The Flock: Mote Sensors Sing in Undergraduate Curriculum

Integrating wireless sensor networks in an undergraduate embedded systems course exposes students to an important emerging technology in the core of the computer engineering curriculum.

Bruce Hemingway
Waylon Brunette
Tom Anderl
Gaetano Borriello
University of Washington

Computer engineering curricula have evolved dramatically over the past 20 years. The early focus was on computer architecture and CPU design. In the 1990s, attention shifted to systems built around highly integrated microcontrollers. By 2000, the software for these embedded systems assumed a more important role, and embedded operating systems provided high-level design abstractions. More recently, wireless communication capabilities have greatly expanded the embedded applications space, leading to new system paradigms such as sensor networks.

Educational excellence requires exposing students to the current edge of research. To ensure that student projects are along the same trajectory that the industry is traveling, educators must continually introduce emerging techniques, practices, and applications into the curriculum.

At the University of Washington's Department of Computer Science & Engineering (UW CSE), we have integrated the emerging field of wireless sensor networks into our undergraduate computer engineering curriculum. While many graduate-level classes focus on sensor networks,[1-3] they do not provide a good template for the undergraduate curriculum because they assume a much greater breadth of knowledge than undergraduate students usually have as well as greater maturity to absorb new topics on their own.

Other efforts to introduce undergraduates to this important application area have focused on using sensor networks in capstone project courses that challenge senior students to apply their acquired knowledge and skills to a final project. The UW CSE "Flock of Birds" project integrates the theory and practice of wireless sensor networks into the mainstream curriculum early enough to form a basis for all students' understanding of embedded computing—not just a short-lived application exercise for some capstone design projects. Figure 1 shows one sensor node—the bird is optional—from the project.

COMPUTER ENGINEERING CURRICULUM

The UW CSE computer engineering program relies on a core curriculum shared with the computer science program. It consists of introductory courses in programming, discrete mathematics, data structures, formal methods, comparative programming languages, logic design, and computer architecture. Computer engineering students specialize with additional second-tier courses in electrical engineering, operating systems, networks, embedded software, and digital system design, as well as a capstone design course that takes a product idea from concept to prototype.

The embedded software, digital design, and capstone design courses give the computer engineering

Published by the IEEE Computer Society
0018-9162/04/$20.00 © 2004 IEEE

program its character. They integrate software and hardware design skills and prepare students to build modern digital systems from start to finish.

In the embedded software course, students learn to use microcontrollers and their interfaces effectively to build systems that control physical devices. The digital design course teaches them to program algorithms into hardware. In the capstone design course, the students apply all their skills to a product that we try to make similar to those on which industry engineers are currently working—in other words, products that will appear on the market one or two years after the students graduate.

CSE 466: Software for Embedded Systems

The students in our embedded software course (CSE 466) have completed computer architecture and digital design courses, and most have studied operating systems as well.

This course exposes the students to the design issues that characterize embedded systems. These include constrained resources, such as limited memory space, I/O, and CPU frequency; the absence of an operating system to handle low-level tasks such as interrupts and peripheral device interfaces; and a variety of protocols for communication between components.

Further, embedded systems often require a debugging tool set different from that of the full-scale systems familiar to our students. Print statements are seldom available, nor are breakpoints always an option. Programmers and, in this case, students must find other methods to signal code milestones.

Through prerequisite courses on data structures, digital design, and machine architecture, CSE 466 students have a solid understanding of how the hardware works and how to build efficient data structures for their algorithms. This gives them the foundation they need to write software at the low level required by embedded systems. The students build on these prerequisites with a hands-on project that is designed to lead them through the process of applying what they already know while exposing them to the new issues that arise with embedded systems.

When students finish the 10-week class, they should be comfortable writing code that directly manipulates I/O registers and establishes communications between multiple devices. They should also understand how interrupts work and how to handle them as well as how to interpret the datasheet of whatever chip they decide to work with in the future.

