AC 2008-1553: AN APPROACH FOR VERTICALLY INTEGRATED EMBEDDED SYSTEMS DESIGN

J.W. Bruce, Mississippi State University
Lee Hathcock, Mississippi State University
An Approach to Vertically Integrated Embedded Systems Design

Introduction

Historically, commercial market trends trickle down into engineering program curricula. In the computing systems marketplace, customers are demanding ever more complex features as computing systems become more capable and affordable. Today, engineering educators are feeling the pressure to provide more realistic, comprehensive, and complex lab experiences to the students in order to remain relevant and keep students’ attention. These demands are especially difficult in the university environment where students may lack several basic skills, and the professor and student work under an intense 15-week time-to-market.

In order to take students from neophyte to the accomplished designer of a vertically integrated embedded system, a design framework must be used to offload many housekeeping and flow control tasks. We need an operating system. Many embedded systems executives and real-time operating systems exist; however, many are commercial and cost-prohibitive to deploy in the classroom, and almost none of them are suitable for use in a extremely resource-limited (read: affordable) designs. Therefore, we chose to write our own operating system.

Another benefit to using an operating system is to teach students to work within the confines of a system in which they may not have full control. PC operating systems do not give you full control of all subsystems. Moreover, as embedded systems become increasingly complex, subsystems are increasingly hidden and black box design skills are required. Our OS emulates these design constraints, but unlike commercial OS-es, professors and lab assistants have full knowledge of the OS behavior and can provide detailed support and guidance not possible if the OS were acquired from commercial companies or the open-source community.

An additional pedagogical objective for our embedded systems class is to use object-oriented design (OOD) techniques. When one thinks about it, using an OOD approach is actually a natural fit for embedded systems. A customer is not going to come to you with the details of the hardware, they are going to say their product should read temperatures, or monitor parts of a vehicle, or other such things. Embedded systems, by definition, interacting with real-world objects, make OOD a perfect choice for design and teaching higher-level subsystem interactions.

Background

XXXXX UNIVERSITY has recently revised its undergraduate computer engineering (CPE) program with input from alumni and advisory employers. The CPE program has focused on embedded computer systems. Embedded systems form a rich application source through which CPE education can be made relevant. Embedded computer systems are a timely subject that is immediately useful to students in their senior design projects. Furthermore, a large number of our CPE graduates currently use or design embedded computer systems in their jobs.
Recently, a team-based progressive embedded systems design course was developed that, in addition to providing the technical embedded systems knowledge, develops team and communication skills in situations emulative of industry\(^1\). The course was a success by many accounts; however, student teams abandoned sound design practices in attempt to meet the demanding 15-week “time-to-market” constraint. Team members produced defect-riddled designs and the design schedules slipped due to an unproductive test-redesign-test development cycle.

Later\(^2\), the course was retooled to use a lightweight design process based loosely on proven software engineering standards\(^{4,5,6}\) which detects defects during design. This development process has been used with success in the subsequent offerings of the design course based on a more complex project\(^3\). The resulting student designs are typically on time and of high quality. Furthermore, students report satisfaction with the experience, because of both the visible results at course end and the perceived relevance of the process that they used.

All of these course offerings\(^{1,2,3}\) have made a visible impact on the computer engineering program at XXXXX UNIVERSITY. Computer engineering student projects in the capstone design course have greater complexity and are of higher quality compared to previous years. It was decided that these useful embedded system design skills be introduced to students earlier in the program, so a simplified embedded systems was adopted in the basic microprocessors course\(^4\). With fundamental embedded systems content being taught at the junior level, the senior level embedded systems course is free to concentrate more on system concepts and integration issues that are more common in engineering practice and industry, like team-based design\(^{2,3}\), industry-based software engineering standards\(^{5,7}\), and the higher-order design constraints that students lacked in the capstone design course. This paper describes an object-oriented design (OOD) approach and a simple real-time operating system to develop a complex, vertically-integrated design project – an Internet weather station.

The Project – An Internet Weather Station

The goal of the project, an internet weather station, is to provide the world with the current weather conditions and keep an archive of past weather. Our College of Engineering has required students to purchase laptops since 1999 and many of our courses make substantial use of these valuable student-provided resources. Each student creates their own weather station “server” running on their laptop connected to the University network. High-level internet processing and data management is done in Python.

