Teaching Embedded Systems the Berkeley Way[*]

Edward A. Lee
EECS, UC Berkeley
Berkeley, CA, USA
eal@eecs.berkeley.edu

Sanjit A. Seshia
EECS, UC Berkeley
Berkeley, CA, USA
sseshia@eecs.berkeley.edu

Jeff C. Jensen
National Instruments Corp.
Berkeley, CA, USA
jjensen@ni.com

ABSTRACT

This paper describes an approach to teaching embedded systems from the perspective of cyber-physical systems. We place less emphasis on the mechanics of embedded system design and more on critical thinking about design technologies and on how the design of embedded software affects the behavior, safety, and reliability of cyber-physical systems. The course gives students experience with three distinct levels of design of embedded software, namely bare-iron programming (software that executes in the absence of an operating system), programming within a real-time operating system, and model-based design. In each case, students are taught to think critically about the technology, to probe deeply the mechanisms and abstractions that are provided, and to understand the consequences of chosen abstractions on overall system design. This paper describes a laboratory experience that first exposes students to the three levels of abstraction through a structured sequence of exercises, followed by an open-ended capstone project. Several example projects are described.

1. INTRODUCTION

Five years ago at Berkeley, we began to refine and develop a *cyber-physical systems* approach to teaching embedded systems. We continue to refine this approach, which focuses on understanding and controlling the conjunction of software and physical dynamics. This contrasts with a more classical approach to embedded systems which treats the subject as a set of technical skills to be mastered by effective embedded systems engineers. The more classical approach focuses on *mechanics* of microcomputer interfacing and C programming; our goal is more ambitious. We too require students to master these mechanics, but more importantly, we require them to think *critically* about these mechanics and they affect interplay between computation and the physical world. The goal of this paper is to explain how we introduce skills in critical analysis and design methodology to students of embedded systems and computer science.

There is quite a bit of background to this work. We have written a textbook [1], and have published in this same venue the approach taken in the textbook [2]. The textbook is organized around three interplaying threads that we call modeling, design, and analysis, as illustrated in Figure 1. The *modeling* part of the book focuses on models of dynamic behavior of physical systems in a temporal continuum, discrete systems, and the conjunction of the two, leading into concurrent models of computation. The second part of the book focuses on the *design* of embedded systems, with emphasis on the role they play *within* a cyber-physical system. This part includes discussions of microcomputer architecture, memory architectures, input/output techniques, multitasking, and scheduling. The final part of the book is about *analysis* and introduces temporal logic, notions of equivalence and refinement, reachability analysis, and program analysis. These three threads are meant to be studied concurrently.

Although the design portion of the textbook has a very "hands-on" flavor, the book deliberately does not include a laboratory component. Instead, we have been developing a complementary lab manual, the principles of which we have previously described [3], and a draft of which is available from the course website, `http://chess.eecs.berkeley.edu/eecs149`. At Berkeley, the lab activity has two phases. In the first phase, which lasts six weeks, students complete a sequence of well-defined exercises with the end goal of programming a microcontroller to control the robot in Figure 2 to autonomously climb a hill while avoiding cliffs and navigating around obstacles that may be present along the way. The second phase is a capstone project where students form teams and construct projects of their own creation. Both phases of the lab are described below, with emphasis on how they complement the textbook and support the key principles of the course.

2. PROGRAMMING EMBEDDED PROCESSORS

Our key goal is to teach a *critical* approach to the subject of embedded systems. Rather than just teaching technical skills, or "how to," our goal is also to get students to think about how things could be improved (or at least why they should be). Our view is that the field of cyber-physical systems is very young, and it would not serve our students well to leave them with the illusion that completing the course

[*]This work was supported in part by the the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (CPS: Large: ActionWebs), and #1035672 (CPS: Medium: Ptides)).

equates to mastery of the subject. The techniques in the field are very likely to evolve – possibly quite dramatically and rapidly – over the next few years, and engineers who believe they already “know how to do it” will be at a huge disadvantage. Those who are taught only to master the mechanics of microcontroller programming will adapt poorly to the coming changes, especially given a lack of exposure to design methodologies that consider the broader dynamics of cyber-physical systems.

To underscore this point, we repeatedly show students that when writing programs for embedded processors today, we often have to step around programming abstractions to get things done. While programming abstractions are well-suited for manipulating data, they do not do well, for the most part, at manipulating the physical world. Concretely, the way we approach such critical thinking is to give students experience in design at three sharply different levels of abstraction, illustrated in Figure 3. At the “bare-iron” level (indicating the absence of an operating system), students directly confront the mismatch between programming abstractions and even the simplest of interactions with the physical world, specifically the use of interrupts for timed operations, and memory-mapped registers for programmatic access to hardware peripherals. Stepping higher into the layer of real-time operating system (RTOS) design, many of the warts associated programming abstractions are hidden behind mysterious device drivers and scheduling algorithms, but they themselves introduce artifacts that must be understood. Lastly, stepping into the model-based design level, abstractions become available that are closer in spirit to the way the physical world works, but this sometimes requires thinking about software in ways that are foreign to many students.