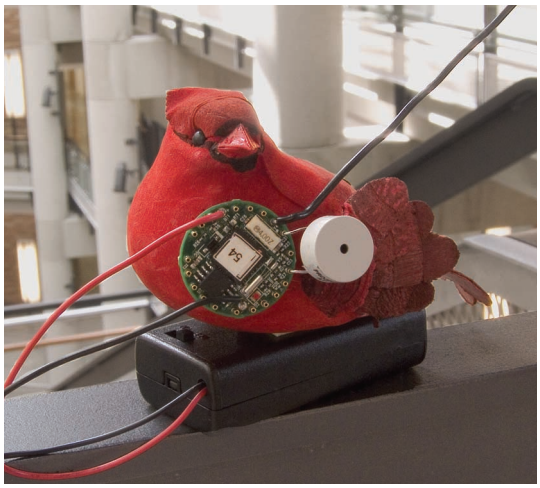

Figure 1. A mote-sensor bird. The "Flock of Birds" embedded systems project integrates the theory and practice of wireless sensor networks into the mainstream classroom curriculum.

Project design for wireless sensor networks

Each edition of CSE 466 must define a project that addresses the complexities of embedded systems within the constraints of a 10-week course. For pedagogical reasons, we want a project that students can complete individually. At the same time, students are generally more engaged by class-wide projects, so we like the individual work to contribute to an overall class effort.

Students are also more interested in projects that use current technology as opposed to obsolete components that they will never see again after they graduate. As both memory and CPU cycles become more plentiful in most of today's microprocessors, designing a project that is resource constrained poses another challenge.

The computer engineering curriculum already used Atmel ATmega AVR-series microprocessors (www.atmel.com) and the AVR-GCC (www.openavr.org) C compiler, so students were familiar with these core materials. To incorporate wireless sensor networks into the curriculum, we selected the Crossbow (www.xbow.com) Mica2Dot platform. The product is an implementation of TinyOS sensor motes (www.tinyos.net), originally created at the University of California, Berkeley.[4] Mica2dot features a standard platform with built-in hardware, the existing TinyOS code base, and a convenient form factor for adding predefined sensors.

The event-based style of TinyOS helped students understand time constraints and code structure by forcing them to write short, nonblocking routines. Its modular design simplified the integration of components like the radio stack, saving countless hours of coding. TinyOS also provided a degree of abstraction within the embedded system context and introduced students to a style of event-based coding that stresses real-time as well as functional issues.

We made sound generation a major focus of the project and the overall class. Students can understand and implement sound generation in a month, yet it taxes a system's cycle and memory capacity enough to make efficiency an important design con-

Figure 2. The Microsoft Atrium in the Paul G. Allen Center for Computer Science & Engineering, University of Washington. The "Flock of Birds" project concluded with a concert in the atrium consisting of 50 "bird" motes that sang songs based on what other birds within radio range were singing. Photo courtesy of Ed LaCasse.

straint. A sample rate in the tens of kilohertz range can generate usable sound. Visual displays such as video on LCDs demand many more computation cycles, but human observers seldom notice visual timing errors on the order of tens of milliseconds. On the other hand, errors of even a few microseconds are discernible in sound generation, giving students quick feedback on program accuracy.

A FLOCK OF BIRDS

The "Flock of Birds" project is a simple distributed system that meets our course objectives by combining sound generation with emergent behavior in an ad hoc network. Each student programs a mote to act as a bird that has several songs stored in its local memory. The programs execute a common rule base, but each bird acts independently—deciding which song to sing based on what the other birds within radio range are singing. In combination, the songs create the sound a flock of birds makes.

The flock is designed to work in any random configuration with any number of nodes. We implemented the project in the Microsoft Atrium of the University of Washington Paul G. Allen Center for Computer Science & Engineering, shown in Figure 2.

