RIOS: A Lightweight Task Scheduler for Embedded Systems

Bailey Miller
Dept. of Computer Science and Engineering
University of California, Riverside
bmiller@cs.ucr.edu

Frank Vahid
Dept. of Computer Science and Engineering
University of California, Riverside
Also with CECS, UC Irvine
vahid@cs.ucr.edu

Tony Givargis
Center for Embedded Computer Systems (CECS)
University of California, Irvine
givargis@uci.edu

ABSTRACT

RIOS (Riverside-Irvine Operating System) is a lightweight portable task scheduler written entirely in C. The scheduler consists of just a few dozens lines of code, intended to be understandable by students learning embedded systems programming. Non-preemptive and preemptive scheduler versions exist. Compared to the existing open-source solutions FreeRTOS and AtomThreads, RIOS on average has 95% fewer lines of total C code for a sample multitasking application, a 71% smaller executable, and 70% less scheduler time overhead. RIOS source code and examples are available for free at http://www.riosscheduler.org. RIOS is useful for education and as a stepping stone to understanding real-time operating system behavior. Additionally, RIOS is a sufficient real-time scheduling solution for various commercial applications.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Multiprocessing/multiprogramming/multitasking

General Terms

Performance, design.

Keywords

Embedded systems, task scheduler, preemption, real-time operating system, C programming, education.

1. INTRODUCTION

Multitasking embedded systems with precise timing may use a real-time operating system (RTOS) to schedule tasks at runtime using priority-based cooperative or preemptive scheduling techniques. Many existing RTOSes provide scheduling services and other features useful in multitasking systems like semaphores, mutexes, queues, etc. [1][7][8][13]. A new embedded systems programmer who needs basic support for multiple tasks may not require the many features of an RTOS. Furthermore, attempts to study RTOS implementations can be hindered by code sizes of thousands of lines spanning dozens of files. RIOS is an alternative to an RTOS, providing real-time scheduling of tasks with only tens of lines of extra code directly inserted into an application C program, requiring no special compilation. The small scheduler is easy for students to understand, and is not hidden through API (application programming interface) calls as in traditional RTOSes.

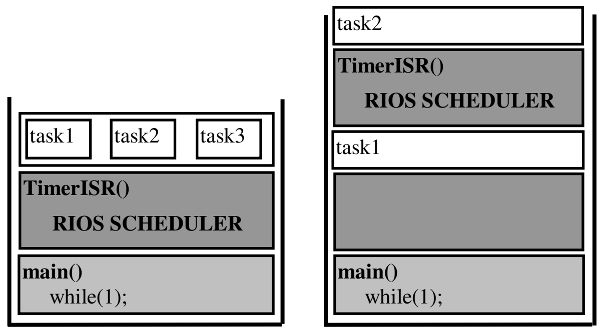
We present non-preemptive and preemptive versions of RIOS. Both versions utilize a peripheral timer to generate an interrupt that contains the RIOS scheduler. Tasks in RIOS are executed within the interrupt service routine (ISR), which is atypical compared to traditional RTOSes.

Figure 1(a) shows the typical program stack of the non-preemptive RIOS scheduler. The main function loops infinitely and performs no real behavior, other than to be periodically interrupted by a timer ISR. The ISR hosting the RIOS scheduler checks if a task is ready to execute, and executes the task if necessary, each such execution known as a *task tick*. For the non-preemptive version, only one task exists on the program stack at any time, and the task must finish before the ISR is called again. The programmer must define each task to be a run-to-completion task, meaning the task executes some actions and then returns, and specifically does not wait on an event, block, or contain an infinite loop. Otherwise, ticks of other tasks might be missed. Run-to-completion tasks are a form of cooperative tasks [1].

The preemptive scheduler in Figure 1(b) allows nested interrupts to occur, which provides higher priority tasks the ability to preempt lower priority tasks. Stack overflow occurrence is mostly prevented by disallowing self-preemption, meaning at most one instance of each task may be present on the stack at any one time. The highest priority executing task will always be at the top of stack, and the number of stack frames is limited by the number of defined tasks in the system. The programmer should define tasks as mostly cooperative for the preemptive scheduler to operate efficiently. The two versions of RIOS thus provide much of the basic functionality necessary to execute concurrent tasks.

This paper is structured as follows. Section 2 discusses the implementation of the existing solutions FreeRTOS and AtomThreads. Section 3 discusses the timer abstractions used in RIOS. Section 4 details the non-preemptive and preemptive RIOS schedulers. Section 5 details our experiences with teaching RIOS in embedded systems courses. Section 6 compares RIOS,

Figure 1: Stack snapshots of: (a) non-preemptive RIOS that schedules one of (task1, task2, task3) at a time, (b) preemptive RIOS that uses nested timer interrupts to preempt tasks.