The Python virtual machine (VM) is an interpreter and is very capable in dealing with abstract data structures and flow control. Python is also very easy to learn, provides rapid code development cycles, and is very extensible. However, timing critical tasks are not Python's forte. So, our Internet weather station project uses a microcontroller subsystem to control the weather sensor package and interpret the resulting weather sensor data. The Python VM communicates with the microcontroller via the Universal Serial Bus (USB). The next two sections will describe the architecture of the microcontroller and the web service subsystems.
Low-level sensor control and interfacing is performed by the USB-connected microcontroller. Sensor control requires accurate timing and is not compatible with the interpreted nature of the Python VM, and the unknown, variable delays introduced by TCP/IP networks. Student teams were instructed to use the Microchip PIC18F2550 microprocessor to control and measure weather sensor data. The course requires the same lab parts kit purchased by the students in the earlier, junior-level microprocessor course. The previous course uses a RS232-based Microchip PIC18F2420, so this experience extends their experience to USB without introducing too many additional new processor complexities. The microcontroller subsystem has four major responsibilities:

- Read and time stamp raw sensor outputs
- Filter sensor outputs
- Maintain sensor reading extrema
- Respond to sensor value requests from the web server

Development for the microcontroller is done with the same tools as the prior course. However, students are required to follow the development, design review, and documentation process described in an earlier ASEE paper.

A calibrated, industrial quality weather sensor package (EnviroMonitor from Davis Instruments) was secured and mounted on the roof of the four story Electrical and Computer Engineering building. Sensors are multiplexed and transmitted via an eight-conductor Cat5/RJ45 cable to the student-accessible Microprocessors laboratory on the third floor.

The sensor signals represent six quantities in three signal characteristic categories: analog, digital closure, and frequency. The weather sensor signals are described in Table 1. Analog sensors create a voltage output corresponding to their quantity, wind direction, air temperature, and wind direction compass heading. Closure signals represent discrete weather events (anemometer revolution, rain drops) by temporarily closing a contact switch. Switch closure pulls a logic-high voltage (5V) to logic-low (ground). Frequency signals represent a weather quantity (humidity) by a square wave frequency. The microcontroller generates a control signal used by the sensors to multiplex two weather sensor signals on the same RJ45 cable conductor.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>RJ45 conductor</th>
<th>Type</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wind direction</td>
<td>2</td>
<td>Analog</td>
<td>Low</td>
</tr>
<tr>
<td>Wind speed</td>
<td>4</td>
<td>Closure</td>
<td>Low</td>
</tr>
<tr>
<td>Temperature</td>
<td>7</td>
<td>Analog</td>
<td>High</td>
</tr>
<tr>
<td>Solar Radiance</td>
<td>2</td>
<td>Analog</td>
<td>High</td>
</tr>
<tr>
<td>Rainfall</td>
<td>5</td>
<td>Closure</td>
<td>N/A</td>
</tr>
<tr>
<td>Humidity</td>
<td>8</td>
<td>Frequency</td>
<td>High</td>
</tr>
</tbody>
</table>

### Table 1: Sensor Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>RJ45 conductor</th>
<th>Notes</th>
</tr>
</thead>
</table>

Page 13.172.4
<table>
<thead>
<tr>
<th>Control</th>
<th>1</th>
<th>Low=0V, High=5.0V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor supply</td>
<td>3</td>
<td>Requires at least 6.5V w.r.t. ground</td>
</tr>
<tr>
<td>Ground</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Sensors and sensor cable specification

The wind direction sensor creates an output voltage between 0 and 2.5 V that changes linearly as the anemometer revolves. Due north (0°) is 0V, east (90°) is 0.625V, south (180°) is 1.25V, and west (270°) is 1.875V. Wind direction is measured by the microcontroller’s internal analog-to-digital converter (ADC). Wind speed is represented by switch closures on every anemometer revolution. The switch closure signals the microcontroller via its input capture timer that records the period between closure which is directly related to wind speed.

