# An Experimental Laboratory Environment for Teaching Embedded Hardware Systems

Dennis Brylow

Marquette University
MSCS Department – Cudahy Hall
1313 W. Wisconsin Ave., Milwaukee, WI 53226
brylow@mscs.mu.edu

## Abstract

*This paper describes Marquette University's efforts to build an experimental embedded systems laboratory for hands-on projects in an introductory hardware systems course. Our prototype laboratory is now serving as the basis for a coherent sequence of class projects threaded throughout subsequent courses in operating systems, networking, and embedded systems, among others. We describe the major components of our laboratory environment, how it is used in our hardware systems course, and how this has contributed to significant improvements in other core courses in our curriculum.*

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; K.3.2 [**Computer and Info Science Education**]: Curriculum

Keywords: Embedded systems education, XINU, curriculum

## 1. Introduction

This paper presents our prototype for an experimental embedded systems laboratory and our design for an introductory hardware systems course based on that laboratory.

The degree to which undergraduate computer science majors study topics related to computer organization and hardware systems varies enormously within the world of university and college education[11]. Programs can range from multiple required courses in digital logic, assembly language programming and architecture, to programs with only a smattering of computer organization topics combined with another course, like operating systems. While the community as a whole seems to acknowledge that some aspects of computer hardware remain a core topic in the curriculum, there is considerable debate on the proper scope and contents of computer architecture courses in computer science education. Against this backdrop, we intend to address the shortcomings of a particularly common point in this wide spectrum, the introductory hardware systems course aimed primarily at second- or third-year undergraduates majoring or minoring in computer science. (These courses are also sometimes called an "introduction to computer architecture," or "computer organization," or even just "assembly language"; these titles can have significant distinctions in meaning, but for the remainder of this paper, we will refer to them collectively as "introductory hardware systems" courses.)

There are several inherent barriers that can prevent mainstream computer science (CS) departments from offering well-integrated hardware systems courses. Particularly in departments on the software and mathematical ends of the CS spectrum, most faculty members themselves have had only one or two undergraduate courses in hardware systems and are reluctant to teach a course so far outside of their research area. Hardware systems courses are among the most likely to be assigned to visiting or adjunct faculty, or to be relegated to an electrical engineering or computer engineering department. The natural result of this is a tendency to have hardware systems curricula that are at best loosely coupled to the other computer science core courses, and at worst wholly unrelated. With this in mind, our first goal in this work can be summarized as follows:

> Our hardware systems curriculum must integrate strongly and smoothly with the other core computer science courses.

Strong integration entails that subsequent courses draw and build upon the most important themes in the hardware systems course. Smooth integration requires that this not unduly distort the natural contents of other courses, or hijack the overall goals of the undergraduate curriculum.

As a matter of teaching philosophy, we prefer courses that take a hands-on, bottom-up approach to teaching complex systems, whether in software or hardware. Yet, we do not have time for our second-year CS majors to absorb an entire computer engineering degree within their hardware systems course. Therefore we must draw lines of abstraction most appropriate for students primarily concerned with software. Our second goal is then:

> Our laboratory environment must allow students to work with the lowest levels of software, interacting directly with hardware.

The primary challenge here is allowing a free range of experimentation for the students without compromising the production computing environment.

In addition to curricular challenges, we also face resource limitations. Our department, like many others, must carefully choose how to allocate its space and equipment budget. Rather than see this as a disadvantage, we view this as a valuable focusing constraint for the current project. Our laboratory environment has been designed from the outset to require low investments in both cost and dedicated space, making it suitable for adoption at other universities and colleges that could not afford to construct a large, specialized facility for these purposes.

> Our laboratory environment must be inexpensive, flexible, and readily duplicable.

Finally, our infrastructure will be of greatest use if it not only supports a wide variety of teaching needs, but also enables a closely related research agenda. This brings us to our fourth major goal:

> Our laboratory environment must serve as a platform for research and development of embedded and real-time software systems.

