Embedded Software Education: An RTOS-based Approach

James Archibald
Electrical and Computer Engineering Dept.
Brigham Young University
jka@ee.byu.edu

Doran Wilde
Electrical and Computer Engineering Dept.
Brigham Young University
wilde@ee.byu.edu

ABSTRACT
Embedded computer systems are proliferating, but the complexities of embedded software make it increasingly difficult to produce systems that are robust and reliable. These challenges increase as embedded systems are connected to networks and relied on to control or monitor physical processes in critical infrastructure. This paper describes a senior-level course that exposes students to foundational characteristics of embedded software, such as concurrency, synchronization and communication. The core of the class is a sequence of laboratory assignments in which students design and implement a real-time operating system. Each student-developed RTOS has the same API, so all can run the same application code, but internal implementations vary widely. The principal challenges that arise in the design and debugging of a multi-tasking RTOS tend to be instances of the general problems that arise in embedded software. In our experience, the activity of creating a working RTOS is effective in helping students acquire the knowledge and skills required to be successful embedded software developers.

Keywords
Embedded systems, embedded systems education, real-time operating systems, concurrency, real-time computing, software bugs

1. INTRODUCTION
Mobile and embedded devices are commonplace in modern society, and embedded systems are increasingly deployed to support critical infrastructure. Consumers naturally expect their embedded devices to operate reliably; embedded systems that provide essential functionality in an airplane or a municipal water system will be held to the highest standards of dependable operation. Unfortunately, reports of vulnerabilities or outright failures in critical embedded systems are increasing in frequency and attracting the attention of the general public.

Under the best of circumstances, the creation of functional software for non-trivial applications is demanding. Virtually no substantive software programs have been found to be error-free when subjected to careful analysis [1]. Embedded developers face all the challenges of conventional software and more. For example, embedded software typically must interact directly with hardware, it must respond to time-critical events in specified time windows, and it often relies on concurrency (processes, threads, interrupts) to meet response time requirements.

Given the formidable task facing firmware developers, it is not surprising that the creation of embedded software is quantifiably more demanding than conventional software [8], nor is it surprising that the same kinds of mistakes turn up repeatedly during design and development [24]. Given the importance of embedded systems, there is considerable interest among researchers and practitioners in improving the efficiency with which embedded software is created. Efforts focus on a wide range of alternatives, ranging from improved tools and languages to optimized testing methodologies. These approaches may well improve productivity and reduce error rates, but it is highly unlikely that they will eliminate bugs in embedded software.

An undergraduate education that prepares students to become embedded system developers should expose students to the fundamental challenges of embedded software, including the use of typical constructs and related problems that too-often accompany their use. As practicing professionals, graduates should understand the underlying issues so they can avoid problems in their own designs, and so they know the most likely causes of behavioral anomalies in their systems during development and testing. It is widely accepted that students learn best about embedded systems through their own hands-on experience [10] [13] [18] [14].

Within the dynamically evolving academic disciplines related to embedded systems, there is constant pressure to create new classes and to expand coverage of emerging topics, but changes are limited by faculty resources and the number of course that can be packed into an undergraduate curriculum [25]. In practice, the well founded desire to expose each student to a broad spectrum of technical topics limits the extent to which they can focus on specializations within their major discipline. As a result, many students will end up working in a sub-discipline in which they have taken just one advanced undergraduate class. Educators thus fre-

This work is based on an earlier work: Preparing Students for Embedded Software Development: An RTOS-based Approach. In Workshop on Embedded and Cyber-Physical Systems Education, WESE, © ACM, 2015.
http://dx.doi.org/10.1145/2832920.2832926.

quently face this challenge (as we have): if our graduates will go into the workplace as embedded software developers having taken just one class with that focus, what is the best content we can provide in that course to prepare them?

This paper describes our answer to that question: a senior-level course in which students work with a partner to design and implement a *real-time operating system* (RTOS) or *kernel* from the ground up. In a sequence of labs over the course of the semester, students create interrupt service routines, then core kernel routines that perform context switching, and then other kernel functions that support synchronization and communication between concurrent tasks. A final lab focuses on writing code for a time-critical application that uses their RTOS. A central thesis of this paper is that the completion of this project sequence gives students critical insight into and experience with the fundamental challenges inherent in embedded software.

In the remainder of this paper, we discuss a sampling of related work from the literature, we explore the challenges of embedded software, we describe the RTOS-based course that has been in place in our curriculum for over a decade, and we consider those aspects of the course that are most responsible for its success.

2. RELATED WORK

A survey of the literature on embedded system education confirms that a wide variety of approaches have been taken, depending on the technical nature of the associated degree program, whether the courses are graduate or undergraduate, and the mix of other (non-embedded) courses in the curriculum [11]. We focus here on papers that make specific mention of a course that involves the design or use of an RTOS, or that includes very similar content.

Chen *et al.* describe a set of six short courses on embedded systems intended to address deficiencies in first-year graduate students [7]. One of those courses focuses on kernel implementation issues, including context switching, scheduling, message passing, and interrupt service routines. Students study the mechanisms of an embedded operating system so that they understand the services it provides, but they do not implement an RTOS. The authors note that it requires "sophisticated programming skills to develop a kernel."

Hsu and Liu describe an introductory course taught in the sixth semester that is founded on a project-based learning strategy [9]. The primary objective of the students was to create application-specific code that ran with the μC/OS kernel, but students were unsuccessful in porting the RTOS to their target platform so the multi-tasking application code never ran as intended.

Jamieson describes a project-based class in which students prototype and design three embedded systems using their choice of implementation platforms [12]. RTOS coverage is one of two key components that is explicitly mentioned as missing in their approach.

Sangiovanni-Vincentelli and Pinto describe the embedded system curriculum in the EECS department at UC Berkeley, which they note was shaped by their research projects [21] [22]. They describe an advanced graduate class that focuses on RTOS concepts, including scheduling and communication. Among other assignments in the class, students implement a simple RTOS.

