Timing is Everything – Embedded Systems Demand Early Teaching of Structured Time-Oriented Programming

Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside, USA
Also with the Center for Embedded Computer Systems, UC Irvine
vahid@cs.ucr.edu

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, USA
givargis@uci.edu

ABSTRACT

Computing was originally dominated by desktop and hence data-oriented systems. However, embedded and hence time-oriented systems, which must measure input events or generate output events of specified time durations, or must execute at regular time intervals, are increasingly commonplace. Blinking a light on and off for 1 second represents a "Hello World" example of a time-oriented system. Time-oriented programming differs significantly from the more common data-oriented programming, and developing correct maintainable time-oriented programs is challenging. The current situation of embedded courses being senior-level courses hampers effective teaching of time-oriented programming, as early-learned programming habits can be hard to break. Early freshmen or sophomore-level introduction of time-oriented programming, involving the right balance between abstractions and resource awareness, may provide a better foundation. A clean microcontroller with a timer, coupled with the synchronous state machine computation model, can provide such a balance.

1. INTRODUCTION

The Rise of Embedded Systems

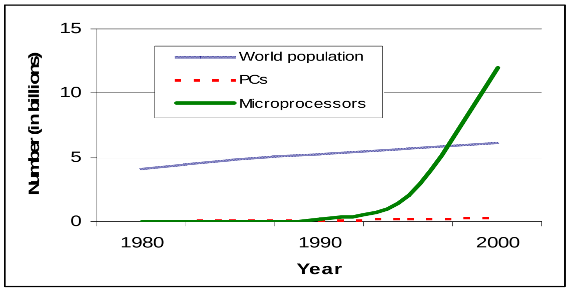
Embedded systems include computing systems that interact extensively with physical real-world devices. Examples are consumer electronics (cameras, cell phones, portable games), automotive electronics (cruise control, navigation), communications equipment (base stations, network routers), factory automation equipment (robotics, sensors, inventory control systems), office automation equipment (scanners, copiers, printers), medical devices (pacemakers, ventilators, ultrasound machines), home automation (security systems, temperature control, smart appliances), and much more. The decreasing cost and size and the increasing performance of computing chips, following Moore's Law, have led to a dramatic proliferation of embedded systems in recent decades, as indicated by the tremendous growth in numbers of microprocessors worldwide shown in Figure 1. Novel embedded systems applications are introduced at a rapid rate, including items like smart ingestible pills, household robots, and bodily-worn health monitoring networks. Of the approximately 150,000 U.S. patents granted per year, roughly 10,000-20,000 are embedded systems related [14].

Current Courses: Details, Details, Details

The teaching of embedded system programming, however, has progressed only moderately in the past two decades, and lacks a solid discipline. Many courses from the 1980s were oriented around an embedded processor chip known as a microcontroller, which has a small low-cost microprocessor coupled with key embedded processing features including program-accessible input/output pins, timers, serial communication devices (UARTs), analog-digital converters, and other peripherals. These small resource-constrained devices required detailed assembly programming and electronics knowledge to set up and use the microprocessor chip and its various peripherals. Many such courses have since replaced assembly programming by C programming and use some libraries to elevate the programmer's focus, and many have adopted the "embedded systems" name. However, most such courses continue to emphasize microcontroller-specific programming and electronics details necessary to get a basic embedded system working, with little attention paid to higher-level embedded system design concepts. This is akin to driver education emphasizing how to add gasoline to a car, check the tires, adjust the mirrors, start the car, and go forward and stop, rather than emphasizing higher-level concepts like how to maintain defensive distances, approach intersections, or plan routes.

Instructors are not to blame for the low-level emphasis. Setting up and running embedded systems courses is *hard*. Unlike desktop computing courses in which fairly standard and stable platforms and tools exist, embedded systems courses must deal with a rather chaotic technical landscape. Dozens of widely-used microcontroller families exist, including 8-bit devices like the Intel 8051, Motorola 68HC05, 68HC08, 68HC11, Microchip PIC, Atmel AVR, Zilog Z80, and much more. For each family, dozens and sometimes hundreds of variations exist (different numbers of pins, size of on-chip RAM, support for external memory, etc.), produced by tens of different companies. 16-bit and 32-bit devices are also available and have similar variety. For a chosen device, a physical programming device (which downloads machine code into the physical chip) must be found. Assemblers and cross-compilers must be found that target the particular device. Development boards must be made or purchased that support the device, such that the device can interface with buttons, switches, LEDs, displays, and other

Figure 1: Growth in number of microprocessors worldwide in past decade due to growth in embedded systems market.



components. Simulators or debuggers may be incorporated. Lab assignments must then be developed around this multitude of items, working around the various items' bugs, sensitivities, or other imperfections (of which many exist). Hardware parts suffer damage (e.g., a microcontroller chip may burn out just by inserting it backwards into a programmer device, or an ageing programmer device may fail to consistently program chips correctly), requiring continual troubleshooting and correction. Furthermore, microcontrollers are tough devices to work with – due to historical artifacts or mass-production needs, the devices tend to require extensive configuration for a particular purpose. Due to each device or tool having a smaller audience than a PC or Windows-based C compiler, documentation is usually scarce (and is sometimes wrong), and simulation and debug tools are scant and often with bugs too. New chips, platforms, operating systems, compilers, debuggers, and even companies providing such items, come and go every few years.

Given the challenging nature of building working embedded systems, instructors often focus primarily on teaching the student the details of how to use the myriad software and hardware tools, to configure and use the microcontroller and each of its key peripherals, to understand how interrupt service routines interact with a main program, and other details necessary to build functioning systems. Figure 2 gives some idea of the challenge, showing the initialization code required to set up a particular microcontroller for a particular usage; note the many distinct items that must be configured (and one small mistake may cause the system to fail). After that teaching process, which may take months, a course may have just enough time left for a student to build an interesting project, with little or no time for teaching a discipline of embedded systems programming. If a second embedded systems course does exist, the course is typically a project course rather than a course that teaches a disciplined embedded programming approach.

