Incorporating Embedded Programming Skills into an ECE Curriculum

Kenneth G. Ricks
The University of Alabama
Electrical and Computer Engineering
Tuscaloosa, Alabama 35487-0286
(205)-348-9777
kricks@eng.ua.edu

David J. Jackson
The University of Alabama
Electrical and Computer Engineering
Tuscaloosa, Alabama 35487-0286
(205)-348-2919
jjackson@eng.ua.edu

William A. Stapleton
The University of Alabama
Electrical and Computer Engineering
Tuscaloosa, Alabama 35487-0286
(205)-348-1436
wstapleton@eng.ua.edu

ABSTRACT

In this paper, the typical electrical and computer engineering (ECE) curriculum is examined to determine its effectiveness at presenting embedded programming skills. The software concepts and programming techniques necessary for embedded systems are somewhat different than those seen in other engineering domains. Thus, it makes sense to specifically address embedded programming needs within the formal programming education ECE students receive. Several topical areas of concern are identified, and two possible ways to incorporate these areas into an ECE curriculum are presented. The experiences gained within the ECE curriculum at The University of Alabama are presented and are used to develop recommendations for incorporating these topics into typical ECE curricula.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *curriculum, computer science education.*

General Terms

Languages

Keywords

Embedded systems education, embedded systems programming, C programming language, engineering curriculum

1. INTRODUCTION

The need for embedded systems engineers is well documented and supported by the recent attention the field has gained within academia. Embedded systems education presents some interesting problems however. One of the most important problems is the breadth problem [7], i.e. how to feasibly incorporate the broad spectrum of embedded systems topics into the curriculum. To gain a perspective on exactly how broad this scope is, one can look at the IEEE/ACM model computer engineering curriculum report describing recommended topical coverage for embedded systems [9]. In this report, there are 11 knowledge areas covering 59 different topics assessing 34 learning outcomes.

In this paper, we address a small part of the breadth problem, specifically focusing on the programming skills needed by embedded systems engineers. Since programming is already a fundamental component of nearly every electrical and computer engineering (ECE) curriculum, one might think students are well exposed to the necessary skills in this area. However, this paper will address concerns that the general programming skills being taught in typical ECE curricula are not addressing all embedded programming skills needed.

Embedded software programming is a critical aspect of embedded systems education. In [5], it is estimated that the amount of embedded software created doubles every ten months, and will reach 90% of all software being written by the year 2010. To address this demand, embedded software programming skills must be incorporated into embedded systems education curricula immediately.

The rest of this paper is organized as follows. In Section 2, we describe a typical ECE approach to programming and point out several pitfalls to such an approach. Section 3 describes several need areas for embedded programming skills not usually addressed in this typical approach. Section 4 presents some possible solutions for correcting the problem. The experiences gained from incorporating embedded programming skills into The University of Alabama (UA) ECE curriculum are presented in section 5. Section 6 presents the conclusions.

2. TYPICAL ECE APPROACH

A typical ECE curriculum includes a two-level approach to teaching programming skills. First, one or more introductory general programming courses are typically offered early in the curriculum. The goals of these courses vary considerably between programs, but usually a high-level language (HLL) is presented. The motivation for offering this material at an early point in the program is to provide engineering students a tool with which to solve problems as they progress through the curriculum.

Later in the curriculum, students are exposed to assembly language usually in the context of a microprocessors or microcontrollers course. At this time, students write assembly language programs to control and interface with various hardware devices thus acquiring low-level hardware interfacing skills. In many cases, students are shown how a HLL program is decomposed into the assembly and machine language equivalents, thereby tying the two programming levels together.

By and large this two-level approach to programming in ECE curricula is effective and provides students with many useful skills. However, from an embedded programming perspective, there are pitfalls in this approach. First, general programming courses often incorporate many different concepts that do not map directly to the skill set needed by embedded systems engineers. This is especially the case for general programming courses supporting students from many different engineering disciplines.

It is quite common for these courses to serve a multipurpose role within the curriculum covering all sorts of topics including teaching programming language syntax [1, 3, 15], problem solving [4, 14], teaming [4], communication skills [4], program design [4], algorithm design, and object-oriented (OO) programming techniques. While these concepts are valuable, they do not replace the fundamental programming skills needed by embedded systems engineers, and they tend to dilute the programming aspects of the course from the embedded systems perspective. There are other effective techniques to teach problem solving using the computer that can be incorporated into other parts of the curriculum [1, 4]. Also, OO approaches are not applicable to all engineering disciplines including some embedded applications [14]. In some cases, the general programming courses are given an engineering flavor by focusing on engineering applications and problems [14]. This makes the applications more appropriate to embedded programming but does not necessarily improve the programming fundamentals presented.

The assembly language courses also have drawbacks. Since these languages are tied very closely to the hardware, class time is required to address syntax, basic assembly software design, particular hardware interfacing issues for specific devices, system integration, debugging, and concepts such as pulse width modulation and analog-to-digital conversion. These are skills and concepts needed in many embedded applications, but this is a large amount of information to present in one course and tends to overwhelm the basic programming skills aspects of the course. Also, assembly language is not the most popular language used for embedded applications. In 2000, 80% of all embedded systems applications were written in a HLL, specifically the C programming language [19], and this number increases as more capable tools introduce more abstraction.