Embedded systems are an important class of computers in the modern world, deeply intertwined in applications ranging from medical devices to vehicle control, from consumer appliances to advanced communication equipment. Embedded systems are often microcosms of more complex computer systems; they possess more limited resources, yet are representative of many of the most important concepts covered in a hardware systems curriculum.

In the following sections, we present our prototype for a laboratory environment designed to support all of the aforementioned goals. Our design emphasizes real exploratory exercises; students run their software directly on the embedded processor, without simulators or emulators. The target platform is an inexpensive embedded system commonly available in home electronics stores, and our software is open-source. This paper is intended to be the first of a sequence outlining our prototype design and its impact on our courses, in this case, our hardware systems course. For the remainder of the paper, we concentrate on previous and related work, describe our laboratory environment hardware and software, and outline the resulting hardware systems course design.

### 1.1. Prior and Related Work

Purdue University's XINU Laboratory[5] has served as a model of experimental, hands-on project environments for two decades. Originally designed to teach operating systems courses on the PDP-11 platform, the XINU operating system was later ported to the Motorola 68000 processor[9] and the Intel x86 architecture[7]. Various extensions were made to support highly successful courses in networking and internetworking[8, 19]. Although highly successful at Purdue, several factors kept the Purdue XINU Lab from being easily duplicated elsewhere. As the PC market exploded, making surplus Intel x86 systems plentiful, a very narrow range of supported network interface cards made it difficult for other schools to acquire compatible target machines. More importantly, the Purdue lab relies on both custom-made hardware and software that requires specialized knowledge, as well as a fair amount of time and money, to produce elsewhere. Even at Purdue this laboratory scaled poorly; this author was an integral part of a year-long effort to revitalize and expand the Purdue XINU Lab to triple its original size in 2001, an effort that exposed many of the shortcomings of the customized components required.

The TinkerNet Project[10, 20] reproduces the overall architecture of the Purdue XINU Laboratory without the need for specialized hardware or a current port of the XINU operating system. Designed to support hands-on, bottom-up projects in networking courses, TinkerNet relies on a base operating system with working network driver on the target platform in order to do its work. However, courses that build software at the operating system driver level or below cannot safely make this assumption.

Several others have proposed various embedded systems laboratories. Northeastern[2] and Cal Poly State[12] have both proposed courses targeted at the Blackfin DSP architecture. Michigan State[4] proposed integrating embedded systems concepts across several of their courses, but did so using a variety of simulation tools and platforms. University of Alabama[17] has incorporated a sequence of seven courses focused on embedded systems into their core computer engineering curriculum. Their platform is based on the VMEbus architecture, and their methodology is quite similar to ours, albeit on a much larger, computer engineering-focused scale that may not translate easily into a computer science major.

FernUniversität Hagen[1] has proposed an impressive array of web-accessible laboratory infrastructure for microprocessors, which faces some of the same remote access and reservation problems we have dealt with.

On the non-academic side, the OpenWRT Linux community[16] has been instrumental in exploring and documenting the platform we have adopted, and continues to provide a high-quality Linux distribution for this system.

## 2. The Platform

In selecting a target platform for our hardware systems laboratory, we desired an embedded RISC architecture with multiple peripheral types, and at least one simply-programmed I/O channel. We wanted an inexpensive, readily available configuration that could support basic assembly language assignments for exploring aspects of the machine, yet one that was not too small to support projects of realistic complexity. There are many possibilities, but we have found success with a line of consumer network devices that are based upon a sophisticated, general purpose embedded platform.

Our laboratory is based on the LinkSys WRT54G[14] series of wireless routers, which ships with a wide area network (WAN) port, four local area network (LAN) ports, and a wireless network interface. Internally, the WRT54G contains one of several models of the Broadcom BCM 47XX/53XX family of "system-on-a-chip" ("SoC") platforms, a 200MHz, 32-bit embedded MIPS[18] architecture with 16 MB of RAM, and 4 MB flash ROM. (Some newer models are now shipping with smaller memory configurations.) Unpopulated pins on the circuit board include two UART serial ports and a JTAG interface suitable for interfacing directly to the processor.