Figure 2: Sample C initialization code for a particular microcontroller – extensive knowledge of details is necessary to properly configure a microcontroller for a particular use.

```
// -----------------------------
// configure output ports
// -----------------------------
ADCON0 = 0x00; // disable A/D converter
CM1CON0 = 0x00;
CM2CON0 = 0x00;//disable comparators */
ANSELH = 0x00;
ANSEL = 0x00; // configure pins as digital channels
TRISA = 0x08; // all bits output except RA3
TRISB = 0xF0; // Port B inputs
RABPU = 1;
WPUB4 = 1;    // enable weak pull ups on RB4
 IOCB4 = 1;   // enable interrupt on change for RB4
TRISC = 0x00; // PORTC all set to outputs
PORTA = 0x00;
PORTB = 0x00;
PORTC = 0x00; // initialize ports
// -------------------------------
// Timer0 setup
// -------------------------------
CLRWDT();   // turn off watch dog timer
OPTION = 0x07;    // setup prescaler
TMR0 = PRELOAD; // preload timer
T0IE = 1; //enable timer0 interrupts
// --------------------------------
// Setup button interrupts
// --------------------------------
RABIE = 1; //Enable change on PORTB interrupts
GIE = 1;       //global interupts enabled
```

As evidence of the low-level focus on modern embedded systems courses, consider the top-selling books in the embedded systems textbook market of 16,000 books per year, as reported by John Wiley and Sons: (1) The HCS12/9S12, An Introduction to Hardware and Software Interfacing, Huang, Delmar Cengage, 2005 – 12%; (2) The 68HC12 Microcontroller: Theory and Application (2nd edition of earlier book: Embedded Systems Design and Applications with the 68HC12 and HCS12), Barret and Pack, Prentice hall, 2004 – 9%; (3) Software and Hardware Engineering: Motorola M68HC12, Cady and Sibigtroth, Oxford University Press – 7%.; (4) Embedded Microcomputer Systems: Real Time Interfacing, Valvano, Int. Thomson Publishers, 2006 – 7%; (5) Microcomputer Engineering, Miller, Prentice Hall, 2003 – 6%; (6) Computers as Components: Principles of Embedded Computing System Design, Wolf, Morgan Kaufman, 2005 – 6%; (7) Embedded System Design: A Unified Hardware/Software Introduction, Vahid and Givargis, John Wiley and Sons, 2001 – 4%. Books 1-5 all focus on the details of particular microcontrollers. Some do address higher-level concepts, but typically do so late and rather lightly. Books 6 and 7 (the latter authored by ourselves) sought to introduce a higher-level discipline to embedded system design (in contrast to emphasizing programming).

Because learning the myriad details of microcontrollers, interfacing, troubleshooting, etc., require rather sophisticated students, embedded systems courses are typically taught at the senior level, as illustrated in Figure 3. All of the above books are typically used in senior-level courses, and items 6 and 7 sometimes in a second embedded systems course or even graduate course. We performed a Google search for "embedded systems" in .edu sites; the first 20 courses we found were all upper-division or graduate level, with typical names being "Real Time Embedded Systems" or "Introduction to Microcontrollers."

Thus, a student of a modern embedded systems course may develop a rather myopic view of embedded programming, viewing it as a collection of low-level details and methods necessary to configure and use a microcontroller. "High-level"

Figure 3: Typical embedded system courses are senior-level, emphasizing myriad details. Adding time-oriented programming *early* in the training can provide a better foundation for such courses.

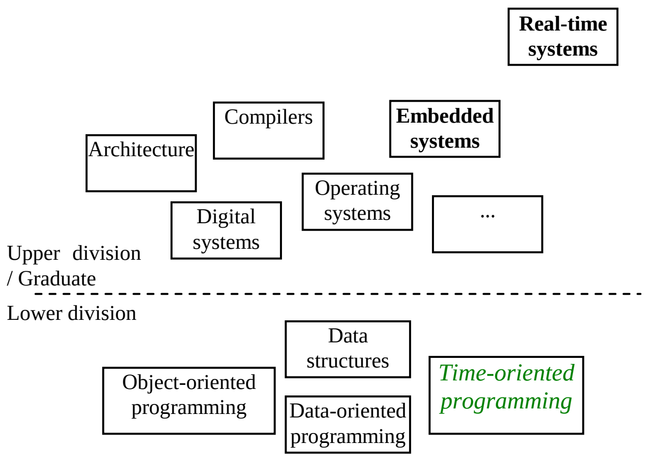
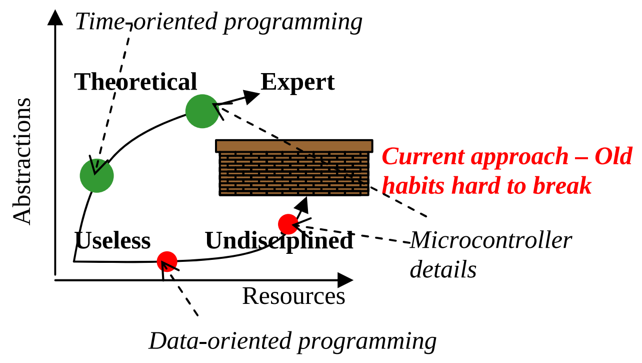


Figure 4: Current curricula teach years of data-oriented programming and then microcontroller details – leading to "undisciplined" embedded programmers having a hard time overcoming the wall of data-oriented programming habits. First teaching time-oriented abstractions establishes a theoretical foundation that may lead to better embedded programmers.