Embedded systems engineers are migrating to a higher level of abstraction, using tools and development environments to handle the lower level details, such as the hardware/software interfacing aspects of embedded systems. But, the typical ECE curriculum depends on general purpose programming courses and low-level assembly language courses to present the programming skills required of embedded systems engineers. This approach to programming does not prepare students to address low-level details from higher levels of abstraction. It is not uncommon for students in such curricula to encounter problems in upper-level ECE courses, like a senior-level Embedded Systems course or a Capstone Design course, where they are expected to use higher levels of abstraction to address the embedded system. A higher level view of embedded programming is needed in the curriculum to address this concern.

3. EMBEDDED PROGRAMMING NEEDS

This section describes four specific areas where ECE students in typical curricula are likely to encounter problems with high-level programming skills needed for embedded applications. These areas, summarized in Table 1, are derived from research, personal communications with educational and industry professionals, and personal observations and assessment data collected within the ECE program at The University of Alabama. UA follows the traditional two-level approach to programming, as described in the previous section, within its ECE curriculum.

Table 1. Summary of areas of concern for embedded programming skills.

Areas of Concern for Embedded Programming Skills
Choice of HLL
Peripheral Interfacing Using Registers
Program Structure
Resource Constraints

3.1 Choice of HLL

Since we are addressing programming skills needed by embedded systems engineers, let's begin with the most basic idea, the choice of HLL. While there are compelling arguments for the use of many different programming languages within education, at this time there is little argument over the need for embedded systems engineers to know the C programming language. As previously mentioned, C is the language of choice for a large majority of embedded applications. It is our experience that students without a working knowledge of C are at a significant disadvantage when competing for jobs related to embedded systems. While having experience with a different HLL certainly flattens the learning curve for those desiring to learn C, there is no reason that C should not be the language of choice for ECE students.

To expand on this idea a bit more, it is our belief that students should begin with ANSI C. This provides a portable programming foundation upon which additional skills can be built. Introductory programming courses that mix object-oriented concepts and other concepts with ANSI C tend to dilute the basic understanding of C and undermine its portability across hardware platforms. Portability is an important concept to embedded programmers because it promotes code reuse thereby shortening time-to-market and impacting design decisions. Software development tools are ranked as the number one factor for microprocessor choice for embedded systems [20], and portability is a key evaluation criterion for these tools and the code they produce. In the following sections, the C programming language is used for all examples.

3.2 Peripheral Interfacing Using Registers

Embedded systems often contain a set of memory-mapped locations, called device registers, used for communication with the device. The ability to access and manipulate registers is critical for embedded systems. Device registers are often located at specific locations in memory. Variables in the HLL must be declared and initialized to represent data located at those specific locations. One cannot declare a variable in ANSI C so that it resides at a specified memory address [17]. Thus, pointers are required, and pointer arithmetic becomes critical to maintain access to the registers [2, 8, 10, 11]. An understanding of the differences between memory-mapped registers and port-based input/output (I/O) registers is needed since the type of registers determines how they can be accessed from the HLL. For example, memory-mapped registers can be treated like any other memory location and read and written using standard assignment and dereferencing operators as shown in Figure 1. But, port-based I/O registers may require special instructions for reading and writing because the underlying processor instruction set differs for accessing these types of registers. Also, bitwise manipulation of register contents is required since in many cases registers contain unrelated bitfields. Thus, it is essential to present the bitwise operators in C, including the bitwise-AND operator "&", bitwise-OR operator "|", logical-NOT operator "~", and the bitfield structure operator ":" [2, 8, 10, 11].

```
int * CSR_ptr = 0xFFAA;
*CSR_ptr = 1;
```

Figure 1. Example C code that accesses a memory-mapped Control/Status register and assigns it a value.

Another important aspect of registers is that in many cases, register values change outside the scope of the program. For example, a register might contain a bitfield used as a status indicator. The user program may initialize this register to a certain value, and the device writes a new value into this bitfield to represent a change in the device's status. If appropriate variable type qualifiers are not used, compiler optimizations can result in erroneous code. For example, consider the unsafe code fragment in Figure 2A. This fragment initializes the CSR device register and then later reinitializes the same register using the same value. The compiler might conclude the second write operation to this location is redundant since the same value is being written twice and no other assignment operation separates the two initialization assignments. Thus, the compiler removes the second write unaware that an external device modified the CSR register contents between the writes. To prevent this, the type qualifier "volatile" can be used, shown in Figure 2B, to inform the compiler that the variable associated with this register may change outside the scope of the program [2, 10, 11, 17]. The resulting code segment is safe from these types of erroneous optimizations.

```
int * CSR_ptr = 0xFFAA;
*CSR_ptr = 1;      /* initialize register contents */
... (During this part of the program, the device's
    status changes and the register value is
    overwritten by the device hardware.)
*CSR_ptr = 1;     /* re-initialize register contents */
```

A. Unsafe code fragment subject to erroneous compiler optimization.

```
volatile int * CSR_ptr = 0xFFAA;
*CSR_ptr = 1;      /* initialize register contents */
... (During this part of the program, the device's
    status changes and the register value is
    overwritten by the device hardware.)
*CSR_ptr = 1;     /* re-initialize register contents */
```

B. Safe code fragment not subject to erroneous compiler optimization.

Figure 2. Example of the importance of variable type qualifiers.

Students can succeed in the two-level programming paradigm without these register manipulation skills. Without an emphasis on embedded programming, a general-purpose C programming class has little motivation to present pointer arithmetic outside the context of particular data structures. Similarly, pointer assignment, variable type qualifiers, and bitwise operators are topics that are often not covered in depth. Instead, variables represent data needed by the program and are stored "somewhere in memory." The translation tools including the compiler, assembler, linker, and loader abstract the details of exactly where in memory the variables reside. The programmer must use an explicit prefix, "&", to determine this information, if it is needed.

