New Orleans JAZZED about Engineering Education

Paper ID #15723

Learning Materials for Introductory Embedded Systems Programming Using a Model-Based Discipline

Prof. Frank Vahid, University of California - Riverside

Frank Vahid is a Professor of Computer Science and Engineering at the Univ. of California, Riverside. His research interests include embedded systems design, and engineering education. He is a co-founder of zyBooks.com.

Dr. Alex Daniel Edgcomb, Zybooks

Alex Edgcomb finished his PhD in computer science at UC Riverside in 2014. Alex has continued working as a research specialist at UC Riverside with his PhD advisor, studying the efficacy of web-native content for STEM education. Alex also works with Zyante, a startup that develops interactive, web-native textbooks in STEM.

Dr. Bailey Alan Miller, Zyante Inc.

Bailey Miller is a software engineer at Zyante Inc., and formerly worked as a software engineer at Space Exploration Technologies Corporation (SpaceX). He received his B.S. in Computer Engineering, and his M.S and Ph.D. in Computer Science, from the University of California, Riverside in 2009, 2011, and 2014, respectively. His dissertation research focused on embedded systems design and novel highly-parallel many-core computer architectures. He has published more than 10 research papers, and served as a university lecturer in computer science.

Prof. Tony Givargis, University of California - Irvine

Tony Givargis is a Professor of Computer Science and Associate Dean in the School of Information and Computer Sciences at the University of California, Irvine. He received his B.S. and Ph.D. in Computer Science from the University of California, Riverside in 1997 and 2001, respectively. He conducts research in the area of embedded systems with an emphasis on embedded systems and software and has authored over 90 peer reviewed papers. He is a named inventor on 11 U.S. patents and has co-authored two popular textbooks on embedded system design. Professor Givargis has received numerous teaching, service, and research awards, including the Frederick Emmons Terman Award, presented annually to an outstanding young electrical engineering educator by the Electrical and Computer Engineering Division of the American Society for Engineering Education.

Learning Materials for Introductory Embedded Systems Programming using a Model-Based Discipline

Frank Vahid, Computer Science & Engin., Univ. of California, Riverside (also with zyBooks)
Alex Edgcomb, zyBooks, Los Gatos, California (also with UC Riverside)
Bailey Miller, zyBooks, Los Gatos, California
Tony Givargis, Department of Computer Science, University of California, Irvine

Abstract

Embedded systems have changed dramatically in recent decades. At their start in the 1970's and growth in the 1980's, embedded systems consisted of relatively simple microcontroller hardware, often programmed in low-level assembly language, to configure a few peripherals and interact with a few input/output pins. Today, improvements in speed, memory size, and power have changed the emphasis from hardware to software, with microcontrollers supporting tens of thousands of lines of code, perhaps programmed in C, often with concurrent tasks, interacting with dozens of peripherals and potentially hundreds of input/output pins in time-multiplexed manner, and dynamically changing among numerous power states. Yet, many university embedded systems courses and textbooks still look similar to those in the 1980's, emphasizing low-level programming of hardware. Little guidance is provided to teach students in a first course how to program using higher-level disciplined methods. The result is an improper foundation and perspective, leading to ad hoc unstructured code, or over-reliance and inefficient use of real-time operating systems. To remedy the situation, we describe a model-based discipline for introductory embedded systems courses, developed over the past decade at the University of California, Riverside and University of California, Irvine. The discipline has students describing desired behavior using synchronous state machines, and capturing those state machines using standard C templates. As such, students develop software paying strict attention to timing issues, and create their own C-based task scheduler, leading to a solid understanding of concurrency and real-time operating system functionality. This paper highlights that discipline, and describes new web-based interactive learning material emphasizing the discipline, which is best used for the lecture portion of a course having an accompanying microcontroller lab. The paper also introduces web-based tools that support the discipline.

Introduction

The embedded systems field is a relatively new engineering field and is thus evolving rapidly. In the 1980s, most embedded systems were primarily low-complexity devices like telephone answering machines or simple factory automation equipment. Embedded systems courses and textbooks began to appear in universities, originally using names like microprocessor-based system design. Emphasis was placed on configuring and/or interfacing with the peripherals like serial communication hardware, pulse width modulators, keypads, and displays. Programming originally focused on the assembly language of a particular microprocessor or microcontroller, evolving to C or other high-level languages in the 1990's/2000's.

Today, even relatively-simple embedded systems in practice may consist of tens of thousands of C code. However, introductory courses and textbooks mainly still focus on configuring and interfacing with peripherals, with little guidance provided to students on how to write programs that are elegant, robust, and scalable. The result is that much embedded systems code, including much commercial code, follows no particular programming discipline, is prone to bugs, and is hard to maintain. Many commercial embedded systems projects fail to become products, or experience failures in the field, as a result[1]. Some universities have intermediate or advanced courses that introduce real-time systems programming including multi-task programming, but many embedded systems students do not have access to such courses, or have already developed bad programming habits. In any case, such courses still may not teach a discipline for how to program each task.

