2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC

Host Sponsor UNC Charlotte

ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #51954

The Evolution of a Real Time Systems Course for Undergraduate Software Engineering Students

Dr. Walter W Schilling Jr., Milwaukee School of Engineering

Dr. Walter Schilling is a Professor and Program Director for Cybersecurity Systems in the Diercks School of Advanced Computing at the Milwaukee School of Engineering (MSOE). He teaches courses in both the Cybersecurity Systems and Software Engineering programs, specializing in software reliability, static analysis, cybersecurity, embedded systems, software verification, and software security. Before joining the MSOE faculty, Dr. Schilling served as a graduate researcher at NASA's Glenn Research Center and worked as a software product design engineer for both Visteon Corporation and Ford Motor Company. He currently serves as an editor for the Computers in Education Journal and is a commissioner with the ABET Engineering Accreditation Commission (EAC), supporting quality assurance in engineering education. Dr. Schilling's contributions to engineering education have been recognized with several honors, including the Ohio Space Grant Consortium Doctoral Fellowship, the ASEE New Engineering Educators Distinguished Service Award, the Merl K. Miller Award from the Computers in Education Journal, and the ASEE Computers in Education Distinguished Service Award.

The Evolution of a Real-Time Systems Course for Undergraduate Software Engineering Students

Abstract

Software engineering encompasses a broad range of essential topics, including software processes, architecture, requirements, and design. These are all foundational components of an undergraduate curriculum aimed at producing disciplined and capable engineers. However, one area of growing importance is the development of real-time systems, particularly as the complexity of modern software continues to escalate. For example, a Level 3 autonomous vehicle may require over 300 million lines of code, while a fully autonomous Level 5 vehicle could exceed 1 billion lines. These demands far exceed those of traditional systems and require deeper expertise in real-time software engineering.

Historically, real-time systems courses have focused on simple applications running on 8-bit microprocessors, typically offered within computer and electrical engineering programs. These approaches, however, fall short of preparing students for the challenges posed by contemporary embedded systems. This paper presents a case study on the design and evolution of a modern real-time systems course tailored specifically for software engineering students. The course leverages advanced embedded platforms and integrates content from networking, C++ programming, design patterns, and operating systems, culminating in the development of a robotic system. Drawing on over a decade of experience, the paper reflects on successes and challenges from both student and faculty perspectives and explores the curricular implications of offering such a comprehensive and integrative learning experience. The paper also offers practical recommendations for institutions seeking to implement or develop a similar course.

1. Introduction

Embedded software continues to account for a significant portion of all global software production. Over 98% of manufactured processing devices are destined for the embedded systems market, a figure that continues to rise. Driven by rapid expansion in the automotive sector, medical systems, IoT deployment, and the integration of AI and sensor networks, the embedded software market is projected to grow from $20.7 billion in 2024 to over $50.5 billion by 2034 [1].

Compared to traditional web or desktop applications, embedded systems are inherently more difficult to design. They often function as real-time systems with complex, overlapping constraints that must be met to ensure safety and performance. [2] Consequently, a substantial number of development positions remain unfilled due to a critical shortage of specialized talent. [3] [4]

While software engineering is often grouped with computer science in collegiate programs, they are distinct disciplines. Software engineering is a rigorous engineering field focused on the design and construction of robust, scalable solutions; computer science focuses on the theoretical foundations of computing. As noted by author Steve McConnell, "Computer scientists test theories and work at the edge of the unknown. Engineers start with knowledge that has already proven reliable." [5]

Because of this distinction, software engineering students require extensive hands-on experience with diverse physical systems. This difference is reflected in accreditation standards: ABET criteria for Software Engineering requires a "culminating major engineering design experience" that incorporates

formal engineering standards and multiple constraints [6]. In contrast, the CAC criteria for Computer Science require only a "major project" focused on the integration of prior coursework [7].

Recognizing these evolving industry needs, the Milwaukee School of Engineering (MSOE) has continually revised its software engineering curriculum. In addition to coursework in mobile, cloud, and security, since 2017, MSOE has required all software engineering students to complete a real-time systems course. Developed with strong support from an Industrial Advisory Committee (IAC), this requirement ensures graduates possess the core skills necessary for the modern workforce. The IAC was instrumental in the process, contributing to curriculum development, lab exercises, and providing grants for essential equipment.

2. Institutional Profile

The Milwaukee School of Engineering (MSOE) has offered an ABET-accredited Bachelor of Science in Software Engineering since 2001. The institution maintains a rigorous focus on applied learning, characterized by small class sizes (a 12:1 student-to-faculty ratio) and extensive laboratory experience. On average, MSOE graduates spend over 600 hours in laboratories related to their major; notably, the campus features more square footage dedicated to lab space than to lecture halls. All classes and labs are instructor-led, as MSOE does not utilize teaching assistants. This faculty-led approach is bolstered by deep industry ties, with faculty members bringing an average of seven years of professional industry experience and active consulting insights into the classroom.

Beyond technical proficiency, all graduates are trained in the MSOE Mindset—a framework that integrates servant-leadership and an entrepreneurial spirit with the university's core mission. This mindset ensures that students are not only technically competent but also value creators and leaders of character. This commitment to "high-impact experiential learning" is further supported by a comprehensive technology package: every student is issued a laptop pre-loaded with program-specific software. Additionally, students have access to the Rosie Supercomputer, providing the high-performance computing resources necessary for modern AI applications and extra-curricular research.

In 2022, MSOE transitioned from an 11-week quarter system to a traditional semester-based calendar (two 16-week terms per year). This shift has allowed for deeper engagement with complex topics. Within this framework, the software engineering program remains grounded in computer science fundamentals, such as discrete math, algorithms, and operating systems, while distinguishing itself through a heavy emphasis on requirements, design, verification, and process.

The software engineering program offers students several unique learning opportunities. One part of the program is a six-credit Software Development Laboratory experience in which students work on large-scale, industry-sponsored projects over two semesters. Most software engineering courses are offered in the 2+2 format, meaning the course meets in lecture two times for one hour and has a two-hour associated lab period each week.