The information presented within an assembly language course does not fill in the gaps left by the general-purpose programming course. The term "register" takes on a different meaning within the context of an assembly language course. In this context, the register set of the processor executing the code is usually what is meant and care must be taken to prevent confusion between the processor registers and the device registers. Also, students are faced with using obscure addressing modes to access the device registers instead of using pointers which is needed at higher-levels of abstraction. While it is true that bitwise operators can be more commonly seen in assembly language, this does not give students the skill to perform these operations in a HLL.

3.3 Program Structure

While basic program structures do not differ significantly between general-purpose programming and embedded programming, the motivations for using certain structures need to be understood by embedded software developers. For example, students need to understand that using subroutines and a modular programming approach may offer "divide and conquer" benefits such that a large problem may be debugged in smaller, more manageable parts. Similarly, a modular, subroutine-based approach may lead to reusable code that reduces "reinventing the wheel" over the long term and shortens time-to-market. On the other hand, utilizing in-line code eliminates the overhead associated with calling subroutines. This can reduce execution time requirements for real-time applications, but may require significantly greater storage requirements since otherwise reused code must be replicated in-line.

Other structural nuances such as global vs. local variables, subroutine parameter forms, and the impacts of these concepts on embedded system performance are critical. Variables that are global in scope may reduce storage requirements and subroutine overhead as they are only stored once and do not need to be copied to the subroutine's context. Unfortunately, global variables may be problematic in a system where multiple routines may all access the same data as additional mechanisms must be put into place to assure data coherency [10, 11]. Similarly, passing data to subroutines by reference rather than by value may reduce the amount of data that must be copied as subroutine overhead when large data structures are involved.

In most cases, students are presented many of the HLL implementation details associated with these concepts in a general programming course. The observed shortcomings in student ability involve understanding the motivations for choosing one alternative over another. For example, most students have seen subroutines in a HLL. The problem is getting students to understand their value and to choose a modular programming approach when appropriate. The goal must be to educate the students to make the best choice for the given application instead of defaulting to what is considered to be easier for the programmer.

In some cases, high-level programming tools abstract structural details of software development that embedded programmers need to understand. For example, embedded software developers in a team-based development environment often use a modular programming approach. The ability to compile HLL code to object code and to link with existing object code to create an executable are basic concepts often encountered in such development environments. But, students in general programming courses typically use high-level tools that abstract the translation process and automatically build executables. Thus, students are seldom aware of the different aspects of program translation required to support different program structures.

3.4 Resource Restrictions

Because embedded systems typically are more resource-limited than general-purpose computing systems, the need to manage system resources carefully is much more urgent for embedded systems than is typically taught in general-purpose programming. This is particularly true when the general-purpose computing platforms incorporate OO programs where the programming paradigm purposefully hides the resource implementation details from the programmer. Large memory-footprint objects such as linked lists or arrays of structures that are easily accommodated on a general-purpose PC may not be possible on an embedded system with limited memory. Even a factor as simple as choosing the most efficient data type can be very important to an embedded system with limited resources. For example, reducing Boolean variables to single bits from "int" size can greatly reduce the memory footprint of a program but requires the use of bit-manipulation operators that may not be considered in a general-purpose programming class. Alternatively, the bitfield structure operator ":" may be used to create variables of various sizes that may be treated functionally much like integers. But, using this operator requires the understanding of how the compiler allocates bits, either starting from the most-significant or the least-significant bit position [2, 10, 11, 17]. Such details are rarely addressed in general-purpose programming courses.

Memory is not the only resource with limitations that must be considered for embedded systems that is generally ignored in general-purpose computing. Time is also a limited commodity. Many embedded systems programs must operate under significant time constraints. General-purpose programming rarely considers hard or soft real-time constraints. Consequently, coding efficiency is typically not given as much emphasis as is required for embedded systems.

Power is another interrelated limited commodity. Many embedded systems operate from batteries or other limited power sources and must consequently operate as efficiently as possible. Limiting power usage and excess heat production often requires embedded system microprocessor clock rates to be limited. This can increase the difficulty of meeting timing constraints.

Embedded systems programmers must be able to balance the tradeoffs involved with execution speed, memory space, and available power. Pre-calculated values in tables can reduce the execution time and power used at the expense of larger, more expensive memories. Clock rates may be reduced to save power at the expense of performance and responsiveness. Smaller, cheaper memories may be used if coding efficiency is increased.

4. CURRICULA REFORM

Now that several need areas have been identified, we must address the problem of adding the embedded programming specific need areas to the typical ECE curriculum. We address two possibilities of how this can be done. First, the embedded programming topics can be integrated into the existing programming courses. Second, additional courses designed specifically to present embedded programming concepts can be added to the curriculum.

4.1 Integration into Existing Curricula

Integration of embedded programming concepts into an existing curriculum begins with the general-purpose programming course. If the HLL presented is C, then the first desired skill is automatically addressed. Integrating the other embedded programming concepts into an existing C programming course does not require a complete re-design of the course. Many of the necessary C constructs are already being presented. Assignments can be modified to address register interfacing, program structure alternatives, or resource limitations. Hardware platforms are usually required for embedded software development. These platforms are typically not used for general-purpose programming courses. In that case, embedded hardware can be simulated using artificial memory constraints, timing deadlines, and register addresses.

