Teaching Microprocessor Systems Design Using a SoC and Embedded Linux Platform [1]

Yann-Hang Lee and Aung Oo
Department of Computer Science & Engineering
Arizona State University
yhlee@asu.edu, aung.oo@asu.edu

Abstract

In traditional microprocessor systems design courses, students learn to develop assembly language programs to control peripherals, handle interrupts, and perform I/O operations. We adopt a 32-bit StrongARM architecture on the Motorola MX1ADS board with Embedded Linux to present a modern microprocessor system design course. With this new platform, we use a high-level language to develop projects that accelerate the students' learning curve. Embedded Linux also provides the necessary flexibility and tool set required for students to debug their own projects. Our students' responded very positively to this change. They were excited about the renewed course structure, the updated learning environment, and the challenging projects.

1. Introduction

Embedded systems are designed for dedicated applications running in control systems. The unique feature of such systems is the capability to perform timely and predictable operations in response to concurrent requests arriving from the external environment. To create an effective embedded system one must properly employ the appropriate system architecture, hardware/software interfaces, peripheral devices, and software components. Currently, embedded systems companies are facing with a shortage of engineers having the appropriate skills to respond to market opportunities [8]. Therefore, embedded software engineering has emerged as a key element for curriculums in Computer Science, Computer Engineering, and Electrical Engineering at universities throughout the world.

To teach the subject of software/hardware integration and I/O interfaces, undergraduate computer science and engineering programs incorporate a microprocessor system and applications course. In the course, students develop assembly language programs to control peripherals, handle interrupts, and perform I/O operations. Then students perform experiments with a target single-board microprocessor system integrated with typical interface circuits such as programmable timers, serial ports and parallel ports. Unfortunately, this approach fails to keep pace with industry technology. This lag is prompted by the advent of rapid prototyping development of microelectronic systems that includes:

a. SoC-based platforms for embedded applications: The system-on-a-chip (SoC) devices have made great progress along with the ever-growing number of transistors that can be integrated on a chip.

b. Abundant I/O interfaces: Besides programmable timers, serial ports, and parallel ports, there are several new I/O standards designed for human interfaces, multimedia, networking, and inter-IC/device communication.

c. I/O programming with high-level languages: For software portability, modularity, and readability, high-level programming languages have been used in all levels of software development. An appropriate use of programming languages and software structures often leads to reusable embedded software.

Our traditional computer engineering curriculum also taught relatively outdated techniques in the subjects of software/hardware integration and interface. The "Microprocessor System Design" course emphasizes assembly language programming and exercises only a limited number of I/O interfaces. The course falls short in addressing state-of-the-art interfacing technology and emerging applications.

In our curriculum development project sponsored by the NSF EIA program, we redesigned the microprocessor system design class. Our goals were to provide a learning environment which aligned with emerging technology and improved the effectiveness of instruction. We also developed a laboratory environment which incorporated cutting-edge programming approaches to manage hardware components in SoC platforms. This renewed course goes beyond the inclusion of various interfaces and devices. The course focuses on the appropriate software

[1] This course development project is supported in part by NSF Educational Innovation Grant EIA-0122600, the Consortium for Embedded and Inter-Networking Technologies (CEINT), and Motorola University Program.

structures using a mixture of high-level and assembly language programming, I/O operations in modern operating systems, and reusable software components.

In this paper, we will explore the challenges and successes we encountered in implementing this new microprocessor system design class. The course serves as the first of three embedded system courses in our curriculum. Section 2 presents background information on the embedded system curriculum at Arizona State University (ASU). In Section 3, we will present the new course design followed by the course objectives, the course material and the setup of the laboratory environment for programming projects. Section 4 will cover some of our lessons learned and feedback from our students. In Section 5 we conclude our discussion.

2. Background

ASU, Motorola, and Intel formed a not-for-profit Consortium for Embedded and Inter-Networking Technologies (CEINT) in 2001 [3]. CEINT developed an infrastructure to support a strong curriculum in embedded systems. The end product was a concentrated path in Computer Systems Engineering, which consisted of an Embedded Systems Development, Embedded Systems Engineering, and Embedded Systems Capstone course [1].

We wanted to provide students with the opportunity to learn practical development techniques using the Embedded Systems Development course. To accomplish this goal, we chose Motorola MX1ADS boards using MontaVista's HardHat Linux Toolkit. Although we discussed both assembly level and high level programming development, C was the main language used for developing projects. This particular combination of programming language, development environment, and microcontroller architecture is rare for an introductory level embedded systems class.

