A Course in Real-time Embedded Software

J. K. Archibald[1*] & W. S. Fife[1]
[1]*Brigham Young University, USA*

Embedded systems are increasingly pervasive, and the creation of reliable controlling software offers unique challenges. Embedded software must interact directly with hardware, it must respond to events in a time-critical fashion, and it typically employs concurrency to meet response time requirements. This paper describes an innovative course that gives students in-depth exposure to the challenges of writing reliable, time-critical, concurrent code. Students design and implement a real-time operating system (RTOS), and they write application code that uses the RTOS they construct. Code development and debugging take place in a simulation environment that offers visibility into the system and strictly repeatable execution while maintaining hardware compatibility. We describe the structure of the class, the custom tools used, and the lab sequence that results in a functional RTOS. We discuss the development of the class and its impact on our students.

Keywords: Real-time systems; Embedded systems; RTOS; Kernel

---

*Corresponding author. Email: jka@byu.edu

Introduction

Given their extensive experience with such devices as cellular phones, digital cameras, and MP3 players, it is natural for students in computer-related majors to seek an understanding of embedded systems and to consider careers in their design and development. However, as Lee (2005) has noted, important technical challenges inherent in modern embedded systems are not well addressed by current computer science.

The most significant unmet challenges of embedded systems, according to Lee, are the need for predictable execution timing, the need to interact directly with specialized hardware, and the need to support concurrency without compromising reliability. He notes that popular programming languages employ abstractions that obscure timing, and that they lack support for the creation of hardware interfaces. While threaded programming models provide an abstraction of concurrency, Lee observes that threaded software is often unreliable. Lee argues that computer science must ultimately reinvent itself to allow the creation of embedded software that is guaranteed to meet tight timing constraints, particularly in networked environments with unpredictable workloads. Short of sweeping, discipline-wide changes, what can educators do to expose students to the unique challenges faced in the development of reliable embedded systems?

Our experience suggests that a well-designed course can do much to educate students about the challenges of time-critical code, of interfacing with hardware, and of creating reliable concurrent software. We describe a senior-level course in real-time embedded systems taken as a technical elective by students in computer science, computer engineering, and electrical engineering. Central to the class are labs in which students work in pairs to design, implement, and debug a fully functional real-time operating system (RTOS), or kernel, followed by the creation of application code that uses their kernel. A single course cannot address all of Lee's concerns, but more fundamental changes are likely to follow only after students are more broadly exposed to the special challenges of embedded software.

In the next section, we discuss some of the challenges and constraints we faced in creating a course that focuses on real-time embedded systems. Subsequent sections detail the development tools used in the class, the nature of the RTOS that students implement, and the sequence of lab assignments that result in a working kernel. We

then discuss the overall structure of the class, lessons learned in its evolution, and assess its overall impact in our curriculum.

Class Origin: Motivation and Challenges

Early Implementations

The class was created to expose students to the unique challenges of creating embedded software. As the functionality of devices such as cellular phones continues to increase, so too does the complexity of the controlling software and its associated development cost.

Initially the class focused on understanding and using an existing RTOS. We used the μC/OS kernel created by Labrosse (1992, 2002) because the source code is readily available and the kernel can be used without fee for academic purposes. Class assignments focused on analyzing μC/OS source code, understanding and augmenting kernel functions, and creating compatible application code. Despite completing these assignments, too few students acquired a sufficient understanding of the RTOS and its underlying principles to guide the creation of reliable, time-critical code of their own.

We elected to restructure the class around the creation and implementation of a kernel. By creating the system from the ground up, students were expected to more fully internalize the principles of embedded software. Moreover, we felt that the creation of a complete working system would empower the students and give them confidence in tackling future challenges.