In cases where the introductory course is organized as one lecture section with smaller laboratory sections, one laboratory section can be dedicated as an embedded programming section for ECE students. In this section, assignments can be tailored for embedded applications and supplemental material can be presented to address embedded programming concepts. Also, at this level it is more plausible to integrate embedded hardware into the course. Such an organization introduces the embedded programming concepts for ECE students while not impacting students from other disciplines and not requiring additional courses be added to the curriculum.

The embedded programming concepts introduced in the general-purpose programming course can be revisited and reinforced in the microprocessors/microcontrollers course where assembly language is presented. In this course, a higher level of abstraction must be incorporated and hardware manipulation from the HLL must be included. Based on the typical content of these courses, this is a natural place within the curriculum to present embedded programming skills. Care must be taken, however, not to substitute HLL programming skills for the low-level skills typically presented in these courses. While it is true that more abstraction is being used in embedded systems, it is still important that students learn an assembly language and the programming skills associated with it. These skills are critical as indicated by their inclusion in the IEEE/ACM model curriculum and are not presented elsewhere in the typical curriculum [9].

4.2 Adding Courses to the ECE Curriculum

If integration into existing programming courses is not possible, then another option is to address these concepts in dedicated coursework. The one main positive to this approach is that existing courses are not affected and widespread coordination within the curriculum is not required to achieve the educational goals. Of course, there are difficulties associated with adding hours to any curriculum if courses cannot be identified for replacement.

For ECE curricula that rely on other departments to teach the general-purpose programming courses, one possibility is to replace these courses with courses that include embedded programming skills and are designed specifically for ECE students. The obvious merits of this include having a course taught by ECE faculty designed for ECE students. The drawbacks include additional teaching loads on ECE faculty and what appears to administration to be redundancy and waste in the curriculum.

Another possibility is to add a higher-level course to the curriculum such as an Embedded Systems course. This course would be taken after the assembly language course and would focus specifically on all aspects of embedded systems. Presenting embedded programming skills in such a course is possible, but does it make sense? Addressing basic programming skills is not usually part of the syllabus for a junior-level or senior-level Embedded Systems course. These courses are loaded with the more advanced knowledge areas, topics, and learning outcomes discussed in the IEEE/ACM model computer engineering curriculum for embedded systems [9]. For example, real-time concepts, interprocess communication, effects of caching on program performance, and scheduling are software concepts that must be covered, not to mention the hardware, design, and interfacing aspects of embedded systems. Taking time to review basic programming skills and teach proper program structure are topics that clearly do not belong at this level.

5. THE UA EXPERIENCE

The following paragraphs describe the UA curriculum, and assessment data supporting the need areas identified, and the two approaches used to incorporate the embedded programming skills into the curriculum.

5.1 The UA Curriculum

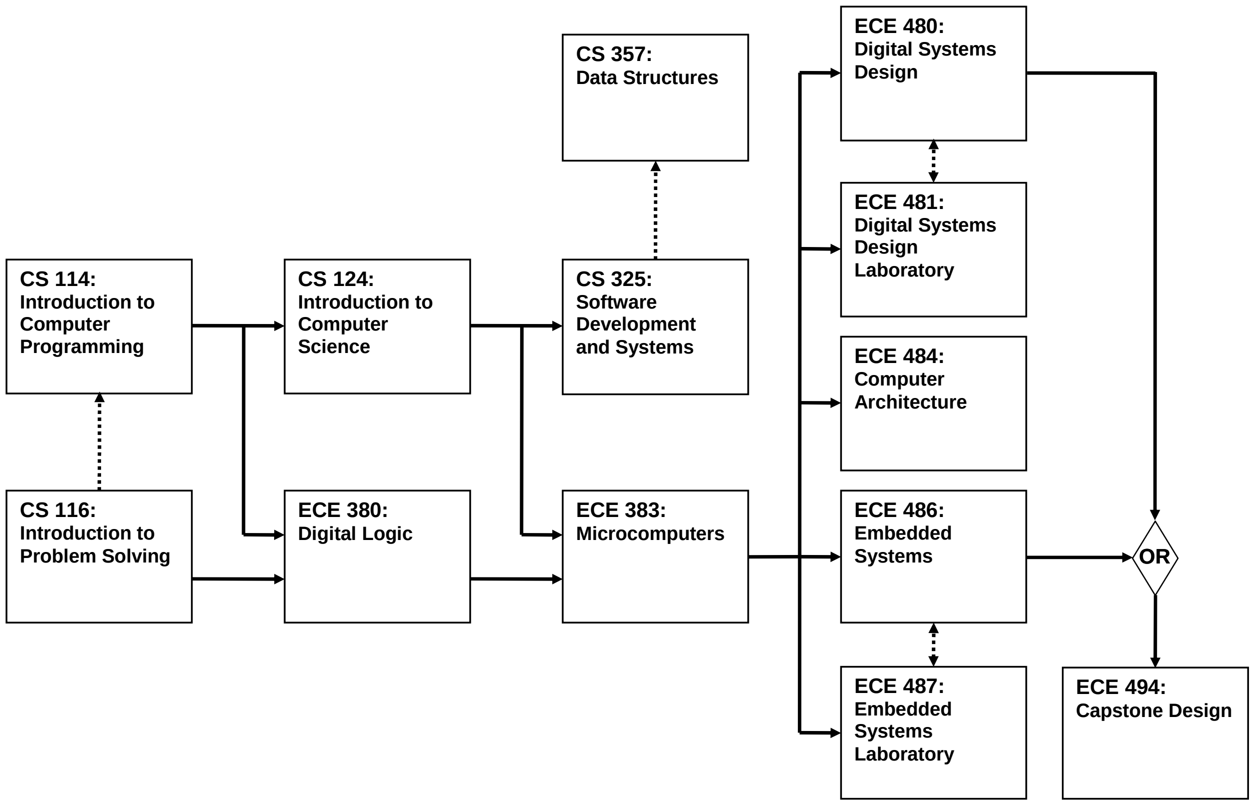
The UA ECE curriculum uses a typical two-level approach to programming. The prerequisite relationships among the programming courses and the core computer engineering courses are shown in Figure 3. Introductory programming skills and problem solving skills are taught in CS 114 and CS 116, where the C programming language, although not strictly ANSI C, is used as the basis. Object-oriented programming is presented in CS 124 by specifically introducing students to C++. These programming courses are taught by the Department of Computer Science within the UA College of Engineering and serve as prerequisites to ECE 380 and ECE 383. ECE 380 is a typical digital logic course. The ECE 383 Microcomputers course covers the traditional topics associated with such a course including assembly language programming and peripheral interfacing. ECE 383 serves as a prerequisite to several traditional computer engineering courses including ECE 480/481, ECE 484, and ECE 486/487. ECE 480/481 is a digital systems design course using VHDL. ECE 484 is a typical microprocessor architecture course. ECE 486/487 is a senior-level embedded systems course recently added to the curriculum. The course sequence culminates with ECE 494 Capstone Design where students must design and implement a complete project within a one-semester timeframe. All the core computer engineering courses within the UA ECE curriculum have an integrated embedded systems component, creating a curriculum with an overall focus on embedded systems [16, 18].