Beginning about 10 years ago, researchers at the University of California, at Riverside and at Irvine, with support by the National Science Foundation, began developing new materials and accompanying simulators intended to bring model-based discipline (well known by the research community) into first courses on embedded systems[4,6]. The materials introduce model-based design just weeks into a semester of a first embedded systems course, using state machines. The researchers developed a special form of state machines intended to be easy to understand by beginners yet sufficiently powerful for commercial use, utilizing C-based semantics. Students think in terms of state machines, and then translate to C via a simple process. Initial emphasis is on capturing desired behavior correctly, then choosing periods to sample inputs and generate outputs appropriately. As desired behavior becomes more complex, students learn to use multiple

concurrent state machines (tasks) to keep distinct behaviors distinct, and to communicate and synchronize among tasks. Students learn to manually translate such multiple tasks to C, and to write a simple task scheduler in C as well, serving well for many commercial systems, as well as providing an excellent foundation for subsequent introduction to real-time operating systems (providing a solid understanding of what is "under the hood"). The material aims to help make introductory embedded systems a serious engineering discipline, with lecture material covering the "theory" of model-based programming (making use of the new material and simulator), and labs covering the "practice" of configuring, interfacing, and programming an actual microcontroller on a breadboard.

The material has been in active use at the University of California at Riverside and the University of California at Irvine for over 7 years, used by several thousand students there so far. The material was published in 2014 as a zyBook[10], and since then has also been used at about 20 other universities, and that number is growing. Contrary to fears that students in a first embedded systems course would not be able to handle model-based design, the result has instead been that students have excelled, and are able to build larger and more complex systems in just one semester, and with fewer bugs. Senior design projects have also become more sophisticated. Student evaluations of the courses have also been very high.

Synchronous state machines

Embedded researchers have long known that the sequential program computation model alone is insufficient for serious embedded systems design. Researchers have developed a variety of computation models for embedded systems, such as synchronous dataflow, statecharts, Kahn process networks, codesign finite-state machines, SpecCharts, SpecC, and many more[2], with many such models having explicit timing constructs. While used in high-end embedded systems, most mainstream embedded systems designers do not use, and often are unaware of, such models. Designers instead program directly in C (or even assembly), using ad hoc timer techniques for dealing with time. When requiring concurrent tasks or time tasks, some designers use a real-time operating system (RTOS), but often do so haphazardly or inefficiently, due in part to a lack of understanding of a real-time scheduler's underlying functionality.

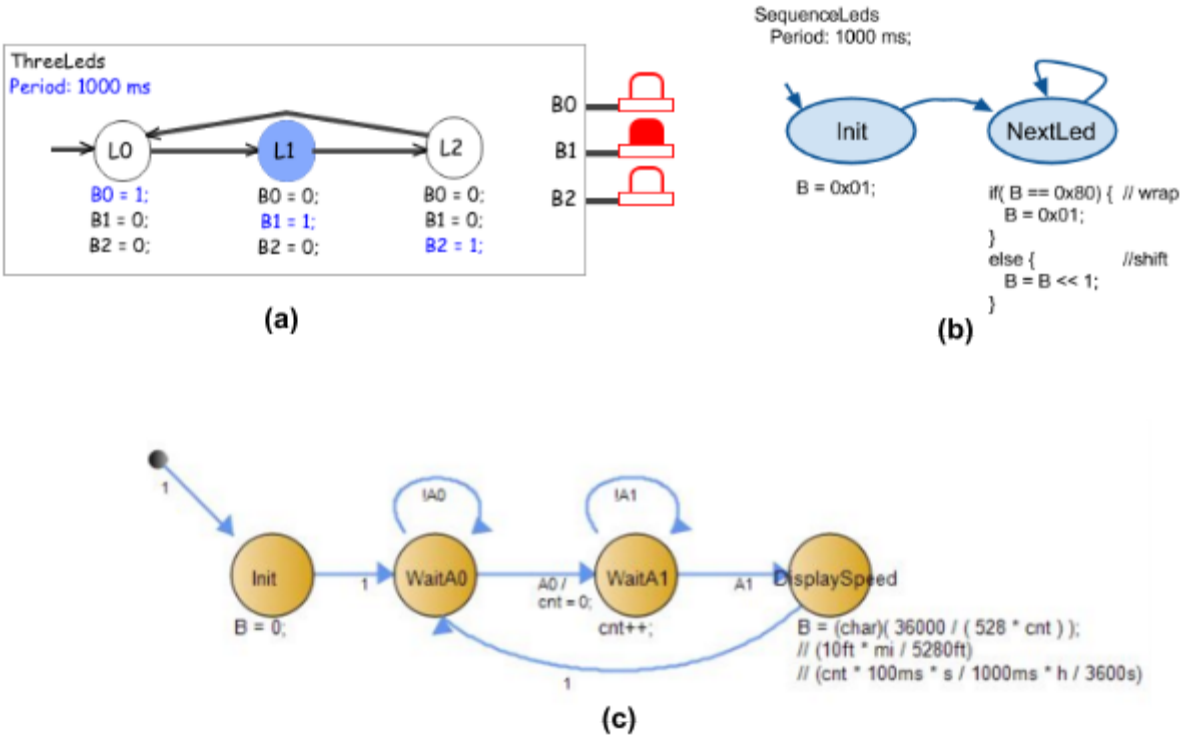
One reason long-existing models haven't been incorporated into mainstream design is that such models are sometimes too complex to understand by a novice. Furthermore, the models were developed without care to ensure a straightforward translation to C exists, so resulting C code to capture those models is complex and hard to read or maintain.