Together with the assistance of engineers at National Instruments, we developed a single laboratory platform that enables students to explore design at all three levels of abstraction: bare-iron, RTOS, and model-based. The microcontroller is shown in Figure 4. The microcontroller is a heterogenous multiprocessor system with a rich set of peripherals. Its general-purpose embedded processor is a Freescale 32-bit PowerPC processor running VxWorks, a popular RTOS from Wind River Systems (a wholly owned subsidiary of Intel Corporation). It can be programmed in C/C++, Java, or by models developed in the LabVIEW graphical design environment. The microcontroller also features a Xilinx field-programmable gate array (FPGA) that we pre-configure with a MicroBlaze 32-bit soft-core processor and peripheral devices. The MicroBlaze processor can be programmed in C at the bare-iron level, and the FPGA can be programmed via VHDL/Verilog and LabVIEW models via VHDL synthesis.

We now explain the principal concepts that we teach using these three levels of abstraction.

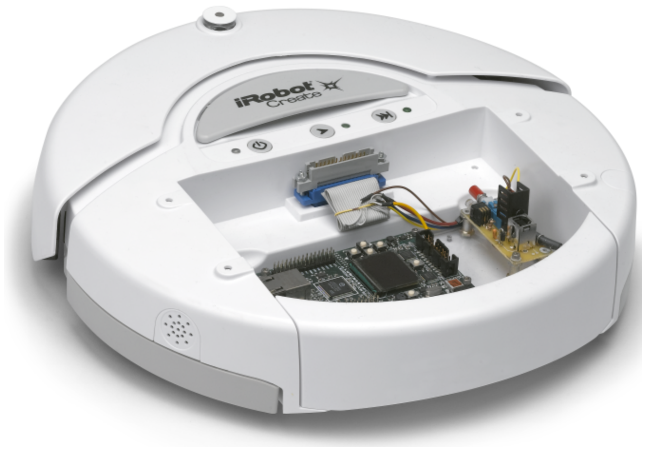


Figure 2: Early prototype of the Cal Climber cyber-physical system. The latest Cal Climber consists of an iRobot Create, a National Instruments Single-Board RIO (sbRIO), and an Analog Devices accelerometer.

2.1 Bare-Iron Programming

A key goal of our course is to demystify what is happening in an embedded system. To accomplish this, we require students to do some bare-iron programming in C, where no operating system or code generator gets between the programmer and the hardware. Moreover, we strive to ensure that students come out of the course with confidence that

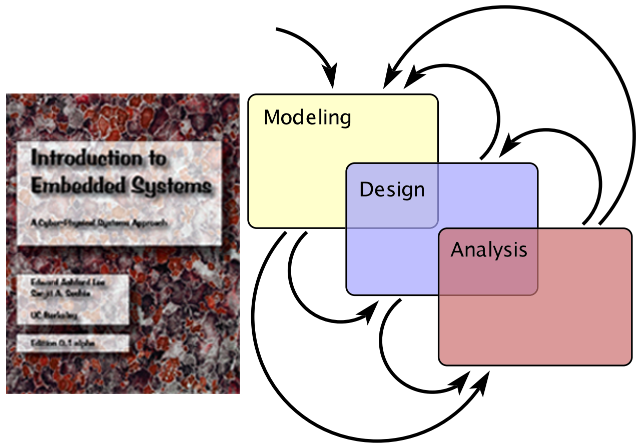
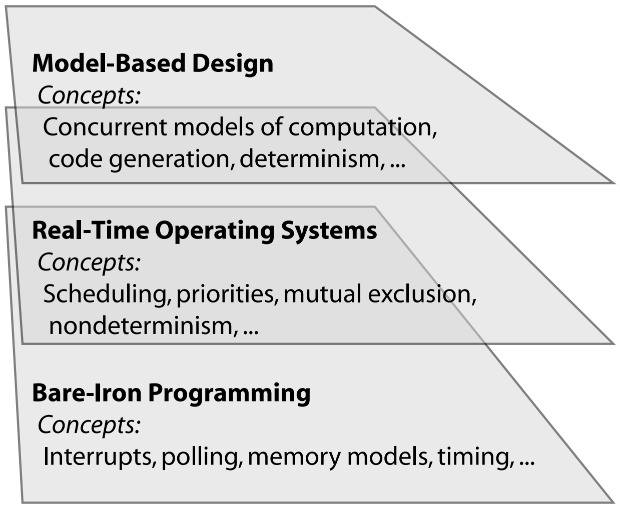


Figure 1: Lee & Seshia Textbook.


