Paper ID #36478

ANON: A Task Scheduler in Source Code for Teaching and Implementing Concurrent or Real-Time Software

Frank Vahid (Professor)

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, since 1994. He is co-founder and Chief Learning Officer of zyBooks, which creates web-native interactive learning content to replace college textbooks and homework serving 500,000 students anually. His research interests include learning methods to improve college student success especially for CS and STEM freshmen and sophomores, and also embedded systems software and hardware. He is also founder of the non-profit CollegeStudentAdvocates.org.

Tony Givargis (Professor, Assoc. Dean for Student Affairs)

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

Abstract

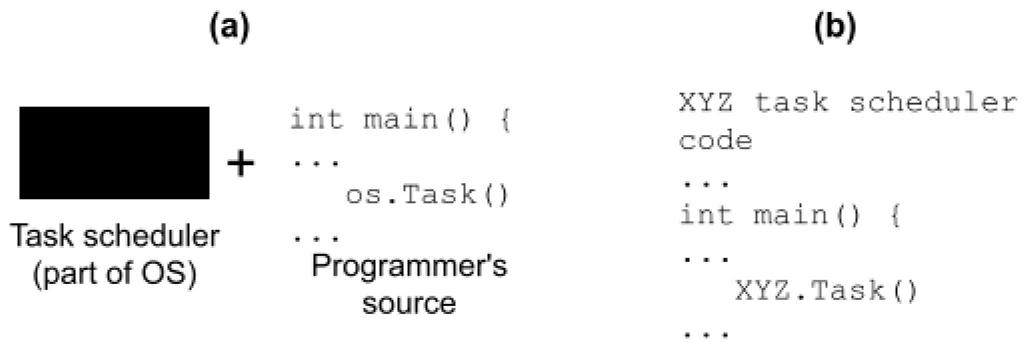
We describe the design and decade-long use of an approach for executing concurrent tasks on a microprocessor without the need for a real-time operating system. We wrote a lightweight non-preemptive task scheduler, called RIOS. The task scheduler is written in C, but that can be implemented in languages like C++, Java, Python, Javascript, etc., rather than in assembly as is commonplace. As such, RIOS can be copy-pasted directly into a project's source code, and modified as desired. The scheduler code includes a structure to hold features of a periodic task like its period and elapsed time since previous execution, an array to hold all tasks, a technique for using a timer and interrupt-service routine (ISR) to keep time, and code to actually call each task at the appropriate time. We describe the core features of RIOS, and its successful usage in embedded systems courses, enabling students to build powerful concurrent-tasks systems correctly and quickly. Students can extend RIOS to further learn real-time concepts, such as including a deadline per task, or creating alternative scheduling algorithms such as rate monotonic, earliest-deadline-first scheduling, or round-robin scheduling. Students can also add functionality to analyze task execution behavior, such as calculating processor utilization or task jitter. As such, students can learn first-hand how the scheduler piece of a real-time operating system operates. Via aggressive code rewriting and minimization over several years, we reduce RIOS's entire code size to just a few dozen lines. RIOS is currently used by dozens of universities to teach real-time software concepts, reaching thousands of students per year. RIOS is also used by hundreds of practicing embedded systems engineers as well, resulting in faster implementation time and much smaller code size than the alternative of linking in a real-time operating system. RIOS is downloadable for free at https://www.cs.ucr.edu/~vahid/rios/.

1 Introduction

Concurrent tasks are needed in various software applications. Operating systems allow programmers to define processes or threads, and 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 operating systems like the popular FreeRTOS [FreeRTOS21] (RTOS: Real Time Operating System), 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.



```
int main() {
...
   os.Task()
...
```

```
XYZ task scheduler
code
...
int main() {
...
   XYZ.Task()
...
```

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 in 2012 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. (The "RI" stands for Riverside/Irvine, ending with OS similar to 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 *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. Thus a task has three possible states: Waiting (should not execute), Ready (should execute), and Running (executing). The above 500 ms Task1 is Ready at 0 ms (as is Task2); if Task1 enters Running and finishes at 3 ms, Task1 will then be in Waiting from 4 ms to 499 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.