Marwedel describes an introductory course in embedded systems for students from multiple disciplines that has been fine-tuned over a period of almost ten years [19]. Relevant topics covered in the class include scheduling, operating systems for embedded systems, and typical properties of an RTOS.

Koopman *et al.* describe embedded system education for undergraduates at Carnegie Mellon University. Students in their first course develop application software for microcontrollers running an RTOS. In a later class, students more fully explore the capabilities of an RTOS in supporting real-time software. In class labs, students implement an embedded RTOS and then write application code that runs on their OS. Topics addressed in the class include concurrency, preemption, real-time synchronization and communication.

Sztipanovits *et al.* describe the embedded curriculum at Vanderbilt University. They note the difficulty in a mid-sized engineering school of creating new classes for new areas of specialization. They describe an undergraduate course on real-time systems that emphasizes design, implementation, and theoretical foundations, and that covers many of the same topics as the course we describe in this paper.

Caspi *et al.* discuss guidelines for a graduate curriculum on embedded software. One area of foundational knowledge they identify is real-time computing, including design, validation and operating systems. They observe that classes with a real-time focus are among the most commonly taught in existing embedded systems curricula.

Ping describes five main areas of emphasis that embedded systems curricula should address [20], one of which is operating systems for embedded systems. The author states that students should master the basic concepts of an RTOS and know enough to port an existing RTOS to a new target.

Wolf and Madsen discuss key factors in education about embedded computing [26]. They observe that, outside embedded system classes, students have virtually no experience with raw CPUs, where they are "free from operating system restrictions and with full access to the halt button." One of their major topics to address in embedded systems education is concurrency; students need to understand why embedded systems use concurrent software, how it is created by the hardware and software, and how it complicates debugging and performance analysis.

Zhang *et al.* present a framework for organizing an embedded system that emphasizes mastery of skills through practice [27]. One of their stated goals is the mastery of at least one embedded operating system. They observe that students are not highly motivated when *studying* an existing embedded system. In contrast, we find students are quite motivated when they are *creating* an embedded system.

3. FIRMWARE CHALLENGES

As noted above, firmware developers face all the challenges inherent in generating conventional software, in addition to specific difficulties that arise because of the nature of embedded systems. In addition to those factors noted in the previous section, embedded system developers must deal with limited visibility into their systems during execution (making it difficult to see what the system is doing), limited memory and processing resources that often prohibit detailed logging of actions and events (making it difficult to reconstruct what happened in a post-mortem analysis), and the infeasibility of exhaustively testing all system inputs and event timings (making it difficult to ensure truly reliable operation).

Students are likely to gain crucial insight into many of these challenges while completing any project that involves the development and testing of a non-trivial embedded system. Not all projects, however, will give students experience with the issues underlying the more elusive and frustrating bugs in embedded software. In some cases, important operational details may be hidden by the programming environment or tools; in others, the system requirements may be met with relatively simple software constructs. (Of course, systems may also be too complicated, frustrating students and making it nearly impossible for them to make a good design and see it through to a working system.)

We feel that an important metric for evaluating class projects is the set of issues and challenges that students are exposed to as they complete the assignments. For example, we have found it helpful to structure our lab assignments so that students *experience* rather than merely *read about* the most common, insidious bugs in embedded software. Obviously, there is a fine line here: we don't want to set the students up to fail, but if the project is rich enough that the bugs in question *can* occur (and *do* occur for some in the class), then the entire class comes to realize that they are not immune and that the issues we talk about are important and relevant. From that point, class discussions on those topics tend to fully engage everyone present in the classroom.

3.1 Common Software Bugs

To highlight the strengths of our RTOS-based embedded software course, consider how it matches up with Michael Barr's top ten list of causes of "nasty embedded software bugs," published in two articles in 2010 [4] [3]. Barr's articles are based on years of professional experience and worthwhile reading for any embedded system developer. We summarize his list and briefly discuss each item.

1. **Race conditions**. These arise—typically in code that writes shared variables—when the resulting state depends on the specific interleaving of instructions from two or more tasks, or threads of execution within an embedded system.

2. **Non-reentrant function.**. A function is reentrant if execution can safely switch from one task running that function to another task that also calls the same function. Non-reentrant functions are a special case of a race condition.

3. **Missing `volatile` keyword**. This keyword instructs the compiler that it cannot optimize memory reads or writes to the specified variable because it may be changed at any point by code executing concurrently (another task or an interrupt routine).

4. **Stack overflow**. This occurs in situations when the programmer must specify the size of the stack associated with a task and it is made too small; whatever happens to be in memory just beyond the stack gets overwritten.

5. **Heap fragmentation**. In systems with dynamically allocated memory, a point can be reached where the heap (the available memory pool) has enough memory to satisfy the next request, but the memory is unfortunately divided into a set of smaller fragments.

6. **Memory leaks**. These happen in systems without garbage collection (the automatic reclamation of unused memory) when blocks of memory are dynamically allocated and used, but not consistently returned to the heap.

7. **Deadlock**. This condition arises when a set of two or more tasks in a system are all blocked, with each waiting for another task in the set to execute first, and with a circular chain of dependences between the tasks.

8. **Priority inversion**. This occurs if a high-priority task blocks on a resource held by a lower-priority task, and then execution switches to a lengthy task with medium priority, causing the high-priority task to miss a deadline.

9. **Incorrect priority assignment**. This can happen when task priorities are assigned on an *ad hoc* basis, rather than using the results of careful analysis ensuring that all task deadlines will be met regardless of event ordering.

10. **Jitter**. This term refers to timing variations in the execution of tasks expected to run at fixed intervals. These variations occur because the processor is executing code associated with interrupts or other tasks.

3.2 Implications for Student Projects

From the wide range of possible student projects, consider those where the issues of the previous section could arise. In the simplest of embedded systems, with a single execution thread (based on round-robin scheduling or a polled loop with no interrupts), students could encounter only those bugs related to dynamic memory allocation. The majority of the bugs on the list arise only in code with multiple execution threads and concurrency.