For control-dominated systems that primarily respond to and generate events, we developed the *synchronous state machine (synchSM) model*. The model uses C syntax and semantics for variables, actions, and conditions. A synchSM has a period. A *tick* occurs at the end of each period, and is defined as:

- Examining the transitions from the synchSM's current state and transitioning to the appropriate next state (and execution any actions on that transition), and
- Executing the actions of that next state, after which the tick is complete.

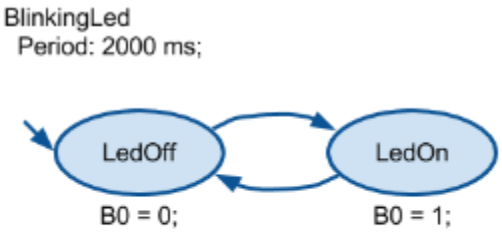
Figure 1 provides three examples. Figure 1(a) shows a simple system that lights one of three LEDs in a sequence, one at a time. Fig 1(b) shows a similar system, but for eight LEDs, and instead using C's bit-shifting capability to set 1 bit in 8-bit output B. Fig 1(c) shows a system that computes the speed of a car passing over two input sensors A0 and then A1 separated by 10 feet.

Figure 1: Synchronous state machines naturally capture behavior of various embedded systems.



A synchSM is captured in C using a standard template for a synchSM's tick function, as shown in Figure 2. Any transition conditions would simply become if-statements within each case of the top switch statement. In main(), the C code configures a microcontroller's hardware timer to the

Figure 2: A synchronous state machine is straightforwardly implemented in C on a microcontroller having a timer.




```c
#include "RIMS.h"

volatile unsigned char TimerFlag=0; // ISR raises, main() lowers

void TimerISR() {
   TimerFlag = 1;
}

enum BL_States { BL_SMStart, BL_LedOff, BL_LedOn } BL_State;

void TickFct_Blink() {

   switch ( BL_State ) { //Transitions
      case BL_SMStart:
         BL_State = BL_LedOff; //Initial state
         break;
      case BL_LedOff:
         BL_State = BL_LedOn;
         break;
      case BL_LedOn:
         BL_State = BL_LedOff;
         break;
      default:
         BL_State = BL_SMStart;
         break;
   }

   switch (BL_State ) { //State actions
      case BL_LedOff:
         B0 = 0;
         break;
      case BL_LedOn:
         B0 = 1;
         break;
      default:
         break;
   }
}

void main() {
   B = 0; //Init outputs
   TimerSet(2000);
   TimerOn();
   BL_State = BL_SMStart; // Indicates initial call to tick-fct
   while (1) {
      TickFct_Blink();      // Execute one synchSM tick
      while (!TimerFlag){}  // Wait for BL's period
      TimerFlag = 0;        // Lower flag
   }
}
```

