduler concepts. Our experiments showed that students could successfully extend RIOS in several ways, but some struggled with more advanced extensions requiring maintaining more complex data structures – we hope to improve learning materials and preparation activities to increase future success. RIOS has been learned by many thousands of students across dozens of universities, and usage continues to grow. RIOS is also used by hundreds of practicing engineers to achieve lightweight multitasking in their products.

```
int timerFlag;
void TimerISR() {
   timerFlag = 1; // Notifies main()
}

// Task1(), Task2() functions omitted

typedef struct task {
  int period;
  int elapsedTime;
  void (*TickFct)(void);
} task;

int main(void) {
  task tasks[2];
  task[0].period      = 500;
  task[0].elapsedTime = 500;
  task[0].TickFct     = &Task0;
  task[1].period      = 750;
  task[1].elapsedTime = 750;
  task[1].TickFct     = &Task1;

  int timerPeriod = 250; // GCD
  TimerSet(timerPeriod);
  while (1) {
    // Scheduler code
    for (int i=0; i < 2; ++i) {
      if (tasks[i].elapsedTime >=
          tasks[i].period) {
        tasks[i].TickFct();
        tasks[i].elapsedTime = 0;
      }
      tasks[i].elapsedTime += timerPeriod;
    }
    while (!timerFlag); // Wait GCD time
    timerFlag = 0;
  }
}
```

Figure 8. RIOS task and scheduler code for a microcontroller without a `Sleep()` function, using the ISR and a global flag to provide time info to the code in `main()`.

Acknowledgments

This work was supported in part by the National Science Foundation Grants CNS#1563652 and DUE#2111323. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

FreeRTOS. (2021) FreeRTOS. [Online]. Available: https://www.freertos.org/

J. Ganssle, *The Art of Designing Embedded Systems*. Newnes, 2008.

E. Lee and S. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*.

MIT Press, 2017.

P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Nature, 2021.

M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. CRC Press, 2008.

F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley Sons, 2001.

J. Valvano, *Embedded Systems*, 2013, self-published book. [Online]. Available: http://users.ece.utexas.edu/~valvano/Volume1/E-Book/

J. Wang, *Real-Time Embedded Systems*. John Wiley Sons, 2017.

T. Aravindh, B. Amgothu, and G. Kalaichelvi, "Multiple active threads with cooperative multitasking in embedded system," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. IEEE, May 2016, pp. 746–750.

P. Koopman Jr, "Heavyweight tasking," *Embedded Systems Programming*, vol. 3, no. 4, pp. 42–52, 1989.

Keith Curtis. (2006) Embedded multitasking with small mcus: Part 2 – cooperative vs. pre-emptive. [Online]. Available: https://www.eetimes.com/embedded-multitasking-with-small-mcus-part-2-cooperative-vs-pre-emptive/

CodeProject.org. To rtos or not to rtos. [Online]. Available: https://www.codeproject.com/Articles/1180332/To-RTOS-or-not-to-RTOS

(2017) Light scheduler. [Online]. Available: https://github.com/Pillou/LightScheduler

embedded.com. (2016) Low-cost cooperative multitasking: Part 1 – building a simple fm player. [Online]. Available: https://www.embedded.com/low-cost-cooperative-multitasking-part-1-building-a-simple-fm-player/

(2022) Github topic on cooperative multitasking. [Online]. Available: https://github.com/topics/cooperative-multitasking?l=c

embedded.com. (2000) Get by without an rtos. [Online]. Available: https://www.embedded.com/get-by-without-an-rtos/

——. Build a super simple tasker. [Online]. Available: https://www.embedded.com/build-a-super-simple-tasker/

J. Kopják and J. Kovács, "Timed cooperative multitask for tiny real-time embedded systems," in *2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE, Jan. 2012, pp. 377–382.

D. Binks. Implementing a lightweight task scheduler. [Online]. Available: https://www.enkisoftware.com/devlogpost-20150822-1-Implementing-a-lightweight-task-scheduler

S. Brennan. Implementing simple cooperative threads in c. [Online]. Available: https://brennan.io/2020/05/24/userspace-cooperative-multitasking/

F. Pereira. Ulwos2: A simple and lightweight cooperative os (part 1). [Online]. Available: https://embeddedsystems.io/ulwos2-a-simple-cooperative-os-part-1/

Non-preemptive priority based scheduler. [Online]. Available: https://github.com/Biggy54321/cooperative-scheduling-C

J. Valvano, *Embedded Microcomputer Systems: Real Time Interfacing*. Cengage Learning, 2011.

S. N. Patel. How to build a scheduler. [Online]. Available: https://homes.cs.washington.edu/~shwetak/classes/ee472/notes/SchedImplementation.pdf

Can embedded systems work without using rtoss? [Online]. Available: https://www.quora.com/Can-embedded-systems-work-without-using-RTOSs

A. Diaz, R. Batista, and O. Castro, “Realtss: A real-time scheduling simulator,” in *2007 4th International Conference on Electrical and Electronics Engineering*. IEEE, Sep. 2007, pp. 165–168.

G. Martinovic, L. Budin, and Z. Hocenski, “Undergraduate teaching of real-time scheduling algorithms by developed software tool,” *IEEE Transactions on Education*, vol. 46, no. 1, pp. 185–196, 2003.

A. Pillai and T. Isha, “Ertsim: An embedded real-time task simulator for scheduling,” in *2013 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE, Dec. 2013, pp. 1–4.

F. Vahid, T. Givargis, and B. Miller. (2013) Programming embedded systems. [Online]. Available: https://www.zybooks.com/catalog/programming-embedded-systems/