Developers turn to concurrent software for systems that require both timely responses to events and lengthy computation. For example, code that responds to a critical event can be placed in a thread with higher priority than the thread containing the code performing the lengthy computation. If the system has a low-overhead mechanism to switch from the current thread to any thread of higher priority as soon the latter is ready to run, then the designer can maintain reasonable levels of processor utilization while still hitting response time targets.

The hardware interrupts supported in most microprocessors provide one such mechanism: in effect the entire class of interrupt code preempts non-interrupt code. More flexibility and control results when preemption is extended to non-interrupt code; the programmer can define multiple tasks, assign each a distinct priority, and then rely on the runtime system to preempt lower-priority tasks when higher-priority tasks become ready to run. This is the execution model provided by a real-time operating system (RTOS). The programmer defines a set of tasks, assigns each a priority, and then relies on the runtime system to run the highest priority ready task until that task completes or is preempted. A typical RTOS provides a set of functions to manage tasks and to allow them to communicate and synchronize their actions.

Many embedded systems are constructed using an RTOS, particularly those intended for platforms with relatively little memory (tens of kilobytes). Dozens of RTOS versions are available from commercial vendors, varying in cost and and available libraries. Developers who use an RTOS are very aware of all issues on Barr's top ten list: at least eight of the ten must be addressed in the normal course of any design with multiple tasks and preemption.

Student projects involving the writing and debugging of application code to run with an existing RTOS could therefore be effective in educating students about the issues from Barr's list. So why build an RTOS? We are certain that both application-level and system-level approaches can be effective, but we feel that the latter offers important advantages that educators often overlook. (As noted in a previous section, undergraduate courses that focus on RTOS implementation are rare.)

There are notable benefits for the students in building a system from the ground up; it removes the mysteries, shows how the pieces fit together, and makes visible all the operations within the system. There is a natural tendency in complex systems to hide more of the implementation and operational details under the surface by having them addressed automatically by development and runtime tools, allowing designers to focus on critical higher-level issues. This can increase productivity, but it can be counterproductive in an educational setting. In particular, it makes it more difficult to understand the details of the overall system operation. (Imperfections in complex tools—think optimizing compilers or complex digital design suites—can also make it difficult for users to achieve desired levels of optimization without reasonable familiarity with the internal operation of those tools.)

In our approach, we have elected to simplify the tools and make virtually everything in the system visible to the students. They see the operational details, down to the precise sequence of machine instructions executed. At this level, there are no mysteries about what library functions do, how the interrupt mechanism works, how contexts are saved and restored, or how tasks block and then are made ready again when their requested resources become available. We claim that important insights come to the students when they interact with the system at this level.

Of course, educators must ensure that any assigned project sequence is of an appropriate size and scope for the class in which it is assigned. In this regard, we know from experience that the design and implementation of an RTOS is a good match with the workload of a 4 credit hour semester-long class.

4. THE RTOS-BASED COURSE

When it was first taught, the course focused on understanding and using an existing RTOS. We used the μC/OS kernel created by Labrosse [16] because the source code was readily available and the kernel could be used without fee for academic purposes. Laboratory assignments focused on analyzing μC/OS source code, understanding and augmenting kernel functions, and creating compatible application code. Unfortunately, after teaching the class with this focus a few times, we concluded that these assignments did not lead to a thorough understanding of the RTOS or to the ability to create reliable, time-critical application code for embedded systems.

At that point, we decided to restructure the class around the design and implementation of a real-time operating system, rather than studying and using an existing kernel. We reasoned that students would more fully internalize the principles of embedded software if they designed and implemented the system software in its entirety. We developed custom tools for our target architecture, initially a modified version of the MIPS instruction set the students had seen in an earlier course. The tools included a compiler, assembler, and a detailed simulator that included breakpoints, single-step execution, and extensive debugging support.

The course has undergone considerable evolution over the 15 year period in which it has been taught. At the core of the class today is a sequence of eight laboratory assignments that result in a fully functional RTOS and application software [2]. Students work in teams of two to create kernels that conform to an application program interface (API) defined for the class RTOS to ensure that kernels can run the same application code. As will be described in the next section, each lab includes application code that each RTOS must run correctly without modification. The common API facilitates class discussions about alternative approaches and tradeoffs without overlying constraining student designs. Within the constraints of the API, there is significant latitude in internal organization and implementation.

The emphasis on creating an RTOS is reflected in the lectures, homework assignments, and exams. Class discussions cover material from the class text and supplemental material that is essential for the labs. Class exams cover general real-time system issues and details specific to the class RTOS.

Students come to this project with different backgounds and abilities, a recognized problem in embedded system courses [17]. We address this by our choice of text, class discussions that review essential background material, carefully selected initial homework and lab assignments, and an assortment of useful supplemental material on the course webpage. The reading and discussions are structured to provide students with essential background information shortly before they need it in their designs, similar in principle to the stepwise spiral approach advocated by Jing *et al.* [13]. It is critical at every step along the way that students see the big picture (the overall objective and context) reasonably clearly.

At present, the target architecture is the Intel 8086, a platform for which a wide range of development tools are readily available. The 8086 was chosen to reduce the learning curve for students already familiar with the Intel IA32 architecture from a prerequisite course using the book and innovative labs of Bryant and O'Hallaron [5]. In the class, we rely primarily on a simulator to provide the environment for development and testing. For several years we used both a detailed simulator and an 8086 hardware development board, but in recent years have used the simulator exclusively. (This had more to do with limits on space and aging hardware than pedagogy; we plan to use both simulator and hardware in the future after a switch to a new target architecture.)

A deterministic simulator offers several advantages that have proven to be critical in the success of the class. First, the simulator permits time to be frozen and the system state to be observed without affecting the execution sequence. In contrast, software execution on embedded hardware is difficult to observe, as added debug code can change the timing and system behavior making it very difficult to track down certain software bugs. With the simulator, students can pause time and observe the detailed actions of their code, and repeated runs produce the same outcome, making it easier to identify and correct behavioral anomalies.