Time-oriented programming
Theoretical
Expert
Abstractions
Current approach – Old habits hard to break
Useless
Undisciplined
Microcontroller details
Resources
Data-oriented programming

may merely mean using C rather than assembly language. The net result is that today's professional embedded systems programmers create code that is amazingly ad hoc, being exceptionally difficult to maintain and often taking a long time to develop. The lack of discipline may explain in part why the majority of embedded systems project are completed late, on average 4 months late for the typical 14 month project [10]. Various companies with whom we interact, including Cisco, Broadcom, Western Digital, Pulmonetics (medical ventilators), Qualcomm, Freescale, Microsoft, and others, have commonly indicated that computing graduates, even those with computer engineering degrees and/or with embedded systems course and project backgrounds, lack the ability to program embedded systems due to "unawareness of resources," "no programming discipline," "inability to deal with time," "a habit of hacking," or even "excessive focus on objects or libraries," requiring "long periods of training" before such graduates can write good embedded code.

2. STRUCTURED TIME-ORIENTED PROGRAMMING

Old Habits are Hard to Break

Given the increasing importance and complexity of modern embedded systems, a more disciplined view of embedded programming is becoming essential (our definition of "disciplined" will be explained subsequently). We have been experimenting with the introduction of disciplined embedded programming methods for many years, and have concluded that an approach that attempts to introduce disciplined methods after an initial low-level microcontroller-details introduction, which is better of course than no introduction of disciplined methods at all, nevertheless is a sub-optimal approach. The reason is because students have spent several years learning data-oriented programming, leading to a perspective and a set of habits that can be hard to change, as illustrated in Figure 4. Instead, we claim that an approach that first introduces disciplined embedded programming methods and later teaches necessary low-level microcontroller details will lead to embedded system designers developing better programs, by creating an improved foundational perspective within the student.

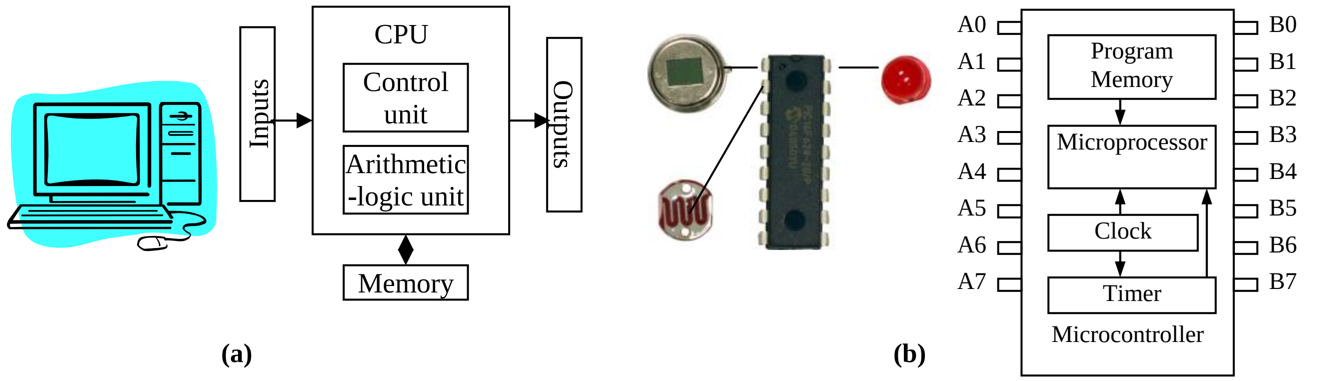
After over a decade of teaching embedded systems in various ways and interacting with dozens of embedded systems teachers and courses worldwide, the authors view *structured time-oriented programming* as one of the key features of a disciplined embedded programming approach. A characteristic of embedded systems, which distinguishes them from traditional desktop (including server) computing systems, is the orientation around the notion of *time*. For example, an embedded system equivalent of a "Hello World" program might repeatedly blink an LED (light-emitting diode) on for 1 second and then off for 1 second, requiring an explicit notion of real time ("1 second"). In contrast, desktop computing has focused on data-oriented programming emphasizing data transformation—reading input data, transforming the data, and outputting new data, with no notion of real time—even since the design of early computers, which was driven by data-oriented applications like processing census data and computing bomb trajectories. Whereas time is a behavioral consequence of desktop programs, time is part of the explicit functionality of embedded systems. Explicit functionality of embedded systems also involves the related notion of *events* – external actions that can occur at any time and to which the system responds. For example, an alternative "Hello World" equivalent might turn on an LED for 1 second every time that a button is pressed, the button press forming an input event. Some desktop programming does incorporate time and event concepts (e.g., blinking a cursor in a graphical display, or responding to mouse click events), but to a lesser extent than embedded programming. For simplicity, rather than always referring to both time and events, we will in this paper take the liberty of using the term time to refer to both concepts, although we realize there are distinctions between the two concepts.

While some time-oriented programming courses exist today, they tend to build on advanced concepts of real-time operating systems (RTOSes) or of parallel programming languages, using extensive abstraction to hide many lower-level details. For our purposes, the appropriate balance must be found between abstracting away low-level details to enable focus on higher-level issues, while also exposing enough low-level details to ensure that programmers have resource awareness and can build systems using today's embedded microprocessors, which may or may not be running RTOSes. Indeed, our interactions with companies that produce embedded systems have revealed an intense desire from those companies for more computing professionals that have a much stronger understanding of underlying computing resources; embedded programmers who only know abstractions and never learned the details of underlying resources may produce grossly inefficient code and be unable to hammer out the details often necessary to get real systems completely working.