(a) Non-preemptive **(b) Preemptive**

Figure 2: Creating and running a task in (a) FreeRTOS, (b) AtomThreads, (c) and RIOS. Code is taken from RTOS examples and manuals and is abbreviated for figure clarity where necessary. Some parameter names are changed for consistency.

```
int main() {

  xTaskCreate(
    &function,
    pcName,
    usStackDepth,
    &parameters,
    uxPriority,
    &pvCreatedTask);

  vTaskStartScheduler();
}

void function() { … }
```

(a) FreeRTOS

```
int main() {
  status = atomOSInit(&stack, SIZE);
  if (status == ATOM_OK) {
     status = atomThreadCreate(
           &threadTaskControlBlock,
           priority,
           &function,
           threadParameter,
           &topOfStack,
           stackSize);
     if (status == ATOM_OK) {
       atomOSStart();
     }
  }
}

void function() { … }
```

(b) AtomThreads

```
TimerISR() {
  //RIOS scheduler
}

int main() {
  tasks[0].period = task0_period;
  tasks[0].elapsedTime = tasks[0].period;
  tasks[0].TickFct = &function;

  TimerOn(task0_period);

  while(1); //Wait for interrupt
}

void function() { … }
```

(c) RIOS

FreeRTOS, and AtomThreads in terms of binary size, overhead, and source code size. Section 7 concludes.

2. EXISTING EMBEDDED SCHEDULERS

Many existing RTOS solutions exist; we have selected two for comparison to RIOS based on popularity, availability, and quality. We chose popular RTOSes because we wish to compare RIOS to relevant and modern software, and also because of the support provided by existing communities that are helpful when developing benchmarks. Availability implies that the RTOSes' code bases are under an open-source license, such as the General Public License (GPL), etc. Many commercial systems use an open-source RTOS. Notable exceptions are large real-time systems with hard critical constraints that require additional features like embedded graphics or security. Quality is an important feature that considers the size and overhead of task scheduling, the memory footprint, etc. We selected FreeRTOS and AtomThreads to compare to RIOS, based on the three metrics. Similar comparisons could be made with other solutions.

Other works present similar schedulers to RIOS. TinyOS [9] is a small open source operating system for embedded systems that shares many event-driven, cooperative task characteristics of RIOS. However, TinyOS utilizes the nesC programming language, requires multiple configuration files to run a simple project, and specifically targets sensor network applications. The Super Simple Task (SST) scheduler [11] also provides a single stack, interrupt driven scheduler. Compared to SST, RIOS is leaner and is targeted to the beginning embedded systems programmer. Phantom [10] uses a compiler-driven cross-language approach, where a POSIX C program is translated to an equivalent, multitasking embedded ANSI-C compliant program. Quantum Leaps provides an event-driven framework utilizing UML (Unified Modeling Language) abstractions [13].

FreeRTOS is a widely known RTOS that has been ported to 31 computer architectures, and is used in numerous commercial products [1]. AtomThreads is a very portable small real-time scheduler written mostly in C [8]. Despite having substantial community support for each of the above schedulers, they are complex pieces of software that may not be easily understood by beginning embedded software programmers. For example, the AVR microcontroller port for FreeRTOS contains approximately 9500 lines of text (C code and comments), making it impractical for a new student to understand the low-level implementation details in the few weeks of time that a typical embedded systems course might allocate to RTOS design.

At the user-level, FreeRTOS (and most other RTOSes) provides an API that allows a programmer to create tasks and add them to the scheduler for execution. Figure 2 shows the various API calls required by FreeRTOS and AtomThreads to initialize and run a single task. FreeRTOS has the most straightforward usage available – merely two API calls to run a task. Other RTOSes like AtomThreads require even more function calls and stack initialization routines, since each task is allocated its own stack.

The use of an API hides the behind-the-scenes action of the scheduler. Hiding RTOS implementation details is good when the focus is on the application; however, for educational purposes having a small, understandable scheduler is also desired. Typical RTOS designs, including FreeRTOS and AtomThreads, utilize inline assembly code to perform context switches during multitasking. The use of assembly is required because the scheduler kernel must save the values of all registers and program counter (PC) of the interrupted task, and restore the registers and PC of the task to be switched to. The use of complicated, low-level context switching routines limits both the understandability and portability of the scheduler. Target platforms must be specifically targeted with low-level context switch routines because of different architecture characteristics, like the number and usage of registers, thus requiring effort to port RTOSes to different targets. Assembly routines are generally a necessary feature of an RTOS, although some past work on schedulers have utilized the standard setjmp.h library to implement thread-switching using C level code only [6]. Engelschall provides an overview of many existing thread libraries, noting that 16 of 18 require inline assembly calls [5]. The non-preemptive version of RIOS does not require inline assembly, since a minimum context is saved by the ISR and nested interrupts are not allowed. The preemptive version of RIOS may require minimal assembly to perform context-switching, depending on the target platform.

Both FreeRTOS and AtomThreads create separate stacks for each task, requiring extra initialization and processing during context switches to switch the stack pointers to the appropriate task. RIOS maintains a single stack of nested task frames, as detailed in Section 0. The use of a single stack relaxes the required stack management procedure during context switches, which can reduce the overhead of scheduling. Note that FreeRTOS does have an API available for the use of co-routines that utilize a single stack only, however many RTOSes such as AtomThreads do not support single-stack programs.