Temperature is represented by an analog voltage between zero and 2.5V. The weather station temperature sensor is a thermistor placed in parallel with a 42.4kΩ resistor. The resistor and thermistor parallel combination is in series with another 42.4 kΩ resistor connected to the 5V sensor reference. The control signals switches the thermistor into the voltage divider. The changing thermistor resistance changes the value of the lower voltage divider resistance. The resulting temperature-voltage relationship is nonlinear and is shown in Figure 1.

![Temperature](image)

Figure 1: Temperature transfer curve
Solar radiance measures the solar energy incident on wavelength selective photo diode on the sensor set. The solar radiance sensor generates a voltage between 0 and 2.5 V. The relationship between voltage and solar radiance is linear with 1.67mV representing 1W/m² of solar radiance impinging on the sensor. Rainfall is measured by a rainfall collection bucket. The rainfall sensor momentarily closes a closure switch for each 0.01” of rainfall. The rainfall signal is obviously asynchronous to all other events and is transmitted on a dedicated RJ45 conductor independent of the weather sensor control signal. Each rainfall closure event signals the microcontroller via an external interrupt.

Relative humidity is measured with a variable capacitor the change with humidity. The variable capacitor is adjusts the frequency of a square wave oscillator. To account for temperature-dependent effects on the oscillator frequency, a reference square wave is generated on conductor 8 when the sensor control signal is low. Relative humidity is determined by the ratio of the two square wave frequencies removing the common temperature dependent effects. Both frequencies are measured with the microcontroller’s input capture timer modules. Figure 2 shows the relationship between frequency ratio and relative humidity.

![Humidity Graph](image)

Figure 2: Humidity transfer curve

The sensor control signal is generated by the microcontroller. Each signal quantity with the exception of rainfall is measured repeatedly over during a 2-3 second interval and a median-filtered value is calculated. The median filtering removes signal noise picked up by the 60m sensor cable that runs between the student’s microcontroller and the roof-mounted weather sensors. The median-filter output, which represents that sensor's sample, is timestamped. Each
sensor reading (where appropriate) is compared with microcontroller-stored values as to maximum and minimum sensor readings. If the current value represents a new extrema, the value and the timestamps are stored in memory. Sensor extrema values can be reset upon a command from the Python VM. The microcontroller acts as an USB slave device that responds to requests from the Python VM web server (USB host). USB communication can be initiated at any time by the Python VM. The web server sends a command packet that is interpreted by the microcontroller, which then performs the actions that correspond to the command. Due to the host-slave nature of the USB system, the microcontroller can not initiate communication with the host (Python VM). It is imperative that the microcontroller is capable of operating on its own and storing sufficient data between host requests.

Web Server Subsystem

The web server is implemented via a Python script, to allow the web server to Python scripts dynamically in response to client actions. While this approach is not industrial strength in robustness or security, it provides the students with very simple and unfettered access to the web server details. (Students were provided the option to use more industry-standard Apache or Apache+Tomcat server configurations.) The web server subsystem had three main responsibilities:

- retrieve weather sensor data from the microcontroller subsystem,
- log weather data for use in long-term weather archives, and
- create and serve dynamic web pages displaying weather data over the internet

The web server runs five different processes to perform its responsibilities.

Weather Sensors – The microcontroller subsystem was encapsulated by a module called Weather Sensors in the web server subsystem. This module implements a weather sensor interface given to students in the design specification. These objects offer a layer of abstraction between the web server and the microcontroller subsystems and allow the web server to communicate with the microcontroller.

Weather Logger – The Weather Logger module periodically retrieves the weather data from the Weather Sensors module and sends it to the Weather Data Store where all the weather data is stored.

Weather Data Store – The Weather Data Store module handles the storing of all the weather data. Any module that needs to save or retrieve weather data must use this module.

Weather Servlet – The weather servlet, a Python script, dynamically generates a web page that contains the current weather data from the Weather Data Store module.
Figure 3 shows the hybrid style diagram for the web server subsystem. The weather logger and HTML server+weather servlet run concurrently and both access weather data via the weather data store. The weather logger periodically uses the weather sensor objects to query the microcontroller subsystem and gather real-time weather data. The HTML server and the associated weather servlet are running and waiting for HTTP requests for web page weather information from users anywhere in the world. Upon a HTTP request via the system’s URL, the weather servlet obtains the latest weather data samples from the weather data store to place dynamically into the weather page.