Custom tools were created for the target architecture, a modified version of the MIPS instruction set introduced in a prerequisite class. Initially the tools included a custom simulator, the lcc compiler (see Fraser, 1995) retargeted to our instruction set, and a custom assembler. The simulator emulated sequential instruction execution and provided debugging support, such as breakpoints, single-step execution, and the ability to examine memory and register contents. An application program interface (API) was defined for the RTOS to ensure class-wide compatibility. Each lab included application code utilizing a subset of kernel functionality that had to be run correctly. The common API facilitated class discussions about alternative approaches and tradeoffs, but it allowed significant latitude in internal organization and implementation.

Current Implementation

The core of the class today is a sequence of eight labs that result in a functional RTOS and application software. This emphasis is reflected in the lectures, homework assignments, and exams. Class discussions cover material from the class text and supplemental material required for the labs. In addition to three weekly lecture sections with the instructor, the class includes two weekly recitation sections conducted by a teaching assistant (TA). In the recitations, information from lectures is reviewed, homework assignments are discussed, and students can discuss design and debugging challenges with the TA. Homework assignments are typically problems from the text, supplemented with questions about the target architecture and potential programming pitfalls in C and assembly language. Class exams cover general real-time system issues and details specific to the class RTOS.

The current target instruction set is the Intel 8086, an architecture for which development tools and a variety of embedded processors are readily available. Thanks to a prerequisite computer systems class based on the book and innovative labs of Bryant and O'Hallaron (2003), students are already familiar with the Intel x86 instruction set. The embedded system class relies primarily on simulation software to provide the environment for development and testing. The current tools are compatible with embedded hardware so that working software can be downloaded to a microcontroller board and run without modification.

Benefits of a Simulation Environment

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 are often plagued with problems such as damaged boards, bad connectors, or boards left with improper settings by previous users. Caused by frequent usage and handling, these hardware problems must eventually be discovered by a handful of unlucky students, often after much debugging. Such challenges are inherent

in hardware and must be addressed when creating real systems, but in an educational setting they detract from the class focus and add to student frustration. Finally, because simulators run on personal computers, they offer advantages of reduced infrastructure and cost relative to embedded hardware and tools.

Software Tools

The choice of software tools is important in any embedded system context. For example, an ideal simulator will execute hardware-compatible binaries, and it will accurately reflect an environment that requires 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 desired functionality.

Our compiler, assembler, and simulator are written in C and run on Windows, Linux, and Unix workstations. 8086 assembly code is generated by a modified version of C86, a publicly available ANSI C compiler. The compiler offers a simple inline-assembly capability with no syntactic or semantic checking. Because inline assembly can lead to perplexing errors, students write and maintain separate C and assembly files. Assembly functions must follow 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 critical of our software tools. The simulator is essentially 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 console 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.

The simulator offers features beyond those of typical debuggers. Most importantly, the simulator offers the possibility of not just stopping execution, but time itself. For students confused by execution involving interrupts, instruction trace 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 pointer problems.

With a simulator, an obvious challenge is representing 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 the video game Tetris. 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 call 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 overhead in both the application code and the RTOS. To ensure that the focus is on software efficiency rather than playing intelligence, the game is reduced to a 6x16 playing area and just two distinct pieces. A screen capture of the simulator is shown in Figure 1. In this example, time has been stopped during a game of Simptris and information about the machine state is displayed, including a memory dump, a disassembly of the beginning of main, a short instruction trace, and the current register state.