Model-Based Design
Concepts:
Concurrent models of computation,
code generation, determinism, ...
Real-Time Operating Systems
Concepts:
Scheduling, priorities, mutual exclusion,
nondeterminism, ...
Bare-Iron Programming
Concepts:
Interrupts, polling, memory models, timing, ...
Figure 3: Layers of abstraction.

they can master *any* microprocessor, not just the ones they use in the lab. For this reason, in class, we deliberately illustrate the ideas using *different* microprocessors than the ones they use in lab.

Consider programming an Atmel AVR, a widely used 8-bit microcontroller, in C, with no operating system. Available documentation has a "cookbook" style, giving sample code like the following:

```
// Set timer1 to interrupt every 1ms
TCCR1A = 0x00;
TCCR1B = (_BV(WGM12) | _BV(CS12));
OCR1A = 71;
```

For historical reasons, in this community, ASCII characters are deemed to be very expensive, so they are used sparingly. Hence the obtuse acronyms like TCCR, which stands for Timer/Counter Control Register. But the above code snippet has a "Harry Potter" flavor to it; it seems to be saying that if you learn the right spell, and express it with conviction, the machine will do what you want.

Our goal, instead, is to give students the confidence that they can puzzle through mysterious incantations like those above and understand the meaning behind the instructions, how instructions bridge software and hardware, and what will be the net effect. This isn't always easy, since vendors seemingly go to great lengths to obscure inner workings. Our view is that providing students with a laboratory framework that protects them from this obfuscation would be a disservice: we want to instill that with confidence, patience, and discipline, students can explore and eventually fully understand what happens inside an embedded system, from architecture to abstraction.

Exploring the above code in this fashion, first notice that the name Timer/Counter Control *Register* suggests that TCCR1A and TCCR1B might be somehow related to the `register` keyword in C. They are not – the register keyword in C is merely a suggestion to the compiler that an automatic variable, which would normally be allocated on the stack, may be better put into a register. So what are

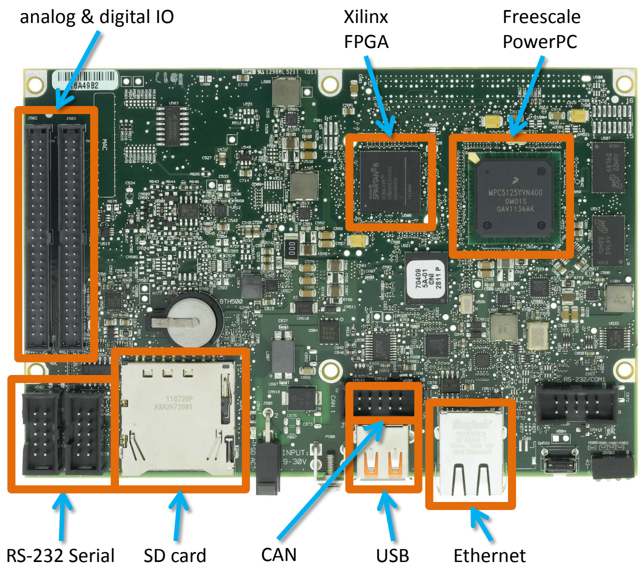


Figure 4: National Instruments Single-Board RIO (sbRIO) 9636.

TCCR1A, TCCR1B, and OCR1A above? They appear on the left side of an assignment, so for this to be valid C, they must be variables, not constants. Since they are controlling a timer, which is a peripheral device, they must be memory-mapped registers, which means that they are variables with a specific fixed memory location. But when you declare variables in C, either as static variables or as automatic variables, you have no control over where in memory they are put. You cannot specify a particular memory address for such variables. Static variables will be positioned in memory by the compiler, and automatic variables will be put on the stack. So TCCR1A, TCCR1B, and OCR1A must not be ordinary C variables. What are they?

Hunting through the header files that must be included for the above C code to compile, we find the following macro definitions:

```
#define _MMIO_BYTE(mem_addr) \
      (*(volatile uint8_t *)(mem_addr))
#define _SFR_MEM8(mem_addr) \
      _MMIO_BYTE(mem_addr)
#define _BV(bit) (1 << (bit))
#define TCCR1B   _SFR_MEM8 (0x81)
#define WGM12    3
#define CS12     2
```

Expanding these C preprocessor macros in the statement from our example code

```
TCCR1B = (_BV(WGM12) | _BV(CS12));
```

yields

```
(*(volatile uint8_t *)(0x81))
                    = (1 << 3) | (1 << 2);
```

If we understand C well enough, we can read exactly what is going on. The left hand side of the assignment is a memory address 0x81 that first gets cast to the data type (`volatile uint8_t *`), a pointer to a volatile unsigned eight-bit integer. The `volatile` keyword in C is a hint to the compiler that the variable may change value independently of code generated by the compiler, preventing certain compiler optimizations that could otherwise result in unexpected use of memory that is shared by an external hardware peripheral and the program. This memory address is then dereferenced by the leftmost * operator, turning the constant memory address into a variable writable in C.