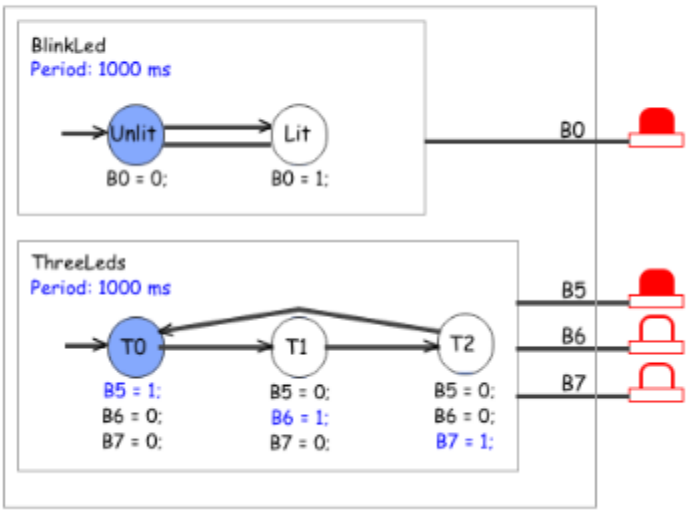
synchSM's period. In main's while(1) loop, the C code calls the tick function, then waits until the timer interrupt service routine is called, as detected by a flag, which main then resets. The key here is that the timer is used in a standard manner. No delay loops or other timing is allowed; all timing is via ticking states. If a period is 2000 ms and a designer wants to wait 10,000 ms, the designer's state machine must wait five states (either five distinct states, or using a variable, actions, and transitions to repeat the same state five times).

To support the above implementation in C, actions in every synchSM state must be run-to-completion; no looping that waits on external values is allowed within a state's actions. Any such waiting must be done using the state and transition capabilities of a synchSM (as in Figure 1(c)), and not using loops within a state's actions.

Concurrent synchSMs

With synchSMs, concurrency is achieved straightforwardly. The designer just describes behavior using multiple synchSMs. Figure 3 illustrates two synchSM's, one blinking an LED 1-sec on, 1-sec off, the other sequencing three LEDs 1-sec on each.

Figure 3: Concurrency is naturally captured as concurrent synchronous state machines.



In C, the concurrency is translated into sequential calls to the synchSM tick functions, as in Figure 4.

Figure 4: Concurrent synchSM's are straightforwardly implemented in C using round-robin scheduling.

```
#include "RIMS.h"

// LedShow C code, having two tasks

volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}

enum BL_States { BL_SMStart, BL_LedOff, BL_LedOn } BL_State;
void TickFct_BlinkLed() {
   ... // Standard switch statements for SM
}
enum TL_States { TL_SMStart, TL_T0, TL_T1, TL_T2 } TL_State;
void TickFct_ThreeLeds() {
   ... // Standard switch statements for SM
}

void main() {
   B = 0; // Init outputs
   TimerSet(1000);
   TimerOn();
   BL_State = BL_SMStart;
   TL_State = TL_SMStart;
   while (1) {
      TickFct_BlinkLed();     // Tick the BlinkLed synchSM
      TickFct_ThreeLeds();    // Tick the ThreeLeds synchSM
      while (!TimerFlag){}    // Wait for timer period
      TimerFlag = 0;          // Lower flag raised by timer
   }
}
```

Because each synchSM cannot have waiting within a state's actions (which instead must be run-to-completion), then each synchSM tick function represents a *cooperative task*. Thus, a simple round-robin calling of the tick functions achieves concurrency.

If periods differ among the concurrent synchSMs, a simple solution exists. The hardware timer is set to the greatest common denominator of the periods. Then, the

main code uses variables to call each synchSM tick function at the appropriate multiple of timer ticks, as in Figure 5.

Figure 5: Differing synchSM periods is handled by simple counting.

```
// LedShow C code
#include "RIMS.h"

volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}

enum BL_states { BL_SMStart, BL_LedOff, BL_LedOn } BL_state;
void TickFct_BlinkLed() {
   ... // standard switch statements for SM
}

enum TL_states { TL_SMStart, TL_T0, TL_T1, TL_T2 } TL_state;
void TickFct_ThreeLeds() {
   ... // standard switch statements for SM
}

...
void main() {
   unsigned long BL_elapsedTime = 1500;
   unsigned long TL_elapsedTime = 500;
   const unsigned long timerPeriod = 100;

   B = 0; //init outputs
   TimerSet(timerPeriod);
   TimerOn();
   BL_state = BL_SMStart;
   TL_state = TL_SMStart;
   while (1) {
      if (BL_elapsedTime >= 1500) { // 1500 ms period
         TickFct_BlinkLed(); // Execute one tick of the BlinkLed synchSM
         BL_elapsedTime = 0;
      }
      if (TL_elapsedTime >= 500) { // 500 ms period
         TickFct_ThreeLeds(); // Execute one tick of the ThreeLeds synchSM
         TL_elapsedTime = 0;
      }
      while (!TimerFlag){}   // Wait for timer period
      TimerFlag = 0;         // Lower flag raised by timer
      BL_elapsedTime += timerPeriod;
      TL_elapsedTime += timerPeriod;
```

In fact, a task structure can be created as in Figure 6. Note that one of the structure's fields is a function pointer to a synchSM's tick function.