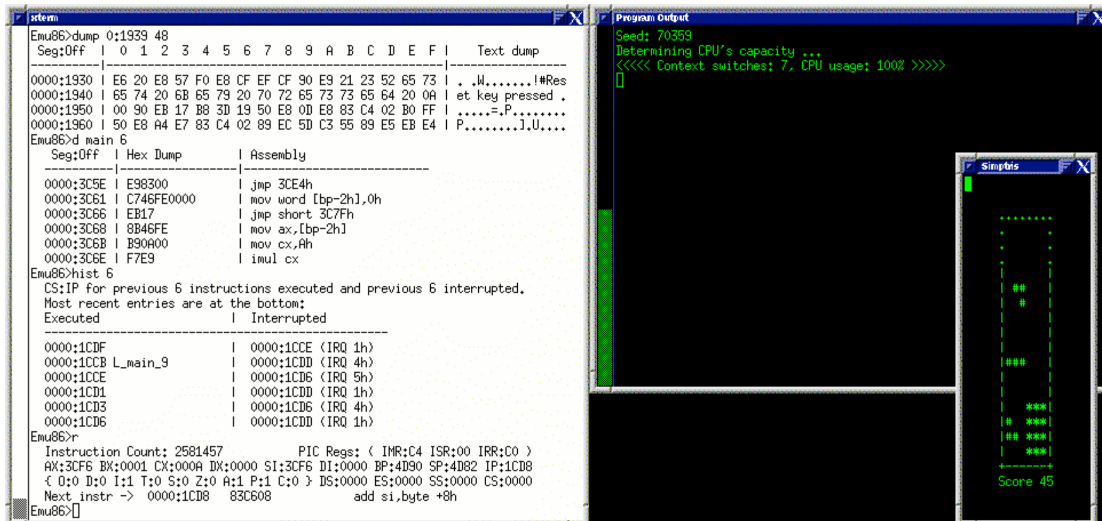

Figure 1: Emu86 simulator during Simptris play

The Kernel

The complete specifications for the kernel include prototypes for the twenty-two functions and a detailed discussion of required behavior. 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 will continue 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 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 reloads a task's context 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 message queues for inter-task communication. 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.

Laboratory 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 debugger capabilities to answer a variety of questions about a program and its execution. For example, students are asked to 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. By this point, each student will have been exposed to the tool chain, the simulator's debugging features, the 8086 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 an interrupt vector table.

In labs four through seven, students design and implement portions of the RTOS. The labs begin with an assignment to complete a detailed design, complete with pseudo code and a specification of data structures. Questions about specific implementation

details must be answered, and students receive feedback about areas of concern. In general, students who complete a thorough design save significant time implementing and debugging their kernels.

With the design complete, students begin to implement the 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 increased timer tick frequency and arbitrary key presses. 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 should begin execution and never relinquish 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 game's five interrupts 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 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. As a final requirement, students download their kernel and application code to a microcontroller board that plays the game via hardware connections to a PC emulating game hardware. Students see that the code they have developed is not specific to the simulator and that it runs on real hardware.

Discussion and Evaluation

Textbook Selection

It can be challenging to find a text that supports project-oriented courses, but we have had good success in using the book by Simon (1999), a text written by an embedded system designer for other practitioners. 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 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.

Ensuring Student Success

Students complete the first labs individually, but then work together in pairs when they begin to design and implement the RTOS. Almost all students are able to complete all labs. By working together, participants learn more and produce better software. Student success is also enhanced by hiring knowledgeable and enthusiastic TAs who can help teams when they get stuck. Because kernel implementations can vary widely, TAs must be able to consider many different approaches and provide help within each team's unique framework.

The 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 measure the impact of changes to the labs.

Near the last lab, students measure and report certain characteristics of their kernel, including total size of source code and executable, longest critical section, and the worst case overhead to release a semaphore. Figure 2 shows the average size of the student kernels for the last seven years. Figure 3 shows the average length of the longest critical section and the average overhead for a context switch, both measured in number of instructions. The 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.