3. The Legacy Embedded Systems Sequence: CE2800 and CE2810

Prior to adopting the Real-Time Systems course, students in the software engineering program were enrolled in a two-course sequence on embedded systems design. This sequence was designed by the computer engineering faculty to meet the needs of the program and their computer engineering constituency. The sequence focused heavily on low-level hardware interaction rather than on higher-level systems architecture, as is more common in modern software engineering.

The first course, CE2800, introduced students to assembly language programming using the ATmega32 embedded board (see Figure 1). The course served as a bridge between hardware and high-level programming, focusing on register files, instruction sets, and basic subsystems like timers and interrupts.

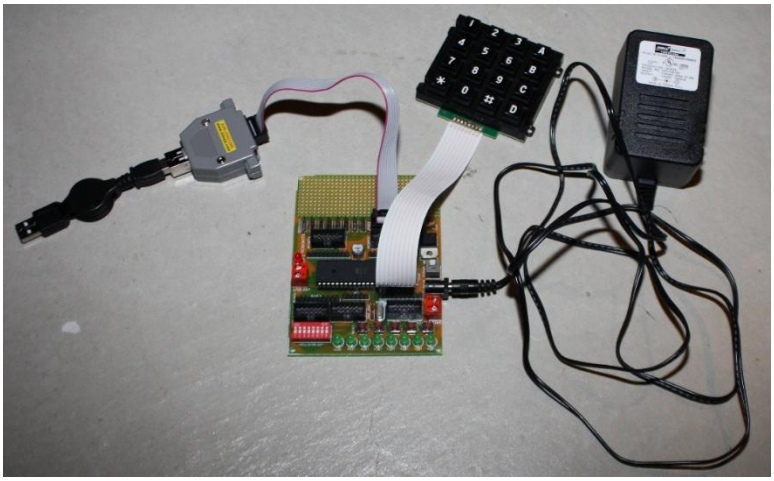

Figure 1 The Embedded Systems Development Board, prior to the real-time systems sequence being initiated.

This course presents assembly language programming as the bridge between hardware and high-level programming languages. Topics covered include the addressing modes, register file, and instruction set of a microcontroller; subsystems such as timers and analog to digital conversion; and interrupts. Software control of hardware is stressed. In the laboratory, students design software to demonstrate proficiency in these areas.

Outcomes:
Upon successful completion of this course, the student will be able to:
1. recognize the role of assembly language programming
2. state the programmer's model of a typical embedded processor (ATmega32)
3. break down the instruction set of a typical embedded processor, recognizing load/store, arithmetic, conditional branch, and unconditional branch instructions
4. construct assembly language programs by using and reusing subroutines
5. apply memory addressing and various addressing modes
6. employ hardware interrupts

Figure 2 CE2800 Course description and outcomes

The sequence continued with CE2810, which transitioned students from assembly to C programming on the same hardware. Students, who primarily had a background in Java, learned C syntax, pointer manipulation, and modular application design using multiple files.

This class builds on CE-2800 and introduces C as a high-level language for embedded systems programming. C pointers are introduced. C functions are introduced. Parameter passing by value versus using pointers is described. Interrupts in C are introduced and then the C/assembly interface is described. Designing modular applications by use of multiple files is described. Several subsystems, such as the USART and Timer system, are introduced. Key concepts are applied in laboratory exercises.

Outcomes:
Upon successful completion of this course, the student will be able to:
1. combine assembly and a high-level language to complete basic embedded system programming tasks
2. employ embedded systems development tools
3. link multiple files to create a larger application
4. design and write C functions
5. use interrupts in C to perform I/O
6. use the various subsystems of the processor in practical applications

Figure 3 CE2810 Course description and outcomes

While students appreciated the practical nature of these courses, several significant pedagogical and technical concerns emerged.

First, there were issues with the tooling and hardware. Students frequently reported issues with the reliability and robustness of the hardware kits. Furthermore, neither course utilized a formal textbook; instead, students relied on ATmega32 programming manuals, which proved difficult for developing students to navigate effectively.

Second, there was the issue of language relevance for software engineering students. While assembly was once the industry standard for embedded systems, by the early 2000s, C and C++ had rapidly replaced it for most embedded development. The heavy emphasis on 8-bit assembly felt increasingly disconnected from the needs of the modern workforce. This sentiment was echoed by the program's industrial advisory committee.

Third, there was an issue of relevancy for the simple systems that could be built on this processor. The legacy sequence lacked exposure to true operating systems and real-time scheduling. Software engineering students typically work on larger, more complex systems that require an RTOS (Real-Time Operating System) and higher-level processors, rather than the isolated 8-bit environments provided by the ATmega32. The ATmega32 also uses Harvard architecture, which can be more difficult for students to understand and is typically not encountered by software engineering students.

4. Overview of Courses

The first real-time systems course taught at MSOE was developed for the quarter-based system and consisted of three lectures and a three-hour lab each week, providing students with 6 contact hours per week. For the purposes of this paper, the course is referred to as SE3910 Version 1.0. This course ran in this format for 4 years.

At this point, an institutional standard was issued requiring that all labs be no more than 2 hours long. This resulted in a 1-hour reduction in the lab length for Real-Time Systems and minor changes to the course descriptions, with the intent of focusing more on system design rather than just system development. Instead of the 6 contact hours per week for the class, it was now limited to 5. This

represents SE3910 Version 2.0. This second course variation ran for five years before the university switched to a semester-based calendar.

The shift to semesters brought a more substantial restructuring. The course moved to two lectures per week across a 15-week semester, paired with a two-hour weekly lab. This format is designated SWE4211 Version 1.0.

5. Changes in Outcomes

Broadly speaking, the course description has remained very stable from the initial offering to the present, as the basics of real-time systems have not changed significantly. The changes in outcomes predominantly occurred as the course transitioned from quarters to semesters. This occurred mainly because a quarter-based networking course had to be removed from the semester-based structure due to institutional constraints. Thus, the networking content was spread across two courses, with real-time systems meeting students' needs for designing and constructing software using networking. Figure 4 contains the initial course description and course outcomes for SE3910 Version 1.0.