With only slight modifications, we have access to two serial ports on the embedded device, and the first of these is already preconfigured as a firmware console interface.

Like many modern architectures, the WRT54G boots to a flexible firmware that performs basic power-on testing, configures devices, and selects an operating system (O/S) to boot. Under normal circumstances, the WRT54G selects a Linux- or VxWorks-based embedded O/S image in the flash ROM, and proceeds from there. With a couple of simple Common Firmware Environment (CFE)[13] commands at the boot prompt, the system can instead perform a relatively complex remote boot, in which it 1) configures the network interface, 2) downloads a file into RAM over the network connection, and 3) boots the kernel image it has loaded into RAM. In this way the system can be made to run student-produced code directly, with only a few seconds delay. At the same time, the production firmware image on the device is not disturbed, and the system can once again become a fully-functional wireless router after merely a power-cycle.

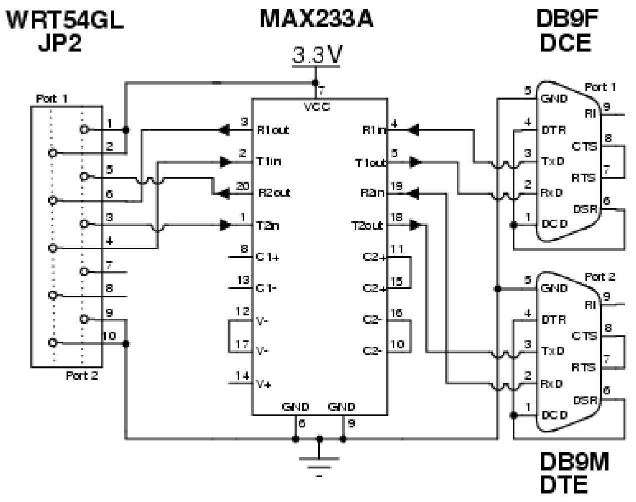

**Figure 1. Dual Serial Transceiver**

### 2.1. Input and Output

The unpopulated, internal serial port pins for the WRT54G are operating at the processor's TTL supply voltage, 3.3 volts. Two important pieces are needed to establish serial communications: 1) a serial transceiver to convert the TTL logic voltages into RS-232 voltages, and 2) an external jack for connecting a serial port.

The first problem can be conquered using a dual RS-232 transceiver (Figure 1) that takes care of producing both positive and negative RS-232 voltages of higher magnitude than the power supply voltage.

External jacks are not necessary for experimentation, but long-term laboratory application is safer and easier if the serial ports are accessible while the device is closed in its case. Our project website[3] details construction of the transceiver circuit and modification of the plastic case to make room for standard DB9 serial connectors.

## 3. The Warzone

The WRT54G can be used alone at any workstation with a serial port and a network connection. This configuration may be suitable for very small laboratory installations concerned only with hardware systems assignments, but to take maximum advantage of the power of our platform it is far more useful to collect many WRT54G's as a pool of *back-ends* doled out by a central server. Particularly in the context