Figure 6: A task structure created in C helps manage synchSMs.

```
typedef struct task {
   int state;                  // Task's current state
   unsigned long period;       // Task period
   unsigned long elapsedTime;  // Time elapsed since last task tick
   int (*TickFct)(int);        // Task tick function
} task;
```

With such a structure, a simple for loop can iterate through a tasks array, and call the tick function of any ready tasks, as shown in Figure 7. Such code essentially represents a non-preemptive task scheduler that underlies a typical RTOS, but written fully in C and requiring only a few lines of code beyond the designer's main functionality. Several versions of such a scheduler (including a preemptive version), known as RIOS[3], are freely available at http://www.riosscheduler.org/, and have been downloaded thousands of times by students and engineers.

Figure 7: With a task structure and an array of such tasks, a simple for loop carries out task scheduling.

```
// For each task, call task tick function if task's period is up
for (i=0; i < tasksNum; i++) {
   if (tasks[i].elapsedTime >= tasks[i].period){
      // Task is ready to tick, so call its tick function
      tasks[i].state = tasks[i].TickFct(tasks[i].state);
      tasks[i].elapsedTime = 0; // Reset the elapsed time
   }
   tasks[i].elapsedTime += tasksPeriodGCD;
}
```

Low-power software is then almost trivially achieved by moving the scheduler code into the timer's ISR, and having main's while(1) loop merely go to sleep, as in Figure 8. The while(1) loop will put the microcontroller into a low-power sleep mode. The timer will wake the microcontroller and jump to the timer ISR, causing the task scheduler to call any ready tasks, after which control returns to the while(1) loop which immediately goes back to sleep.

Figure 8: Low power is achieved straightforwardly by moving the scheduler code into an ISR, and having main() repeatedly call sleep().

```
void TimerISR() {
  unsigned char i;
  for (i = 0; i < tasksNum; ++i) { // Heart of the scheduler code
     if ( tasks[i].elapsedTime >= tasks[i].period ) { // Ready
        tasks[i].state = tasks[i].TickFct(tasks[i].state);
        tasks[i].elapsedTime = 0;
     }
     tasks[i].elapsedTime += tasksPeriodGCD;
  }
}

int main() {
  unsigned char i=0;
  tasks[i].state = TL_SMStart;
  tasks[i].period = 500;
  tasks[i].elapsedTime = tasks[i].period;
  tasks[i].TickFct = &TickFct_ThreeLED; //function TickFct_ThreeLED not shown

  TimerSet(tasksPeriodGCD);
  TimerOn();

  while(1) {
     Sleep();
  }
  return 0;
}
```

A designer is thus taught to use the largest possible periods for each synchSM while achieving sufficient reactivity to input events, in order to maximize sleep time and thus minimize power.

Other topics

The above model-based discipline is central to the material, but the material covers additional topics, including:

- Bit-level manipulation in C
- Input/output topics, such as sampling rates, glitching, expandable I/O, and latency.
- Peripherals: Pulse-width modulation, UARTs, and analog-digital conversion
- Embedded programming issues: Lookup tables versus functions, fixed-point programming.
- Utilization and scheduling: Worst-case execution time, utilization computations, jitter, preemptive scheduling.
- FPGAs: Converting synchSMs to hardware desciption languages for synthesis to FPGAs.
- Domains: The material includes one chapter introductions to each of three key embedded system domains: Control (in particular, PID control), digital signal processing, and pattern recognition.