This course introduces students to software development for real-time systems, which often have stringent timing constraints that must be satisfied even under adverse circumstances caused by component failures. Real-time applications include flight control systems, vehicle control systems, industrial processes, life-support systems, robotic manipulators and multimedia applications. Special attention is paid to scheduling, latency minimization, bandwidth constraints, and other design issues that impact the design of these systems. Laboratory assignments provide experience in the design and implementation of realistic applications using a real-time operating system.

Outcomes:

1. *Understand concepts of time-critical computing and identify real-time systems*
2. *Get familiar with a host-target development environment for time-critical systems.*
3. *Write multitasking computer programs with inter-task communication and synchronization.*
4. *Apply concepts of inter-task communication and synchronization via shared memory, message queues, signals, semaphores, mailboxes.*
5. *Understand real-time kernels and task scheduling.*
6. *Understand concepts of reliability in relation to real-time software*
7. *Construct distributed real-time applications using a commercial Real-Time Operating System*
8. *Analyze the performance of a real-time system.*

Figure 4 Original Course Description and outcomes for Real-Time Systems Version 1.0

The change from SE3910 Version 1.0 to Version 2.0 prompted a revisit to the course. Version 2.0 of the course adds "intense design course" up front, signaling a stronger design orientation to students and shifting the class's focus from development to design. Version 2.0 also added the phraseology "embedded development board", clearly indicating to the students that they would be working with real, embedded hardware and not simulations. In Version 2.0 of the course, the technical topic of mailboxes is replaced with mutexes. Mutexes are a standard part of contemporary RTOSes, while mailboxes are generally being replaced with message queues for better flexibility and scalability. Outcome 7 drops the word "commercial" before "Real-Time Operating System," indicating that the projects were built on open-source hardware and software and did not require a proprietary OS. The outcomes for SE3910 Version 2.0 are shown in Figure 5.

This intense design course introduces students to software development for real-time systems, which often have stringent timing constraints that must be satisfied even under adverse circumstances. Real-time applications include flight control systems, vehicle control systems, industrial processes, life-support systems, robotic manipulators, and multimedia applications. Special attention is paid to scheduling, latency minimization, bandwidth constraints, and other design issues that impact the design of these systems. Laboratory assignments provide experience in the design and implementation of realistic applications using a real-time operating system and embedded development board.
Outcomes:

1. *Understand concepts of time-critical computing and identify real-time systems*
2. *Get familiar with a host-target development environment for time-critical systems*
3. *Write multitasking computer programs with inter-task communication and synchronization*
4. *Apply concepts of inter-task communication and synchronization via shared memory, message queues, signals, semaphores, and mutexes*
5. *Understand real-time kernels and task scheduling*
6. *Understand concepts of reliability in relation to real-time software*
7. *Construct distributed real-time applications using a real-time operating system*
8. *Analyze the performance of a real-time system*

Figure 5 Modified Course Description for SE3910 as the time spent in the laboratory was reduced by one-third

The shift from quarters to semesters led to more substantial changes in the course outcomes. The course itself shifted from being framed as an intense design experience to more of a team-based project, reflecting that students were working in teams and needed to do so to be successful, as well as a reduction in the intensity of the lab experience made possible by the longer semester timeframe. The course also focused more on the system's distributed nature, as it involves multiple processes running on different computers. Because of changes in the prerequisite chain, the semester-based course explicitly added outcomes related to networking, UDP, and TCP, as these were no longer covered in a required course in the curriculum. Cross compilation was also explicitly added, as students need to understand how the toolchain works rather than merely use it. The strength of the verbs was also increased in many of the new outcomes, replacing "understand" with higher-level Bloom's verbs such as construct, justify, verify, and identify. An outcome was also added to explicitly indicate that students would use configuration management in their project, an aspect that had been present before but was more emphasized during the semester-long project. The course description and outcomes for SWE4211 are given in Figure 6.

This course introduces students to software design and development of real time distributed systems. Real-time applications include flight control systems, vehicle control systems, industrial processes, life-support systems, robotic manipulators, and multimedia applications. Special attention is paid to scheduling, latency minimization, bandwidth constraints, and other design issues that impact the design of these systems. Team based laboratory assignments provide experience in the design and implementation of realistic applications using a real-time operating system, cross compilation, and embedded development boards.

Outcomes:

1. *Understand concepts of time-critical computing and identify real-time systems*
2. *Understand real-time kernels and task scheduling*
3. *Understand concepts of reliability in relation to real-time software*
4. *Develop software using cross compilation development environments*
5. *Construct software requiring inter-task communication and synchronization using shared memory, message queues, signals, semaphores, and mutexes*
6. *Construct distributed real-time applications using a real-time operating system*
7. *Justify the design of a network protocol based upon requisite performance characteristics.*
8. *Construct software using UDP and TCP/IP protocols*
9. *Identify real time constraints for a software system based upon physical behavior*
10. *Analyze the performance of a real-time system and verify that software is meeting real time constraints*
11. *Use software configuration management practices to work as a team on a software development project*

Figure 6 Modified Course Description for Real-Time Systems and the semester version.

While the course description has remained generally consistent, the main changes in the course have been in the lab sequence and the hardware used to teach real-time systems. This is the topic of the next section.

6. SE3910 Version 1.0 Lab Sequence

From its inception, the course's lab component has required students to work cooperatively in teams to design and implement software for a distributed real-time system. Initially, the selected platform consisted of the BeagleBoard-xM development board running the RT Linux operating system, a BeagleBoard-xM LCD7 7" TFT display, and a Leopard Imaging camera module. These completed systems communicated over a dedicated Ethernet network and were capable of exchanging video and audio in real-time.

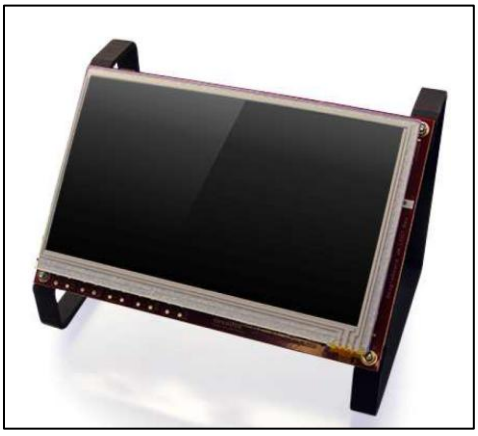

Figure 7 Initial SE3910 hardware

For development and debugging, students required access to oscilloscopes and multimeters, as well as an Ethernet network isolated from the campus backbone. To meet these needs, Digilent Analog Discovery units were purchased. This represented the first use of this equipment on our campus and was one of the early adoptions of what was then a new technology. These units were intuitive for students unfamiliar with hardware and were relatively inexpensive to acquire.