The right hand side of the statement specifies the constant `0x0C`, a byte that has its second and third bits set. These bits are set by shifting the constant 1 left three times and twice, respectively. Puzzling through the documentation, we can discover that setting bit three causes the timer to clear when it reaches the designated count value given by the line `OCR1A = 71` (where OCR stands for Output Compare Register, again economizing on ASCII characters). The effect is that once a counter has reached a specified threshold, it is reset to zero and begins again. Setting bit 2 in TCCR1B specifies a "prescaler" of 256 for the counter, meaning that the counter ticks at 1/256 the clock rate of the processor. If the processor is running at 18MHz, then counter will tick at approximately 18MHz/256 = 71kHz. Hence, setting `OCR1A = 71` will cause the interrupt to occur approximately every millisecond. The spell is demystified.

Students complete the above exercise as part of homework assignments from the textbook. A complementary laboratory exercise on MicroBlaze tasks students with creating music, with the hint that a note can be synthesized over a small speaker driven by a digital pin that toggles to create a square wave of corresponding fundamental frequency. In the laboratory exercises, students face the C expression

```
#define TLR0 \
    (*((volatile int *) \
    (TIMER0_ADDR + 0x04)))
```

as part of the MicroBlaze timer peripheral. Students use these registers to configure timed interrupts, toggle a digital pin, and generate a 440Hz square wave. In later exercises, they learn to periodically read an analog-to-digital converter (ADC) without blocking the processor while a conversion is taking place, a critical component of timed input and output. Lastly, students increase the rate of a timed interrupt until unexpected behavior occurs, demonstrating some of the side-effects of non-sequential programming models.

Our hope is that through this exploration, students learn that the control of a machine is gained not through hacking together of magic spells and code snippits, but rather understanding of its an underlying logic – perhaps several layers – that with persistence can be decoded and used to construct embedded software. Vendors do not make this easy and sometimes obscure mechanisms in an effort to make things easier for the programmer, but a good embedded system designer will always seek to understand the mechanisms rather than just trust incantations. Moreover, if successfully conquered, the above exercise greatly strengthens a student's understand of C as a programming language, the memory model in C (particularly where static and automatic variables live in memory), and the toolchain (particularly the role of the preprocessor, which expands seemingly invalid C code into valid and understandable C code). Similar exercises can strengthen the understanding of the role of a compiler and linker.

In the spirit of encouraging critical thinking, we note from the above example that the C language does not, in fact, provide appropriate abstractions for what we need to accomplish. The fact that TCCR1A, TCCR1B, OCR1A, and TLR0 above are not, and cannot be, C variables, and yet appear in the program as if they were, suggests a mismatch of abstractions. Surely there will be better ways in the future. We encourage students to look for these future improvements, to embrace technological change, and not fear it as is often the case for those who learn to master mechanics alone.

```
#include <stdlib.h>
#include <stdio.h>
// Value that gets updated.
int x;
// Type of the notify procedure.
typedef void notifyProcedure(int);
struct element {
  // Pointer to notify procedure.
  notifyProcedure* listener;
  // Pointer to the next item.
  struct element* next;
};
// Type of list elements.
typedef struct element element_t;
// Pointer to start of list.
element_t* head = 0;
// Pointer to end of list.
element_t* tail = 0;

// Procedure to add a listener.
void addListener(notifyProcedure* listener) {
  if (head == 0) {
    head = malloc(sizeof(element_t));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(element_t));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
// Procedure to update x.
void update(int newx) {
  x = newx;
  // Notify listeners.
  element_t* element = head;
  while (element != 0) {
    (*(element->listener))(newx);
    element = element->next;
  }
}
// Example of notify procedure.
void print(int arg) {
  printf("%d ", arg);
}
```

Figure 5: A C program with subtle threading problems.

2.2 RTOS Programming

Once students understand the mechanics of timer interrupts and the C memory model, it is straightforward to explain how multitasking works in an operating system. When working with C programs under an RTOS like VxWorks, interactions with the physical world are mediated by device drivers. A little understanding of the bare-iron level is sufficient to understand some of the pitfalls associated with timing disruptions due to interrupts and missed events due to disabled interrupts. The benefits of elevating the level of abstraction to that of an RTOS come at the cost of a host of new issues that must be understood, including thread scheduling, priority inversion, race conditions, mutual exclusion, and scheduling anomalies.

In the spirit of encouraging *critical thinking*, we consider an example program shown in Figure 5, which is a C version of an example discussed in [4]. This program implements an extremely common design pattern called the observer pattern [5]. In this pattern, a static variable `x` may be updated by multiple threads. Each time it is updated, a number of registered listeners must be notified by calling a procedure that the listener has specified by calling `addListener()` from some thread. As shown in the figure, the code is not thread safe, and in general could result in a corrupted linked list storing the list of listeners.