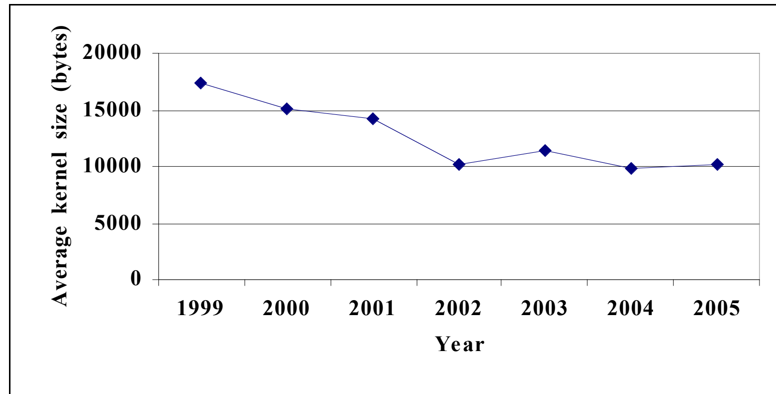

Figure 2: Average student kernel size in bytes for each year

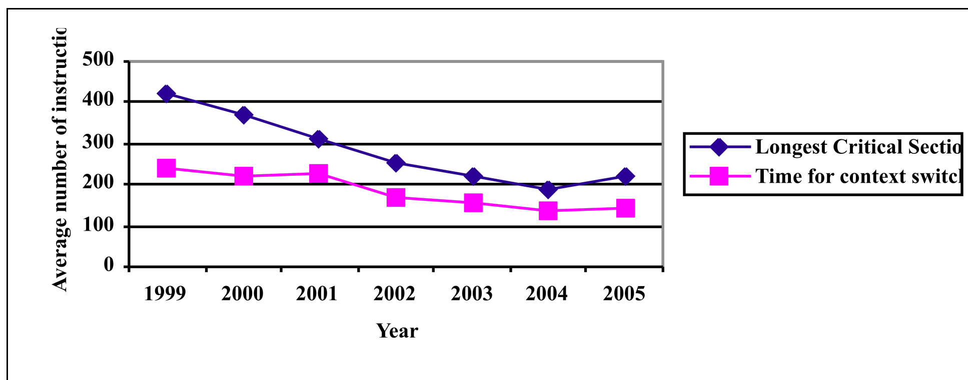

Figure 3: Average student kernel time for the longest critical section and for a context switch

Course Impact

The class is popular among students and is often cited as a favorite class in exit interviews with graduating seniors. Students enjoy building the complete system and feel that the overall experience gives them confidence in working with embedded applications. Official data is not available, but informal discussions with graduates indicate the course has increased the number seeking and finding employment relating to embedded systems. Naturally, few graduates are employed to develop an RTOS, but the knowledge of RTOS operation obtained in this class allows them to become

productive quickly as developers of application code for embedded systems. Not only do students know the types of functions generally supported, but they are sensitive to the overhead of RTOS operations and potential pitfalls in using and misusing kernel functions. Moreover, we have seen an increase in follow-on student projects using 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.

The course contributes significantly to student knowledge of embedded systems, but its greatest outcomes are more general. Students 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 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.

Conclusions

In summary, this course in real-time embedded software plays a significant role in our curriculum. Its impact goes far beyond preparing students for employment as embedded system developers. The course has undergone several important changes and will continue to evolve in response to changes in technology and other changes in our curriculum. The core focus of the class – developing embedded software from the ground up – has proven to be very educational for the students and well suited for a single semester undergraduate course. We are confident that the class will continue to meet our needs in the long term, and we feel that students at other universities would be well served by similar courses.

References

Lee, Edward A. (2005). Absolutely positively on time: what would it take? *Computer, July*, 85-87.

Fraser, Christopher W. & Hanson, David R. (1995). *A Retargetable C Compiler: Design and Implementation*. Redwood City, CA. The Benjamin/Cummings Publishing Company.

Labrosse, Jean J. (1992). *µC/OS: The Real-Time Kernel*. Lawrence, KS. R&D Publications.

Labrosse, Jean J. (2002). *MicroC/OS-II: The Real Time Kernel*, Second Edition. San Francisco, CA. CMP Books.
Simon, David E. (1999). *An Embedded Software Primer*. Boston, MA. Addison-Wesley.
Bryant, Randal E. & O'Hallaron, David R. (2003). *Computer Systems: A Programmer's Perspective*. Upper Saddle River, NJ. Pearson Education.