5.2 Assessment of Programming Skills

The authors collected assessment data from UA ECE students at the junior level to specifically evaluate overall programming skills and embedded programming skills. Table 2 shows some representative questions asked in the assessment process and a summary of the results from each question. The results represent the percentage of students who provided a reasonably correct answer to each question. In this case, “reasonably correct” translates into more correct than not.

The assessment data collected corroborate several observations made by the authors related to students’ performance within the UA ECE program and generalized in earlier sections of this paper. First, ECE students show a general lack of overall programming skills. This is evident from the fact that 70% or more of students responded reasonably correctly to only four of the 14 questions. While it is true that many of the questions are specific to ANSI C, some of the concepts such as variable scope and pointers are not limited to ANSI C.

Second, there seems to be little retention by upperclassmen of the HLL programming concepts presented early in the curriculum. For example, only 66% of the students tested were able to write a reasonably correct version of the most basic C program, the “Hello world” program. Similarly, only 44% of students had a working understanding of the *#include* directive and only 9.4% understood how to pass parameters between subroutines. This is despite the fact that these students have passed basic C and C++ programming courses which certainly would have/should have presented this material. It is interesting to note that 84% of students could figure out the basic functionality of existing C code. This verifies their exposure to some form of C programming. But it also reinforces the notion that understanding code and writing code are two completely different skill sets.



CS 357: Data Structures
ECE 480: Digital Systems Design
ECE 481: Digital Systems Design Laboratory
CS 114: Introduction to Computer Programming
CS 124: Introduction to Computer Science
CS 325: Software Development and Systems
ECE 484: Computer Architecture
CS 116: Introduction to Problem Solving
ECE 380: Digital Logic
ECE 383: Microcomputers
ECE 486: Embedded Systems
OR
ECE 487: Embedded Systems Laboratory
ECE 494: Capstone Design

Figure 3. Prerequisite relationships among programming courses and core computer engineering courses within the UA ECE curriculum.

Third, students demonstrate a complete lack of understanding of the HLL constructs necessary for embedded systems programming. In some cases, this is due to a lack of retention as was discussed above. In other cases, the material is not presented to the students. General programming courses often have no reason to present programming skills applicable to embedded software. For example, programming skills related to bitwise operators, the address operator (“&”), pointer specifics, and ANSI C variable declaration keywords are very specific to embedded software. In many cases, general programming on a PC offers no valid reason to present this material. Whether it is lack of retention or lack of exposure, the poor assessment results associated with these types of concepts cannot be ignored.

5.3 Using the Embedded Systems Course

For the first attempt to integrate the desired embedded programming skills into the UA ECE curriculum, the senior-level ECE 486/487 Embedded Systems course was used. Supplemental material was presented in this course to fill in the gaps resulting from the typical two-level programming approach. To avoid the loss of significant lecture time within the Embedded Systems course, informal presentations of embedded programming skills were incorporated into the laboratory portion of the course. The specific embedded programming skills presented in this course closely followed the need areas presented in Table 1. Specifically, the strict use of ANSI C was required as the programming language. Device register interfacing using ANSI C constructs was required including the use of pointers, bitwise operators, and the address operator (“&”). Program structure was presented and students were required to write modular code using ANSI C subroutines. This program structure included emphasis on passing parameters by reference and by value as well as the advantages and disadvantages of global and local variables. Finally, resource constraints were presented by analyzing variable declarations and the memory requirements associated with those declarations.

The hardware environment used by the students for the ECE 486/487 course included multiple single-board-computers interfacing with each other and an analog-to-digital peripheral board within a VMEbus enclosure. The operating system used was a non-real-time version of Linux with specific custom C libraries for interprocess communication and shared memory. The analog-to-digital peripheral board was controlled through a set of control/status registers memory-mapped into the VMEbus address space.

Table 2.

Assessment questions and the corresponding percentage of students providing a reasonably correct response.