We use this example to introduce mutual exclusion (using the `pthreads` library). But the most obvious solution using mutexes has a deadlock risk. Fixing the deadlock risk turns out to be extremely treacherous. One seemingly obvious and simple technique involves copying the list of listeners while holding a mutex, then releasing the mutex before notifying the listeners. However, this fix has a nasty insidious error that may never show up in testing. In particular, it becomes possible (albeit improbable) for listeners to be notified of updates to the variable `x` *in an order opposite to the updates* *that occur.*

We pose a scenario where a sensor is measuring engine temperature, and if the temperature exceeds some critical threshold, it updates the variable `x` accordingly. A listener running in another thread is updating a cockpit display for a pilot, say. If the listener is notified of events in the wrong order, the cockpit display could be stuck indefinitely in a state indicating that the temperature is fine, when in fact the temperature is critical. A subtle insidious bug that never showed up in testing becomes life threatening. For details of this example, see [1], Chapter 11.

For such examples, we show how constructing models of the system is more effective than testing for detecting and understanding such potential errors. In particular, the models we focus on are concurrent compositions of state machines, where we analyze both deterministic and nondeterministic composition methods and show how systematic state-space exploration can reveal problems hidden to testing alone.

2.3 Model-Based Design

The third layer of abstraction in Figure 3 is model-based design. Our approach is described in [6, 7]. To get experience with model-based design, students use LabVIEW to program the Cal Climber robot in Figure 2. LabVIEW compiles graphical representations of structured dataflow and Statecharts, or the imperative language MathScript, and deploys the resulting binary on the embedded microcontroller.

The iRobot Create is modeled in the LabVIEW Robotics Environment Simulator, a physics-based ordinary differential equation solver with a 3D rendering engine (Fig. 6). The simulator publishes physical quantities of interest, including measurements of the iRobot sensors and an analog accelerometer. The simulator executes a control algorithm developed by students using the Statecharts model of computation (Fig. 7). The same Statechart is deployed to the real robot without modification, introducing students to a model-based design methodology.

The solution to the hill climb problem is fundamentally different when developed in LabVIEW because of the inherent concurrency of structured dataflow. Like the RTOS solution, the LabVIEW solution hides the low-level architecture interactions, but the style is quite different. Instead of calling procedures of device drivers, the model-based solution has a parameterized block representing an I/O device that produces a stream of sensor data. In Figure 8, for example, you can see the accelerometer blocks at the upper left with "wires" connecting them to the control logic. The model-based approach also approaches timing differently, as timing is a primitive construct in LabVIEW. Our goal is to broaden the thinking of students so that they don't leave with the notion that the one way they have learned to do this is the only way (or even the best way).

3. MODELING PHYSICAL SYSTEMS

Consistent with the cyber-physical systems theme of the course, a significant component of the course involves modeling the physical world and conjoining those models with models of the cyber world. In the lab, we include two specific efforts in this direction. First, students study an Analog Devices accelerometer. Referencing the sensor datasheet, students translate voltage signals into meaningful physical units such as g-forces. The students study the problem of disam-

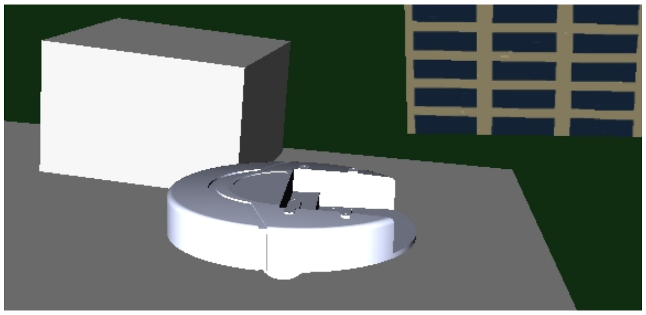


Figure 6: Cal Climber simulation in LabVIEW.