Figure 3: Implementations of the timer-related functions for a 8 MHz AVR ATMEGA324P.

```
ISR(TIMER1_COMPA_vect) { //(TimerISR) Timer interrupt service routine
  //RIOS kernel code
}

TimerSet(int milliseconds) {
  TCNT1 = 0;
  OCR1A = milliseconds*1000;
}

TimerOn() {
  TCCR1B = (1<<WGM12)|(1<<CS11); //Clear timer on compare. Prescaler = 8
  TIMSK1 = (1<<OCIE1A); //Enables compare match interrupt
  SREG |= 0x80; //Enable global interrupts
}

TimerOff() {
  TIMSK1 &= (0xFF ^ (1<<OCIE1A)); //Disable compare match interrupt
}
```

3. TIMER ABSTRACTION

The RIOS scheduler C code can run on a wide variety of target microprocessors. The only requirement is that the microprocessor have a peripheral timer that can be set to tick at a specific rate, with each tick calling an interrupt service routine (ISR). Most embedded microprocessors satisfy that requirement. RIOS assumes the following timer-related functions exist:

- TimerISR() -- An ISR function called when a peripheral timer ticks.
- TimerSet(int) -- A function that sets the rate at which the peripheral timer ticks.
- TimerOn()/TimerOff() -- Functions that enable/disable the peripheral timer.

Before using the RIOS scheduler code, a programmer must therefore implement the timer-related functions for the particular target microprocessor. Figure 3 shows an example for an AVR ATMEGA324P microprocessor. Implementing the timer-related functions for other microprocessors is typically straightforward.

4. RIOS

RIOS provides a simple, C-based approach to providing simple multitasking functionality in embedded designs. The technique hinges on the calling of task tick functions within peripheral timer interrupts. Every task in RIOS has an associated state, such that a call to the tick function of the task results in an update of the task state. Thus, tasks are non-blocking and require global storage of state. RIOS is built around the model of synchronous state machines, such that a call to a task tick function results in execution of the actions of a single state. RIOS is not limited to state machines however; normal code blocks can also be used if desired. The only requirement for the use of RIOS for a target platform is that nested interrupts are allowed either by default or by re-enabling interrupts immediately once the Interrupt Service Routine (ISR) has been entered, as is the case with most common embedded processors. In the following sections, we present non-preemptive and preemptive versions of RIOS.

4.1 Non-preemptive scheduler

The first presented version of RIOS is a non-preemptive multitasking scheduler. Each task must be defined by the programmer to be run-to-completion. We present two uses of the scheduler: tasks defined as basic code blocks, and tasks defined as state machines. We highlight the use of state machines in RIOS

Figure 4: A complete sample program using the non-preemptive RIOS scheduler with simple tasks.

Definitions

```
typedef struct task {
  unsigned long period; // Rate at which the task should tick
  unsigned long elapsedTime; // Time since task's last tick
  void (*TickFct)(void); // Function to call for task's tick
} task;

task tasks[2];
const unsigned char tasksNum = 2;
const unsigned long tasksPeriodGCD = 200; //Timer tick rate
const unsigned long periodToggle      = 1000;
const unsigned long periodSequence   = 200;

void TickFct_Toggle(void);
void TickFct_Sequence(void);
```

RIOS scheduler

```
unsigned char processingRdyTasks = 0;
void TimerISR() {
  unsigned char i;
  if (processingRdyTasks) {
    printf("Timer ticked before task processing done.\n");
  }
  else { // Heart of the scheduler code
    processingRdyTasks = 1;
    for (i=0; i < tasksNum; ++i) {
      if (tasks[i].elapsedTime >= tasks[i].period) { // Ready
        tasks[i].TickFct(); //execute task tick
        tasks[i].elapsedTime = 0;
      }
      tasks[i].elapsedTime += tasksPeriodGCD;
    }
    processingRdyTasks = 0;
  }
}
```

Entry point

```
void main() {
  // Priority assigned to lower position tasks in array
  unsigned char i=0;
  tasks[i].period = periodSequence;
  tasks[i].elapsedTime = tasks[i].period;
  tasks[i].TickFct = &TickFct_Sequence;
  ++i;
  tasks[i].period = periodToggle;
  tasks[i].elapsedTime = tasks[i].period;
  tasks[i].TickFct = &TickFct_Toggle;

  TimerSet(tasksPeriodGCD);
  TimerOn();

  while(1) { Sleep(); }
}
```

Loop interrupted by TimerISR()

Task 1

```
// Task: Toggle an output
void TickFct_Toggle()  {
  static unsigned char init = 1;
  if (init) { // Initialization behavior
    B0 = 0;
    init = 0;
  }
  else { // Normal behavior
    B0 = !B0;
  }
}
```

Task 2

```
// Task: Sequence a 1 across 3 outputs
void TickFct_Sequence() {
  static unsigned char init = 1;
  unsigned char tmp = 0;
  if (init) { // Initialization behavior
    B2 = 1; B3 = 0; B4 = 0;
    init = 0;
  }
  else { // Normal behavior
    tmp = B4; B4 = B3; B3 = B2; B2 = tmp;
  }
}
```

because synchronous state machines provide a consistent and logical programming model for teaching students embedded design.

4.1.1 *Basic tasks*

A program that demonstrates the use of non-preemptive RIOS with basic code blocks as tasks is shown in Figure 4. The program toggles and strobes outputs on port B as defined in the Toggle task and the Sequence task. A *task* struct is described near the top of the program. The struct defines all of the components of a task, which include the following variables:

- *period*: the interval that the task should be executed.
- *elapsedTime*: the amount of time that has passed since the previous execution of the task.
- *TickFct*: a pointer to the task's tick function.