This hardware setup was piloted as a technical elective with a group of eight students. These students worked through prototype labs on the BeagleBoard-xM, ultimately building a simple, Skype-like communications system.

However, before the BeagleBoard-xM could be fully deployed, it was superseded by the BeagleBone and BeagleBone Black products. The original BeagleBone (White) offered easier development and integration using external "capes" and was more cost-effective. These capes allowed for the development of lab activities in a modular fashion. The BeagleBone Black eventually replaced the BeagleBone White, adding a higher-performance processor and onboard flash memory, which eliminated the need for students to provide their own microSD cards. Because of the lower costs, students could purchase their own BeagleBone devices for use in other projects. Consequently, the course was refactored to utilize the BeagleBone platform.

For this early version of the course, the goal was to develop a simple video conferencing application. This domain was specifically chosen to distinguish the coursework from robotics—the primary focus of our EE and CE programs—and to explore an emerging problem space, as video conferencing applications were neither ubiquitous nor reliable at the time.

Lab	Lab Title	Outcomes
1	Getting started with the development environment and the BeagleBone Black	• Understand how to create an sd card image of an operating system • Understand the concept of cross-compilation • Understand how a makefile functions. • Understand the operation of a cross compiler.
2	Basic IO with the BeagleBoard	• Explain how Embedded Linux can perform low level input and output through device drivers. • Construct an embedded Linux program which performs low level Linux I/O to read and write to an output pin. • Measure the systemic latency of input and output control using embedded Linux. • Construct software which uses both polling and interrupts to detect input value changes.
3	The Game of Anticipation	• Construct software which uses multiple POSIX threads. • Construct simple embedded software which controls GPIO devices. • Use GPIO interrupts to manage a given software package. • Design your own hardware interface
4	Networked IO with the BeagleBone	• Construct an embedded Linux program which performs low level Linux I/O to read and write to an output pin. • Construct a program which uses unix sockets to communicate over Ethernet. • Measure the systemic latency of input and output control using embedded Linux. • Measure the processor utilization on the BeagleBone
5	Building a Camera using GStreamer	• Interface with the camera port • Interface with GStreamer to capture an image.
6	Building a QT Interface	• Understand how to use QTCreator to create a simple GUI Application • Use QTCreator to create an application which can run on both the desktop and on a remote embedded target.
7	QT and a Camera	• Construct a compiled qt program which displays a test video pattern on the display • Construct a program which will capture video and show it on a display • Understand how one can interface QT • Understand how to use QTCreator to create a simple GUI Application • Use QTCreator to create an application which can run on both the desktop and on a remote embedded target
8	Building a Video Conferencing Application	• Design a system to meet a given set of real time constraints. • Use QTCreator to create an application which can run on both the desktop and on a remote embedded target

Figure 8 Initial lab topics and outcomes for SE3910.

Overall, this sequence worked well initially. Students gained exposure to real-time programming in a Linux environment and successfully built a real-time system, even if limited in scope. Crucially, they produced a tangible project to showcase beyond simple software running on a laptop.

The initial hardware was used by more than 150 software engineering students and nearly 100 students in various summer camps. As a hobbyist product, the BeagleBone offered several advantages: it was relatively inexpensive and easily extensible. However, these same traits led to complications. As a hobbyist-grade product, it was less robust than industrial alternatives, and we encountered significant issues regarding quality and embedded Linux support.

Between roughly 2014 and 2018, the BeagleBone and BeagleBone Black experienced significant operating system churn as the platform navigated rapid changes in embedded Linux and upstream kernel development. Early releases shipped with Ångström Linux, a minimalist embedded distribution, but BeagleBoard.org formally transitioned to Debian in 2014 to gain a broader package ecosystem, longer-term maintenance, and improved developer familiarity [8].

Concurrently, Linux kernel support for the Texas Instruments AM335x SoC was in flux. BeagleBone images alternated between TI-maintained kernels and newer mainline kernels as device drivers, PRU

(Programmable Real-Time Unit) support, and power management matured upstream. The enforced migration from legacy ARM board files to Device Tree-based hardware descriptions, along with the introduction and later refactoring of Device Tree overlays for cape support, frequently broke pin configuration workflows and required repeated OS image revisions [9].

This instability was further amplified by the ecosystem-wide adoption of systemd and evolving udev behavior within Debian, which affected boot sequencing, service startup, and hardware enumeration. Because the BeagleBone project prioritized open hardware and mainline Linux compliance over strict backward compatibility, distribution changes were frequent and highly visible, producing notable churn during this period.

Supply chain issues also emerged. Between late 2014 and much of 2015, the BeagleBone Black experienced a pronounced supply shortage when RadioShack became a large-scale retail distributor. BeagleBoard.org reported that RadioShack's nationwide rollout of *Make: Getting Started with BeagleBone* kits rapidly absorbed production capacity previously allocated to educational buyers, contributing to a "drought" in traditional channels [10], [11].

Over the years, normal wear and tear and expanding class sizes necessitated additional hardware purchases. However, several original components became unavailable, including the expansion capes, the initial audio boards, the prototype capes used for latency labs, and the original LCD touch panels. Consequently, we had to source alternatives, such as external USB sound adapters and different LCD panels. These newer panels often featured subtly different layouts, requiring multiple sets of lab instructions for the same class.

Technical performance also degraded over time. We encountered issues with newer cameras where interactions between resolution settings and updated Linux distributions caused performance lags. In the final offering using this hardware, the distribution kernel was not compiled with PREEMPT_RT extensions, significantly impacting our ability to demonstrate real-time Linux concepts. These issues detracted from the student experience and moved the focus away from system development.