The primary goal was subjective: to generate behavior that mimics the effect of birds cooperating to sing the same song but vary the particular song over time. We implemented monitoring software to measure the effect, but the final judgment of success was aesthetic: Does it sound right? The subjective measure contrasts with the usual quantitative measures we used in most of our past projects. A visual display of the data associated with the sound reinforced the aural perception.

The algorithm for generating behavior uses radio packets collected for random amounts of time from other motes. After the time lapses, the mote decides which song to sing based on the data about what songs the neighboring motes are singing. The mote then shuts its radio off, sings its song, turns the radio back on, and transmits a packet announcing which song it just sang.

This algorithm requires using timers, radio communication, and direct manipulation of hardware registers for sound production. Students had to first develop a complex software module on a breadboard and then explore the challenges of porting it to the mote within the TinyOS constraints.

A monitoring node served as the gateway to a control laptop that used special radio packets to start and stop the algorithm and to modify global parameters, such as the min/max limits for the random number generation that determined an individual node's behavior. This added a dynamic aspect to the flock process. For example, we could control the temporal density of bird songs by changing the maximum value for the listen time between song events. Lowering the radio transmit level in the large space of the atrium prevented distant motes from hearing each other and made the flocking effect operate in multiple local areas.

We presented the flock concept to the students in the context of emergent behavior and the basics of cellular automata and gave them the simplified template algorithm shown in Figure 3.

The initial flock specification was purposely incomplete, as we wanted student feedback and participation in completing its definition. This is analogous to what students might experience in the real world, where others have decided on a protocol and high-level algorithm, and the developer's job is to implement a system that realizes the functionality.

The incomplete specification also exposed students to different interpretations of the design documentation. They had to identify the ambiguities and then clarify them either with other developers or with us, "the customer."

We asked students to invent a methodology for predicting the success of this algorithm and to suggest three improvements to it. We incorporated many of these suggestions into the final algorithm to enhance students' experience of participating in the project design. A complete procedural docu-

ment is available on the class Web site (www.cs.washington.edu/education/courses/cse466/03au/).

COURSE WORK

Most students in the course had no prior embedded programming experience. To keep the first few assignments simple, easy to debug, and somewhat familiar, we had the students begin programming with embedded C rather than TinyOS. The assignments introduced some basic embedded systems programming concepts such as decoding binary numbers to pins for a pair of seven-segment displays, using an analog-digital converter to measure sensor voltage and current levels, and writing interrupt handlers for timers.

Breadboard basics

The students compiled their programs with AVR-GCC and uploaded them to an ATmega16 processor on a solderless breadboard. These chips come in a standard dual-inline package that is easy to place on a standard breadboard. The breadboard allowed space for students to add peripheral devices and debug their circuits by connecting probes for oscilloscopes and logic analyzers.

After introducing the basics, the course moved quickly to more specific applications—most notably, sound generation with a piezoelectric transducer. The first sound assignment involved using a wave table to generate sine waves at various frequencies. This introduced the students to the notion of using a fixed sampling rate with a phase increment to control frequency.

Also, because the ATmega microprocessor series has no digital-to-analog converter, the students had to use pulse-width modulation instead to produce analog outputs to drive the piezoelectric transducer. Using PWM for this purpose was new to many students, and few of them saw immediately that they could use a low-pass filter in conjunction with PWM to produce a reasonably accurate DAC. By the assignment's end, however, all the students had at least a basic grasp of both PWM and simple hardware-based low-pass frequency filtering.

For the next assignment, students had to build a system that could play several different songs. The system implemented familiar songs based on cell phone ring tones, ranging from *Für Elise* to *Theme from "The A Team."* Using familiar songs helped the students quickly determine whether their implementation worked. If they recognized the song, they were on the right track; if not, they likely had a problem. Using familiar songs also raised the interest level in the class. This was one of the rare times that computer engineering students could show off their projects to nontechnical friends.