Virtual Microcontroller

We propose the use of a *virtual microcontroller* as a step towards achieving the appropriate balance between abstraction and resource awareness. Fundamental resources in time-oriented embedded programming of a microcontroller include a *microprocessor*, a *timer*, and an *interrupt service routine*. A main program can initialize and activate a timer, which in turn automatically calls at specified intervals an interrupt service

Figure 5: "Clean" computing platforms: (a) A simplified view of a computer provides the foundation for most desktop-programming courses, (b) likewise, a simplified view of a microcontroller can provide a good foundation for a time-oriented embedded programming course.



routine (ISR), pausing the main program's execution during such calls. Creating time-oriented programs using just those basic resources, without the aid of an RTOS, represents a fundamental embedded programming skill, akin to a surgeon learning to perform surgery with a scalpel but without the aid of modern robotic surgical tools. The programmer (or surgeon) develops an intuitive understanding, which not only may help when using the more advanced tools, but also enables competence even when the more advanced tools are unavailable.

The virtual microcontroller therefore consists of a simple microprocessor and a timer (along with required program memory and general-purpose I/O), as illustrated in Figure 5(b). All items are present in very straightforward form. For example, the I/O consists of eight inputs A0-A7 and eight outputs B0-B7 accessible by name in C or assembly (microcontrollers often have I/O that can be configured for either input, output, or both, or can serve as memory address/data lines instead, and thus require configuration). The timer is set in C by calling a predefined function called TimerSet(T) where T specifies the interrupt interval in milliseconds (most microcontrollers instead require extensive configuration of various registers, such as mode registers, frequency registers, prescaler registers, and more, to obtain a particular interrupt rate). This simplified, or "clean," computing platform is akin to the simplified platform commonly used in data-oriented programming courses, shown in Figure 4(a). The clean platforms provide a sufficient abstraction on which to program, introducing just enough resource concepts for solid understanding, but without overexposing the student to resource details.

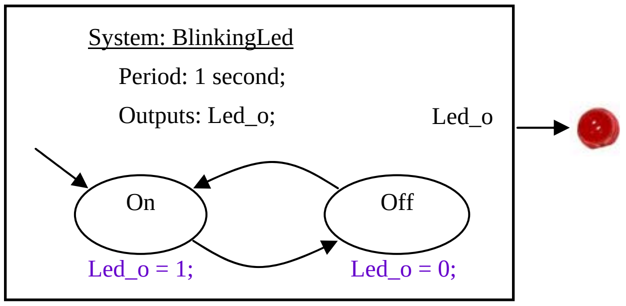
Synchronous State Machines – SynchSMs

While a virtual microcontroller provides a clean platform for developing and executing time-oriented code, the C language (which is used in a majority of embedded systems [10], though our discussion applies equally to C++, Java, or assembly language) is not particularly well-suited for time-oriented programming. Left alone to figure out how to implement timed behavior using a microcontroller with a timer and ISR, students develop an "impressive" variety of methods, with some putting most code in the main function and others putting most code in the ISR, some using sequenced code and others used heavily iterated code and global status variables, and so on.

Instead, we propose the use of *synchronous state machines*. Good computation models for data-oriented desktop processing include sequential programming or object-oriented programming models. For embedded processing, however, other computation models are common. In particular, state machines excel at modeling basic sequential control behavior, as well as more complex behavior. State machines (SMs) allow not only straightforward description of processing of events, but also provide an elegant basis for specifying timing. For programming purposes, a state machine can utilize declared variables and possess arithmetic operations and conditions, as in the FSMD model of [15]. In fact, each state can have a sequence of associated actions described using sequential programming constructs such as assignment statements, if-then-else statements, loops with constant bounds, and even (non-recursive) subroutines having such statements as described above. Such SMs are commonplace in hardware descriptions targeting synthesis to circuits.

State machines provide a solid basis for a disciplined approach to time-oriented embedded system programming. The authors have developed and utilized such approaches in their courses during the past several years. A simple approach involves using a synchronous SM, or *synchSM*, whereby all states last for a defined time interval known as a tick rate or period, as in Figure 6. Thus, if the synchSM's period is 1 second, then the shown synchSM will set output Led_o to 1 for 1 second, then to 0 for 1 second, then to 1 for 1 second, and so on, causing

Figure 6: SynchSM for the Blinking LED (time-oriented "Hello World" equivalent) example.



the LED to blink as desired.

SynchSMs mesh well with the basic microcontroller time elements. A programmer can set the timer's interrupt rate to the desired tick rate, and program the timer's ISR to raise a global flag. The programmer's main code, implementing the synchSM, can detect the raised flag (and reset it) and then proceed to the next state.

Students may be well served by being encouraged to work with the synchSM abstraction extensively on a wide variety of examples, to develop a solid perspective of describing timed behavior using a timed computation model.

Implementing SynchSMs in C – Models Matter, Not Languages

Of course, programs are typically written using C code (or similar), bringing us to another important concept in structured time-oriented programming. The concept is that of conceptualizing system behavior using a *computation model* (e.g., synchSMs), and then implementing that model in C using a straightforward template-based approach. A straightforward template approach is shown in Figure 7, wherein the main program waits for a tick, and then processes the state machine's transitions and state actions using switch statements. In contrast, today every programmer combines main code, timers, and timer ISRs in unique and creative ways. While creativity in some endeavors is a wonderful thing, creativity in programming makes programs harder to understand, debug, and maintain. A synchSM approach represents a highly-structured approach, akin to structured programming guidelines for data-oriented programming [12] but instead focused on time.

An additional advantage of the state machine approach is that multiple concurrent state machines can be handled via straightforward round-robin processing. Straightforward counter methods can handle synchSMs with different periods. Priority schemes among the synchSMs could be introduced too, leading towards the basic idea of RTOSes.