7. SE3910 Lab Version 2.0

Due to broader curriculum changes, including reducing the lab from three to two hours, the decision was made to reevaluate the lab structure and platform.

Between 2014 and 2017, the BeagleBone and Raspberry Pi platforms exhibited markedly different patterns of software stability. While BeagleBone prioritized upstream compliance, which led to breaking changes, the Raspberry Pi Foundation utilized a more vertically integrated ecosystem. Raspberry Pi OS (formerly Raspbian) remained Debian-based, but changes were backported and abstracted behind stable tools like raspi-config and Foundation-maintained kernel branches. This insulated users from Linux volatility, making the evolution feel incremental and predictable.

Given this stability, the lab hardware was transitioned to the Raspberry Pi using the PREEMPT_RT patch. Simultaneously, we reevaluated the lab sequence. As time passed, more students became involved in robotics through programs like FIRST Robotics and VEX Robotics. These organizations have shaped modern engineering education by making robotics a mainstream interest for incoming students.

Additionally, in this same time period, the Computer Engineering program shifted away from using robotics based projects.

With this background, the project was revised to involve the design and operation of a simple robot, the AlphaBot2. This platform integrates with the Raspberry Pi and features multiple onboard sensors. The camera is connected via the CSI interface with X-Y positioning servos, and software-based PWM (Pulse Width Modulation) is used for motor control.

Figure 9 AlphaBot 2 robot.

With the robot serving as the primary vehicle for teaching real-time systems, the lab sequence and outcomes were adjusted accordingly. These new labs and their respective sequences are detailed in Figure 10. This revised structure reinforces the concept of systems managing multiple, competing, realistic constraints. For example, increasing the polling rate of the distance sensor allows robots to operate reliably at higher speeds; however, it also consumes more CPU cycles, which can reduce the resources available for image streaming and diminish stream clarity.

Lab	Lab Title	Outcomes
1	Getting started with the development environment and the Raspberry Pi	• Create a microSD card image of an operating system • Understand the concept of cross-compilation • Understand the operation of a cross compiler. • Construct a simple application using the Raspberry Pi. • Explain how to use remote debugging capabilities to debug a program on an external device. • Compare the latency of a Real time system with a non-real time system.
2	Basic IO with the Raspberry Pi	• Explain how Embedded Linux can perform low level input and output through device drivers. • Construct an embedded Linux program which performs low level Linux I/O to read and write to an output pin. • Measure the systemic latency of input and output control using embedded Linux and two different languages. • Construct software which uses both polling and interrupts to detect input value changes. • Work with cmake and Eclipse to compile and link cross compiled programs
3	The Game of Anticipation	• Construct simple embedded software which controls GPIO devices. • Use GPIO interrupts to manage a given software package. • Construct software which uses multiple C++11 threads. • Debug multithreaded programs using gdb on a remote host. • Document C / C++ source code using Doxygen • Generate documentation from your completed project
4	Networked Light Control	• Develop embedded code using network sockets • Interpret data within a trivial communications protocol • Construct a synchronized queue to process received messages. • Control GPIO using the network.
5	Basic Robot Control	• Integrate simple control systems into a robot. • Control a robot remotely using simple network protocols • Develop embedded code using network sockets • Reuse a previously developed queue for aiding in the development of a system. • Use PWM to adjust the speed of a motor • Construct software which uses periodic tasks to complete a simple task.
6	Distance Measurement	• Design and perform an experiment to analyze latency • Construct software which uses timing to determine distances • Implement software to control an ultrasonic distance sensor. • Integrate multiple queues into a control system.
7	Sensor Usage	• Integrate multiple queues into a control system. • Use input sensors to manage a robotic system. • Calibrate an input sensor using an analog value. • Make decisions based upon analog values when controlling a robot. • Add threads to an existing system to extend basic functionality. • Measure CPU utilization.
8	Camera Streaming	• Capture images from a CCD camera using OpenCV • Transmit images across the network to base computer • Cross compile with libraries on a local machine • Develop multithreaded software subject to real time performance constraints
9	RMA Based Robot	• Perform RMA on a system to obtain optimal performance • Tune the performance of a system to make it operate in the smoothest fashion possible • Develop multithreaded software subject to real time performance constraints

Figure 10 Lab topics and outcomes for first robot based system.

This change, coupled with updates to the ABET Continuous Improvement Process, resulted in the addition of new assessments to the course. Because the course is taken during the senior year and sits at the end of a prerequisite chain, it serves as an ideal location to assess student learning. Coinciding with these curricular updates, ABET transitioned its student outcomes from the legacy “a–k” criteria to the current “1–7” outcomes. Consequently, the real-time systems course was assigned two Key Performance Indicators (KPIs) for assessment, as illustrated in Figure 11.

ABET Outcome 1: *an ability to identify, formulate, and solve complex engineering problems by applying principles of engineering, science, and mathematics.*

SESO1P6 *A student will apply mathematical models and principles to design and implement solutions.*

ABET Outcome 6: *an ability to develop and conduct appropriate experimentation, analyze and interpret data, and use engineering judgment to draw conclusions.*

SESO6P2b *A student will apply scientific experimentation in software testing and defect resolution.*

Figure 11 ABET KPIs evaluated in the real-time systems course.

To assess students' ability to solve engineering problems, an assessment was developed based on the final Rate Monotonic Analysis (RMA) conducted to verify the robot's operation. If this analysis is performed correctly, the mathematical model and design are provably optimal from a scheduling standpoint. Generally, performance issues become immediately visible if the robot is not tuned properly. This assessment was also integrated into the final exam; while it lacked the depth of the hands-on lab, it allowed for the verification of individual understanding of design concepts, ensuring accountability in what was otherwise a partner-based lab environment.

A specific lab sequence was designed with input from Industry Advisory Board members to experimentally validate the capabilities of the ultrasonic distance sensor. In this experiment, students observed the impact of latency on sensor accuracy and identified error margins based on the object's distance. Students were able to see how latency increases variance in distance measurements, thereby decreasing the reliability of individual readings.