![Diagram](image)

Figure 3: Hybrid Diagram for web server

Structure of Design Experience

The experiential design experience is addressed in a three-hour weekly lab. The design experience is progressive, where each project design milestone generally builds upon the successes of the previous. Teams worked in instructor-selected groups of three to four team members each. Students are given a 60+ page design specification document at the beginning of the course. Design specifications typically describe the interfaces of each subsystem and their desired function. The details of implementation are left up to the team. Milestones are to be completed in one to two weeks each depending on the academic holiday calendar and milestone complexity. This section briefly describes each project milestone in turn.

**Milestone 1 (M1)** – The main software tools and components that the students will use are installed, including: Python, third-party Python libraries, and microcontroller IDE. Students work through tutorials and simple program builds to provide a familiarity with the tools.

**Milestone 2 (M2)** – Write Python script to retrieve any arbitrary URL from the web. Use created script to retrieve NOAA XML weather data for an arbitrary airport weather station, then parse, display, and log data to a local file.

**Milestone 3 (M3)** – Develop initial MCU firmware (in C) for monitoring weather sensors. In this milestone, use Microchip PIC182550 to record current, maximum, and minimum temperature values.
Milestone 4 (M4) – Develop MCU firmware to control an arbitrary I2C device based on commands sent by Python (PC) over USB. Use firmware to record interior room temperature with a I2C temperature sensor.

Milestone 5 (M5) – Extend firmware in Milestones 3 and 4 to adds additional weather sensors: wind direction, wind speed, solar radiance, humidity, and rainfall. Simultaneous reading of all weather sensors is a challenge, due to timing and signal conditioning issues.

Milestone 6 (M6) – Encapsulate firmware in Milestone 5 with command set sent from Python (PC) via USB. Also provide a simple user interface on PC for testing and demonstration.

Milestone 7 (M7) – Create Python “objects” to encapsulate weather sensors and associated communication. This encapsulation hides the details of each sensor and communication so that it is possible to change underlying physical sensors, communication links, and/or sensor command set without causing change to sensor server code developed in following milestones.

Milestone 8 (M8) – Create basic web server (using Python or Apache/Tomcat) Periodically log weather values and display current weather conditions via a dynamic webpage.

Milestone 9 (M9) – Improve firmware and “weather server” to handle unconnected sensors and data archival to the ECE department server. Students can create further extensions to the original specification, which may include but are not limited to: accessing historical weather logs via the web, graphically displaying the weather logs, creation of “static” web content from the above weather logs, real-time display of current conditions, or displaying current radar images retrieved from internet.

Object-Oriented Design

Embedded systems by definition sense their environment and act upon it. Therefore, embedded systems must communicate with and/or control real devices, or objects. Thus, an object-oriented design (OOD) is very natural for embedded systems. However, most resource-constrained embedded systems lack the processing power, memory, and software tools to implement an OOD directly. Of course, modern personal computers have plenty of resources and OOD is readily implemented. The project described here will use the microcontroller to perform only the tasks for which it is required (timing-sensitive and sensor specific). The Python VM on the laptop can implement an OOD by hiding the hardware and interfacing specifics in “private” object methods. This section will briefly outline our approach.

The overriding desire to hide hardware complexities and promote code reuse lead us to encapsulate as much as possible within objects that represent various pieces of the weather sensor hardware. A virtual, or template, class object called wxSensor was specified to the student. This class cannot be instantiated. This class must be extended to include details of the exact communications used.
The first portion is to encapsulate the lowest level reading and writing of data. Python and many of its support modules hide many of platform-dependent subsystems behind standard Python objects. PySerial was used to provide a MacOSX-, Windows-, and Linux-compatible interface to the serial communications and USB subsystem. Since some other communications schemes could be specified later, or the interface to the weather sensors may change, students were required to implement a wxSensorSerial class to encapsulate the weather sensors and their various functions.

Secondly, the actual sensors need to have an interface through which to retrieve data. The actual implementation of the commands may change based on the sensors, or the command set that controls the sensors may change between implementations. So functions are defined to send the proper commands to the MCU, and retrieve the data corresponding to the command set accordingly. To illustrate, students are required to retrieve the solar radiance from the sensors. First, a wxSensorSerial object is created, with the correct communications port parameters. Then, the only call is to the getSolarRadiance() function, which returns the desired solar radiance value.