Goals for the flock:
Birds sing the same song for a little while.
Songs start, spread, then die out.
Over time, different songs emerge as dominant for some period of time.
Flock process flow:
1. Initialization tasks; select x = random(0-15).
2. Radio off; sing birdsong[x]; radio on.
3. Listen for Random(min1, max1) sec.
4. SendMessage "I sang song x".
5. Listen for Random(min2, max2) sec.
6. Decide which song to sing next:
   a. Determine nearest songs.
   b. If my song is the same as any of the nearest songs, then I'll repeat the same song.
   c. If all nearby motes are singing the same song, then I'll switch to a different song.
   d. If all nearby songs are distinct, then I'll switch to a different song.
7. Go to step 2 and repeat.

Figure 3. Goals and simplified algorithm for each bird in the flock.

Sound generation uses both wave and sequencing tables, which require large amounts of memory—more than main memory could accommodate. Because the standard method of simply putting these tables in RAM as static arrays was not possible, students had to put the data in program space. This was slightly less convenient because the lookup required special calls. However, it introduced students to standard practice for handling static data in an embedded environment. This reinforced the concept of memory mapping and resource allocation.

The final assignment on the ATmega16 was to play 16 different bird songs. The songs were in MIDI format, which is slightly different from the previous format and required the students to convert to a storage format of their own devising. However, the sound-generation portion of the assignment was very similar to the earlier assignment, so students were able to reuse much of the code they had already written.

Students were not as familiar with the birdsongs as they were with the tunes from the previous assignment. By playing the MIDI files on their PC and comparing the sound with their implementation's, they were able to get their code up and running quickly.

The ATmega128 processor that the Mica2dot motes use is very similar to the ATmega16, so much of the code from this assignment was reused as a component in the final project.

Motes and TinyOS

Once the students were comfortable with basic embedded-systems programming issues, we introduced motes and TinyOS. TinyOS is based on an event-driven paradigm instead of the polling paradigm that was standard for previous student assignments. Some students had been exposed to the

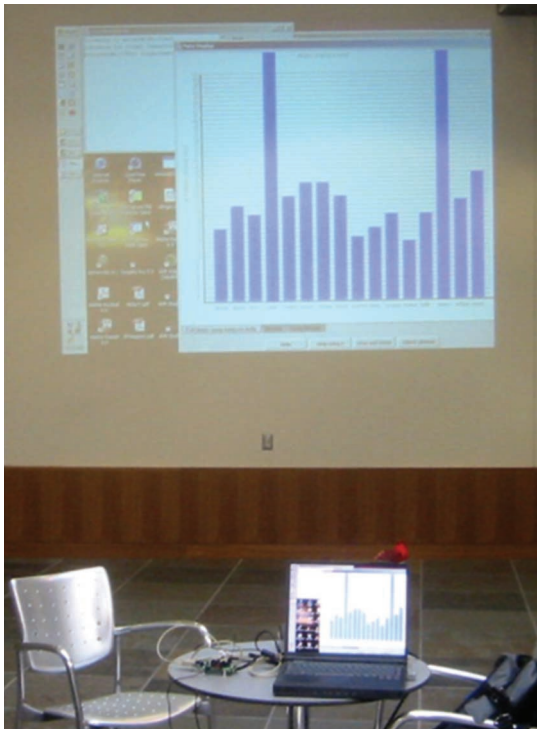

Figure 4. "Flock of Birds" monitoring laptop and wall display. The frequency of different bird songs was projected in real time during the concert.

event-based paradigm through operating system or user-interface programming classes, but many of them had not.

Students spent the first few weeks learning what a component, module, and interface were and how to wire modules together. To add to the confusion about event-driven programming, the students struggled to learn NesC (http://nescc.sourceforge.net/), an extension to the C programming language designed at UC Berkeley for TinyOS.[5]

We used tutorials from the TinyOS Web site and simple assignments to introduce this subject to students. After a rough start, students began to see how quickly they could build programs with considerable functionality and only a little custom code connected to existing software components.