Secondly, the simulation approach avoids problems that inevitably arise in labs using real hardware. In our experience, despite reasonable efforts by staff to maintain lab infrastructure, students often encounter problems such as damaged boards, bad connectors, or boards left with improper settings by previous users. For a few unlucky students, these problems can significantly increase the time to debug their systems. Finally, because our simulator is readily available on a variety of platforms, it is convenient to use and does not require dedicated infrastructure.

We note that others have advocated the use of simulators in embedded system courses for a variety of additional reasons. These include allowing software development to begin before custom hardware is available, reducing costs and space requirements, and abstracting away hardware details that get in the way of the intended learning experience [18]. Others have observed that simulators are well suited for development environments in which RTOS concepts are taught [6]. In our experience, the ideal development environment for teaching purposes includes both a highly functional and detailed simulator and a real hardware target.

It can be challenging to find a text that supports a project-oriented course, but we have had good success in using Simon's book, an RTOS-oriented introduction to the field of embedded software [23]. Without limiting discussion to a single RTOS, Simon addresses most critical issues that arise in the creation and use of a kernel. The text is very accessible, and the topics it treats match the needs of our class labs surprisingly well. The book even includes a CD with the source code to μC/OS which students can consult for additional insight if they choose.

Our course includes feedback mechanisms that help us make regular improvements. For example, each lab submission must include a short report of problems encountered and the time spent completing the lab. Problem summaries sometimes alert us to issues with tools or documentation, but typically they report errors in the students' design or code. Edited error summaries are posted with each lab to give future teams with matching symptoms ideas about possible causes. High, low, and average times required for each lab are computed each semester and posted on the lab webpage. This helps students plan and alerts them when their time investment becomes excessive. Instructors use this information to identify possible improvements and to evaluate the impact of changes to the lab assignments.

Near the end of the semester, students measure and report certain characteristics of their kernels, including total size of source code and executable, longest critical section, and the worst case overhead to release a semaphore. Encouragingly, their measurements show that the student kernels are generally getting smaller, faster, and more efficient. We believe that this is a result of ongoing improvements to the course.

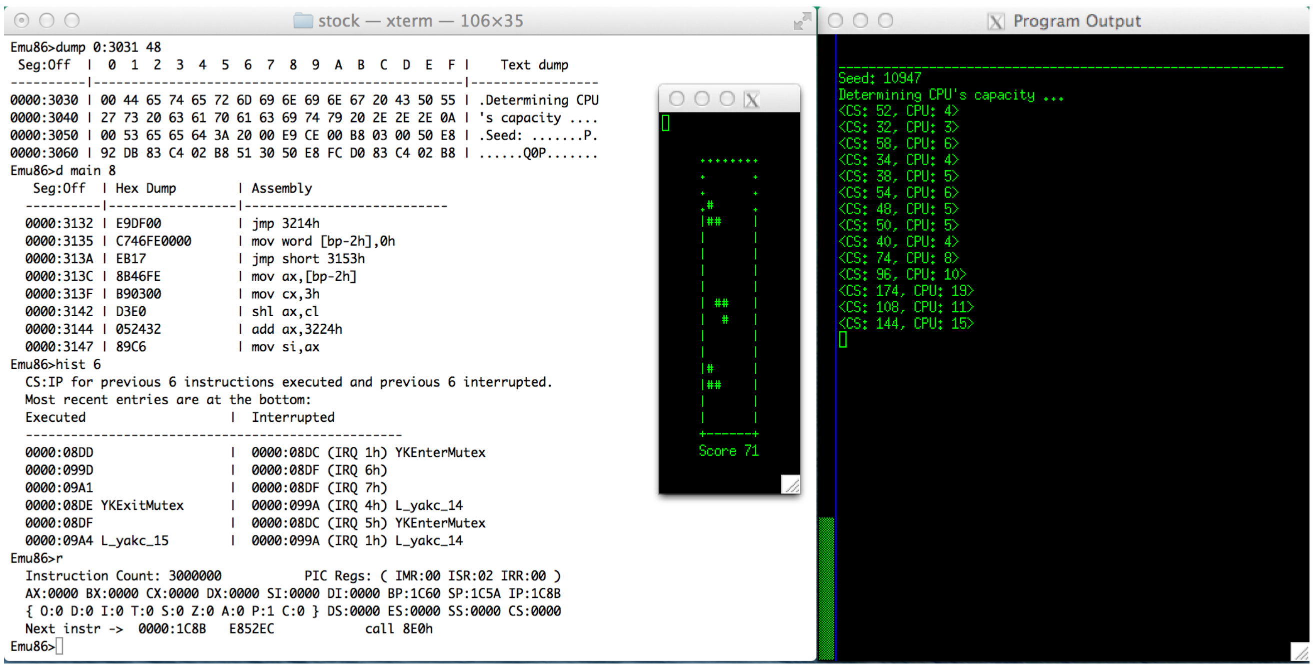
4.1 Software Tools

The choice of software tools is important in any embedded system project. For example, an ideal simulator would run the same binaries as the hardware, and it would accurately reflect an environment requiring time-critical code. We elected to create much of our own software, in part because of prior experience with architectural simulators and compilers, and in part because it appeared to be the best way to get the functionality we wanted.

Our compiler, assembler, and simulator are written in C and run on Windows, Linux, and OS X machines. 8086 assembly code is generated by a modified version of C86, a publicly available ANSI C compiler. The compiler offers simple inline-assembly functionality with no syntactic or semantic checking. Because flawed inline assembly can lead to particularly perplexing errors, students are required to write and maintain separate C and assembly files. Assembly functions follow compiler conventions so they can call and be called by functions written in C. To convert assembly files to an executable, we use the Netwide Assembler (NASM), a free, portable assembler. Since a portion of each kernel must be written in assembly—most notably interrupt service routines (ISRs), the code to save and restore contexts, and the code to dispatch tasks—it is important that the assembler be user-friendly. NASM supports straightforward syntax and directives.