At the same time, the students were challenged to get quickly up to speed on the fundamentals required to use the new development environment and tools. Most of the students did not have strong backgrounds in developing software for Linux. To lessen this steep learning curve, we provided laboratory demonstrations and walked through simple development projects in small groups. We also provided online tutorials, sample Linux drivers, and low level C code examples for students to study.

In this course, we introduced students to memory devices, memory controllers, buses, handling interrupts, DMA, timers, counters, UART, SPI, I2C, parallel I/O, keypad, LCD, touch panels, and A/D - D/A converters. The students also developed device drivers for timers, PWM, UART, gpio, and SPI eeprom as class projects. Other available features such as watchdog timer, blue tooth technology, USB, and CMOS sensors were left for more advanced courses in the sequence.

Assembly language teaches the students about the detailed architecture of the hardware. This gives students an appreciation for high level constructs implemented in assembly language [2]. However, implementing all software programs in assembly language neither practical nor desired. In fact, assembly-language programming is no longer the best choice for developing embedded systems, due to the availability of excellent compilers and the rising complexity of software projects [6][9].

3. Course Design

3.1. Course Objectives

The objectives of the course are to familiarize the students with hardware-software interfaces, hardware designs of microprocessor systems and peripheral devices and their communication protocols. Students work at acquiring technical knowledge and applying this knowledge to the development of programs for controlling peripheral devices and interfaces. Thus, the students learn to analyze and synthesize suitable solutions for building integrated hardware/software systems capable of interacting with external world.

3.2. Course Content

The revamped course places emphasis on software/hardware integration and I/O programming, the incorporation of the state-of-the-art SoC platforms, and emerging embedded system development tools. Our plan is to gear the integration of hardware modules to construct embedded systems and the programming models and characteristics of various I/O interfaces and peripherals. The course syllabus is established as follows:

Course Syllabus: Microprocessor System Design

Course Goals:

- Develop an understanding for using a CPU core as a component in system-level design.
- Develop the ability to integrate the CPU core with various interface units in embedded systems.
- Gain the necessary skills for programming and debugging I/O operations to manage peripherals for embedded applications.

Major topics covered:

- Introduction and review of instruction set and assembly language programming, instruction execution cycle and timing (4 lectures)
- C programming for embedded systems (2 lectures)
- Interrupts and I/O multiplexing (2 lectures)
- Parallel I/O interface and signal handshaking (1 lecture)

- Timers and counters (2 lectures)
- Serial communication: UART, SPI, and I2C (4 lectures)
- Keypad and LCD interfaces (3 lectures)
- Transducers and sensors, touch panels, A/D-D/A converters (3 lectures)
- Memory devices, SRAM, DRAM, flash memory, and SDRAM controller (3 lectures)
- Buses, access arbitration, timing, and bus protocols (2 lectures)

Laboratory projects:

- Introduction project on understanding the programming environment on a target development board.
- 3-4 small (1-2 weeks) assignments on programming and interfacing with various peripheral units.
- 2 medium (3-4 weeks) sized projects to build applications integrating multiple devices.

As shown in the syllabus, the course started with an introduction to the ARM architecture and instruction sets. We then discussed C programming for embedded systems which included accessing I/O registers, bit manipulation, C calling convention, and in-line assembly. The students used the ARM Software Development Toolkit (ARM SDT 2.02u) to develop and debug their assembly/C programs in an ARM instruction set simulator called an *ARMULATOR*.

Following the introduction to ARM architecture and programming, we presented the overall architecture of MX1 processor and the connection to peripheral interfaces. For the I/O interfaces and interrupt signals, we started the discussion with the general-purpose input/output (GPIO) and handshaking signals. Since most I/O functions and peripheral interfaces are multiplexed at the I/O pads, the lectures focused on the programming techniques for configuring I/O pins and functions. Similarly, interrupt multiplexing and configuration techniques were discussed, followed by interrupt vectors and ISR operations. This allowed us to look into each peripheral interface in subsequent lectures.

The peripheral interfaces covered in the class included a timer, pulse-width modulator, UART, SPI, I2C, LCD controller, and touch panel controller. The lectures addressed the basic design principles, the internal register configuration of the peripheral interfaces, and interrupt mechanisms. The timing diagrams of the signal waveforms at I/O pins were discussed to illustrate the interaction of programming model and device operations. In addition, the schematics of the MX1ADS development board were used to show the connections of MX1 processor with external interface circuits and devices. While discussing LCD and touch panel controllers, the lectures also encompassed general raster display devices and A/D converters.