To create and schedule a task, a new task struct instance is declared, the above variables are assigned, and the task struct instance is inserted into the *tasks* array at the start of the main() function. Compared to the previously examined RTOSes that require multiple API calls, RIOS provides a simple and transparent process for task initialization.

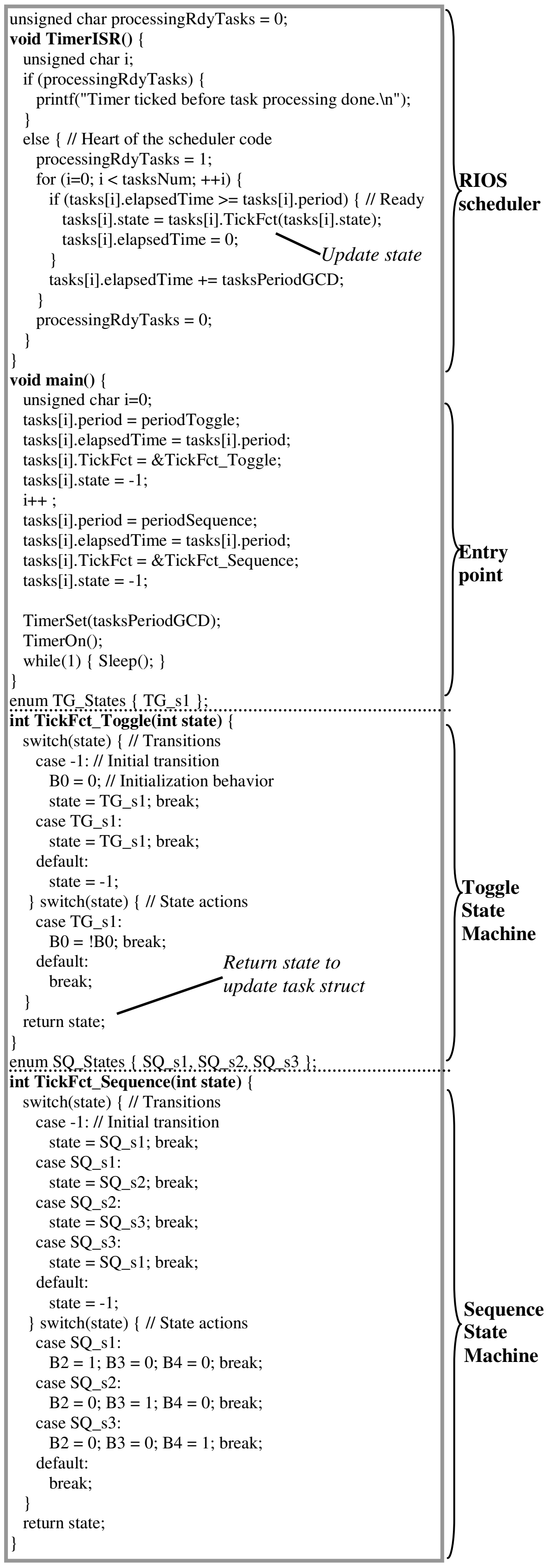
The main() function first initializes the *tasks* array as noted above, and then configures the peripheral timer to periodically signal an interrupt. The timer should be configured to call an ISR at a rate equivalent to the greatest common divisor of all task periods to ensure that the ISR will always execute exactly when at least one task is ready. The main() function then enters an infinite loop, to be interrupted periodically by the ISR.

The ISR hosts the RIOS scheduler code, as seen in Figure 4. The non-preemptive scheduling code requires only about 10 lines of C code. Compared to previously evaluated RTOSes, RIOS can fit into a single C file alongside application code and can be easily understood by beginning embedded system students. The heart of the scheduling code is a loop that iterates over the *tasks* array. If the *elapsedTime* of a task is greater than or equal to the *period* of the task, than the tick function *TickFct* is executed. RIOS is built to execute a single tick of a task when the task period expires, thus tick functions should be run-to-completion, meaning they should not block, wait, or contain infinite loops. Once the tick function returns, the *elapsedTime* of each task is incremented by the timer period.

Priority is given to tasks that have a lower position in the *tasks* array, as the loop in the scheduler evaluates the *elapsedTime* of task[0] first. In the non-preemptive version of RIOS, a flag *processingRdyTasks* is set while the scheduler is active and is reset when the scheduler finishes. If *processingRdyTasks* is set at the start of an ISR, than the previous task could not complete within the timer period and an error is thrown to avoid stack overflow situations. Thus, task tick functions scheduled by the non-preemptive scheduler should never block or wait for resources that may not be available by the end of the timer period.

A program utilizing the non-preemptive RIOS scheduler has a maximum stack depth of three frames, as shown earlier in Figure 1(a). A program always contains at least the main() function stack that is stuck in an infinite loop, as well as periodic ISR calls and a single running task. The non-preemptive version of RIOS does not require any inline assembly routines, since no executing task should ever be interrupted and the call to the ISR will save the return address to main() automatically in the function prologue.