A similar approach can be used for other computation models common in embedded systems. Other common computation models in embedded systems include dataflow models, such as synchronous dataflow. With care, different models can even be implemented in a single C program on a single microcontroller.

Figure 7: Template approach for implementing a synchSM in C.

```
#define Led_o B0

int BL_Clk=0;
void TimerISR()
{
   BL_Clk = 1;
}

void main(void)
{
   enum BL_StateType {BL_On, BL_Off}
      BL_State;
   B=0;//Init outputs
   TimerSet(1000); // 1 second
   TimerOn();

   BL_State = BL_On;
   while (1) {
      switch (BL_State) {// State actions
         case BL_On:
            Led_o = 1;
            break;
         case BL_Off:
            Led_o = 0;
            break;
      } // State actions

      while (!BL_Clk); BL_Clk = 0;

      switch (BL_State) { // Transitions
         case BL_On:
            BL_State = BL_Off;
            break;
         case BL_Off:
            BL_State = BL_On;
            break;
      } // Transitions
   } // while(1)
} // main
```

3. TOOLS

Microcontroller Simulator

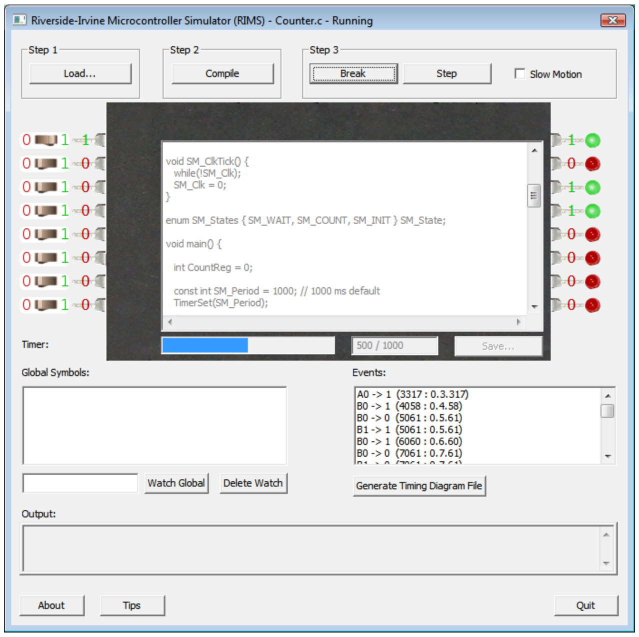
Appropriate tools are needed to support the proposed approach to teaching structured time-oriented programming. We are developing a tool, the Riverside-Irvine Microcontroller Simulator (RIMS), to support programming of the proposed virtual microcontroller, seen in Figure 8.

The tool graphically depicts a microcontroller with 8 input and 8 output pins. Concreteness is emphasized to ease the comprehension task for young students. Each input pin is connected to a switch that the user can move to the "0" or "1" position, and each output pin to an LED that illuminates when its pin is 1. The microcontroller's C program is shown inside the microcontroller. The user is shown a 3-step process: (1) Load a C program (or write one directly in the C window and save it), (2) Compile the program, (3) Run the program. The compiler is built into the tool (using the open-source LCC compiler [28]) and the generated assembly code is then executed by an instruction set simulator, which we wrote and built into the tool. The running program may be "break'ed" at any time, executed step-by-step, and have any symbol values viewed, as with a standard debugger. All events on input and output pins, in addition to being graphically shown, are printed to an event text display also.

To help teach the concept of the timer and timer ISR, a "status" bar shows the current value of the timer. Thus, as a program executes, the status bar fills, the ISR is called, and the status bar starts over. The student thus visually sees the "ticking" of the microcontroller's timer and the associated ISR call.

To further support the teaching of timing concepts, the textual event printout can automatically be converted to timing diagrams. Such conversion is accomplished by the RIMS tool generating standard VCD files, which can then be viewed using any of several VCD to timing diagram viewing tools, such as the freely-available Wave tool [44]. All of the above functionality (excluding the VCD viewer) exists as a single Windows executable, making installation and operation easy.

Figure 8: Screenshot of the virtual microcontroller simulation tool, RIMS.

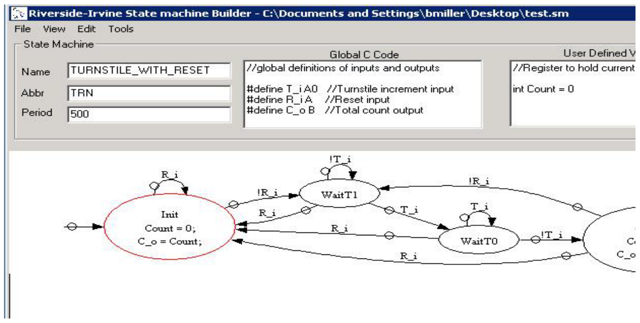


```
void SM_ClkTick() {
  while(!SM_Clk);
  SM_Clk = 0;
}

enum SM_States { SM_WAIT, SM_COUNT, SM_INIT } SM_State;

void main() {

  int CountReg = 0;

  const int SM_Period = 1000; // 1000 ms default
  TimerSet(SM_Period);
```

500 / 1000

```
A0 -> 1  (3317 : 0.3.317)
B0 -> 1  (4058 : 0.4.58)
B0 -> 0  (5061 : 0.5.61)
B1 -> 1  (5061 : 0.5.61)
B0 -> 1  (6060 : 0.6.60)
B0 -> 0  (7061 : 0.7.61)
```

Figure 9: Screenshot of the state machine builder tool, RISB.