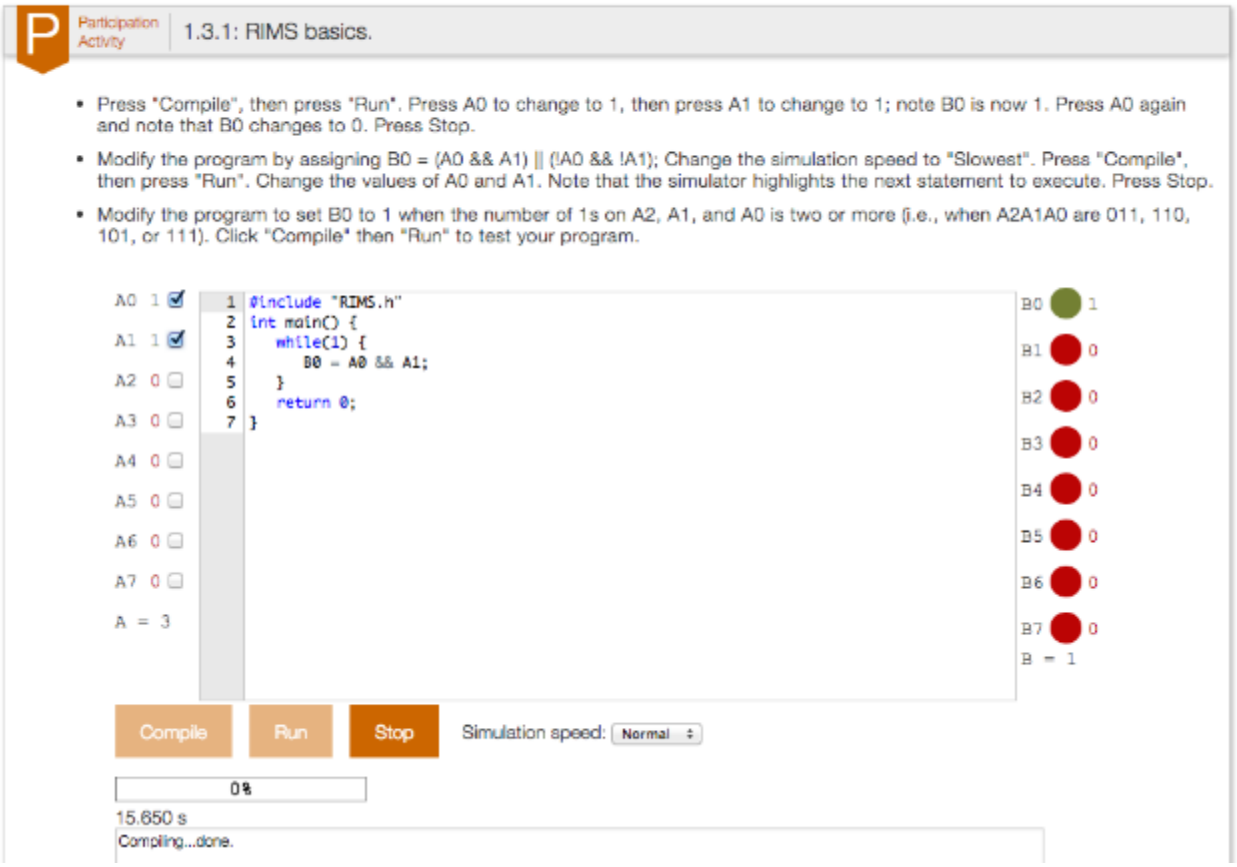
RI tools for lecture component

Too often, the lecture portion of embedded systems intro courses focuses on microcontroller details. Such focus fails to teach students that embedded systems is a serious software discipline. While such details obviously are important, we believe they should either be relegated to the lab portion of a class, or taught in clearly denoted "lab details" segments within lecture time. Lecture time instead can focus on the above topics. To support those lecture topics, we have developed several web-based tools (initially released in 2009 as Windows-based tools, but we have migrated much of the functionality to the web now). The *Riverside-Irvine Microcontroller Simulator*, or *RIMS*, is a virtual microcontroller having a simple 8-input 8-output structure, with

each input accessed via C global variables A0, A1, ..., A7, and outputs as B0, B1, ... B7. All eight inputs can be accessed as A, and all eight outputs as B. Ex: A0 = 1, A7 = 0, or A = 0xFF.