After discussing the selected peripheral interfaces and the programming techniques, the lectures focused on the memory structure of microprocessor systems. Both the abstract model and physical memory architecture of the SRAM and DRAM were explored. We paid special attention to synchronous DRAM, their timing characteristics, and access modes. We used the Micron MT48LC32M8A2 as an example of SDRAM.

The interconnection mechanism of microprocessor systems is also an important subject of the course. We focused on the bus architecture and the protocols of PC's XT, AT, ISA, and PCI buses. The general bus designs, including synchronous/asynchronous, bus arbitration, and block transfer were also covered. The final topic covered optimization techniques of bus performance such as pipelined transfers and split transactions.

3.3. Hardware Platform for Lab Projects

Although our goal was to teach the general principles of the microcontroller architecture and system design, we desired to have a target platform available to students to use for experimentation. We decided to use a 32-bit RISC platform instead of a traditional 8-bit architecture such as the Intel 8051 and Motorola 6811. There were three motivating factors in choosing a 32-bit RISC architecture over an 8-bit architecture. First, we wanted to use a current technology so that students would be well prepared for a career in the embedded systems industry. Second, we wanted to introduce multiple peripheral devices and bus technologies that were only available on 32-bit architectures. And finally, we had received a large endowment from industry partners to provide equipment and classroom support for the 32-bit architecture.

The target hardware platform had to include a high performance SoC microprocessor for which popular interfaces were available and configurable. To acquire additional support to build the experimental environment, we contacted the Motorola's Dragonball University Program, sponsored by Motorola SPS in 2003. The University Program considered our approach for software/hardware integration as an effective instructional method for embedded systems software development, and donated thirty Dragonball MX1 development boards (MX1ADS) for our lab. Motorola also agreed to provide all necessary technical support to expedite the installation of lab equipment.

To facilitate various projects, the SoC-based development boards are accompanied with a peripheral board on which various devices are installed. Figure 1

depicts a typical development system that enables programming development for different I/O projects.

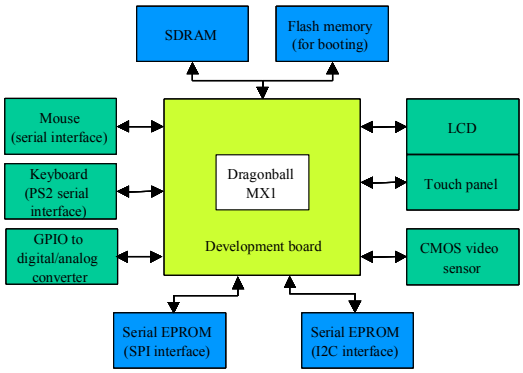

Figure 1. The target development system for lab assignment

3.4. Software Platform for Lab Projects

Embedded Linux was chosen as the software platform on the MX1ADS boards. The fine modularity of Linux components allowed us to customize the Linux kernel for the course. Only the device drivers required to boot the target board were kept in the Embedded Linux build. This enabled students to load their drivers as modules. Additionally, Linux provided a rich set of freely available debugging tools and environments, such as *printk, strace, gdb*, *ksymops*, and *klogd*. With MontaVista's Linux, we established the software development environment shown in Figure 2.

Influence from industrial trends also played a

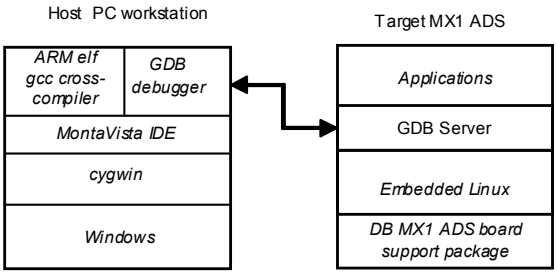

Figure 2. The target software development environment for MX1 ADS

significant role in our decision to use Linux. Currently, Linux is one of the preferred choices in the embedded system industry due to the availability of kernel source code without loyalties. This has lead toward recent trends of Linux becoming a dominant platform in embedded controllers. According to a survey conducted by the Venture Development Corporation, the estimated worldwide shipments of embedded Linux operating systems, add-on components, and related services reached over $60.0 million in 2003. This number is projected to reach over $115 million in 2006 [4].