The Emu86 simulator is the most crucial of the software tools that support our class. The simulator is an 8086 emulator with a textual interface to which a variety of useful debugging functions have been added. A wide range of capabilities are offered:

- Program binaries can be loaded into the simulated memory.
- Program input and output are tied to a separate con-



```
Emu86>dump 0:3031 48
 Seg:Off  |  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F |    Text dump
----------|-------------------------------------------------|-----------------
0000:3030 | 00 44 65 74 65 72 6D 69 6E 69 6E 67 20 43 50 55 | .Determining CPU
0000:3040 | 27 73 20 63 61 70 61 63 69 74 79 20 2E 2E 2E 0A | 's capacity ....
0000:3050 | 00 53 65 65 64 3A 20 00 E9 CE 00 B8 03 00 50 E8 | .Seed: .......P.
0000:3060 | 92 DB 83 C4 02 B8 51 30 50 E8 FC D0 83 C4 02 B8 | ......Q0P.......
Emu86>d main 8
   Seg:Off  | Hex Dump         | Assembly
  ----------|-----------------|---------------------------
  0000:3132 | E9DF00           | jmp 3214h
  0000:3135 | C746FE0000       | mov word [bp-2h],0h
  0000:313A | EB17             | jmp short 3153h
  0000:313C | 8B46FE           | mov ax,[bp-2h]
  0000:313F | B90300           | mov cx,3h
  0000:3142 | D3E0             | shl ax,cl
  0000:3144 | 052432           | add ax,3224h
  0000:3147 | 89C6             | mov si,ax
Emu86>hist 6
  CS:IP for previous 6 instructions executed and previous 6 interrupted.
  Most recent entries are at the bottom:
  Executed                   |  Interrupted
  -----------------------------------------------
  0000:08DD                  |  0000:08DC (IRQ 1h) YKEnterMutex
  0000:099D                  |  0000:08DF (IRQ 6h)
  0000:09A1                  |  0000:08DF (IRQ 7h)
  0000:08DE YKExitMutex      |  0000:099A (IRQ 4h) L_yakc_14
  0000:08DF                  |  0000:08DC (IRQ 5h) YKEnterMutex
  0000:09A4 L_yakc_15        |  0000:099A (IRQ 1h) L_yakc_14
Emu86>r
  Instruction Count: 3000000          PIC Regs: ( IMR:00 ISR:02 IRR:00 )
  AX:0000 BX:0000 CX:0000 DX:0000 SI:0000 DI:0000 BP:1C60 SP:1C5A IP:1C8B
  { O:0 D:0 I:0 T:0 S:0 Z:0 A:0 P:1 C:0 } DS:0000 ES:0000 SS:0000 CS:0000
  Next instr ->  0000:1C8B   E852EC            call 8E0h
Emu86>
```

```
------------------------------------------------------------
Seed: 10947
Determining CPU's capacity ...
<CS: 52, CPU: 4>
<CS: 32, CPU: 3>
<CS: 58, CPU: 6>
<CS: 34, CPU: 4>
<CS: 38, CPU: 5>
<CS: 54, CPU: 6>
<CS: 48, CPU: 5>
<CS: 50, CPU: 5>
<CS: 40, CPU: 4>
<CS: 74, CPU: 8>
<CS: 96, CPU: 10>
<CS: 174, CPU: 19>
<CS: 108, CPU: 11>
<CS: 144, CPU: 15>
```

Figure 1: A screen shot of the simulator.

sole window.

- Breakpoints can be set on arbitrary instructions in the executable.
- Programs can be executed to completion, to a breakpoint, or for any fixed number of instructions.
- The contents of registers can be viewed and set to arbitrary values.
- Memory contents can be viewed as numerical values or disassembled instructions, and set to arbitrary values.
- Lists of the most recently executed instructions and most recently interrupted instructions can be viewed.
- Interrupts can be manually asserted at any point in the program.
- Addresses of variables and functions can be read from the symbol table.
- Breakpoint monitors can be set to stop execution when a register or memory location is accessed, modified, or reaches a particular value.

As can be seen, the simulator offers features beyond those of conventional debuggers. Most importantly, the simulator offers the possibility of not just stopping execution, but time itself. For students confused by execution involving interrupts, instruction and interrupt histories prove invaluable. Thorough testing of interrupt nesting and critical sections is made possible by the ability to assert an interrupt manually at any point. Breakpoint monitors can watch arbitrary regions of memory rather than single variables. Powerful breakpoint conditions make it easy to find the instruction that writes a particular value to a given memory location—a useful capability for stack overruns and problems with pointers.

Figure 1 shows the interface for the simulator. The console window on the left shows the results of user commands to display (dump) the contents of a range of memory locations, to disassemble some instructions following the label *main*, to show the history of the instructions most recently executed and interrupted, and to display current register contents. The other two windows show the output generated by the simulated system.

With any simulator, the challenge must be addressed of how to represent the physical world with which the system interacts. Given our software focus, we use a simplified physical model in which essential interrupts are tied to the user's keyboard. The highest priority interrupt, a system reset, is asserted when Ctrl+R is pressed. The interrupt associated with the system heartbeat timer is generated automatically at fixed intervals (10,000 instructions by default) and manually by pressing Ctrl+T. Any other key press causes the key value to be stored in a dedicated memory location and a keyboard interrupt to be asserted.