(a)

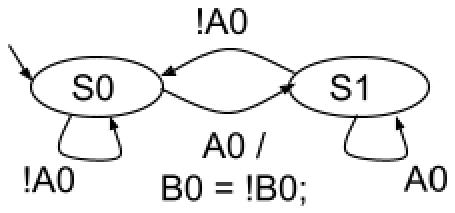
```
void Task0() {
   while (1) {
      B0 = !B0;
      Delay(500);
   }
}
```

(b)

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

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 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, so the Task executes and exits, then when called again resumes at the correct state.

Note that the task maintains an internal 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 [Ga08] [LeSe17] [Ma21] [Sa08] [VaGi01] [Va13] [Wa17].

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 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.

3. Implementing periodic cooperative tasks in source code

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".

```
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.

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.

```
...
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.

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 than just ==, a safety measure in case a programmer accidentally 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 to 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.

4. The RIOS approach for executing periodic cooperative tasks

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

```
// 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.

The above 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 evolved over several years of aggressively striving for the simplest-possible code for RIOS.

5. 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.

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.

```
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.

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.

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.

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, 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.

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 [DiBa07][MaBu03][PiIs13]. 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.

6. Experiences and usage

RIOS was released for free use 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 20,000 times by 14,000 users; in 2020 there were 4,400 visits by 3,300 users. In 2020, 85% of visitors came through search (mostly google searches). About 8% 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 in 2014 as a core chapter of a popular embedded systems textbook [PES14]. That material has been used by 12,500 students at 78 universities (285 distinct class offerings), with usage increasing about 15% per year. Figure 8 summarizes.

Metric	Value
Website visits (since 2016)	20,000
Website users (since 2016)	14,000

Subscribers to textbook (since 2014)	12,500
Distinct universities adopting textbook (since 2014)	78

Figure 8: RIOS usage metrics.

At our own universities, RIOS is a fundamental part of our introductory embedded systems course, taught to several hundred 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.

In UCR's 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 Figure 9. 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.

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

Figure 9: Scores for students attempting to extend RIOS.

7. RIOS code without Sleep()

We also designed RIOS to operate even if a microcontroller does have a sleep function. In those cases, the microcontroller surely still has an interrupt service routine (ISR) that gets called automatically by a timer configured with a period. Thus, RIOS requires the programmer 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 10.

```
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 10: 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().

8. 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 likely used by hundreds of practicing engineers to achieve lightweight multitasking in their products.

9. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1563652 and 2111323.

10. References

[DiBa07] Diaz A, Batista R, Castro O. Realtss: a real-time scheduling simulator. In2007 4th International Conference on Electrical and Electronics Engineering 2007 Sep 5 (pp. 165-168). IEEE.

[FreeRTOS21] FreeRTOS, https://www.freertos.org/, 2021.

[Ga08]] Ganssle J. The art of designing embedded systems. Newnes; 2008 Jul 3.

[LeSe17] Lee EA, Seshia SA. Introduction to embedded systems: A cyber-physical systems approach. Mit Press; 2017.

[MaBu03] Martinovic G, Budin L, Hocenski Z. Undergraduate teaching of real-time scheduling algorithms by developed software tool. IEEE Transactions on Education. 2003 Feb 28;46(1):185-96.

[Ma21] Marwedel P. Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things. Springer Nature; 2021.

[PES14] Vahid F, Givargis T, Miller B. Programming Embedded Systems. zyBooks 2014-2022 (digital textbook), https://www.zybooks.com/catalog/programming-embedded-systems.

[PiIs13] Pillai AS, Isha TB. ERTSim: An embedded real-time task simulator for scheduling. In2013 IEEE International Conference on Computational Intelligence and Computing Research 2013 Dec 26 (pp. 1-4). IEEE.

[Sa08] Samek M. Practical UML statecharts in C/C++: event-driven programming for embedded systems. CRC Press; 2008 Oct 3.

[VaGi01] Vahid F, Givargis TD. Embedded system design: a unified hardware/software introduction. John Wiley & Sons; 2001 Oct 17.

[Va13] Valvano JW. Embedded Systems. (Self-published book); 2013 May. http://users.ece.utexas.edu/~valvano/Volume1/E-Book/

[Wa17] Wang J. Real-time embedded systems. John Wiley & Sons; 2017 Aug 14.