Furthermore, these experiments demonstrated how "performance mode" yielded more accurate measurements due to reduced context-switch latency compared to "power save mode." This exercise allowed us to assess the students' ability to design and conduct a formal experiment. Students also observed the impact of the Linux PREEMPT_RT kernel firsthand by comparing a microSD card running the RTE against one running a standard kernel.

This lab sequence remained stable for the first two years. Students purchased Raspberry Pi 3 units for the course; although the Pi 4 was released during the second year, it arrived too late for integration. Minor hardware issues occurred, such as stripped motor gears resulting from an improperly selected PWM period. However, the primary challenge was power management: the robot batteries were small, lacked a charge-monitoring circuit, and frequently depleted without warning.

The third year proved to be the second most challenging in the course's history due to the COVID-19 pandemic. With students moving between remote and in-person instruction, the lecture format transitioned easily, but the lab sequence required a total overhaul. Pandemic safety protocols required students to work individually rather than in pairs. Because we lacked a sufficient inventory of the original robots for individual distribution, we switched to a more available model and assembled over 60 units. These were either shipped to students or picked up on campus. Instructors assisted with debugging via Microsoft Teams, often navigating the technical hurdles of taxed home networks.

Figure 12 The robots aligned and being prepared to ship to students during the pandemic.

These replacement robots were a compromise in terms of quality. While they were used for three additional years until our transition to a semester-based calendar, they presented significant maintenance issues. During the same period, we experienced a severe supply chain crisis with Raspberry Pi hardware.

Throughout the COVID-19 pandemic, the Raspberry Pi ecosystem was heavily impacted by global supply chain disruptions, particularly the semiconductor shortage that intensified throughout 2021. The surge in demand for consumer electronics, combined with constrained fabrication capacity for the 40 nm process nodes used in Raspberry Pi devices, led to severe shortages. Production volumes in 2021 stagnated despite high demand, forcing institutions to adapt curricula or seek alternative hardware. While the Raspberry Pi Foundation indicated in late 2021 that these constraints would persist, supply chain pressures began to ease by late 2022. By mid-2023, production returned to pre-shortage levels, effectively ending the shortage [12] [13] [14] [15].

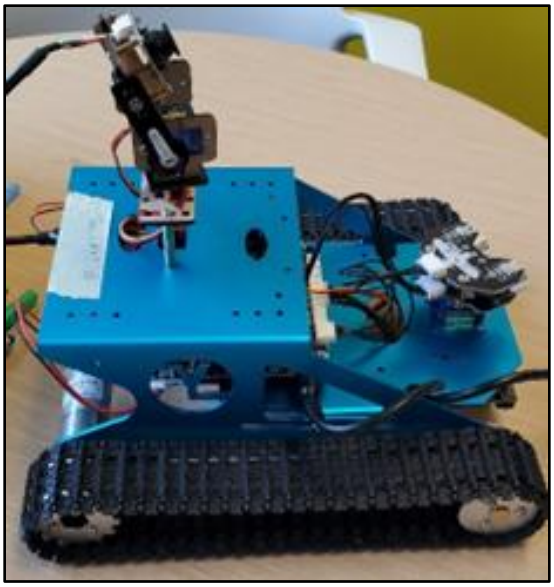

Figure 13 The robot selected for semester-based courses.

8. SWE4211 Lab Sequence

When the university transitioned to a semester-based calendar, a new robot was selected, as shown in Figure 13. This platform was specifically chosen for its more robust design and its support for an appropriate number of sensors and controllable features. This hardware allows for year-to-year variability in the lab assignments without requiring extensive hardware modifications or a complete rewrite of the course curriculum.

The transition to the semester format presented a unique challenge. While the total number of lecture sessions remained the same (30 sessions), they were distributed over 15 weeks instead of 10. Additionally, the course required five additional lab sessions. This necessitated two strategic adjustments. At the start of the course, the pace of the labs had to be slowed, as it was not possible to deliver new theoretical material fast enough to keep pace with the expanded lab schedule. At the conclusion of the course, the additional sessions allowed for and required greater technical depth and the implementation of more advanced features.

The updated lab outcomes for the semester-based course are provided in Figure 14.

Lab	Lab Title	Outcomes
1	Getting started with the development environment and the Raspberry Pi	• Create a microSD card image of an operating system • Configure a Linux virtual machine for cross compilation
2	Basic IO with the Raspberry Pi	• Describe the role of device drivers in enabling low-level I/O operations within embedded Linux systems. • Develop and deploy an embedded Linux application that performs direct I/O operations to read from and write to hardware output pins. • Configure and utilize CMake and Eclipse to cross-compile, link, and debug embedded programs for target hardware platforms. • Implement GPIO-based control logic to manipulate the state of other GPIO pins through software. • Monitor and analyze real-time CPU usage of running programs using the htop utility.
3	Raspberry Pi Performance Analysis	• Compare the performance of GPIO operations implemented in Python, native C/C++, and Rust. • Measure signal latency and analyze how design constraints affect system responsiveness. • Explain how embedded Linux can perform low level input and output through device drivers.
4	The Game of Anticipation	• Construct simple embedded software which controls GPIO devices. • Use GPIO interrupts to manage a given software package. • Construct software which uses multiple C++11 threads. • Debug multithreaded programs using gdb on a remote host. • Document C / C++ source code using Doxygen • Generate documentation from your completed project
5	Networked Light Control	• Develop embedded code using network sockets • Interpret data within a trivial communications protocol • Construct a synchronized queue to process received messages. • Control GPIO using the network. • Control the brightness of a light using software PWM. • Verify the robustness of your implementation by using defensive testing
6	Performance Analysis and Robot Start	• Compare the real-time performance of two embedded computers • Control a robot remotely using simple network protocols • Develop embedded code using network sockets • Reuse a previously developed queue for aiding in the development of a system. • Construct software which uses periodic tasks to complete a simple task. • Integrate simple network-based controls for a robot
7	Basic Robot Control	• Integrate simple control systems into a robot. • Control a robot remotely using simple network protocols • Develop embedded code using network sockets • Reuse a previously developed queue for aiding in the development of a system. • Construct software which uses periodic tasks to complete a simple task. • Control the creation of sound through a piezo buzzer
8	Sensor Usage	• Integrate multiple queues into a control system. • Use input sensors to manage a robotic system. • Add threads to an existing system to extend basic functionality. • Measure CPU utilization. • Measure robot performance characteristics
9/10	Distance Measurement	• Implement software to control an ultrasonic distance sensor. • Design and perform an experiment to analyze latency • Construct software which uses timing to determine distances • Design and integrate an algorithm using publish-subscribe related concepts to control a robot
11	Camera Streaming	• Capture images from a CCD camera using OpenCV • Transmit images across the network to base computer • Cross compile with libraries on a local machine • Develop multithreaded software subject to real-time performance constraints
12/13	A Collision Sensing Streaming Robot	• Integrate multiple queues into a control system. • Use input sensors to manage a robotic system. • Add threads to an existing system to extend basic functionality. • Using experimentally captured data, design a solution to a real-world problem • Measure CPU utilization
14/15	A Secured RMA Based Robot with Image Detection	• Integrate AES encryption into a data channel • Integrate a heartbeat strategy to detect the operational health of a software component • Use Generative AI to implement an image detection algorithm to detect stop signs and stop the robot • Perform RMA on a system to obtain optimal performance • Tune the performance of a system to make it operate in the smoothest fashion possible • Develop multithreaded software subject to real-time performance constraints