The simulator also includes built-in support for Simptris, a simplified version of a Tetris-like game. Application software written in the final lab plays the game by calling functions to move and rotate pieces and by responding to interrupts that signal changes in the game state. Each call to move or rotate a piece incurs fixed communication overhead—a function cannot be called until a signal is received indicating that the previous placement command finished. Since pieces appear and fall at an increasing rate, even carefully crafted code will eventually fail to place a piece as desired and the game will end. The number of lines cleared is therefore an indication of the overhead and inefficiencies in both

Function	Description
void YKInitialize(void)	Perform all kernel initialization
void YKEnterMutex(void)	Disable interrupts
void YKExitMutex(void)	Enable interrupts
void YKIdleTask(void)	Lowest priority background task
void YKNewTask(void (* task)(void), void *taskStack, unsigned char priority)	Create new task
void YKRun(void)	Start the RTOS
void YKDelayTask(unsigned count)	Block task for *count* ticks
void YKEnterISR(void)	Record ISR entry
void YKExitISR(void)	Record ISR completion
void YKScheduler(void)	Pick the next task to run
void YKDispatcher(void)	Cause the next task to run
void YKTickHandler(void)	Decrement count for delayed tasks
YKSEM* YKSemCreate(int initialValue)	Create a semaphore
void YKSemPend(YKSEM *semaphore)	Obtain a semaphore
void YKSemPost(YKSEM *semaphore)	Release a semaphore
YKQ *YKQCreate(void **start, unsigned size)	Create a message queue
void *YKQPend(YKQ *queue)	Get message from queue
int YKQPost(YKQ *queue, void *msg)	Put message in queue
YKEVENT *YKEventCreate(unsigned initialValue)	Create an event group
unsigned YKEventPend(YKEVENT *event, unsigned eventMask, int waitMode)	Wait for event flag
void YKEventSet(YKEVENT *event, unsigned eventMask)	Set an event flag
void YKEventReset(YKEVENT *event, unsigned eventMask)	Clear an event flag

Table 1: Required Kernel Functions

the application code and the RTOS. To ensure that the primary focus is on software efficiency rather than playing intelligence, the game is reduced to a 6x16 playing area with just two distinct pieces. The screen capture in Figure 1 includes the two windows displayed in Simptris mode. The small window in the center shows the current game state as pieces fall and touch bottom, and the larger window on the right shows the output of the system as the application code executes.

4.2 The Kernel

The complete specification for the kernel includes prototypes for twenty-two functions and a detailed discussion of their required behavior. The required kernel functions are summarized in Table 1. (The prefix for the function names stems from the name of the kernel: YAK, originally an acronym for *Yet Another Kernel*. The naming strategy of kernel functions makes it easy to identify calls to kernel functions in the application source code.)

The RTOS supports application code consisting of distinct tasks with separate stacks and unique, static priorities. Each task is either ready to run or blocked, in which case it will be made ready by the kernel when sufficient time passes (if the task delayed itself) or when the resource requested by the task (e.g., semaphore, message) becomes available. Once running, a task continues to execute until it blocks or until it is preempted by a higher priority task.

Application code includes ISRs that execute when an interrupt is both enabled and asserted. Each ISR must call specific RTOS functions on entry and exit that allow the kernel to track the interrupt nesting level and to distinguish between function calls from task and interrupt code.

The RTOS requires specific rules to be followed when initializing the kernel and starting the application code. It is assumed, for example, that user code is the first to execute on reset. This code must call functions to initialize the kernel, create and initialize tasks, and begin task execution. The code also typically creates RTOS resources used by the application, such as semaphores and queues.

The most fundamental components of the RTOS are the scheduler, which selects the highest priority ready task, and the dispatcher, which loads the saved context of a task and transfers control to it. Other functions allow tasks to delay themselves for a specific number of system clock ticks, to use semaphores for synchronization and mutual exclusion, and to communicate using message queues. An attempt to obtain a semaphore that is not available will cause the caller to block, and releasing a semaphore will cause the highest-priority task blocked on the semaphore to be made ready. User code must allocate space for each message queue, but the RTOS manages the queue itself so that a task requesting a message can block if the queue is empty and be unblocked when a message is written to the queue.

4.3 Lab Assignments

The lab sequence begins with an assignment to write, compile, and run code using the tools. Students write a simple function in assembly language that is called from a C program. The function computes an expression involving parameter values and a global variable and returns the result. In the second lab assignment, students use the simulator and its debugging capabilities to answer a variety of questions about a program and its execution. For example, they must determine the memory address associated with a global variable and with the first instruction of a function. More challenging questions ask for the maximum amount of memory used by the stack and the exact memory location of a local variable within the second call to a recursive function. After completing these first labs, students have been exposed to the tool chain, the simulator's unusual debugging features, the target machine's instruction set, and stack frame and function call conventions.

In the third lab assignment, students write ISRs in assembly for the three basic interrupts in the simulator. Each ISR saves the register context, calls a C function (an interrupt handler) to respond to the interrupt, restores the context, and returns. Interrupt code must reliably support nested interrupts—tested by delaying the keyboard handler for a certain key long enough to ensure that a timer tick occurs while still in the keyboard ISR. In completing this lab, students come to understand the interrupt mechanism, how to save and restore state consistently, and how to initialize the interrupt vector table.

In labs four through seven, students design and implement portions of the RTOS. These labs begin with an assignment to complete a detailed design, complete with pseudocode and a specification of the data structures that will be used. Students must also submit responses to a series of questions about how a variety of issues will be addressed in their implementation. TAs and the instructor read their submissions and provide feedback that includes alerts to aspects of their designs that might prove problematic. Naturally, it is difficult to tell from their descriptions how everything in their designs will work, but the assignment gets them thinking about the important issues, even if they do not yet appreciate the consequences of the various design choices that they make. The design step is critical; in general, students who complete a thorough design spend less time implementing and debugging their kernels.

With the design complete, students begin to implement their RTOS. Each lab builds on the previous and includes application code that tests the new functions and that must execute correctly on the student RTOS, even with arbitrary key presses and increased timer tick frequency (modifiable from the command line in the simulator). The simplest application code creates a single task that, in turn, creates a low priority task and a high priority task. Once the highest priority task is created, it begins execution and never relinquishes control. Application code for later labs is more complex, exercising kernel functions that include support for semaphores, message queues, and event groups.