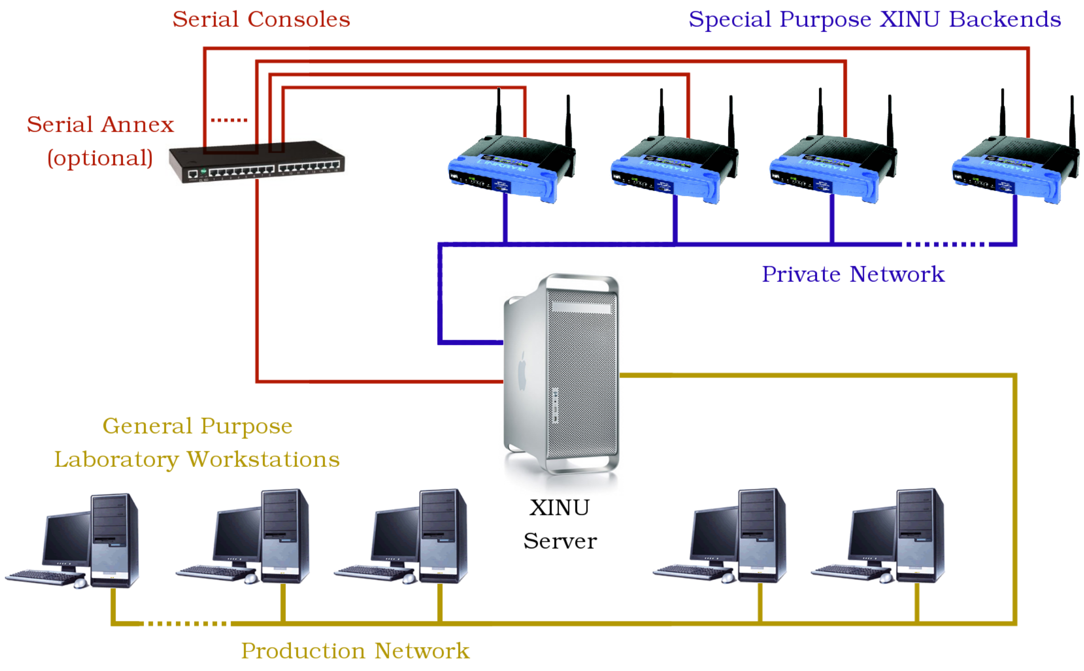

**Figure 2. Laboratory Topology**

of more advanced courses in networking, it is best that this pool of backends be managed on a private network, isolated from the production network by a gateway.

Figure 2 outlines the layout first used at Purdue[8] and later nicknamed the *warzone* by the TinkerNet Project[20]. A cadre of dedicated target machines resides on the private network (warzone) where students may load and run experimental kernels on demand. These dedicated, backend machines load a new O/S image from the central server across the network each time they reboot, allowing students to refresh the image frequently as they build their system.

Marquette's warzone is managed by a single G5 XServe with one network interface on the production network, and a second network interface connected to a private switch that links all of the backend machines to their own subnet. Our infrastructure can handle a heterogeneous mix of backend architectures, allowing both newer and older models of backend, as well as a mix of alternative architectures (RISC machines, CISC machines, processors from a variety of manufacturers.) This heterogeneity can be both cost effective and a desirable pedagogical feature.

It is important to note at this point that since our backends are in fact fully operational wireless routers, the wireless NICs should normally be deactivated within their native firmware, in order to protect both the warzone and the production wireless network from stray traffic. In more advanced networking courses, explicit use of the wireless interface may be highly desirable – the impact of this on nearby production wireless networks should be carefully gauged accordingly. We keep our wireless interfaces turned off, the antennae unscrewed, and the routers locked in a grounded metal case during the normal course of hardware systems and operating systems courses.

## 3.1. Serial Ports

Collected into a pool of backends, the number of interfacing serial ports required quickly exceeds the number present on any typical server machine. As shown in Figure 2, an optional serial annex can provide dozens of additional serial ports relatively cheaply. Details of our serial annex configuration can be found at the project website[3].

## 3.2. Reservation System

The central server regulates access to the backend console connections, kernel upload area, and warzone network via a simple reservation system, called the *console daemon*. The console daemon is network accessible, and can be contacted from any general purpose workstation. In this way, actual development and compilation can be carried out from any machine on the Internet; students compile their project, connect to the console daemon, upload, boot the backend, and interact with the running operating system console at will. We have automated this entire process using a simple, extensible script built on top of the console daemon utilities.