Figure 14 Lab topics and outcomes for semester based course.

To accomplish these goals, the scope of the early labs was adjusted to provide more depth in areas that did not require synchronous lecture coverage, while the introduction of complex new concepts was paced more deliberately. For example, performance analysis was extended over several weeks, allowing students to compare different programming languages from a real-time perspective. Previously, these comparisons were limited primarily to different OS distributions and whether the Linux PREEMPT_RT was enabled in the kernel image.

This slower initial pace also allowed more time to emphasize experimental discipline and technical report writing. In previous iterations, students often demonstrated suboptimal performance in experimentation, specifically regarding data collection and visualization (such as generating appropriate graphs). By utilizing the expanded semester timeline, early labs could reinforce these foundational skills—competencies that should theoretically have been mastered in prior courses but historically remained inconsistent. While this focused coverage has improved assessment scores in experimentation, students continue to struggle with mathematical precision and unit management, particularly with the proper use of SI prefixes and the conversion of results between them (e.g., milliseconds to microseconds).

New concepts were integrated into the final weeks of the course. The first was embedded systems security and the encryption of data channels, a critical concept for software engineers that was not feasible under the previous quarter system. Additionally, we introduced the use of Generative AI to develop source code for stop-sign recognition within the camera feed. Both topics increase the computational load on the processor, thereby increasing the complexity of the Rate Monotonic Analysis (RMA). The objective is that by requiring additional lab work after the introduction of RMA, students will show a corresponding improvement in their ability to develop and assess mathematical models for design.

From a pedagogical standpoint, the concluding lectures provide perhaps the most valuable aspect of the course through detailed case studies. Students are guided through significant software failures directly related to real-time systems, including the Ariane 5 rocket failure, the Apollo 11 task overflow, and the Therac-25 incidents. While these cases are older, their ethical and professional lessons are invaluable and can be presented with greater technical depth now that students understand the underlying technology.

The course culminates in a deep dive into the Toyota Unintended Acceleration case study [16] [17]. Students are given the opportunity to review legal testimony transcripts while being guided through the technical architecture of the system. This allows them to understand how the legal process and engineering failures may intersect and impact their professional lives.

9. Recommendations and Observations

The Real-Time Systems course is a challenging course to teach because of its hands-on experiential learning aspects. The faculty member assigned to teach it needs to have a working knowledge of all the coursework that leads to this course and exhibit strong diagnostic skills. However, these challenges are themselves instructive, as they are the hallmark skills of a practicing engineer. By modeling these skills for students, instructors help them to grasp their importance.

Throughout this evolution of courses, student feedback has been consistently favorable and has helped shape the course's development. Figure 15 provides a sample of student comments regarding the most

impactful aspects of the class. While this is not an easy course for the students, they appreciate its depth and thoroughness. Most importantly, they have fun learning the material.

Student 1: "I liked that the robot was continually being added to, i.e. each lab built on the last."
Student 2: "The labs were definitely my favorite part of this course and some of my favorite labs at MSOE."
Student 3: "I really enjoyed seeing the design architecture of the system and its components."
Student 4: "The labs, they were very interesting & fun to figure out. These were some of the most fun labs I've had."
Student 5: "Working with hardware for the first time that actually felt like using hardware."
Student 6: "As a SWE student, it is often difficult to get non SWE and CS friends to appreciate our work, but the outcomes of this course are far more tangible to non-computer oriented colleagues."
Student 7: "I liked how the labs built on each other till it was a 'full functioning' robot."
Student 8: "I really enjoyed programming the robot. It was a lot of fun doing something and then seeing it interact with things in real life."
Student 9: "Even though building the robot was tough & stressful, it was still fun. I felt like I learned so much more by building it."
Student 10: "The best part of this class was getting to see physical results of our programming efforts. As SEs, that's not really something we've been able to experience thus far."
Student 11: "It was fun running the finished product and seeing it all work together."
Student 12: "The best part was getting some experience controlling hardware as that is lacking in the SE major."
Student 13: "I really enjoyed the labs. I always thought they were the most helpful part of this course."
Student 14: "I really liked how the labs built upon each other and were not just standalone."

Figure 15 Sample student comments related to Real-Time Systems.

With the current trend toward decreasing enrollments in Computer Science, which is also impacting the software engineering discipline, it is important to design appropriate curricular experiences. Robotics integration improves engagement and motivation, strengthens coding and computational skills, supports hands-on experiential learning, and enhances STEM learning outcomes [18] [19]. A course such as this provides the necessary real-world context and domain expertise that is not possible in more theoretical or simulated courses. This also helps to engage students who are participating in the many K-12 robotics programs, such as FIRST Robotics and VEX Robotics.