In the final lab, the focus shifts to writing application code. In the context of the simulator's Simptris game, students write interrupt code for each of the five additional interrupts used by the game and they create task code that can play the game. Students often employ a dedicated task that decides where to place each piece and a second task that makes the function calls to move pieces while dealing with the communication delays. Typical designs include a semaphore (blocking the second task until a previous call completes) and a message queue (communicating move information between tasks).

Application code for the lab varies widely, but it generally uses a significant subset of the RTOS. For full credit, student code must clear a specified minimum number of lines in the game. Kernel inefficiencies can be a limiting factor, but the required threshold can be reached without highly optimized RTOS functions or sophisticated placement algorithms.

Category	Score	Avg.	Scale
Overall course rating	7.0	6.5	0-8
Amount learned	6.8	6.5	0-8
Materials and activities effective	6.9	6.3	0-8
Well organized	7.0	6.4	0-8
Intellectual skills developed	7.0	6.5	0-8
Good use of time outside class	83.3	74.6	0-100

Table 2: Student Evaluations

5. EVALUATION AND OBSERVATIONS

Student evaluations for the last completed offering of the class are summarized in Table 2. The average listed in the table is of all other classes in our department. These ratings place the class among the highest-rated courses in the department. In addition, the course is frequently cited as a favorite in exit interviews (at graduation). We note that the success rates in the class are impressive: of the roughly 750 students who have taken the class, just 11 failed to produce a working RTOS. We feel this is due to the effective infrastructure that exists within the course.

What do students actually learn in the course of designing and implementing an RTOS? Based on our observations, they have unusual insight into the causes of bugs on Barr's top ten list. For example, virtually every team must chase down at least one bug caused by a race condition or non-reentrant function—generally because they failed to see that a particular portion of their kernel code was a critical section and they failed to disable interrupts during its execution. Similarly, many teams experience stack overflow at least once when they create tasks and task stacks for their own application code. The overwriting of adjacent code or a nearby data structure can be a baffling bug, but the debugging capabilities of the simulator (the memory monitors in particular) make this relatively easy to track down.

The only entry on Barr's list that never comes up in the context of our class labs is heap fragmentation. When blocks of memory are needed dynamically (for a task control block or a message queue, for example) they are allocated by simple routines that allocate entries from a fixed size array of C structs. For virtually all other entries in Barr's list, the students come to understand what the real issues are: how the condition might arise, what is at stake, and how it can be avoided.

Of course, students learn a great deal about embedded systems beyond the causes of bugs on Barr's list. They are well positioned to develop application code for any RTOS, not just the one they created. Since they have written the context-switching code (to save and restore task contexts), they are keenly aware of the consequences of switching to another context at an inopportune time. This, in turn, helps them understand critical sections and the importance of disabling interrupts, using semaphores, or calling functions that lock the scheduler. Since they have implemented semaphores and message-passing primitives, they are aware of what happens behind the scenes when similar functions are called and they are aware of important questions to ask when learning about a new RTOS. (Examples: Does this semaphore provide priority inheritance? Is the next task to run the waiting task with highest priority or the task that has been waiting the longest?) Students are also sensitive to the overhead of RTOS operations and potential pitfalls in using and misusing kernel functions.

Students completing the class show increased confidence in working with embedded software. Over the years, we have observed an increase in follow-on student projects in which embedded software plays an important role, and in many of these projects the students have chosen to use an RTOS. For example, a recent senior project used a kernel on a custom FPGA board to support real-time, on-board vision processing for small autonomous vehicles. Reliable data are not available, but informal discussions with graduates indicate that the course has increased the number that seek and find employment in the area of embedded systems.

Although the course focuses on embedded systems, we believe that its most significant outcomes are not restricted to the embedded domain. Most students taking the class do not go on to become embedded system developers, but all of them become better programmers. They are more likely to pursue a thorough design before coding and less likely to use inefficient constructs. They are more likely to correctly manipulate low level data and less likely to be stumped by obscure software bugs. Students have increased confidence in their ability to create reliable systems because they successfully completed a challenging project. Students have a better understanding of system fundamentals, including task or process-level actions of the operating system and the interaction between hardware and software. These conclusions are supported by anecdotal evidence from students, alumni, and recruiters.

As with any class, there are costs in setting up the infrastructure, but once established, the overhead associated with teaching this course compares favorably to other classes that focus on computer systems. Given the critical role of the lab assignments, it is helpful if the instructor and TAs have previously created their own RTOS so they can consult their own code when questions come up. For example, student concerns early in the semester about the work required to write an operating system from scratch are reduced when the instructor announces that his kernel is a total of 955 lines of C code and 175 lines of assembly.

The overhead of evaluating lab submissions is far from excessive; students sit down with a TA and allow their code to be tested, and it is usually immediately obvious if it works correctly or not. Knowledgeable TAs are critical in the success of the class, however. Teams occasionally get stuck and need help. Because kernel implementations can vary widely, TAs and the instructor must be able to consider many different approaches and provide suggestions within each team's unique framework. A reasonable level of programming maturity in the students is essential, but the requirement to work in teams helps those students with less experience or lower confidence.

We have had few instances of plagiarism, or inappropriate reuse of code from others. The nature of the assignments makes it easy to maintain an archive from previous semesters that new submissions can be compared against. In practice, the nature of the assignments makes it unlikely that an internet search will produce code that can be used without substantial modifications. Moreover, the text for our class comes with a CD that includes the full source code of μC/OS that students can study if desired. Although the overall operation of μC/OS is similar to our RTOS, there is very little code that can be taken from that kernel and used without significant modification.

In any technical class, the course content must be updated periodically. We have made many refinements over the years and will continue to do so in the future. We are currently working to revise the class projects to use an inexpensive microcontroller as the target platform. We are intentionally choosing a platform with significant memory constraints to emphasize the advantages of an RTOS and to allow students to purchase their own hardware if desired. We are currently identifying the target platform and development tools we will use. We plan to use both a hardware target and a binary-compatible simulator. We hope to add an additional lab (creating a monitor or remote debugger) that gives students additional exposure to system-level behavior.