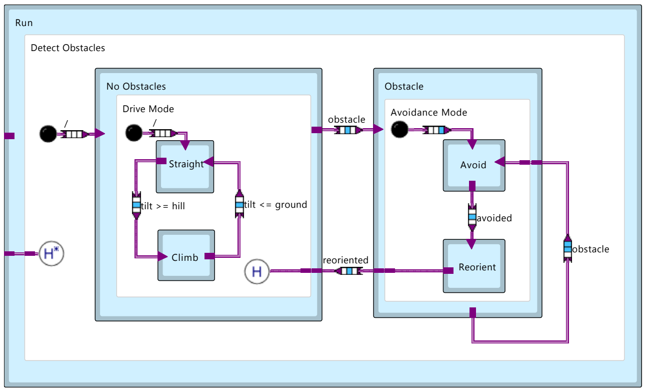
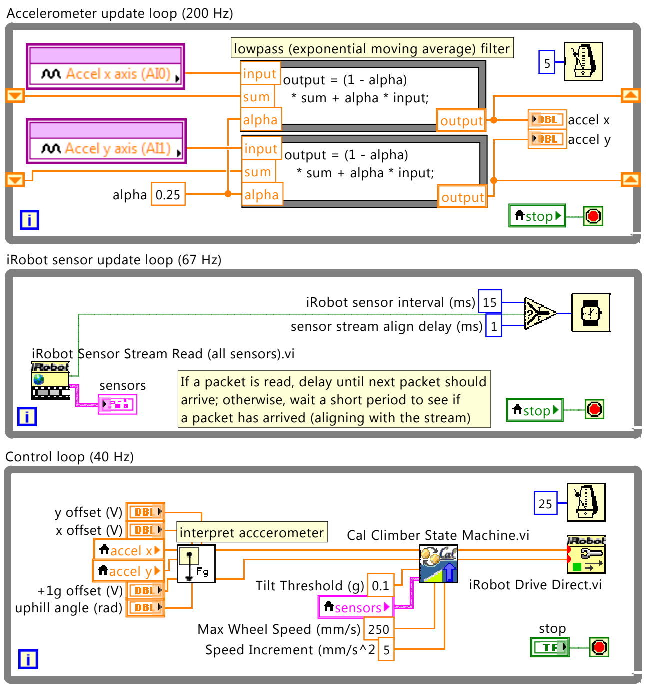


Figure 7: Cal Climber control algorithm in LabVIEW.


Accelerometer update loop (200 Hz)
Accel x axis (AI0)
Accel y axis (AI1)
lowpass (exponential moving average) filter
input
sum
alpha
output
alpha 0.25
5
accel x
accel y
stop
iRobot sensor update loop (67 Hz)
iRobot sensor interval (ms) 15
sensor stream align delay (ms) 1
iRobot Sensor Stream Read (all sensors).vi
sensors
stop
Control loop (40 Hz)
y offset (V)
x offset (V)
accel x
accel y
+1g offset (V)
uphill angle (rad)
interpret acccerometer
Cal Climber State Machine.vi
Tilt Threshold (g) 0.1
sensors
Max Wheel Speed (mm/s) 250
Speed Increment (mm/s^2 5
iRobot Drive Direct.vi
25
stop

output = (1 - alpha)
  * sum + alpha * input;

If a packet is read, delay until next packet should arrive; otherwise, wait a short period to see if a packet has arrived (aligning with the stream)

Figure 8: Cal Climber sensor and actuator processing in LabVIEW.

biguating coordinate acceleration from gravitational force, how to calibrate a sensor using an affine function model, and how to use an accelerometer to measure pitch and roll.

The second effort in modeling of physical systems extends the understanding of the accelerometer to how it can interact with both cyber and physical systems. As students design control algorithms to navigate the Cal Climber to the top of a hill, an inevitable early design leads them to explore further the problem of disambiguating coordinate acceleration from gravitational force: the measurement of tilt depends not only on the position of the robot, but also acceleration induced by the control algorithm they design.

4. CAPSTONE PROJECT

After six weeks of structured lab exercises that expose students to the three levels of abstraction, the remaining nine weeks of the course are dedicated to capstone design projects. Each project culminates in a demonstration session and poster presentation, and satisfies a senior design requirement for graduation from the department of Electrical Engineering and Computer Sciences. Though we encourage students to choose from a list of suggested projects, many students opt to design their own projects from scratch, using concepts learned throughout the course. Videos from project presentations can be found at `http://chess.eecs.berkeley.edu/eecs149`.

Capstone design projects are kicked-off with a session on project management led by an experienced project manager. The goal is as much to buoy student success as it is to prepare them for interactions with project managers in industry. Students submit a one-page charter that is an overarching project specification; they later submit a project plan of action, a timeline for milestones, and a division of responsibilities. Projects must apply at least two of the following concepts: concurrency, modeling of physical dynamics, reliable real-time behavior, modal behavior governed by finite state machines coupled with formal analysis, real-time networks, simulation strategies, and design methodologies for embedded systems design.

Team are paired with a mentor with relevant experience who is a professor, graduate student, researcher, or industry professional. Progress is checked weekly, alternating between in-class presentations and one-page milestone reports comparing progress to the original project charter. Halfway through the project, students are asked to host live demonstrations during a department open-house, which encourages students to achieve functional milestones. Students periodically submit peer evaluation forms to identify any issues with a particular team member; to our surprise, students are often more critical of themselves than their teammates are of them. Project management is a crucial component of the capstone design project; we refer the reader to Koopman [8] for his complementary treatment of software engineering practices used in embedded systems courses.