At a lower level, the getSolarRadiance() function writes the command to retrieve the data, reads the data that is sent back, and converts the raw data to its proper readable form. This lower level interface is defined by the programmer, but will always eventually come back to encapsulating the wxSensorSerial read and write commands, along with some basic math functions. Furthermore, students must be able to read solar radiance sensor values, along with its extrema data, timestamps, and reset these values. Finally, solar radiance is just one of the dozen or so weather sensor readings that must be maintained.

Simple Real-time Operating System

At this point, it should be clear that the described project is complex. This project would be difficult to complete in a normal academic semester if students were expected to design, write, and test all hardware and software components of the project themselves. Furthermore, an industrial project of this magnitude would likely rely on existing designs or software libraries. To this end, we wrote a simple real-time operating system based on the very clever protothreads library by Adam Dunkels.

Protothreads provide a nearly zero-overhead (and stack-free) implementation of lightweight threads (similar to co-routines). Protothreads differ from normal threads in that they can not be preempted and are stack-less. Protothreads provide event-driven flow control in a very readable fashion and a mechanism for "blocked" protothreads to give up processor focus so other protothreads can run. Protothreads facilitate very complex flow control without resorting to full state machine software structures. The protothread implementation relies on a Duff's device, a loop-unwinding technique as applied to the C language switch-case construct. Duff's device seems a bit counterintuitive at first, and may seem like a C language hiccup. However, the Duff's device behavior is mandated by the C language specification. Therefore, Duff's device, and protothreads in turn, are readily implemented in nearly every quality C language compiler. In short, the simple operating system developed by the authors is exceedingly portable to other compilers and processors.
In this section, we will describe the structure and operation of the Embedded Systems Operating System (ESOS). The ESOS is a cooperative, multi-tasking embedded systems framework based on the protothreads concept. The protothread library is extremely well-suited for our embedded systems programming needs because (i) most embedded system user requirements ultimately reduce into a event-driven reactive system architecture, and (ii) most small microcontrollers have extremely limited data memory, such that the comprehensive and extensive stack support required for preemptive threads is not feasible.

The ESOS provides a cooperative multitasking environment for ESOS and user tasks to coexist. Since tasks must cooperate, it is imperative that user tasks and program statements strictly follow the conventions set forth in this section.

- ESOS has complete control over the HW that creates the ESOS system tick, Users shall not manipulate the HW (or its software structures) that create the ESOS system tick in any way.
- Users shall not manipulate any HW interrupt enables or interrupt flags directly. All modifications to these bits shall be done through ESOS-provided API functions. (ESOS must be kept aware of the status of these bits at all times, and does so through the IRQ API functions that ESOS provides to user.)
- User tasks shall not block for any appreciable duration. If a task must be blocked for an unknown, arbitrary, or lengthy period of time, the user task should be constructed to ``wait" or ``yield" in some way so that other tasks can get focus and execute. (Many ESOS systems operations are coded as tasks and will not execute unless user tasks yield properly.)
- The ESOS task functions WAIT_???? will yield if the associated condition fails. However, if the association condition is asserted, the task will not yield control to ESOS. Therefore, each user task should have at least one yielding statement that is guaranteed to yield under all circumstances to prevent monopolizing the CPU.
- User task variables that are required across any ``wait" or ``yield" function must be declared as static. If the user task variable is not declared as static, the value of the variable after the yield is unpredictable.
- Users shall not use the C language switch statement construct across ``wait" or ``yield" functions. This can be harder to detect and avoid than you think. Your user task could `"spawn" a child task which contains yields. So a switch statement in the parent task would not be allowed across the `"spawn" statement that creates a yielding child(ren). I recommend that you avoid the switch statement constructs completely. Use the if - elseif - else construct instead; this C language construct is safe in all ESOS contexts.