In the target environment, students test their software components to manage peripheral devices. Since the I/O addresses are a part of the kernel address space and are protected, software components are developed as loadable device drivers modules. User applications use the drivers through standard file operations such as *open, close, read, write,* and *ioctl*. Interrupt service routines can also be registered as the modules are installed. This approach is quite attractive since the software for hardware interfaces are modular and embedded as a part of the operating system to support user applications. For students who have not taken any operating system courses, it may be challenging to comprehend the software structure and kernel APIs, and to develop kernel modules.

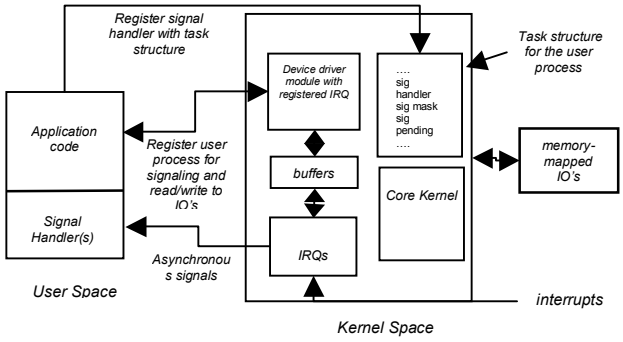

Figure 3. A pseudo driver for exercising kernel I/O address space and interrupts

To assist students with Linux specific driver development, we provided several example driver modules to illustrate the interactions between user applications and device drivers. One example is a pseudo driver, shown in Figure 3, which allows a user application to access memory locations in the I/O address space. When read or write functions are called, a command structure consisting of an I/O address and a data field is passed from the user application to the driver. The driver then reads from or writes to the I/O address. Hence, the student's application program can manipulate and access various control and status registers of peripheral controllers. To illustrate interrupt-driven data transfer, we added a ring buffer in the pseudo driver with which I/O data can be saved for subsequent read calls. Blocked driver function calls and the interaction with ISRs are demonstrated using a wait queue, *interruptible_sleep_on,* and *wake_up_interruptible* kernel functions. In addition, the pseudo driver makes use of asynchronous notification to emulate interrupts to user application programs. An ISR can invoke *kill_fasync* to signal a user application

handler once it is registered. The signal handler can then take an action or pass the status changes to the main program. This pseudo driver also provides a great example to build character device drivers for some peripheral devices.

3.5. Sample Projects

To reduce the learning curve on Linux device driver development models and Linux kernel application programming interfaces (API), we provided a driver framework for each assignment. This allowed the students to concentrate on writing the hardware/software interface code rather than worrying about Linux's internal device driver interface. For example, the following segment of code is part of the driver framework we provided to students to develop a timer driver.

```
int init_module()
{
   int result;
   /* register our character device */
   result = register_chrdev(IO_major, driverName,  &IOBridge_fops);

   if (result < 0) {
      printk("<1>%s: Can't get major %d\n", driverName, IO_major);
      return result;
   }

   if (IO_major == 0)
   {
      IO_major = result;
   }

   // initialize hardware timer
   timer_init();

   // Register timer interrupt from the kernel.
   if (request_irq(TIMER_IRQ, timerISR, 0, "Timer2", NULL)) {
      printk("<1> Unable to get IRQ for Timer 2\n");
      unregister_chrdev(IO_major, driverName);
      return -EBUSY;
   }
   return 0;
}

void cleanup_module()  /* This function is called when we do rmmod. */
{
   printk("<1>Freed %s\n", driverName);
   free_irq(TIMER_IRQ, NULL);
   unregister_chrdev(IO_major, driverName);
}

void timer_init() {
}

void timerISR(int irq, void *dev_id, struct pt_regs *reg) {
}
```

In terms of projects, the platform enabled many development assignments with peripheral device controllers and hardware configurations. The following lists some sample projects given in the Fall of 2004.

1. Measurement of execution of the CRC-32 procedure with a hardware timer. The measurement was done in the eLinux environment on MX1ADS target board using MontaVista's DevRocket IDE on a Windows PC or Linux workstation.
2. Development of an interrupt-driven mouse driver for a serial mouse. The project employed a Microsoft 2-button serial mouse (Version 2.0A) attached to UART serial port. The driver compiles three mouse movement data packages and then reports any movement to the user applications.
3. Development of a driver for an external memory device. A Microchip 25LC640 EEPROM which consisted of 256 32-byte pages (or blocks) was used. The EEPROM contained an SPI interface. Hence, all commands and data transfer operations are done via a SPI bus controller. The project introduced students to the important concept of timing in device driver programming.