Figure 5: Sample program using non-preemptive RIOS with state machine tasks.


RIOS scheduler
Update state
Entry point
Toggle State Machine
Return state to update task struct
Sequence State Machine

```
unsigned char processingRdyTasks = 0;
void TimerISR() {
  unsigned char i;
  if (processingRdyTasks) {
    printf("Timer ticked before task processing done.\n");
  }
  else { // Heart of the scheduler code
    processingRdyTasks = 1;
    for (i=0; i < tasksNum; ++i) {
      if (tasks[i].elapsedTime >= tasks[i].period) { // Ready
        tasks[i].state = tasks[i].TickFct(tasks[i].state);
        tasks[i].elapsedTime = 0;
      }
      tasks[i].elapsedTime += tasksPeriodGCD;
    }
    processingRdyTasks = 0;
  }
}
void main() {
  unsigned char i=0;
  tasks[i].period = periodToggle;
  tasks[i].elapsedTime = tasks[i].period;
  tasks[i].TickFct = &TickFct_Toggle;
  tasks[i].state = -1;
  i++ ;
  tasks[i].period = periodSequence;
  tasks[i].elapsedTime = tasks[i].period;
  tasks[i].TickFct = &TickFct_Sequence;
  tasks[i].state = -1;

  TimerSet(tasksPeriodGCD);
  TimerOn();
  while(1) { Sleep(); }
}
enum TG_States { TG_s1 };
int TickFct_Toggle(int state) {
  switch(state) { // Transitions
    case -1: // Initial transition
      B0 = 0; // Initialization behavior
      state = TG_s1; break;
    case TG_s1:
      state = TG_s1; break;
    default:
      state = -1;
  } switch(state) { // State actions
    case TG_s1:
      B0 = !B0; break;
    default:
      break;
  }
  return state;
}
enum SQ_States { SQ_s1, SQ_s2, SQ_s3 };
int TickFct_Sequence(int state) {
  switch(state) { // Transitions
    case -1: // Initial transition
      state = SQ_s1; break;
    case SQ_s1:
      state = SQ_s2; break;
    case SQ_s2:
      state = SQ_s3; break;
    case SQ_s3:
      state = SQ_s1; break;
    default:
      state = -1;
  } switch(state) { // State actions
    case SQ_s1:
      B2 = 1; B3 = 0; B4 = 0; break;
    case SQ_s2:
      B2 = 0; B3 = 1; B4 = 0; break;
    case SQ_s3:
      B2 = 0; B3 = 0; B4 = 1; break;
    default:
      break;
  }
  return state;
}
```

4.1.2 *State machines*

State machines are a powerful model that can be used to teach structured methods of embedded system design [11][13]. We have specifically designed RIOS for use with state machines by including a *state* attribute into the task struct. The RIOS scheduler will update the state of a task by executing a tick of the state machine, which results in the state machine transitioning to the next state and executing the actions within the new state.

Figure 5 shows an abbreviated sample program using the non-preemptive RIOS scheduler with state machines. The same tasks from Figure 4 have been implemented as state machines, where task Toggle toggles pin 0 on port B, and task Sequence strobes pins 5-7 on port B. The Toggle task tick function will execute every 1000 milliseconds, while the Sequence task function executes every 200 milliseconds. Every 1000 milliseconds both tasks will be ready to execute, and Toggle will be executed because Toggle has a higher priority (lower position in *tasks*). Since both tasks are relatively simple and require little computation time, preemption is not necessary for this example.

The scheduler supports state machines directly by passing the state of a task as an argument to the task tick function, and updating the task state with a new value when the tick function returns. Initially, state machines are assigned a state value of -1, implying that the state machine has not yet executed. On the first call to the tick function, the state will transition from the -1 state to the real initial state, in this case SQ_s1 and TG_s1, and execute any actions of the state. This technique prevents initial state actions from being skipped on the first call of the tick function. State machines are written as two consecutive switch statements, the first determining the next state to transition to, and then second executing the actions of that state. This structured method of writing state machines provides a useful template for beginning students, since state machines can be designed at a higher abstraction level (e.g., a drawing), and easily translated into code. The state machine code can immediately be incorporated into a multitasking RIOS-based system by performing the simple initialization of the *tasks* struct. Other RTOSes evaluated in this work require the use of infinite loops and API function calls to implement periodic functions. For example, FreeRTOS requires the use of the vTaskDelayUntil() function within an infinite loop in the tick function to specify a frequency with which to wake up and execute the task. Structured state machines that tick periodically provide a conceptually simpler framework for students than blocks of code with timed delays.

4.2 Preemptive scheduler

To support systems that require finer-grained timing than run-to-completion tasks, we introduce a RIOS scheduler that supports preemption of lower priority tasks. Figure 6 shows the entire preemptive scheduler kernel – approximately 15-20 lines of C code. The scheduler in Figure 6 is similar to the non-preemptive RIOS version in that an array of tasks is iterated over to determine if the task is ready to run. A new data structure *runningTasks* is introduced that tracks the tasks that are executing. *runningTasks* effectively acts as a stack where the highest priority task is located at the top of the stack. When a task completes, the task is removed from the top of *runningTasks* and the next lower-priority task begins execution. An idle task is always allocated at the bottom of the *runningTasks* stack and can not be removed.