As a minimum, the user must provide at least two program items for the ESOS to generate a usable application. First, the user must provide an user_init() function. Secondly, the user must provide at least one user task or user interrupt task which is registered in user_init().

user_init -- The user must provide this function to initialize all user software structures and user hardware timers. This routine will be called during ESOS system initialization. User can manipulate host CPU hardware registers, except those (interrupts) that are provided through ESOS system functions. User should register at least one user task during user_init().
User tasks – The user must provide at least one user task which is registered in user_init(). All user functions have the following prototype: uint8 userTask(struct pt* pt). An user function can be easily prototyped or declared with the macro: ESOS_USER_TASK(routine), where routine is the name of the function that implements the user task.

Each user task must begin with TASK_BEGIN(pt) and end with TASK_END(pt). The two macros denote the beginning and end of the user task code, and are required by ESOS for proper operation.

User tasks are called repeatedly by the ESOS scheduler and, as such, user variables that must be preserved between yielding statements must be declared as static. Since the protothread library that is used to generated the cooperative multitasking uses an unique feature of the C language switch statements, user tasks must not use switch statements across yielding calls. It is recommended that user tasks and helper functions not use switch statements at all, as subroutine or spawned child tasks might contain yielding statements that are not apparent to the calling routines above.

ESOS provides a reasonably hardware-independent API for user applications. The API supports task control, semaphores, communications, user-defined interrupt service routines, etc.

Task functions utilizing protothreads are as follows (all caps indicates macros):

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esos_RegisterTask</td>
<td>Registers task with ESOS system, with no assumptions on periodicity or order of execution.</td>
</tr>
<tr>
<td>esos_UnregisterTask</td>
<td>Removes the task from the ESOS scheduler.</td>
</tr>
<tr>
<td>TASK_EXIT</td>
<td>Exits a task. Control is passed back to the parent, either another task or ESOS.</td>
</tr>
<tr>
<td>TASK_RESTART</td>
<td>Restart a task from the beginning of the task (TASK_BEGIN)</td>
</tr>
<tr>
<td>TASK_YIELD</td>
<td>Yields control back to ESOS, and resumes at yield point.</td>
</tr>
<tr>
<td>TASK_YIELD_UNTIL</td>
<td>Yields control back to ESOS regardless, and resumes execution when the condition is TRUE at the yield point.</td>
</tr>
<tr>
<td>TASK_YIELD_WHILE</td>
<td>Yields control to ESOS while the condition is TRUE. When FALSE, execution is resumed at the yield point.</td>
</tr>
<tr>
<td>TASK_WAIT_UNTIL</td>
<td>Yields task execution until the given condition is TRUE. If already TRUE, execution continues with no yielding of control.</td>
</tr>
<tr>
<td>TASK_WAIT_WHILE</td>
<td>Yields task execution while the given condition is TRUE. If the condition is FALSE, execution continues with no yield.</td>
</tr>
<tr>
<td>TASK_SPAWN</td>
<td>Spawns a new child thread. The parent is blocked until the child exits, and resumes at the TASK_SPAWN point.</td>
</tr>
<tr>
<td>TASK_SEM_INIT</td>
<td>Initializes semaphore with the value of initial count. Value is between 0-255 in ESOS.</td>
</tr>
<tr>
<td>TASK_SEM_WAIT</td>
<td>Task yields until the semaphore described is non-zero.</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>(sem_struct_ptr)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TASK_SEM_SIGNAL</th>
<th>Increments the semaphore described, which eventually causes tasks waiting on the semaphore to resume execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sem_struct_ptr)</td>
<td></td>
</tr>
</tbody>
</table>

The following is a set functions to access flags provided by the ESOS system for user use. These flags consist of eight 1-bit flags, and are specifically for user use. Also, the user has access to the system tick, and may retrieve it, or some offset value of the current tick value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esos_SetUserFlag</td>
<td>Sets specified user flag. User masks valid if only 1 bit set.</td>
</tr>
<tr>
<td>esos_ClearUserFlag</td>
<td>Clears specified user flag.</td>
</tr>
<tr>
<td>esos_IsUserFlagSet</td>
<td>Returns non-zero if the user-specified flag is set.</td>
</tr>
<tr>
<td>esos_IsUserFlagClear</td>
<td>Returns non-zero if the user-specified flag is clear.</td>
</tr>
<tr>
<td>esos_GetSystemTick</td>
<td>Returns 32-bit ESOS system tick value (1ms per tick)</td>
</tr>
<tr>
<td>esos_GetFutureSystemTick</td>
<td>Returns a 32-bit value representing the current 32-bit ESOS system tick plus an offset delta (in ms).</td>
</tr>
<tr>
<td>(delta)</td>
<td></td>
</tr>
</tbody>
</table>