The console daemon system allows users to request backends by name or by type. Associated tools let users view the status of the entire backend pool, and manually bump users that have held a backend connection for longer than an administrator-configured maximum time period.

The software for the console daemon is largely based on the connection server ("cserver") daemon developed originally at Purdue[8]. Our contributions have been to port the original tools to a modern dialect of C, update them to work with current Linux and Solaris distributions, add TCP Wrapper security controls to the network daemon, adapt it to a new serial annex, and write proper documentation for the system. The code for the console daemon and frontend utilities is now available as an autoconf-enabled release from our website[3], and builds successfully on a wide variety of POSIX-compliant platforms. Our significant improvements to the console daemon have been released as open source with the permission of original contributors.

### 3.3. Rebooting

When developing complex systems, students can expect some crashes. In stand-alone installations a developer resets the target machine, but in a larger scale laboratory environment students may not have easy access to the backend machines, and remote users may have no access at all.

Remote reset of backend machines is accomplished in Purdue's laboratory using special, custom-made hardware and software designed by this author in 2001. Cognizant of the high costs in manpower and equipment that this entails, we have sought a simpler, readily scalable solution. Fortunately, a laboratory targeting embedded devices with no moving parts is amenable to simple powercycling, rather than formal resetting of the motherboard. Various serial-controlled power strips are now commonly available for applications in server rooms at a fraction of the cost of custom-building a comparable system of serial-controlled, solid-state relays. Our project website[3] details the off-the-shelf hardware and open-source software we have employed for this.

Using such hardware, remote users of the laboratory can send a command via the console daemon to immediately powercycle their backend with a single key stroke.

An important consequence of this feature is that backend and server hardware need not be adjacent to student work space at all. The laboratory hardware can be locked away on a rack in a data closet or server room, and courses can be taught using the infrastructure from any general purpose computer laboratory.

Having detailed the basic hardware requirements of the experimental laboratory, we now describe the use of this equipment in the context of a hardware systems course.

## 4. The Hardware Systems Course

Marquette's introductory hardware systems course (COSC 065) is three credit hours and is designed to meet the "Architecture and Organization" requirements (AR1 through AR8) in the ACM/IEEE Model Curriculum[15], corresponding closely with the model course CS220 Computer Architecture.

The stated goals for COSC 065 are as follows. Students completing the course successfully will be able to:

- Understand the principles underlying computer hardware systems, and see how they shape software,
- Define and describe the purpose of major components in modern computer hardware, and understand how they work together to accomplish computing,
- Solve problems in assembly language for a modern computer processor.

Additional, unstated goals for the course include:

- Preparing students with platform-specific knowledge they will need for subsequent courses in operating systems, networking, embedded systems, and compilers.
- Familiarizing students with development tools they will need for system software development in C or other lower-level languages.

The current textbook for the course[6] covers the essential topics suggested by the ACM/IEEE Model Curriculum[15] for a hardware systems course targeted at computer science majors, and the course outline follows more or less along the book's outline.

- Digital logic, combinational and sequential logic,
- Data representation,
- Processors and pipelines,
- Instruction sets, both RISC and CISC,
- Assembly languages and addressing modes,
- Activation records,
- Memory and storage, virtual memory and caching,
- I/O, buses, and interrupts.

All of these topics can be directly addressed with hands-on laboratory assignments that both build on previous topics and serve as excellent preparation for subsequent courses.

The textbook is supplemented with technical documentation for the MIPS architecture and instruction set, along with copious examples provided in lecture.

### 4.1. Laboratory Exercises

The first third of the course supports lecture topics with digital logic assignments. Students build combinational logic circuits out of 74LS00-series logic gates, resistors, switches, and LEDs. The fifth laboratory culminates in building a full-adder, which $n$ lab teams then combine together into an $n$-bit ripple adder.