In addition to checking if a task is ready to run, the condition in the scheduler is updated to check if the priority of the task is greater than the currently executing task, and if the task is not already running. Recall that priority is established by position in the array, where lower elements in *tasks* have priority. Priority based on position in the *tasks* array is useful because we can simply compare the loop iterator *i* to the task at the top of *runningTasks* to determine if the task would have priority in the scheduler. Also, we note that RIOS does not allow self-preemption as self-preemption is rarely useful for applications targeted by RIOS and introduces situations where stack overflows can occur easily without special handling.

There are small critical sections in the scheduler code, in order to prevent nested ISR calls while the scheduler is performing administrative tasks. Immediately preceding the call of a tick function, the task must first be marked as running and pushed onto the stack of *runningTasks*. If another call to TimerISR occurs before *runningTasks* was updated with the current tasks' priority, but after *currentTask* had been incremented, than a lower priority task could possibly execute within the new scheduler stack frame, since the value of *runningTasks[currentTask]* would be *idleTask*. Similarly, a critical section that follows the execution of the tick function protects *runningTasks* from corruption if TimerISR is called immediately before decrementing *currentTask*.

The ISR entry and exit points are bounded by context switching helper macros SaveContext() and RestoreContext(). Depending on the platform, a call to the ISR may not save all of the registers required to be able to seamlessly return to the interrupted position in the program. For example, on the AVR architecture only the registers used by the ISR, i.e. the call-saved registers r2-r17 and r28-r29, are pushed onto the ISR stack during the prologue. Thus, an assembly routine must be provided that also stores the temporary registers and any other special registers provided by the architecture. Since RIOS utilizes only a single stack, the assembly calls are generally limited to pushing or popping of registers on or off the stack only.

Figure 6: Preemptive RIOS scheduler

```
unsigned char runningTasks[4] = {255}; //Track running tasks-[0] always idleTask
const unsigned long idleTask = 255;      // 0 highest priority, 255 lowest
unsigned char currentTask = 0;    // Index of highest priority task in runningTasks

void TimerISR() {
  unsigned char i;
  SaveContext(); //save temporary registers, if necessary
  for (i=0; i < tasksNum; ++i) {                           // Heart of scheduler code
    if (  (tasks[i].elapsedTime >= tasks[i].period)  // Task ready
       && (runningTasks[currentTask] > i)    // Task priority > current task priority
       && (!tasks[i].running)        // Task not already running (no self-preemption)
      ) {

      DisableInterrupts(); // Critical section
      tasks[i].elapsedTime = 0;  // Reset time since last tick
      tasks[i].running = 1;         // Mark as running
      currentTask += 1;
      runningTasks[currentTask] = i;      // Add to runningTasks
      EnableInterrupts(); // End critical section

      tasks[i].state = tasks[i].TickFct(tasks[i].state);      // Execute tick

      DisableInterrupts(); // Critical section
      tasks[i].running = 0;
      runningTasks[currentTask] = idleTask; // Remove from runningTasks
      currentTask -= 1;
      EnableInterrupts(); // End critical section
    }
    tasks[i].elapsedTime += tasksPeriodGCD;
  }
  RestoreContext();//restore temporary registers, if necessary
}
```

Figure 7 shows a stack trace of an example program using the RIOS preemptive scheduler. The example consists of three tasks as outlined in the given table; each task is assigned a period with which to execute their given tick functions, a runtime that indicates how long the task takes to complete, and a priority (lower number indicates higher priority). The greatest common divisor of the set of tasks is 250 ms, thus the timer ISR is configured to tick at a rate of 250 ms. Each task is initially marked as ready to run at the start of the system, thus on the first tick of the ISR, every task is ready to execute. The scheduler executes the tasks in order of their priority, and is initially able to complete T1, T2 and a portion of T3 prior to the next tick of the ISR. A second ISR frame is pushed onto the stack, and T1 is executed by RIOS. Since T2 is not yet ready, and T3 is marked as running in a lower stack frame, RIOS skips them and the ISR returns to yield control back to T3. T3 can not complete its 500ms runtime requirement before the next ISR tick, thus again a second ISR frame is pushed onto the stack and the tasks that are ready and have higher priority are executed. Eventually, after approximately 900 milliseconds, T3 is able to complete, and the original ISR frame returns to yield control to the sleeping main() function.

5. CLASSROOM EXPERIENCES

We have utilized the non-preemptive version of RIOS in introductory embedded systems courses for the past 3 years. RIOS is used in tandem with a digital e-book and a software toolset, named PES (Programming Embedded Systems), both of which were created specifically for the course [12]. The PES tools include: RIMS, a MIPS-based microcontroller simulator, RIBS, a graphical state machine designer, and RITS, a timing diagram viewer to evaluate I/O.