Interrupt functions are also very important, and are also required to be registered much as tasks are. Interrupts must be registered before they are enabled. User interrupts are given less priority than those required for ESOS to function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esos_RegisterUserInterrupt</td>
<td>Registers a given interrupt with ESOS.</td>
</tr>
<tr>
<td>esos_UnregisterUserInterrupt</td>
<td>Unregisters the specified user interrupt.</td>
</tr>
<tr>
<td>esos_EnableAllUserInterrupts</td>
<td>Enables all user interrupts.</td>
</tr>
<tr>
<td>esos_DisableAllUserInterrupts</td>
<td>Disables all user interrupts.</td>
</tr>
<tr>
<td>esos_EnableUserInterrupt</td>
<td>Enables a previously registered user interrupt.</td>
</tr>
<tr>
<td>esos_DisableUserInterrupt</td>
<td>Disables a previously registered user interrupt.</td>
</tr>
<tr>
<td>esos_IsUserInterruptEnabled</td>
<td>Returns a non-zero value if the specified interrupt is enabled.</td>
</tr>
<tr>
<td>esos.DoesUserInterruptNeedServicing</td>
<td>Returns non-zero value if the specified interrupt is in need of servicing.</td>
</tr>
<tr>
<td>esos_MarkUserInterruptServiced</td>
<td>Clears the interrupt flag for the specified interrupt.</td>
</tr>
</tbody>
</table>

Also provided by ESOS is a communications buffer object. Data is written and read from this structure when ESOS reaches the communications buffer task in its schedule.
esos_GetCommSystemBuffDesc | Returns the pointer to the communication buffer structure.

esos_GetCommSystemVersion | Returns the current revision number of the ESOS communication system. This indicates whether the system is USB or RS-232, and both major and minor revision numbers.

esos_GetCommSystemMaxOutDataLen | Returns the maximum number of bytes that the ESOS communications system can transmit at a time.

esos_GetCommSystemMaxInDataLen | Returns the maximum number of bytes that the ESOS communications system can send to a user at a time.

Assessment

The foremost assessment of the course’s effectiveness is that all five teams in the Fall 2007 semester successfully completed the project requirements and submitted a working design. Students initially relished the earlier design milestones (M2-M4) dealing with the MCU firmware because of their familiarity with C language firmware development in the earlier microprocessor course. By the later firmware milestones (M5-M6) where the system functionality grew rapidly, student yeared for a higher level of abstraction. Although students had little to no previous Python experience, they taught themselves very quickly since the language is straightforward and they had a tangible objective (M6-M9). Students especially enjoyed the sensor encapsulation in M6-M7 that allowed very rapid and elaborate use of a wide variety of sensor data.

Quantitative and objective course assessments came from the software product metrics reported by students throughout the semester\(^2\). Metrics were collected by each student as to the number of hours spent in development, lines of code (LoC) developed, defects created, type and severity of defect. Metrics were collected according to student, team, subroutine, subsystem, and milestone. Although the amount of data collected can be overwhelming (especially for the instructor), it provides a great deal of insight into the course. However, one must always remember that the data is self-reported by the students with each student and team having a different standard by which they choose to measure themselves. Large variations in the data may be attributed to variation in discipline and expectations on the part of the students. Nevertheless, students reported spending a semester average of 76 hours effort (σ=28, min=44, max=129) writing an average of 1790 LoC (σ=1100, min=526, max=4949) and creating an average of 12 major defects (σ=7, min=0, max=27) and 20 minor defects (σ=13, min=0, max=41). The most prevalent qualitative complaint from the cohort at semester’s end was that the design experience was too time-intensive. This complaint was most vocal from student who were simultaneously enrolled in the capstone senior design course.
Figure 4: (a) Team effort and (b) Student Effort per week

The left chart in Figure 4 shows the effort expended by each of the five teams (denoted by the letters A, F, N, S, and T) for each of the nine milestones. Since some teams have more members than others, the right chart in Figure 4 shows the effort for each student normalized by their team size. From the charts, we can see that Milestones 3-7 are more labor-intensive than the others even though these milestones are given two weeks duration. With this quantitative measure, the instructor see where to place more resources, assistance, and time to alleviate the students’ concerns about workload.