The more successful projects incorporate many aspects of embedded design taught in the course, but tend to place a significant focus in one specific area of design. Projects that have too broad a focus, or focus on more than one core concept, are often held up by issues such as mechanical construction or control algorithm design. Students who choose an embedded microcontroller simply because they are comfortable with it often find they have to overengineer a problem because of a limitation; more careful consideration in selecting a microcontroller would have greatly reduced this. One team, for example, used a microcontroller compatible with an open-source driver for the iRobot Create, only to discover it lacked an external communication port. The students "engineered around" the problem by emulating a UART serial port in software; we found this to be a worthwhile effort that demonstrated several concepts taught in the course, however, the students viewed this as a costly effort that prevented them from completing some of their project objectives. We have not yet found a way of guiding teams to appropriately define and scale projects in a fashion that naturally emphasizes a small set of core concepts without incurring mechanical setbacks or the need to over-engineer.

4.1 Example Projects

We summarize a selection of projects below pursued over the last five years of course offerings. Videos of all of these are available at `http://chess.eecs.berkeley.edu/eecs149`.

WiiCegBot: One notable project from the Spring 2008 course offering was a self-balancing two-wheeled robot driven by the Nintendo WiiMote. The low-level control algorithm for the robot was programmed in C on a bare-iron microcontroller. The accelerometer data from the WiiMote went to a LabVIEW model on a PC via bluetooth. A supervisory controller was implemented in LabVIEW, which communicated wirelessly with the robot. The project incorporated networking, wireless communication, and control; however, the primary focus of the project was real-time computing. The students were Abraham Liao, Ashik Raj Manandhar, Yoav Peeri, and Derek Tia.

Autonomous Helicopter: One project from the Spring 2009 offering involved a remote-controlled helicopter platform that was re-engineered to perform an autonomous hover. The project was centered on the control design to achieve this, specifically a proportional feedback controller implemented in LabVIEW, which performed carefully calibrated gain scheduling, with a focus on using the tail rotor to stabilize the helicopter. The project also handled software implementation issues such as synchronization of concurrent components and restricting the use of floating-point to optimize code size and performance. The students were Emily Cheng, Jason Cuenco, Keaton Chia, Terry Liu, and Vivian Chu.

Decentralized Pacman Game: This project from the Spring 2009 offering is a decentralized version of the Pacman game played with NXT Lego robots. A centerpiece of the project was the finite-state strategies for Pacman and the "ghost" robots, implemented using concept from the modal modeling content in the course. Another important aspect involved discretizing physical dynamics so as to map moves of Pacman and ghost robots to driving "unit" distances in the maze. A related aspect involved accounting for sensor inaccuracies in maze navigation. The students were Michelle Au, Brian Lam, Ingrid Liu, and Pohan Yang.

Robot Vehicle Convoy: One of the more common project topics over the years has involved a robotic vehicle convoy, with a leader vehicle and several followers, typically following in a sequential, train-like fashion. In the Spring 2009 course offering, the lead vehicle was a remote-controlled Rovio robot followed by a sequence of iRobot Creates. Key components of the project included the design of the control of leader and follower robots, typically as finite-state machines, and the sensor design, e.g., using arrays of infra-red light emitting diodes (LEDs). The students were Walter Li, William Li, Kevin Liu, and Daniel Wei.

Robotic Xylophone: One project in Spring 2011 tackled the problem of designing a robotic music player that plays back a given sequence of notes at a specified beat. In particular, the project aimed to construct a robotic xylophone. A key technical challenge was to meet precise timing specifications dictated by the musical piece to be played. The team found that the traditional xylophone design of having a rectangular row of keys was too difficult to achieve reliable timing, and hence came up with an innovative circular xylophone design. Practical issues were also handled such as mechanical design of the arm that moves at a safe yet fast enough speed and with limited noise allowing the music to be heard. Formal analysis was used to determine a possible failure case where a interrupt service routine would infinitely interrupt, preventing the actor controlling the motor from performing any computation. In the end, the system recognizably replayed common melodies such as the Imperial March, Twinkle Twinkle Little Star, and Old MacDonald. The students were Palash Agarwal, Oren Blasberg, Luke Calderin, and Sameep Tandon.

Robotic Lift Operator: Again in Spring 2011, a team constructed a robot that autonomously rode an elevator (Fig. 9). Using the the ClearPath Robotics Husky A100 mobile robot, a National Instruments CompactRIO (cRIO), and a Robotis Bioloid robotic arm, the students used vision algorithms to detect the location of an elevator shaft and its calling button, the robotic arm to actuate the call button, and vision to detect when the elevator had arrived and its door had opened. The robot then selected a floor and used feedback from an accelerometer to determine when the appropriate floor had been reached. The students employed producer/consumer design patterns for communication with the robot and an Android mobile phone. The robot moved remarkably well, though the students encountered a limitation with the robotic arm as it was not powerful enough to actuate elevator buttons. The students were Karthik Lakshmanan, Apoorva Sachdev, Rafael Send, Gayane Vardoyan, and Jacob Dickenson.