For the first project, we provided a Linux character driver capable of writing and reading registers on the target board. The students were tasked with developing an application to measure the execution time of a given program by using the hardware timer. This assignment introduced students to the Linux device driver model and software-hardware interface.

Next, the serial mouse driver project allowed students to apply their theoretical understanding of UART to develop an interrupt driven mouse driver. The driver uses an asynchronous I/O signal to communicate between the application and device driver in the kernel. We provided a framework for asynchronous I/O implementation in the Linux device driver.

The overall goal of the assignments was to reinforce classroom learning by providing the students with interesting projects. This gave them a greater understanding of theoretical concepts and a feeling of satisfaction upon completion of the projects [2].

4. Outcome and Evaluation

At the end of the semester, we surveyed the students about their learning experience. Twenty-eight out of forty-four students responded to the survey (64%). The survey questions are grouped into five categories: C programming, the Linux development environment, system architecture and system-level design, peripherals and projects, and overall satisfaction.

According to the survey, over 80% of the students agreed their understanding of C programming language has increased and that they were comfortable with developing device drivers using C. Even though the students were not familiar with the tools and development platform we used in class, we found that they were able to

learn them quickly. About 73% of the students suggested that they were able to use the tools effectively at the end of semester.

The most challenging issue was the lack of proficiency in C programming and Linux development environments. We are planning to integrate a Linux environment in some prerequisite classes and add more emphasis on C in basic programming courses in the future.

5. Conclusion

Similar to many computer engineering curriculums, the microprocessor system design course at ASU has focused on teaching hardware/software interfacing and the management of peripheral devices. The previous approach of using assembly language and microcontroller-based platforms had been in place for more than a decade. It allowed the students to appreciate machine level processor operations and hand optimization to achieve the efficiency of assembly programs. However, with the advent of modern software development tools and the wide-spread use of embedded systems applications, a change in course material becomes inevitable.

There are a few important initiatives used in our approach for the microprocessor system design course. First, the use of assembly language for software development to control peripheral interfaces should be minimized. Students must be able to assess the cases where the use of assembly code can be justified. This would include encapsulating assembly code in well-defined interfaces and incorporating the code in software components as required. Second, the use of a broad set of peripheral interfaces including serial buses, LCD controller, touch panel, and data acquisition should be introduced. Finally, a practical software development and execution environment should be utilized so that students can gain familiarity with modern tools to build structured software components for embedded applications.

With these initiatives, the microprocessor system design course was transformed and introduced in the Fall of 2004. It was anticipated that knowledge gaps would exist in some of the prerequisite courses. Hence, we assumed that students may encounter difficulty with the required learning curve. However, we were surprised and satisfied with students' reception to the course. In general, students were excited about the new course structure, the updated learning environment, and the challenging projects, although complaints over the large amount of manuals and data sheets still existed. Overall, we believe this course was successful and we look forward to the development of the more advanced courses in the Embedded Systems curriculum.

6. References

[1] Gerald C. Gannod, et. al., "A Consortium-based Model for the Development of a Concentration Track in Embedded Systems", *Proceeding of the 2002 American Society for Engineering Education Annual Conference & Exposition.*

[2] Chris Hudson, "Teaching Microcontroller Technology – learning through play", *IEEE International Symposium on Engineering Education: Innovation in Teaching, Learning and Assessment*, Volume: Day 1, 4 January 2001.

[3] David C. Pheanis, "CEINT Internship Program", *33rd ASEE/IEEE Frontiers in Education Conference,* November 2003.

[4] Chris Lanfear, Steve Balacco, "The Embedded Software Strategic Market Intelligence Program, 2004", Venture Development Corporation, July 2004.

[5] Seongsoo Hong, "Embedded Linux Outlook in the PostPC Industry", *Proceeding of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[6] Frank Vahid, "Embedded System Design: UCR's Undergraduate Three-Course Sequence", *Proceedings of the 2003 IEEE International Conference on Microelectronic Systems Education*, 2003

[7] Naehyuck Chang and Ikhwan Lee, "Embedded System Hardware Design Course Track for CS Studnets", *Proceeding of the 2003 IEEE International Conference on Microelectronic Systems Education,* 2003.

[8] Shlomo Pri-Tal, John Robertson, Ben Huey, "An Arizona Ecosystem for embedded Systems", *IEEE International Conference on Performance, Computing, and Communications*, 4-6 April 2001.

[9] Konstantin Boldyshev, "Linux Assembly HOWTO," http://www.linuxselfhelp.com/HOWTO/Assembly-HOWTO/index.html.