Name: TURNSTILE_WITH_RESET
Abbr: TRN
Period: 500

```
//global definitions of inputs and outputs

#define T_i A0   //Turnstile increment input
#define R_i A    //Reset input
#define C_o B    //Total count output
```

```
//Register to hold current

int Count = 0
```

State machine builder

Because describing behavior as synchSMs is so central to our proposed approach, we are also developing a graphical synchSM capture tool – the Riverside-Irvine State machine Builder (RISB), a screenshot of which is shown in Figure 9. The tool allows a user to add states and transitions between states, to type actions for states and to type conditions for transitions, and to declare the state machine's name, period, and any variables. Although the user works with a graphical depiction of the state machine, all graphical placement and routing of the state machine is automatically handled by the open source GraphViz tool [16] that we encapsulated within RISB, allowing the user to focus on the state machine's functionality and not details related to the state machine's graphical display. The tool can automatically translate a state machine to C code. The C code can then be executed on RIMC, with the current state being automatically highlighted in RISB, to help students visualize the execution of the state machine. Such automatic translation to C emphasizes the fact that writing the C code is not a creative endeavor, and instead should follow strict, automatable, guidelines. As such, structured time-oriented programs result.

Physical prototypes

Although the above described framework allows for learning the structured time-oriented embedded programming approach, creating working physical implementations can help crystallize embedded concepts. Thus, mapping the virtual microcontroller to physical platforms may be desired by many instructors of courses having lab components. We are therefore mapping the virtual microcontroller to various physical platforms, and plan to publish instructions for such mapping that instructors and their teaching assistants can follow to create numerous instances for their labs[1]. We have thus far mapped our virtual microcontroller onto three physical platforms – an Atmel AVR microcontroller, a Xilinx Spartan FPGA board, and a desktop PC using a USB interface to an external breadboard for the general-purpose I/O pins. Figure 10(a) shows a mapping onto the AVR microcontroller using a breadboard approach. Students could add input and output components, perhaps using a second board, to interact with the I/O. Figure 10(b) shows the same mapping but in a self-contained box where all inputs are connected to switches and outputs to LEDs, allowing for standalone operation of simple embedded programs having 8 input switches and 8 output LEDs – the same as the RIMS PC-based visual simulator. Figure 10(c) shows mapping to an FPGA board (at the bottom of the picture). For any of these implementations, additional display functionality can be added using the VMC's serial UART, which can be connected through a PC's serial port to output on a serial terminal program, or connected through a custom serial-to-VGA device (which we have designed) to output directly to a monitor.

To simplify the process of mapping programs to these physical implementations, we have developed an approach whereby a textual assembly file can be downloaded onto a USB stick, and that USB stick then plugged into the physical implementation for uploading to the VMC, as illustrated in Figure 10(b). Such an approach makes clear to the student what program is being executed, and enables easy migration of different programs to a physical platform or from one platform to another. We implemented the necessary USB read functionality and the just-in-time assembler for this textual assembly USB file approach, resulting in fully-functioning physical implementations.

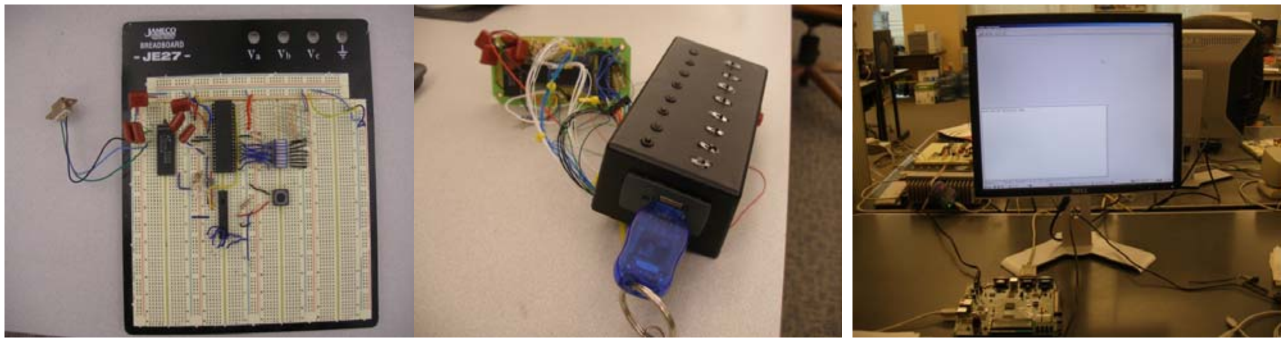
4. RELATED WORK

An earlier section described the main existing embedded systems courses that are the target of the project. Several other items can be considered related to this project also.

Another category of courses emphasizes real-time systems, typically based on an RTOS. Jian [23] presents a survey of the state of real-time systems education, showing that very few universities offer courses in the topics. Schwarz [35] observes

[1] Note: We do not plan to provide physical prototypes to other universities, but rather descriptions of how to build them for a few common physical platforms, which other universities can use or adapt.

Figure 10: Virtual microcontroller implementations: (a) on an AVR microcontroller breadboard, with input/output wires that can be connected to other circuits, (b) in a black-box, with internal AVR-microcontroller-based circuitry exposed, and (c) on a Xilinx Spartan 3E FPGA, which happens to also use a connection to provide additional output to a terminal. All three can execute the same virtual microcontroller program identically.



(a) (b) (c)

that undergraduate real-time course offerings vary and lack any accepted standard of contents or discipline. The textbook by Burns [8] presents real-time systems and the programming languages that work well with them, used primarily at the senior or graduate level. Kornecki [26] argues that universities do not pay enough attention to practical software development in the field of time-oriented reactive programming. ACM's curriculum guidelines make only short mention of the subject [1]. A number of real time operating systems have been introduced to provide a higher level of abstraction between the application software and embedded hardware, including the open source eCos [13], and VxVorks and RTLinux from WindRiver [45].