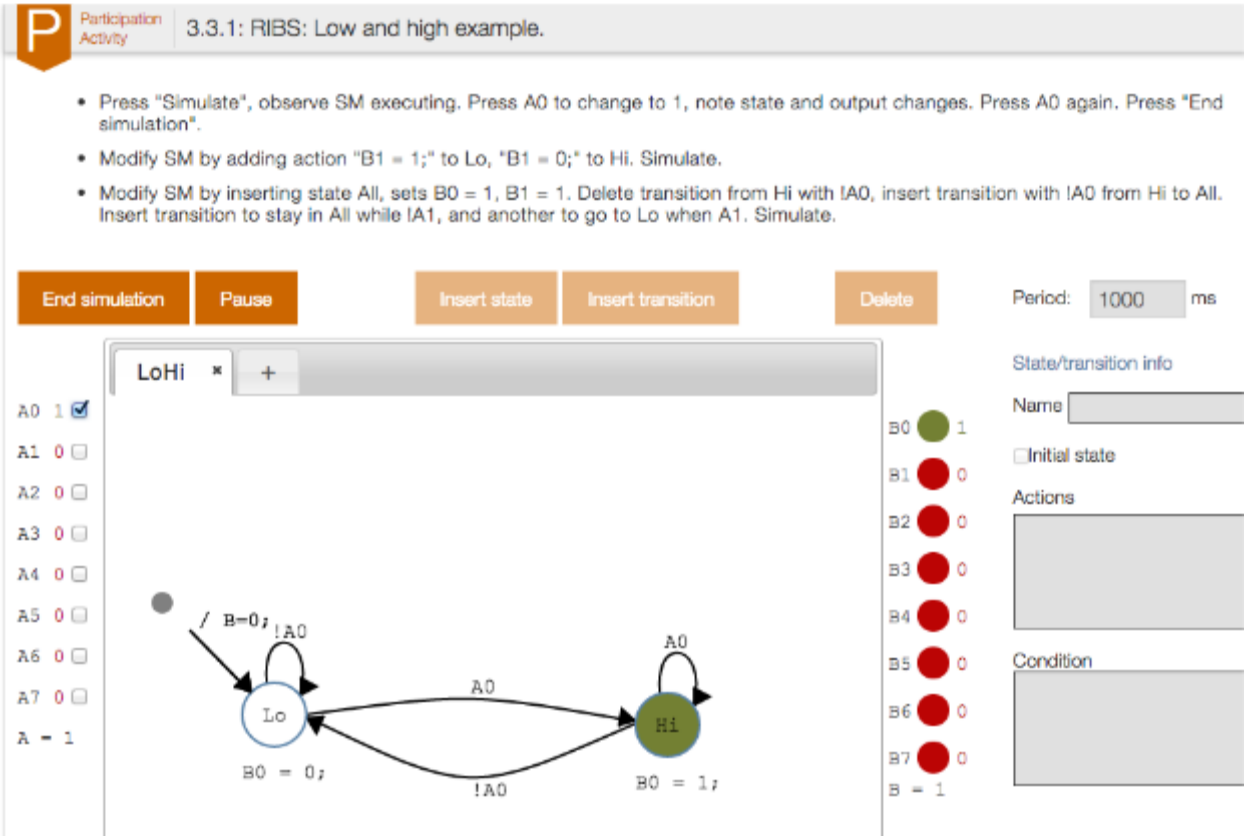
The simulator (which includes a C compiler) actually runs remotely on Amazon Web Services (AWS); the student interacts with a light web-client window (see Figure 9) to that simulator. The simulator supports standard stop and step behavior, and can be run faster or slower than real time (with the slowest speed useful when explaining concepts during lecture).

Figure 9: The RI simulator allows students to program a virtual microcontroller directly in a web-browser, enabling focus on concepts rather than non-essential microcontroller details and application installation.



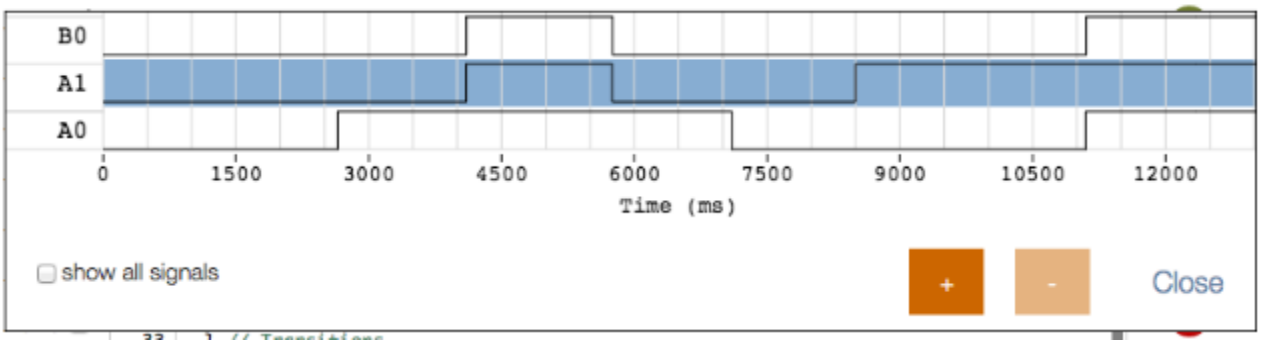
We also provide a state machine capture and simulation tool, known as *RIBS* (Riverside-Irvine Builder of State machines), depicted in Figure 10. Presently, a subset of C is supported for the actions and conditions in the web-based RIBS (the Windows version supported most of C; work is ongoing to expand the subset for the web-based version).

Figure 10: RIBS supports capturing and simulating state machines in a web browser.



A third tool is a timing diagram viewer, shown in Figure 11.

Figure 11: A web-based timing diagram viewer shows results of simulations of RIBS (state machines) or RIMS (C on the virtual microcontroller).



Using the above tools, students can be assigned homework problems initially involving writing C on RIMS, and then evolving towards describing synchSMs. With the Windows version, students can export synchSM files and submit for grading, including timing diagrams and auto-generated C. Such functionality is being developed for the web-based version during the coming year.