PES focuses on teaching time-oriented programming techniques, culminating with the introduction of RIOS and multitask scheduling. Initially, PES introduces the concept of a synchronous state machine and how state machines can be used to capture desired behavior. The first examples are single-state machine systems that do not require scheduling. More advanced systems are gradually introduced that add additional tasks, and PES develops the programming techniques to support the additional complexity. Multitasking is initially introduced by providing a template for round-robin scheduling that inserts an infinite loop in the main function code to call separate functions for each task. Timing is supported by a periodic timer interrupt that sets a flag. This technique is simple, but does not support tasks with different periods. To introduce support for scheduling tasks with different periods, the task struct is described and a RIOS scheduler is placed into the main code. To allow power-saving modes, RIOS is moved to the timer ISR and replaced in the main function code by a call to Sleep(). PES also describes a version of RIOS used to support event-driven state machine designs. Event triggers can be added into RIOS easily by adding an additional flag to the task struct that indicates if the task has been triggered, and by checking the status of the flag when determining if the task is ready to tick. The preemptive version of RIOS has not yet been incorporated into PES, but we hope to include it in future courses. Table 1 describes the progression of PES from simple systems to complex multitasking systems requiring RIOS.

Table 1: Course topics culminate with RIOS

1.	Capture simple behavior as a graphical state machine.
2.	Translate graphical state machines to C code.
3.	Capture more complex systems as multiple state machines.
4.	Round-robin multitasking of state machines with same period.
5.	RIOS multitasking of state machines with different periods.

The culmination of our introductory embedded systems course is a two-week project. Since we started using PES and RIOS in the classroom, student projects have become noticeably more complex. Three years ago, the typical submission used a single microcontroller, one or two peripherals, and a maximum of three state machines. The projects yielded by PES-instructed students typically contain two or three communicating microcontrollers, multiple peripherals, and five or six state machines. We have found that students can handle complicated multitasking situations much easier, and thus the project quality has increased. Examples and comparison of student projects are available online at http://www.riosscheduler.org.

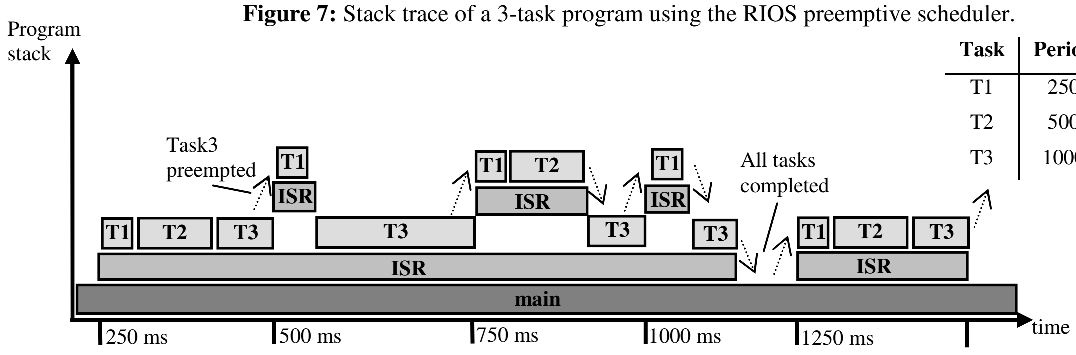
6. RIOS VS. RTOS COMPARISON

6.1 Scheduling overhead

A primary metric of the quality of a scheduler is the overhead, or how much time a program spends determining which task to run next, and switching contexts to execute the task. Every RTOS utilizes a timer interrupt to provide a basic method for tracking time elapsed in the system. We calculate overhead for each RTOS by starting a separate hardware timer at the beginning of the timer interrupt, and recording the difference in time when the interrupt returns. In RIOS, the tick function call is executed within the ISR, thus we stop the hardware timer whenever a tick function is currently being executed. Note that due to the inclusion of profiling code, the overhead is slightly higher in all cases.

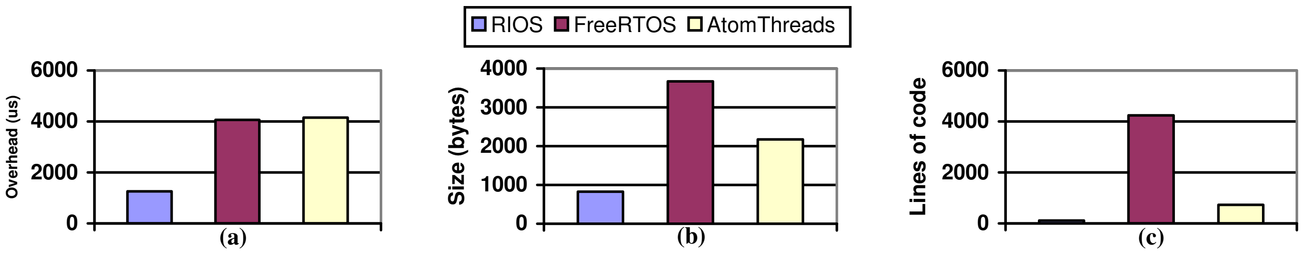
The program executed consists of 3 tasks of varying runtime, priority, and period. Task1 is a short-lived (1ms runtime), high priority task that executes rapidly (25ms period). Task2 has a medium-length runtime (5ms), medium priority, and a medium-length period (50ms). Task3 has a long runtime (25ms), low priority, and executes rarely (100ms). Each task consists of a single delay() function call that simulates some computation. The

Figure 7: Stack trace of a 3-task program using the RIOS preemptive scheduler.