Much progress has been made in the past two decades on capturing embedded system functionality. These include several event-based languages. Esterel [6] is a synchronous programming language for the development of complex reactive systems. The notion of time is replaced with the notion of order, called the multiform notion of time, which means only simultaneity and presence of events are considered. Statecharts [17][18] is a visual formalism for complex systems. Statecharts extend traditional state diagram transition diagrams with notions of hierarchy, concurrency, and communication. They also include some timing mechanism (e.g., "timeout" events). Numerous other languages can be viewed as supporting event/time based descriptions, including VHDL [21], Verilog [41], Rapide [32], LOTOS [7], CSP [19], Ada [39], and more.

Several approaches focus extensively on time. Kopetz [25] presents the time-triggered architecture, which is a computing infrastructure for the design and implementation of dependable distributed embedded systems. Applications are decomposed into autonomous clusters each with a fault-tolerant global time base. The global time base simplifies communication and guarantees timeliness of real-time applications. Commercial companies like TTTech [40] stemmed from this work. Lamport [27] discusses time-oriented programming in the context of events in a distributed system and how to synchronize items. Ouimet [33][34] introduces timed abstract state machines. ASMs consist of a set of mutually exclusive rules each guarded by a condition of variables. A rule whose condition at a given execution step will update external and internal variables. Timed ASMs involve timing extensions to ASMs, such as specifying the time duration of a rule's execution (which may be a range; a specific value in the range will be randomly chosen during runtime).

Other related approaches emphasize models of computation. The Polis framework [4] defines codesign finite state machines (CFSMs) as a formal model for communicating FSMs to which a system captured in a language like Esterel can be translated for analysis and synthesis. Lee [31] presents a framework for comparing models of computation, including Kahn process networks, dataflow, sequential processes, concurrent sequential processes with rendezvous, Petri nets, and discrete-event systems. Other works [24][38] have focused on quantitatively comparing various computation models, specifically targeting parallel models. Jeukens [22] investigates how best to use various computation models to design increasingly complex systems. Andrews [3] investigates how best to leverage computation models for hybrid CPU-FPGA platforms. Lee [29] discusses various requirements of future embedded programming, including time concepts.

Most of the above involve advanced concepts and thus not ideal for introductory undergraduate courses. Increasing attention is being paid to embedded systems education, through workshops (e.g., the Workshop on Embedded Systems Education, WESE, held since 2005), special issues of journals, and numerous special sessions and papers appearing in mainstream research venues. Numerous research projects attempt to improve engineering education. Hodge [20] introduces the concept of a Virtual Circuit Laboratory, a virtual environment for a beginning electrical engineering course that mimics failure modes in order to aid students in developing solid debugging techniques. The environment not only provides a convenient test environment, but also allows an instructor to concentrate more on teaching. Butler [9] developed a web-based microprocessor fundamental course, which includes a Fundamental Computer that provides students in a first year engineering course a less threatening introduction to microprocessors and how to program.

Other researchers have concentrated on developing or evaluating computing architectures for beginning students or non-engineers. Benjamin [5] describes the BlackFin architecture, a hybrid microcontroller and digital signal processor. The architecture provides a rich instruction set based on MIPS with variable width data, and parallel processing support. Ricks [9] evaluates the VME Architecture in the context of addressing the need for better embedded system education. The Eblocks project [11] concentrated on developing sensor blocks that people without programming or electronics knowledge could connect to build basic customized sensor-based embedded systems.

Much research has involved virtualization [30][36], with several commercial products developed in response to the need for portable virtual machines. VMware [43] and the open source product Xen [46] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual Machine [37] allows the programmer to write operating system independent code, and tools like DOS Box and console emulators allow the user to run legacy applications in modern operating systems.

5. CONCLUSIONS

Embedded systems have experienced massive growth in the past two decades. Embedded system education has improved, but exists largely as a late add-on to a traditional data-oriented computing curriculum, leading to ad hoc modified data-oriented programming methods rather than a disciplined method targeted to the time-oriented nature of embedded systems. Early introduction of a structured time-oriented programming approach may help. Key features include a virtual microcontroller that exposes just enough low-level resources along with higher-level abstractions, a synchronous state machine (synchSM) computation model for explicit time-oriented programming and a clear method for capturing synchSMs in the prevailing C language, a set of easy to use tools that support synchSM and C capture, compilation, simulation, and debugging, and the ability to readily develop physical prototypes of a programmed virtual microcontroller. These items may catalyze adoption of early teaching of time-oriented programming, which we hope to see become part of a standard computing curriculum in the coming decade.

6. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CNS-0614957).

References

[1] Association for Computing Machinery. Computing Curricula 2005. http://www.acm.org/education/curricula-recommendations

[2] Alice Programming Environment, http://www.alice.org.

[3] ANDREWS, D., NIEHAUS, D., JIDIN, R., FINLEY, M., PECK, W., FRISBIE, M., ORTIZ, J., ED KOMP. AND ASHENDEN, P. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. IEEE Micro. On page(s): 42- 53, Volume: 24, Issue: 4, July-Aug. 2004

[4] BELARIN, F., M. CHIODO, H. HSIEH, A. JURESKA, L. LAVAGNO, C. PASSERONE, A. SANGIOVANNI-VINCENTELLI, E. SENTOVICH, K. SUZUKI, AND B. TABBARA, Hardware-Software Co-Design of Embedded System: The POLIS Approach Norwell, MA: Kluwer, 1997..

[5] BENJAMIN, M., KAELI, D., AND PLATCOW, R. Experiences with the Blackfin Architecture in an Embedded Systems Lab.. WCAE '06

[6] BERRY, G. AND GONTHIER, G. The ESTEREL Synchronous Programming Language: design, semantics, implementation. *Sci. Comput. Program.* 19, 2 (Nov. 1992), 87-152.