4.2 Other Projects

In general, students have been very creative in coming up with project ideas. Two other projects focused on tracking a human face; one of these used a movable camera mounted on a wheeled robot, and another used a low-cost commercially available quadrotor aircraft. Several projects focused on gesture tracking, some using data glove inputs and others using wearable accelerometers and gyroscopes. One project used the Microsoft XBox Kinect vision system to control a mobile robot with a grasping arm. Two projects focused on automatic parallel parking of a vehicle; one of these, interestingly, incorporated a cooperative algorithm where already parked vehicles would make room for a new vehicle.

5. RESULTS

This course is a work in progress, and its actual impact is difficult to measure objectively. Students are generally interested and engaged, celebrating their projects as well as those from other teams. Students are proud of what they accomplish, and often post project presentation videos to the internet. In several cases, the course has profoundly affected the career trajectory of students, including the last author of this paper. A rather unexpected outcome came from another student, now a graduate student at Berkeley, who has applied some of the analysis concepts of the class to the problem of automatically generating problems that are variants of the ones in the text [9]. In some cases, the projects that almost totally failed to meet their stated goals were the best learning experiences; we encourage students to be ambitious, and where they fall short of their goals, to provide an analysis of the reasons.

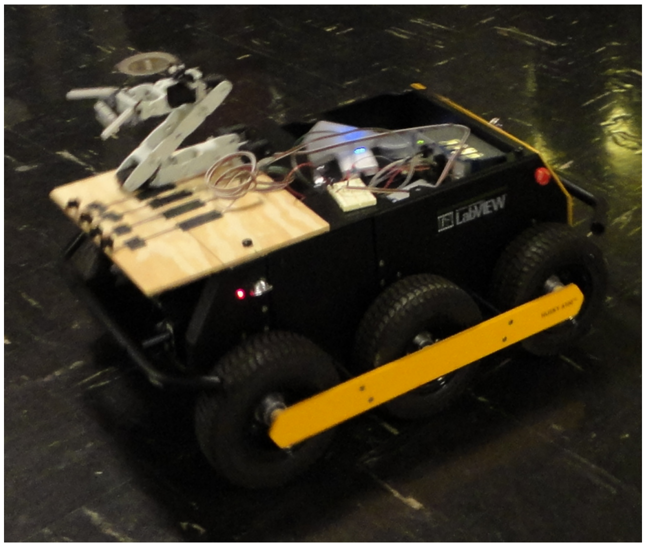


Figure 9: Autonomous robot capable of changing floors via elevator.

Any success we have had with the laboratory platform is not uniquely attributable to the sensors, the robot, or the microcontroller we chose; the value of the Cal Climber is that it offers a simple but illustrative cyber-physical system, and the value of the microcontroller is that sbRIO is programmable in tools developed by Xilinx, Wind River, and National Instruments, exposing students to tools that facilitate design at different levels of abstraction. A kit-based approach to the lab that is commercially supported and costs about the equivalent of a traditional textbook could go a long way towards facilitating export of the laboratory curriculum. The iRobot Create meets many of these requirements, and greatly influenced laboratory development. We seek a kit that offers mobility, sensing, and actuation together with a suitably powerful and adaptable embedded controller, as a platform for model-based design of cyber-physical systems.

Acknowledgment

This work was supported in part by the National Science Foundation awards #0644436 (CAREER: Robust Reactive Systems through Verification and Learning), #0931843 (CPS: Large: Action Webs) and #1035672 (CPS: Medium: Timing-Centric Software), and by National Instruments (especially Trung N. Tran).

6. REFERENCES

[1] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach.*

Berkeley, CA: LeeSeshia.org, 2011. [Online]. Available: http://LeeSeshia.org

[2] S. A. Seshia and E. A. Lee, "An introductory textbook on cyber-physical systems," in *Workshop on Embedded Systems Education (in conjunction with ES Week)*, Scottsdale, AZ, 2010.

[3] J. C. Jensen, E. A. Lee, and S. A. Seshia, "An introductory capstone design course on embedded systems," in *International Symposium on Circuits and Systems (ISCAS)*. Rio de Janeiro, Brazil: IEEE, 2011, pp. 1199–1202.

[4] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[6] J. C. Jensen, D. H. Chang, and E. A. Lee, "A model-based design methodology for cyber-physical systems," in *First IEEE Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems (CyPhy)*, Istanbul, Turkey, 2011. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/837.html

[7] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13–28, 2012.

[8] P. Koopman, *Better Embedded System Software*. Drumnadrochit Education, 2010. [Online]. Available: http://www.koopman.us/book.html

[9] D. Sadigh, S. A. Seshia, and M. Gupta, "Automating exercise generation: A step towards meeting the mooc challenge for embedded systems," in *Workshop on Embedded Systems Education (WESE)*. Tampere, Finland: ACM, 2012.