Many students used a 15.625-kHz sampling rate on the Crossbow Mica2dot motes. Running at the standard 4 MHz, the system outputs samples every 256 cycles. This forced students to code their sampling algorithms carefully so that the processing could finish within the 256 cycles.

Debugging the Mica2dot motes presents interesting challenges. Traditionally, students debug their programs by embedding print statements in their code or using interactive debuggers with breakpoint and inspection capabilities. The Mica2dot has one light-emitting diode for use in feedback. While TinyOS includes a simulator, many of the mote components that control hardware, such as the radio and sound-generation components, cannot be effectively debugged with the simulator. Students could use some print statements with the serial port while the mote is on the programmer, but they had to be more inventive when debugging in the wireless communication environment.

Given that the flock algorithm runs on a mote among many other motes running the same algorithm, each node receives large amounts of input that the test cycle must simulate to emulate all the other nearby nodes. This required a substantial test fixture that could produce large numbers of packets at predetermined times and then report on the packets sent by the mote under test.

Instead of having the students implement the entire test fixture, we provided a fixture that took as input a simple description of packets to send and the time delay between them and sent these to the mote under test. This test fixture handled all of the necessary low-level interactions and allowed the students to write arbitrary and repeatable tests. This freed them from having to worry about the details of sending packets from a computer to the mote under test.

Writing these tests also helped the students to better understand the timing details of the communication protocol. The students not only had to figure out how to generate a comprehensive test for the protocol but also how to create situations that would produce specific desired responses. Where they lacked understanding or the documentation lacked detail, they had to chase down their bugs systematically.

Concert exam

The conclusion of the project was a two-hour "concert" of 50 motes in the Allen Center Atrium. Students had to qualify their birds for admission by passing a special test designed to exclude rogue birds from the flock. Well-behaved birds graduated to the Allen Center atrium for the performance. Students reprogrammed failed motes with code from motes that did pass, allowing everyone to participate in the concert.

For the performance, we had to modify the testing program to send specific packets that would trigger the motes to begin their process. The underlying system was already implemented, so we simply built a GUI to it. The GUI also allowed us to monitor the motes and project a real-time wall display of the frequency of different birdsongs the motes were singing. Figure 4 shows a snapshot of the central monitoring laptop and projected wall display during the concert.

We used the existing TinyOS SerialForwarder component to put control motes at various places in the room and have them forward their received packets to the monitoring laptop via 802.11 wireless Ethernet. In this way, we could turn the trans-

mit strength down very low on the bird motes while still ensuring that the monitoring system recorded every packet sent.

The overall sound effect was pleasing, and even worked in three dimensions when some students moved their birds to upper balconies. The aural feedback contrasted with more conventional projects that require extended data analysis to understand the results. The results were also easier to understand than many simulation schemes.

The project succeeded from an instructional perspective—integrating communication protocols, constrained resources, hardware control, and a novel application that required student projects to interact. The motes supported a project that would otherwise have been too complex to implement over a 10-week quarter.

PROJECT CHALLENGES

TinyOS is a work in progress. The large-scale installation in our laboratories was daunting, particularly relative to conflicts with preinstalled tools such as Cygwin and Java. These tools are used systemwide by many more classes and students than our immediate group, and we spent much staff time solving configuration problems. These problems will change with each new TinyOS version, so this aspect of the course will remain time-consuming until the tools and installation procedures mature.

The tutorials provided with TinyOS helped to introduce the students to the basics. However, students could implement the tutorial examples by simply copying code or following instructions by rote. To make sure students understood the underlying concepts, we gave an exam to test their knowledge. This motivated the students to take the time necessary to learn the concepts. After the exam, many students claimed that TinyOS was simple once they understood it. Future versions of the tutorials might benefit from scattering conceptual questions along the way to help students focus on the high-level structures of TinyOS in addition to its mechanics.