[7] BOLOGNESI, T. AND BRINKSMA, E. Introduction to the IS0 specification Language LOTOS.. Amsterdam: North-Holland, 1989, pp. 23-73.

[8] BURNS. A. AND WELLINGS, A. Real-Time systems and Programming Languages. Third Edition. Pearson Education Limited. 2001.

[9] BUTLER, J. AND BROCKMAN, J. Web-based Learning Tools on Microprocessor Fundamentals for a First-Year Engineering Course. 2003. Proceedings of the American Society for Engineering Education.

[10] CMP. Embedded Systems Design State of Embedded Market Survey, 2006, http://www.embedded.com/columns/survey.

[11] COTTRELL, S. AND F. VAHID. A Logic Enabling Configuration by Non-Experts in Sensor Networks. Conference on Human Factors in Computing. 2005

[12] DIJKSTRA, E. Notes on Structured Programming. T.H.-Report 70-WSK-03, Second Edition, April 1970.

[13] Ecos. http://ecos.sourceware.org/

[14] FAST GmbH. Study of worldwide trends and R&D programmes in embedded systems, 2005. ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf

[15] GAJSKI, D., DUTT, N., WU, A. AND LIN, S. High-Level Synthesis: Introduction to Chip and System Design. Springer 1992.

[16] Graphviz – Graph Visualization Software, http://www.graphviz.org.

[17] HAREL, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (Jun. 1987), 231-274.JIAN, K. Constructing a Solid Real-Time Operating Systems Course in Computer Science Major. *J. Comput. Small Coll.* 22, 4 (Apr. 2007), 65-74

[18] HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., AND SHTUL-TRAURING, A. 1988. Statemate: A Working Environment for the Development of Complex Reactive Systems.. International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 396-406..

[19] HOARE, C.A.R. Communicating Sequential Processes. Comm. of the ACM, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[20] HODGE, H. HINTON, H.S, AND LIGHTNER, M. Virtual Circuit Laboratory. ASEE. American Society for Engineering Education. 2000

[21] IEEE, INC., IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076-1987. Los Alamitos, Calif.: IEEE CS Press, 1987.

[22] JEUKENS, I. AND STRUM, M. On the Choice of Models of Computation for Writing Executable Specifications of System Level Designs. In Proceedings of the 13th Symposium on integrated Circuits and Systems Design.2000.

[23] JIAN, K. 2007. Constructing A Solid Real-Time Operating Systems Course in Computer Science Major. *J. Comput. Small Coll.* 22, 4 (Apr. 2007), 65-74

[24] JUURLINK, B. H. AND WIJSHOFF, H. A. A Quantitative Comparison of Parallel Computation Models. *ACM Trans. Comput. Syst.* 16, 3 (Aug. 1998), 271-318

[25] KOPETZ, H. AND BAUER, G. The Time-Triggered Architecture. Proceedings of the IEEE. On page(s): 112- 126, Volume: 91, Issue: 1, Jan 2003

[26] KORNECKI, A. Real-Time System Course in Undergraduate CS/CE Programs. IEEE Transactions on Education. Volume 40. Number 4. November 1997.

[27] LAMPORT,L. Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM, July 1978, pp. 558-565.

[28] LCC, A RETARGETABLE COMPILER. http://www.cs.princeton.edu/software/lcc/

[29] LEE, E.A. What's Ahead for Embedded Software? IEEE Computer, vol. 33, no. 9, pp. 18-26, Sept. 2000.

[30] LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. SIGOPS Oper. Syst. Rev. 36, 5 (Dec. 2002), 85-95.

[31] LEE, E.A AND SANGIOVANNI-VINCENTELLI, A. A Framework for Comparing Models of Computation IEEE Trans. CAD Integrated Circuits and Systems, Dec. 1998, pp. 1217-1229

[32] LUCKHAM, D. C. AND VERA, J. An Event-Based Architecture Definition Language. *IEEE Trans. Softw. Eng.* 21, 9 (Sep. 1995), 717-734

[33] OUIMET, M. AND LUNDQVIST, K. Incorporating Time in the Modeling of Hardware and Software Systems: Concepts, Paradigms, and Paradoxes. In Proceedings of the international Workshop on Modeling in Software Engineering (May 20 - 26, 2007). International Conference on Software Engineering.

[34] OUIMET, M. AND LUNDQVIST, K. The TASM Language and the Hi-Five Framework: Specification, Validation, and Verification of Embedded Real-Time Systems. APSEC 2007

[35] SCHWARZ J.J., SKUBICH J., MARANZANA M., AND AUBRY R., Graphical Programming and Real-Time Design Instruction, in Real-Time Systems Education, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 20-25.

[36] SMITH, J. AND NAIR, R. VIRTUAL MACHINES: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005.

[37] STARK, R., SCHMID, J, AND BORGER, E. Java and the Virtual Machine- Definition, Verification, and Validation. 2001.

[38] SKILLICORN, D.B., AND TALIA, D. Models and Languages for Parallel Computation. *ACM Comput. Surv.* 30, 2 (Jun. 1998), 123-169

[39] TAFT S.. Ada 2005 Reference Manual, LNCS 4348, Springer-Verlag. 2006

[40] TIME-TRIGGERED TECHNOLOGY. http://www.tttech.com/

[41] THOMAS, D.E AND MOORBY, P.R., The Verilog Hardware Description Language. Kluwer Academic Publishers, 1991

[42] VAHID, F AND GIVARGIS, T. Embedded System Design: A Unified Hardware/Software Introduction. John Wiley and Sons, 2001.

[43] VMWARE. http://www.vmware.com/

[44] WAVE VCD Viewer, http://www.iss-us.com/wavevcd/, 2008.

[45] WINDRIVER Systems. http://www.windriver.com/

[46] XEN. http://www.xen.org