Overall, we are in complete agreement with Koopman *et al.* that "students learn more effectively when motivated by exciting course projects," and that students learn better "when working through actual implementations in realistic environments that force them to confront the very real limitations and quirks of embedded systems" [15].

6. CONCLUSIONS

In this paper, we have described a senior-level course focused on embedded software development in which students design and implement a real-time operating system (RTOS). This course gives students significant experience with the constructs in embedded software that are typically associated with the most troublesome bugs that developers encounter. Students develop practical skills in developing time-critical code, but the impact of the class in our program goes far beyond preparing students for employment in the area of embedded systems.

The core focus of the class—developing embedded software from the ground up—has proven to be very effective for our students and to be well suited for a single semester undergraduate course. The class is an enjoyable and fulfilling class to teach, to take, or to serve as a TA for. Because it provides insight into such issues as system operation, multithreading, synchronization, and execution efficiency, recruiters like students to take this class to be considered for a variety of positions related to computer systems. We feel that students at other universities would be well served by a similar course.

7. REFERENCES

[1] Coverity: Open source code has fewer defects. *linux-magazine.com*, April 2014.

[2] J. K. Archibald and W. S. Fife. A course in real-time embedded software. *Computer Science Education*, 17(2):97–106, June 2007.

[3] M. Barr. Five more top causes of nasty embedded software bugs. *www.eetimes.com*, November 2010.

[4] M. Barr. Five top causes of nasty embedded software bugs. *www.eetimes.com*, April 2010.

[5] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2nd edition, 2011.

[6] P. Caspi, A. Sangiovanni-Vincentelli, L. Almeida, A. Benveniste, B. Bouyssounouse, G. Buttazzo, I. Crnkovic, W. Damm, J. Engblom, G. Folher, et al. Guidelines for a graduate curriculum on embedded software and systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(3):587–611, 2005.

[7] J. Chen, H.-M. Su, and J.-H. Liu. A curriculum design on embedded system education for first-year graduate students. In *Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–6. IEEE, 2007.

[8] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, pages 42–52, April 2009.

[9] R. C. Hsu and W.-C. Liu. Project based learning as a pedagogical tool for embedded system education. In *Information Technology: Research and Education, 2005. ITRE 2005. 3rd International Conference on*, pages 362–366. IEEE, 2005.

[10] S. Hussmann and D. Jensen. Crazy car race contest: Multicourse design curricula in embedded system design. *Education, IEEE Transactions on*, 50(1):61–67, 2007.

[11] D. J. Jackson and P. Caspi. Embedded systems education: future directions, initiatives, and cooperation. *ACM SIGBED Review*, 2(4):1–4, 2005.

[12] P. Jamieson. Arduino for teaching embedded systems. are computer scientists and engineering educators missing the boat? *Proc. FECS*, pages 289–294, 2010.

[13] L. Jing, Z. Cheng, J. Wang, and Y. Zhou. A spiral step-by-step educational method for cultivating competent embedded system engineers to meet industry demands. *Education, IEEE Transactions on*, 54(3):356–365, 2011.

[14] I. Kastelan, M. Barak, V. Sruk, M. Anastassova, and M. Temerinac. An approach to the evaluation of embedded engineering study programs. In *Information & Communication Technology Electronics & Microelectronics (MIPRO), 2013 36th International Convention on*, pages 742–747. IEEE, 2013.

[15] P. Koopman, H. Choset, R. Gandhi, B. Krogh, D. Marculescu, P. Narasimhan, J. M. Paul, R. Rajkumar, D. Siewiorek, A. Smailagic, et al. Undergraduate embedded system education at carnegie mellon. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(3):500–528, 2005.

[16] J. J. Labrosse. *MicroC/OS-II: The real-time kernel*. CMP Books, 2nd edition, 2002.

[17] C.-S. Lee, J.-H. Su, K.-E. Lin, J.-H. Chang, and G.-H. Lin. A project-based laboratory for learning embedded system design with industry support. *Education, IEEE Transactions on*, 53(2):173–181, 2010.

[18] H. Lim, H. Yu, and T. Suh. Using virtual platform in embedded system education. *Computer Applications in Engineering Education*, 20(2):346–355, 2012.

[19] P. Marwedel. Towards laying common grounds for embedded system design education. *ACM SIGBED Review*, 2(4):25–28, 2005.

[20] W. Ping. Research on the embedded system teaching. In *Education Technology and Training, 2008. and 2008 International Workshop on Geoscience and Remote Sensing. ETT and GRS 2008. International Workshop on*, volume 1, pages 19–21. IEEE, 2008.

[21] A. L. Sangiovanni-Vincentelli and A. Pinto. Embedded system education: a new paradigm for engineering schools? *ACM SIGBED Review*, 2(4):5–14, 2005.

[22] A. L. Sangiovanni-Vincentelli and A. Pinto. An overview of embedded system design education at berkeley. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(3):472–499, 2005.

[23] D. E. Simon. *An Embedded Software Primer*. Addison-Wesley, 1999.

[24] D. B. Stewart. Twenty-five most common mistakes with real-time software development. *Tutorial at 2006 Embedded Systems Conference*, September 2006.

[25] J. Sztipanovits, G. Biswas, K. Frampton, A. Gokhale, L. Howard, G. Karsai, T. J. Koo, X. Koutsoukos, and D. C. Schmidt. Introducing embedded software and systems education and advanced learning technology in an engineering curriculum. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(3):549–568, 2005.

[26] W. Wolf and J. Madsen. Embedded systems education for the future. *Proceedings of the IEEE*, 88(1):23–30, 2000.

[27] Y. Zhang, Z. Wang, and L. Xu. A global curriculum design framework for embedded system education. In *Mechatronics and Embedded Systems and Applications (MESA), 2010 IEEE/ASME International Conference on*, pages 65–69. IEEE, 2010.