Questions	Results
1) In ANSI C, what are the bitwise operators for AND, OR, NOT, and XOR?	84%
2) In ANSI C, what are the logical-test operators for AND, OR, NOT, and XOR?	75%
3) Describe the difference between the ANSI C bitwise operators and the logical-test operators. When is each appropriate?	19%
4) Describe the difference between the ANSI C bitwise-AND operator and the ANSI C address operator. How are these two uses for the same character ("&") distinguished?	22%
5) In ANSI C, what is a pointer? How is a pointer specified? How is a pointer used? How is the pointer value related to the physical memory system of the computer?	31%
6) Describe the function of the following ANSI C keywords. How/when is each appropriately used? (const; static; volatile; extern)	75%
7) In ANSI C, how would one access a specific memory address?	38%
8) In ANSI C, what format is used for text characters? For text strings? How do you reserve memory space for strings?	6.3%
9) How are parameters passed between C subroutines?	9.4%
10) What is the difference between global and local variables in ANSI C? How are each type specified? What are the advantages of each type?	16%
11) Describe the operation of the ANSI C bit-shift operators "<<" and ">>".	31%
12) Describe the operation of the ANSI C directive "#include".	44%
13) Describe the output of a provided ANSI C code segment.	84%
14) Write an ANSI C “Hello world” program.	66%

There are several lessons learned from this approach. First, based upon the assessment data showing the limited programming skills of our students, any exposure to software development within the curriculum is important. Thus, adding a significant software development component to the ECE 486/487 course is viewed as a positive. Second, incorporating this material into the curriculum tends to break down the stereotypes associated with “software” engineers and “hardware” engineers. Although these distinctions exist in other disciplines, in the embedded systems world, engineers must be proficient in both the hardware and the software domains to be successful. Third, it is difficult to integrate introductory programming concepts into a senior-level course. Although attempts were made to present this material in the laboratory portion of the course to reduce lecture time dedicated to these concepts, the students cannot be expected to conquer more advanced concepts when such fundamental skills are lacking. For example, it is difficult to introduce real-time scheduling concepts to students that do not understand modular program structure and multitasking. Finally, programming skills are more important in an embedded systems course than the hardware platform used. Hardware platforms are plentiful and in many cases, design of embedded systems courses centers around the selection of an appropriate hardware platform. However, it is the programming skills that should take priority. As long as the hardware platform provides access to basic peripherals, device registers, and memory, the focus of the equipment should be the software development environment supported. Students without sound embedded programming skills will struggle regardless of the platform used.

Based upon the experiences at UA, the following recommendations can be made. First, presenting embedded programming concepts in an introductory programming course does not appear to be a good solution. The retention problem would persist when students are asked to recall the information for the first time years later. Also, early in the curriculum, students do not have an appreciation for the skill sets needed for embedded applications. Thus, presentation of these concepts later in the curriculum is recommended. A senior-level Embedded Systems course was used at UA for this purpose and the results were mostly positive. But, not every ECE curriculum has such a course, and there are problems associated with adding courses to a curriculum. Also, there is no debating the fact that introductory programming skills do not really belong in senior-level courses. Incorporating these concepts into ECE 486/487 replaces more advanced embedded systems topics that need to be included. Therefore, it is recommended that incorporation of embedded programming concepts should first occur in the microprocessors or microcontrollers course where assembly programming is presented. This course appears to be the best fit for the embedded programming topics. The interfacing aspects already present in such a course make it easy to include interfacing to device registers. Also, this course serves to introduce assembly programming. So, there is already a programming component to which embedded programming concepts can be attached. Presenting C and linking it to the underlying assembly is an especially attractive possibility in such a course.

5.4 Using the Assembly Language Course

Based upon the lessons learned from using the senior-level ECE 486/487 course for this purpose, embedded programming skills are now incorporated into the UA ECE curriculum using the ECE 383 Microcomputers course. This course traditionally involved assembly language programming and incorporated some treatment of the concepts related to embedded programming skills, although these were never the emphasis of the course. The traditional emphasis was on using assembly language on Intel 80*x*86-family microprocessors operating under Microsoft DOS and Windows operating systems, often as a component in larger HLL programs to address direct control of peripherals or time-critical tasks.

In redesigning ECE 383 to support embedded systems education, it was important to consider each of the areas of concern for embedded programming skills shown in Table 1. The first step in addressing these concerns was to change the computing platform used by the students. While the Intel 80*x*86-family microprocessors are quite powerful they are correspondingly complex. The Freescale (formerly Motorola) MC9S12-family microprocessors, and specifically the MC9S12DP256B, were selected as having a good balance between simplicity of programming model, wealth of incorporated peripherals, and availability of mature programming tools.

The first area of concern addressed in the redesign of ECE 383 is support for ANSI C as the choice of HLL. ANSI C is well-supported for the MC9S12DP256B both by professional toolsets such as MetroWerks CodeWarrior and by free alternatives such as GNU *gcc* [6, 12]. Support for ANSI C does not supplant the use of assembly code. While CodeWarrior supports assembly language code and the *gcc* package includes the GNU assembler *as*, the students will primarily use the freeware Freescale assembler *as12* or the shareware MiniIDE from MGTEK for stand-alone assembly programming [13].

The second area of concern addressed in the redesign of ECE 383 is support for peripheral interfacing using memory-mapped registers. The MC9S12DP256B contains a significant number of memory-mapped peripherals. Those used by the students include the Serial Communications Interface (SCI) that implements RS-232 communications, Serial Peripheral Interconnect (SPI), Inter-Integrated Circuit (IIC) bus, Controller Area Networking (CAN), an Enhanced Capture Timer (ECT) including pulse accumulators (PAC), analog-to-digital (A/D) converters, priority interrupt controller, 4kB EEPROM, 12kB RAM, and 256kB FLASH.