As the lecture moves into machine and assembly language topics, the laboratory projects move to writing MIPS assembler programs that run directly on the WRT54G backends. As outlined in the next section, a small loader and operating system base supports all of the system library calls typically invoked by students learning assembly programming. The infrastructure is sufficiently flexible to support assembler projects that bear no relationship to projects in later courses, but we prefer the projects that build steadily toward our long term goal:

- Basic assembler operations; Typical assignments include simple arithmetic calculations (exponents, averages,) and table-driven output (Celsius to Fahrenheit conversion chart, pounds to kilograms, etc.)
- I/O, converting between character and integer representations; e.g. simple calculator with input in several possible bases, or perhaps iterative greatest common denominator.
- Functions and calling conventions; e.g. breaking the calculator into multiple functions or building functions for printing out chessboard states.
- Recursion and stack frames; e.g. classic recursive problems like Towers of Hanoi, N-Queens or Knights-A-Visiting.
- Pointers and memory; e.g. allocating memory to store and reverse an arbitrary list of integers, or exploring the behavior of the memory allocation subsystem.
- Device I/O; e.g. address serial port control registers directly to send and receive characters. As an added challenge, this requires interpreting technical documentation for the 16550 UART interface.
- Interrupts; e.g. write a small interrupt handler for counting timer clicks or perhaps using the UART in asynchronous mode.

The tools for interacting with the laboratory environment are no more elaborate to use than invoking a typical simulator, or executing a program at the command line.

### 4.2. Embedded Programming Environment

Students run their code assembly code by first linking it to a starter kernel and then running a tool that automatically uploads the executable to the server, restarts the backend, and connects to the serial console. The starter kernel is a relocatable object file containing a simple boot loader and a small library of useful system functions. Software is assembled and linked using a standard version of the Gnu Compiler Collection (gcc) configured as an architectural cross-compiler targeting the embedded MIPS-32 platform. Students need only the starter kernel image and an appropriate Makefile in the current directory, and the lab tools and cross-compiler in their path.

The starter kernel is a stripped down XINU image with a single process, the I/O device drivers for the serial ports, and the basic library functions typically used by introductory assembly programmers:

- getchar() and putchar(), getInt() and printInt(),
- printf(),
- malloc() and free(),
- sleep() and halt().

The XINU image runs without kernel protection, so students have the freedom to explore the entire range of RAM and memory-mapped I/O regions on the device. Over the course of the term, they can take over control of the serial ports directly, and we provide hooks for them to install interrupt handling code, as well.

In the operating systems course (the following term) students see the source code for the starter kernel, and use their own serial driver as a starting point to replace or extend many components of the system. By the end of the two term sequence project partners in the course will have co-authored most of approximately 6,000 lines of C and assembly source comprising a preemptive, multitasking embedded operating system with a variety of components and peripherals.

## 5. Discussion

In this section, we assess the effectiveness of these curriculum changes by evaluating the extent to which we have met our initial design criteria.

### 5.1. Integration

Our first goal was to develop a hardware systems curriculum that would integrate strongly and smoothly with the other core computer science courses. Indeed, the current incarnation of hardware systems draws directly upon material in its prerequisite course on discrete and combinatoric algebra, (representation, logic), and deliberately complements topics, (recursion, functional decomposition, etc.), from the data structures course taken concurrently.

Furthermore, hardware systems dovetails directly into our operating systems course, which actually builds upon assignments from hardware systems. Ongoing revisions to later courses in compilers, networking, and embedded systems will also build directly upon earlier course assignments in both hardware systems and operating systems. In all of these cases, our new thematic approach has only moderately impacted lecture content, while significantly focusing practical assignments.

## 5.2. Interaction

We have also met our goal of building a laboratory environment that allow students to work with the lowest levels of software; our hardware systems assignments manipulate memory and registers directly, and interact with I/O hardware and interrupt processing. The operating systems assignments add process context switching, timer preemption, memory management, and I/O buffering. Proposed networking assignments deal directly with network interface drivers and the construction of a network protocol stack.