Task	Period	Runtime	Priority
T1	250	10	1
T2	500	100	2
T3	1000	500	3

Figure 8: Comparison of (a) scheduling overhead for 1 second of execution time, (b) compiled binary size, and (c) lines of code.



system is run for 1 second of total time, and the amount of overhead for each interrupt is added to a global sum. The targeted architecture is an AVR-ATMega324P, which is a small 8-bit RISC microcontroller from Atmel that is configured to run with an 8MHz clock [1]. The timer tick of RIOS and each RTOS is configured to generate interrupts at the greatest common divisor of each task period, 25 ms, to minimize unnecessary overhead. Within 1 second of time, the scheduler is required to perform 10 preemptions of Task3 in order to yield to Task1 or Task2.

The scheduling overhead of RIOS, FreeRTOS, and AtomThreads are shown in Figure 8(a). RIOS requires ~1250 microseconds of overhead, approximately 30% of the overhead required by FreeRTOS or AtomThreads. RIOS is faster because of its simplicity. RIOS does not check for stack overflow, or if tasks are ready to unblock. Also, RTOSes tend to implement extra function calls within the scheduler, e.g., FreeRTOS calls the functions vTaskIncrementTick() and vTaskSwitchContext(), which requires prologue and epilogue stack management and thus require a few more cycles per tick. For best comparisons, the RTOSes were made as lean as possible. The FreeRTOS build does not include semaphores, queues, mutexes, and most of the API calls except those necessary to enable periodic tasks (vDelayTaskUntil). The majority of extra delay per tick (~50 ms) in FreeRTOS compared to RIOS comes from the subroutine to check for ready tasks.

6.2 Binary and code size

An important metric of a scheduler is the size of the compiled scheduler code. Embedded systems are often limited by memory size, and thus a scheduler should be small so that space exists for the application. Figure 8(b) shows compiled binary sizes of RIOS, FreeRTOS, and AtomThreads for a sample application. All programs were compiled using the –Os flag. All extra modules not required, such as semaphores and mutexes, were not linked into the RTOS builds. As shown in Figure 8(b), RIOS requires an executable of only 830 bytes, compared to 3668 and 2172 bytes for FreeRTOS and AtomThreads, respectively.

The amount of source code of a scheduler is important when considering an educational environment. If the objective is to teach basic scheduling techniques to students, than interpreting thousands of lines of codes spread amongst 10 files is a distraction. The main benefit of RIOS is that a small amount of readable code can enable multitasking, albeit without some of the features supported by complete RTOSes. We used the open source tool cloc [4] to determine the amount of actual lines of code of each RTOS. cloc filters out blank lines and comments, thus Figure 8(c) shows the raw number of lines of code. The RIOS sample program consists of 116 lines of code. FreeRTOS and AtomThreads consist of 4242 and 733 lines, respectively. FreeRTOS is much larger than AtomThreads or RIOS because FreeRTOS contains more features and modules (which can be disabled at runtime for producing comparably sized binaries). Nonetheless, RIOS provides a comprehensible approach for students in an educational environment.

7. CONCLUSION

We presented non-preemptive and preemptive versions of RIOS, a lightweight scheduler for embedded systems. It was shown that basic multitasking of periodic tasks can be performed in approximately a dozen lines of code. Compared to FreeRTOS and AtomThreads, RIOS requires 95% less code, is 70% faster, and results in 71% smaller binaries on average. RIOS requires that tasks are periodic, however when coupled with the synchronous state machine programming model RIOS may provide an effective platform for teaching embedded multitasking. Marked improvements in student project quality has been noted while using RIOS in introductory embedded programming classes.

8. ACKNOWLEDGMENTS

This work was supported in part by a U.S. Department of Education GAANN fellowship and by the U.S. National Science Foundation (CNS 1016792, DUE 0836905, CPS 1136146).

9. REFERENCES

[1] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. 2002. Cooperative Task Management Without Manual Stack Management. USENIX.

[2] Atmel Corporation. 2012. http://www.atmel.com/.

[3] Barry, R. FreeRTOS. http://www.freertos.org/.

[4] Danial, A. 2006. cloc: Count Lines of Code. Northrop Grumman Corporation.

[5] Engelschall, R. 2000. Portable multithreading: the signal stack trick for user-space thread creation. USENIX 2000.

[6] Engelschall, R. 2005. Gnu pth - the gnu portable threads, http://www.gnu.org/software/pth/.

[7] Labrosse, J. J. 1998. Microc/OS-II (2nd ed.). R & D Books.

[8] Lawson, K., AtomThreads. http://atomthreads.com/.

[9] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. 2004. TinyOS: An operating system for wireless sensor networks. In Ambient Intelligence. Springer-Verlag.

[10] Nacul, A.C., Givargis, T. 2005. Lightweight Multitasking Support for Embedded Systems using the Phantom Serializing Compiler. DATA '05.

[11] Samek, M. and Ward, R. 2006. "Build a Super Simple Tasker", Embedded Systems Design.

[12] Vahid, F., Givargis, T., Miller, B. 2012. Programming Embedded Systems, An Introduction to Time-Oriented Programming. UniWorld Publishing, Lake Forest, CA.

[13] Quantum Leaps. http:///www.state machine.