The third area of concern addressed in the redesign of ECE 383 is support for structured programming. The importance of structured programming models to a broad class of problems is emphasized. In their assignments, students write programs utilizing subroutines that are required to use ANSI C parameter-passing conventions so that C and assembly language routines may be readily interoperable. Several student laboratory assignments require the students to work in teams in which the team members are each assigned a different set of subroutines to write, test, and document before the group comes together to write complete programs using these routines. Once these subroutines are added to the students' "library" they continue to use them throughout the remaining assignments. Each student's laboratory report is required to detail their design process for each subroutine they write including the testing procedures for verifying the proper operation of the subroutine.

The fourth area of concern addressed in the redesign of ECE 383 is support for managing resource constraints. Memory usage, execution speed, and power consumption are all significant constraints for embedded systems. Many of the HLL data structures such as OO classes, arrays of structures, and linked lists are difficult or impossible to implement within the 12kB RAM of the MC9S12DP256B. Use of simpler data structures and a consideration of the most efficient data type for a given problem are emphasized. Execution speed is a limiting factor for time-critical operations. Students must be cautious to limit the execution time of subroutines, particularly interrupt service routines (ISR), that are called on a periodic basis so that one instance of the ISR will complete before the next is invoked. Power consumption is a significant concern for systems operating from battery supply, as is typical for many embedded systems. Power management is addressed from several directions. Peripheral units are switched off when not in use. The master system clock can be slowed if this does not impact time-critical functions. The microprocessor can be switched into one of several low-power "sleep" modes until needed.

Various assignments, particularly laboratory exercises, have been incorporated into ECE 383 to address these areas of concern. Students' first homework assignment in the updated ECE 383 is a prerequisite skills assessment covering general programming concepts. The results shown in Table 2 came from this assessment in the Fall 2006 semester and confirm anecdotal observations from previous semesters. The apparently poor retention of certain programming concepts at the onset of ECE 383 strengthens the rationale for incorporating a review of C programming concepts and constructs in parallel with the presentation of the assembly language for the MC9S12DP256B. This also serves to strengthen the concept of interoperability between low-level and high-level languages.

The students' first two laboratory exercises involve creating C-like subroutines for basic input/output capability on the MC9S12DP256B using its RS-232-compatible SCI. The students write assembly language analogues to the C *getc*, *putc*, *gets*, *puts*, *atoi*, and *itoa* subroutines which directly control the necessary memory-mapped peripheral hardware while providing a C-compatible subroutine parameter-passing interface. Comparisons with the memory footprint required by the *iostream*-based *cin* and *cout* methodology presented in the CS 114/116/124 prerequisite courses is provided in support of the need for the assembly equivalents. The subroutines developed for these first exercises will be used throughout the remaining exercises.

The third laboratory exercise involves creating software interfaces for a number of peripheral devices that must be managed simultaneously through the memory-mapped peripheral interface ports. Input is polled from a bank of simple switches and a telephone-style keypad and service by interrupt from the host PC via SCI. Output is to a bank of simple LEDs, four 7-segment LED digits, and to the host PC via SCI. Outputs to the discrete LEDs and 7-segment displays are multiplexed so the students' programs must continually refresh each digit within a given time threshold to prevent visual artifacts such as blinking digits. This provides a "soft" time constraint.

The fourth laboratory exercise involves interfacing with a more complex external peripheral. The students devise an interface for a 2 line x 16 character LCD panel. Successfully interfacing with the LCD requires a bidirectional interface with significant timing constraints. The LCD panel is used as a component, along with the 4x4 keypad from exercise 3, to form a simple four-function calculator.

The fifth laboratory exercise involves utilizing the PWM capabilities of the MC9S12DP256B. Two PWM channels are used to generate telephone-style Dual-Tone Multiple Frequency (DTMF) signals in response to input on the keypad. The input digits are displayed on the LCD.

The sixth laboratory exercise involves utilizing the ECT to create precise timing references to form the basis of a real-time clock with alarm capability. A periodic interrupt is generated to update a series of linked counters to keep track of human-readable time, *e.g.* seconds, minutes, hours, etc. Users of the clock are provided a simple push-button interface to set the current time and an alarm time. The PWM is utilized to create the alarm tone.

The seventh laboratory exercise involves utilizing the A/D capabilities of the MC9S12DP256B for sampling input signals and the SPI unit to interface with an external digital-to-analog (D/A) unit, the LTC1661. Students generate an application in which the A/D is used to sample one of 16 selectable input signals at a selectable sampling rate from 1kHz to 20kHz and the SPI-connected D/A unit is used to recreate the sampled signal. Students then explore the effect of the Nyquist criterion on a series of generated input signals of various frequencies with comparisons of the original signal and D/A regenerated signal on an oscilloscope.

The eighth laboratory exercise involves control of various motor types. Both the standard position-encoded servo motors and servos modified for continuous rotation are controlled in open-loop fashion utilizing PWM. DC motor speed is driven by PWM and closed-loop control is implemented using a tachometer through the PAC to monitor motor speed. A simple speed control algorithm is implemented to set the DC motor speed at a specified fraction of its peak speed. Four-phase unipolar stepper motors are controlled for speed and shaft position.