From a cost perspective, robotics kits represent a modest hardware investment for programs to make, especially in comparison to the extensive laboratory environments required for mechanical, electrical, and civil engineering departments. In many cases, robotics hardware is cheaper than the servers required for even modest AI teaching. The robots themselves, with appropriate maintenance, can be used repeatedly. While the prime users are the students in the program, on weekends and during the summer, they can be used for STEM recruitment activities, helping to expand the pipeline into the software engineering discipline. These platforms also offer something visual for prospective students visiting campus and touring facilities. To a prospective student, much of the computing field can feel lifeless — a computer in a computing lab looks very similar to a computer in an English writing lab. Robots provide something tangible and visual that can be used to explain how students are taught using active learning. Robots also help students prepare for future projects, including Edge Computing and TinyML projects.

One practical decision all programs must make is whether to provide hardware to students or require them to purchase it. Throughout the Real-Time Systems course history, several approaches have been tried. When students purchase their own devices, those devices can be used elsewhere. Many Raspberry Pi units, for example, were later used by students in a network security elective or on their senior design projects. However, this model leaves faculty at the mercy of supply chain disruptions and may force

extensive course changes if the appropriate equipment is unavailable when needed. Purchasing equipment at the program level, except for microSD cards for OS images, allows everything to be reused and, more importantly, greatly reduces the likelihood of an unexpected supply chain issue derailing the course.

10. Summary

Overall, the Real-Time Systems course has undergone three significant revisions while maintaining the same core outcomes.

For software engineering students, teaching real-time systems through the robotics domain helps them appreciate the tangible impact of latency in a safe, low-risk environment. This approach also aids in student recruitment, as many in the field first developed an interest in technology through pre-college robotics programs such as FIRST Robotics and VEX Robotics.

While not strictly mandatory for all software engineering programs, a course in real-time systems is invaluable because it immerses students in engineering-driven software development rather than abstract computing concepts. It emphasizes the realities that define professional practice—standards, constraints, trade-offs, and risk—aligning closely with ABET's expectations for engineering design and science. Students are not merely exposed to operating systems and real-time concepts; they are required to apply them through iterative design, analysis, and experimentation under real-world constraints such as hardware variability and standards compliance. In doing so, the course differentiates itself from traditional computing coursework by reinforcing software engineering as a disciplined engineering practice.

Through a sequence of hands-on laboratories, students learn by measuring, comparing, and validating system behavior in physical environments rather than through simulations. They experimentally evaluate language and hardware choices, investigate timing effects, apply design patterns to distributed embedded systems, and utilize UML sequence diagrams to reason about performance issues. These labs deliberately expose students to the types of failure modes encountered in industry, where configuration errors or minor timing variations can have catastrophic impacts. This emphasis on evidence-based reasoning helps students develop a technical intuition for system behavior that cannot be gained through purely theoretical exercises.

Beyond the laboratory, the course strengthens professional judgment by integrating ethical, legal, and standards-based considerations into technical design. Through case studies of real-world failures—spanning the aerospace, medical, and automotive industries—students examine how technical decisions intersect with safety and regulation. This perspective, combined with the course's industry-aligned origins, ensures that graduates depart with a clear understanding of how to engineer software systems that operate correctly, predictably, and safely in the real world.

Bibliography

[1] Global Market Insights, "Embedded Software Market Size & Share – Industry Analysis Report, 2023–2032," GMInsights.com, 12 2024. [Online]. Available: https://www.gminsights.com/industry-analysis/embedded-software-market. [Accessed 25 4 2026].

[2] Y. Chawla, "Global surge in embedded software demand: here is why," DAC Digital, 4 June 2022. [Online]. Available: https://dac.digital/global-surge-in-embedded-software-demand-here-is-why/.

[3] L. Harvie, "The Great Embedded Engineer Shortage: Why 80% of Job Postings Go Unfilled," Runtime Recruitment, 2025.

[4] S. Bhujbal, "How to overcome talent shortages in embedded systems," UST.

[5] S. McConnell, Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers, Addison-Wesley, 2003.

[6] ABET, *Criteria for Accrediting Engineering Programs, 2016 – 2017,* Baltimore: ABET, 2015.

[7] ABET, *Criteria for Accrediting Computing Programs, 2019 – 2020,* Baltimore: ABET, 2019.

[8] Fustini, "BeagleBoard.org releases Debian for BeagleBone Black," Element14, 4 3 2014. [Online]. Available: https://community.element14.com/products/devtools/single-board-computers/f/forum/24108/beagleboard-org-releases-debian-for-beaglebone-black/159392.

[9] J. Cooper, *Introduction to the Beaglebone Black Device Tree,* AdaFruit, 2024.

[10] J. Krider, "Dude, where's my BeagleBone Black?," Beagleboard.org, 13 April 2014. [Online]. Available: https://www.beagleboard.org/blog/2014-04-13-dude-wheres-my-beaglebone-black. [Accessed 10 1 2026].

[11] J. Krider, "Declaring the drought is officially over," BeagleBoard.org, 28 January 2015. [Online]. Available: https://www.beagleboard.org/blog/2015-01-28-declaring-the-drought-is-officially-over. [Accessed 10 1 2026].

[12] E. Upton, "Supply chain, shortages, and our first-ever price increase," Raspberry Pi, 20 October 2021. [Online]. Available: https://www.raspberrypi.com/news/supply-chain-shortages-and-our-first-ever-price-increase/.

[13] A. Cunningham, "Supply chain woes lead to a "temporary" Raspberry Pi 4 price hike," *Ars*

Technica, 20 October 2021.

[14] A. Yee, "At last, the Raspberry Pi shortage is finally coming to an end," *PC World,* 1 June 2023.

[15] E. Upton, "Supply chain update – it's good news!," Raspberry Pi, 11 December 2022. [Online]. Available: https://www.raspberrypi.com/news/supply-chain-update-its-good-news/.

[16] S. Henry, "Toyota's Unintended Acceleration Scandal: What Happened and Its Lasting Impact," *The Auto Wire,* 4 January 2025.

[17] M. Barr, *Barr's testimonial slides,* Safety Research, 2014.

[18] P. Gupta, "The Role of Robotics in Modern Education – Benefits, Challenges, & Examples," *EdTech Review,* 2024.

[19] F. Ouyang and W. Xu, "The effects of educational robotics in STEM education: a multilevel meta-analysis," *Springer Nature,* vol. 11, no. 7, 2024.