## 5.3. Scalability

Our laboratory environment is inexpensive, flexible, and readily duplicable, addressing our scalability goals. Individual backend units cost about \$50, with around \$10 of additional hardware required for the serial port transceiver. Connecting serial ports cost approximately \$40 per port in bulk (using either a large terminal annex, or USB-enabled serial converters on a hub for smaller quantities.) Assuming that a suitable server and spare network connections are on hand, the total amortized cost per backend unit is less than \$125 for a small installation. Allowing for a deluxe installation with rebooting hardware, surplus serial ports, and assorted cables, a deluxe laboratory with 30 backends still costs under \$150 per unit.

A single backend machine is typically suitable to provision four teams of two students each for a course in hardware systems or operating systems; courses in embedded systems and networking may require a higher ratio of backends to students for projects that make use of multiple backends per running assignment. For example, the Marquette laboratory is a medium-sized installation of 10 Linksys WRT54G backends, and 10 PowerPC G3 backends.

We have taken great pains to use off-the-shelf hardware and software tools to build our laboratory; where this has not been possible, we have carefully documented our hardware work on the project website[3], and made our software freely available.

Other schools are already taking advantage of this; two are in the preliminary stages of reproducing our laboratory design in their contexts, and several others have expressed an interest.

## 5.4. Research

Our laboratory environment also meets the goal of serving as a platform for research into embedded and real-time system software. Based upon our strong results in building this initial prototype, we have received external funding to support research into IP telephony and real-time scheduling on this platform. The limited resources and interesting peripherals already present on this platform make it an ideal target for investigating real-time and interrupt-driven issues in realistic embedded systems. In terms of longevity, the platform is well supported by the Linux community, and the primary vendor has shown every sign of continuing to support and improve this product family.

## 5.5. Student Evaluation

Student reaction to introduction of our laboratory environment has been overwhelmingly positive. Undergraduates have volunteered their time over semester breaks to make serial transceiver modifications to backends. One student who had already taken a prior incarnation of the hardware systems course sat in on the entire semester of the new course in order to experience the practical laboratory environment.

Our department has only just begun a rigorous assessment plan this past term, so we do not have hard data to compare student outcomes against the earlier, unfocused incarnation of the same course. Anecdotal data, however, suggests marked improvement in our undergraduates' preparation for subsequent systems courses, and increased interest in embedded devices and networking.

## 6. Conclusion

We have presented our design for an experimental laboratory environment used to focus and enhance practical assignments in the assembly language phase of our introductory hardware systems course. The laboratory is scalable, inexpensive, and comprised entirely out of readily available parts and software. Several other schools are already working to duplicate our infrastructure for their own courses.

The hardware systems course we have developed in concert with this laboratory integrates smoothly into our existing computer science curriculum, but also makes possible a coherent sequence of cross-course class projects that have significantly enhanced several of our other offerings.

Our operating systems course builds directly upon the hardware systems course, allowing students to build their own small, but powerful embedded operating system within the two terms of their second year.

Our work focuses on hands-on class projects that encourage students to explore both the lowest levels of hardware/software interaction and the breadth of embedded systems acting in the modern world. Our chosen architecture is a ubiquitous home networking appliance, which despite its lowly intended application, makes for a powerful platform for devising a wide range of challenging assignments.

## 6.1. Contributions

Our contributions include:

- a modern adaption of the "warzone" laboratory design using off-the-shelf hardware and open-source software,
- the first public release of a comprehensive set of console tools and reservation management software specifically for running a laboratory of this type,
- the first port of the venerable XINU operating system to a modern, embedded RISC architecture,
- a layer of wrapper functions over the XINU system calls appropriate for use by introductory hardware systems students, and,
- a novel hardware systems curriculum design that meets the ACM/IEEE model curriculum guidelines while supporting a thematic focus on embedded systems software, beginning in the hardware systems course and continuing through potentially at least three more semesters.