The ninth laboratory exercise involves development of a prototype "weather station" suitable for field deployment. Various sensor elements are interfaced by different means such as temperature via A/D, humidity via IIC, and wind speed via PAC. The sampling period is selected as variable from seconds to hours. To conserve power, peripherals and the microprocessor must be put into low-power modes between samples. The aggressiveness of the power management techniques is dependent on the delay between samples. For instance, the humidity sensor requires ten or more seconds from first applying power to stabilize its readings. If sampling times are shorter than a few seconds, it is not reasonable to turn the humidty sensor off. Sampled data is logged in an external EEPROM and, when requested from the host PC via SCI or another microprocessor via CAN, is packaged and sent for later analysis.

5. CONCLUSIONS

As embedded systems continue to increase in number and complexity, ECE curricula must address the embedded programming skills needed by their graduates. The typical ECE programming experience does not address these embedded programming concepts, often leaving graduates to acquire these skills on-the-job. This paper presents four areas of concern regarding embedded programming concepts not addressed in typical ECE curricula. These areas include a lack of C programming skills, inability to interface to registers, incomplete knowledge of appropriate program structures, and the inability to address resource constraints common in embedded systems.

To address these problems, ECE curricula must incorporate these concepts into existing programming courses or introduce new dedicated courses to address these specific topics. Each of these options presents its own set of concerns. The ECE curriculum at UA initially introduced these topics in a dedicated embedded systems course introduced at the senior level. The lessons learned from this experience indicate that these topics belong in the assembly language programming course found in typical ECE curricula. UA is currently implementing this change in its curriculum and is in the early stages of assessing its effectiveness.

6. REFERENCES

[1] Bjedov, G., Andersen, P.K., "Should Freshman Engineering Students Be Taught a Programming Language?", Proceedings of the 26th Annual Frontiers in Education Conference, Volume 1, Nov. 6-9, 1996, pp. 90-92.

[2] Bramer, B., Bramer, S., *C for Engineers, 2nd Edition*, John Wiley & Sons, New York, New York, 1997.

[3] Budny, D., Lund, L., Vipperman, J., Patzer, J.L.I.I.I., "Four Steps to Teaching C Programming", Proceedings of the 32nd Annual Frontiers in Education Conference, Volume 2, November 6-9, 2002, pp. F1G-18 - F1G-22.

[4] Davenport, D., "Experience Using a Project-Based Approach in an Introductory Programming Course", *IEEE Transactions on Education*, Volume 43, Issue 4, November 2000, pp. 443 – 448.

[5] Ganssle, J., "The Demise of the Embedded Generalist", Embedded.com, Available: http://www.embedded.com/showArticle.jhtml?articleID=51202213, November 2, 2004.

[6] GNU M68HC11 Project, Available: http://www.gnu-m68hc11.org/.

[7] Haberman, B., Trakhtenbrot, M., "An Undergraduate Program in Embedded Systems Engineering", Proceedings

of the 18th Conference of Software Engineering Education and Training (CSEET'05), April 18-20, 2005, pp. 103-110.

[8] Harbison III, S. P., Steele Jr., G. L., *C: A Reference Manual, 5th Edition*, Prentice Hall, Upper Saddle River, New Jersey, 2002.

[9] Joint Task Force on Computer Engineering Curricula, IEEE Computer Society, Association for Computing Machinery, "Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering", December 12, 2004, pp. A.43 – A.45, Available: http://www.computer.org/education/cc2001/CCCE-FinalReport-2004Dec12-Final.pdf.

[10] Kernighan, B. W., Ritchie, D. M., *The C Programming Language, 2nd Edition*, Prentice Hall, 1988.

[11] Kochan, S. G., *Programming in ANSI C*, Prentice Hall, Indianapolis, Indiana, 1994.

[12] MetroWerks CodeWarrior, Available: http://www.metrowerks.com.

[13] MiniIDE from MGTEK, Available: http://www.mgtek.com/miniide/.

[14] Nagurney, L.S., "Teaching Introductory Programming for Engineers in an Interactive Classroom", Proceedings of the 31st Annual Frontiers in Education Conference, Volume 3, October 10-13, 2001, Reno, Nevada, pp. S2C - 1-5.

[15] Parrish, A., Borie, R., Cordes, D., Dixon, B., Jackson, J., Pimmel, R., "An Integrated Introductory Course for Computer Science and Engineering", Proceedings of the 29th Annual Frontiers in Education Conference, Volume 1, November 10-13, 1999, pp. 11A3/12 - 11A3/17.

[16] Ricks, K. G., Stapleton, W. A., Jackson, D. J., "An Embedded Systems Course and Course Sequence", in Proc. of the 2005 Workshop on Computer Architecture Education (WCAE), Madison Wisconsin, June 5, 2005, pp. 46-52.

[17] Saks, D., "Representing and Manipulating Hardware in Standard C and C++", Embedded Systems Conference, Session ESC-243, San Francisco, California, March 6-10, 2005, Available: http://newit.gsu.unibel.by/resources/conferences%5Cesc_2004%5CSan_Francisco%5Cesc_243.pdf.

[18] Stapleton, W. A., Ricks, K. G., Jackson, D. J., "Implementation of an Embedded Systems Curriculum", in Proc. of the 20th International Conference on Computers and Their Applications (CATA'05), New Orleans, Louisiana, March 16-18, 2005, pp. 302-307.

[19] 1999/2000 TRON Association Survey, Available: http://www.ncsu.edu/wcae/ISCA2005/submissions/ricks.ppt.

[20] Turley, J., "Survey says: Software Tools More Important than Chips", Embedded Systems Design, Available: http://www.embedded.com/showArticle.jhtml?articleID=160700620, April 11, 2005.