Some nondeterminism in our specification often made it difficult to accurately predict whether a mote under test was going to be listening at any given time. Because of this, our testing required us to provide very clear cases in which several dropped packets would not affect the outcome. We also had to allow for some packets to be dropped when transmitting them back from the node under test.

We determined that a fully automated testing system for grading would not be feasible. Instead, we scripted seven tests totaling roughly 10 minutes, after which the staff ran through the data by hand. This was not a problem, however, since each test produced only 5 to 20 packets from the mote—a small number that was easy to examine.

The test program represented a large undertaking for the staff. It totaled more than 4,000 lines of Java even though it used the SerialForwarder component shipped with TinyOS. The program nevertheless proved quite useful to both staff and students, enabling bombardment of a mote with large numbers of random packets as well as just a few carefully chosen packets.

We have considered making several changes to this project if we use it again in the future. First, the sound quality of the piezoelectric transducers, while sufficient, was not exactly high fidelity. The choice was based on its extremely low power requirement, but in other operating environments this might not be a consideration. Research into other transducers or speakers could enable a wider repertory of sounds. For example, we have considered including insects, rain, other animal sounds, and whispered voices, along with contexts and rules sets for emergent behavior to fit the sound types.

We might add input sensors. Light sensing could allow the virtual entity to react to the diurnal cycle of ambient light. Proximity sensors could support behavior modifications according to the movements of people near a mote.

Expanding the project to include rechargeable batteries and power management could allow a life cycle that would require activity only when power was sufficient, in the style of BEAM robots (www.nis.lanl.gov/projects/robot/).

Controlling light-source color and intensity could add visual interest and another level of complexity.

From the instructional viewpoint, a vital ingredient of the course is having an experienced TinyOS programmer on the staff. Until the accompanying teaching materials for motes and TinyOS mature, this expertise will be an important condition in attempts to scale the project. ■

References

1. A. Chien, "Programming Sensor Networks," graduate course CSE 291, Univ. California, Davis, 2003; www-csag.ucsd.edu/teaching/cse291s03/.
2. M. Beigl et al., "Teaching a Practical Ubicomp Course

with Smart-Its," poster at Ubicomp 2002; www.teco.edu/~michael/publication/smart-it_ubicomp.pdf.
3. M. Welsh, "Wireless Communications and Sensor Networks," graduate course CS 263, Harvard Univ., 2004; www.eecs.harvard.edu/~mdw/course/cs263/fa04/.
4. J. Hill et al., "System Architecture Directions for Network Sensors," *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS-IX), ACM Press, 2000, pp. 93-104.
5. D. Gay et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. Programming Language Design and Implementation* (PLDI) 2003, ACM Press, 2003, pp. 1-11.

Bruce Hemingway *is a lecturer in the University of Washington's Department of Computer Science & Engineering and manager of its W.T. Baxter Computer Engineering Laboratory. His research interests include the development of autonomous computer-music generators that will sense mood from physiological indicators and produce music pleasing to the listener. Hemingway received an AB in music from Indiana University. Contact him at bruceh@cs.washington.edu.*

Waylon Brunette *is a research engineer and teaching associate in the Department of Computer Science & Engineering at the University of Washington. His research interests include mobile and ubiquitous computing, wireless sensor networks, and personal area networks. Brunette received a BS in computer engineering from the University of Washington. Contact him at wrb@cs.washington.edu.*

Tom Anderl *is a teaching assistant in computer engineering at the University of Washington. His research interests include application-specific hardware and resource-limited computation. Anderl received a BS in computer engineering from the University of Washington. Contact him at tanderl@cs.washington.edu.*

Gaetano Borriello *is a professor of computer science and engineering at the University of Washington. His research interests are in ubiquitous computing, principally new hardware devices that integrate seamlessly into the user's environment with particular focus on location and identification systems. Borriello received a PhD in computer science from the University of California, Berkeley. Contact him at gaetano@cs.washington.edu.*