Experiences

The model-oriented discipline has been used by thousands of students at University of California, Riverside (UCR) and University of California, Irvine (UCI), and dozens of other universities. Students' majors include computer science, computer engineering, and electrical engineering. While early evaluators of the research worried that students would not grasp state machines in an intro course, the opposite has proven to be the case: Students find state machines highly intuitive (and state machines seem to be less intimidating to electrical engineering students). The UCR and UCI courses each have a programming course as a prerequisite. Student evaluations have been very strong. Feedback from instructors has been outstanding; one professor from a Florida university indicated "You made this class fun for the students, and they are learning more than before too."

At UCR, student completion rates of the online activities have been over 95% for several quarters. Each student completes about 500 activities in the material during a quarter. If such activities had been submitted as homework, grading time would have varied between 1 minute to 5 minutes per activity per student; we estimate about 2 minutes on average. As each class offering has about 100 students, grading time would have been about 100 * 500 * 2 = 100,000 minutes, or about 1666 hours. Clearly, such an amount of homework could not have been assigned, and thus in the past far less homework was assigned, averaging about 100 hours of grading for a quarter, or about 1/3rd of each teaching assistant's time. Since switching to this material, the TA's have time freed up to instead spend time with students in the lab / office hours / discussion forum, or improving the course itself (developing labs, creating test questions, etc.).

In previous papers, we have reported on randomized controlled studies showing students learn more from such interactive content[7] (16% improvements, with 64% improvement among the initially weakest students), on a cross-semester analysis of four courses at three universities showing that students perform better overall in classes that switch to such material[8] (about a

1/3rd letter grade improvement), and via post-class analysis[9] we found that students spend about 90 minutes/week in the material (averaging 4.5 sessions/week and 20 minutes/session).

At UCR and UCI, end-of-course projects have more functionality than before, and nearly all students have fully functioning multi-task projects (whereas previously, when students would write C code however they chose, non-functioning code was commonplace, especially due to tricky timing issues). At UCR, a second embedded systems course continues the approach, as well as teaching students to use an RTOS (which students firmly understand, having written their own scheduler). Students commonly follow the synchSM model-based discipline in their senior design projects.

Conclusions

The synchSM model-based discipline is suitable for introductory embedded systems courses. The supporting RI web-based tools (RIMS and RIBS) empower instructors to provide homeworks that are clearly distinct from the more-detailed lab activities that often accompany such a class, or allow instructors without such lab setups to still introduce students to embedded systems using the virtual RIM simulator. The RIOS scheduler code, fully in C, is freely available for various uses. The above discipline and the web-based RI tools are published by zyBooks[5], using extensive animations and integrated learning questions throughout. The material is configurable by instructors, and commonly combined with other material on programming in C, digital design, or other subjects.

Acknowledgements

This work was supported in part by the National Science Foundation (CNS1016792, CPS1136146), the Semiconductor Research Corporation (GRC 2143.001), a U.S. Department of Education GAANN fellowship, and the National Science Foundation's Small Business Innovative Research (SBIR) program (1315094 and 1430537). We are sincerely grateful for their support.

References

[1] R. Dewar. The education of embedded systems software engineers: failures and fixes. March 2014, www.embedded.com.

[2] Edward A. Lee and Sanjit A. Seshia, Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition, http://LeeSeshia.org, ISBN 978-1-312-42740-2, 2015.

[3] B. Miller, F. Vahid, T. Givargis. RIOS: A Lightweight Task Scheduler for Embedded Systems. Workshop on Embedded Systems Education (WESE, part of ESWEEK), Finland, Oct. 2012.

[4] S. Sirowy, D. Sheldon, T. Givargis, and F. Vahid. Virtual Microcontrollers ACM SIGBED Review, Vol. 6., Issue 1, 2009.

[5] F. Vahid, T. Givargis, B. Miller. Programming Embedded Systems. zyBooks, 2015.

[6] F. Vahid and T. Givargis. Timing is Everything -- Embedded Systems Demand Teaching of Structured Time-Oriented Programming . Int. Wkshp. on Embedded Systems Education, (WESE), 2008.

[7] Edgcomb, A. and F. Vahid. Effectiveness of Online Textbooks vs. Interactive Web-Native Content, Proceedings of ASEE Annual Conference, 2014.

[8] Edgcomb, A., F. Vahid, R. Lysecky, A. Knoesen, R. Amirtharajah, and M.L. Dorf. Student Performance Improvement using Interactive Textbooks: A Three-University Cross-Semester Analysis, Proceedings of ASEE Annual Conference, 2015.

[9] Edgcomb, A., D. de Haas, R. Lysecky, and F. Vahid. Student Usage and Behavioral Patterns with Online Interactive Textbook Materials, Proceedings of International Conference on Education, Research, and Innovation (ICERI), 2015.

[10] zyBooks, zyBooks.com. Accessed: April 2016.