Our future work includes fleshing out design plans for more embedded XINU-based coursework in advanced operating systems, networking and internetworking, wireless and distributed computing, and embedded real-time systems, all on this platform. The first full release of embedded XINU is planned for this summer, and we hope to extend our hardware and software to support an IP telephone within the year.

We gratefully acknowledge the assistance of Douglas Comer (Father of XINU), and of Cisco Systems and Linksys. The OpenWRT Linux community has been a source of much information that would otherwise have remained hidden. Many thanks go to the persevering students who have worked hard to make the Marquette Systems Lab a success, both by enduring their required courses, and giving graciously of their time just to make things work better.

## References

[1] H. Bähring, J. Keller, and W. Schiffmann. Remote operation and control of computer engineering laboratory experiments. In *WCAE 2006: Workshop on Computer Architecture Education*, 2006.

[2] M. Benjamin, D. Kaeli, and R. Platcow. Experiences with the blackfin architecture in an embedded systems lab. In *WCAE 2006: Workshop on Computer Architecture Education*, 2006.

[3] D. Brylow. Embedded XINU project wiki, 2007. `http://mulug.mscs.mu.edu/wrt-wiki`.

[4] B. H. C. Cheng, D. T. Rover, , and M. W. Mutka. A multipronged approach to bringing embedded systems into undergraduate education. In *Proceedings of ASEE 98: American Society for Engineering Education*, 1998.

[5] D. E. Comer. *Operating System Design: The XINU Approach*. Prentice Hall, 1984.

[6] D. E. Comer. *Essentials of Computer Architecture*. Prentice Hall, Upper Saddle River, New Jersey, 2005.

[7] D. E. Comer and T. V. Fossum. *Operating System Design: The XINU Approach*. Prentice Hall, PC edition, 1988.

[8] D. E. Comer and J. C. Lin. A laboratory environment for experimenting with XINU. Technical Report CSD-TR 96-047, Purdue University, 1996.

[9] D. E. Comer and S. Munson. *Operating System Design: The XINU Approach*. Prentice Hall, Mac edition, 1989.

[10] M. Erlinger, M. Molle, T. Winters, C. Lundberg, and R. Shea. Tinkernet: A low-cost networking laboratory. In *ACE 2004: Sixth Australasian Computing Education Conference*, Australian Computer Society, 2004.

[11] Joint ACM/AIS/IEEE-CS Task Force Computing Curricula. Computing curricula 2005: overview report, March 2006.

[12] D. Franklin and J. Seng. Experiences with the blackfin architecture for embedded systems education. In *WCAE 2005: Workshop on Computer Architecture Education*, 2005.

[13] M. Lichtenberg. *Common Firmware Environment (CFE) Functional Specification*. Broadcom Corporation, Irvine, CA, 1.6 edition, 2004.

[14] Linksys. WRT54G wireless-G broadband router, 2007. `http://www.linksys.com`.

[15] Joint IEEE Computer Society/ACM Task Force on Model Curricula for Computing. Approved final draft of the computer science volume, Dec 2001.

[16] OpenWRT linux documentation, 2007. `http://wiki.openwrt.org/`.

[17] K. G. Ricks, W. A. Stapleton, and D. J. Jackson. An embedded systems course and course sequence. In *WCAE 2005: Workshop on Computer Architecture Education*, 2005.

[18] MIPS Technologies. *MIPS32 4K Processor Core Family Software User's Manual*. Mountain View, CA, 2002.

[19] C. A. Telfer. *Abstractions and Efficient Implementation of Automatically Reconfigurable Network Testbeds*. PhD thesis, Purdue University, 2003.

[20] T. Winters, R. Ausanka-Crues, M. Kegel, E. Shimshock, D. Turner, and M. Erlinger. Tinkernet: A low-cost and ready-to-deploy networking laboratory platform. In *ACE 2006: Eighth Australasian Computing Education Conference*, Australian Computer Society, 2006.