Figure 5: (a) Lines of code written by each team and (b) by each student

The left chart in Figure 5 shows the total lines of code (LoC) written by each team for each milestone with the average over all teams shown as a line. From a coding effort effort exerted by
each team, Milestones 5-6 appear to be most intensive. To account for larger teams being able to write more elaborate (with possibly many more LoC??), the right chart in Figure 5 shows the LoC written per student per “work-day”. Viewing each student's coding productivity, we see that student tend to become more productive coders as the semester progresses. In general, student produce more LoC/day later in the semester than earlier. Milestones 8 and 9 have relatively fewer LoC than than the milestones immediately preceding since these milestones are largely system integration tasks. Correspondingly, we see that the student's coding productivity drops precipitously for M8-9 since this type software deals largely with interfaces and is time-consuming to write.

Figure 6: (a) Software defects created by each team and (b) by each student

The left chart in Figure 6 shows the total number of software defects created by each team over the nine milestones. Students recorded the type of defect and its severity, but that data is not shown here. (Another paper perhaps?) We see that M3, M5, and M6 (and M4 to some degree) seem to be more defect-prone than other milestones. Here firmware (hardware register manipulation in C) and Python defects are lumped together. The tendency of firmware development to produce defects is clearly seen here as M3-M6 are mostly firmware. Milestones with mostly to all Python code development (M1, M7-M9) are less likely to produce defects. This trend is also seen in the student's production of software defects.

Analysis of the student-reported data shows that students produced 194 LoC per workday on average. This is appreciable above the industry reported productivity of 10-60 LoC per workday for projects of this size11. It is unclear whether the difference is due to the methods that students used to measure LoC or where the heavy use of Python (a very-high level language) promotes coding productivity gains. Finally, students reported an average coding defect density of 20.7 defects/KLoC which almost exactly in the middle of the range (0-40 defects/KLoC) reported by the literature11.
Conclusions

This paper describes complex vertically integrated embedded systems project, an Internet weather station, that was implemented by small team of senior and introductory graduate students majoring in EE, CPE, and CS over a 15 week period. Although the project is ambitious, all teams were successful in completing the design requirements, with many teams adding additional “features” by semester’s end. The keys to achieving such student productivity (many students were also completing senior design projects simultaneously) was provided by

- coding standards\(^2\) – insured all team members wrote code that is compatible and readable by other team members
- code reviews\(^2\) – eliminated coding defects early in the design cycle
- object oriented design – a natural framework for describing embedded systems
- Python – a language well-suited for rapid development
- ESOS – a simple real-time operating systems that provides task management and flow control, as well as providing interrupt, timer, and semaphore support

The object oriented design approach pays huge dividends for the professor. The Internet weather project described here is based on an earlier experience\(^3\). After that experience was published, the dedicated Java-based web-server was replace by a Java virtual machine running on a laptop. The design specification and a vast majority of the code written for the first offering was directly used on the second implementation. Only the lowest level hardware interface methods had to be altered. High level “objects” presented the same interface and the servlet ran with zero changes. This is an amazing feat since the underlying hardware change from a Intel 8051 microcontroller to a full-blown personal computer running Windows, Linux or MacOSX. This altered design project was used for one semester with even greater success than reported in the 2005 ASEE paper\(^3\). The project described here replaces the Java VM with Python. The code structure is identical with only a change from Java language syntax to Python syntax. Student choosing to run the Apache\(^+\)Tomcat server could run the servlet from the first incarnation without change even thought absolutely nothing “beneath” the web server level was the same as the first project.

Student outcomes were good; each team successfully completed the design as described in the specification. Many teams added additional features while completing the final design milestone. Effort and time expended by students were mostly acceptable, but a few of the more intensive firmware design tasks did appear to be overly laborious. In future offerings, these tasks will be allocated more time, or additional infrastructure will be provided to the students to ease the workload. Students reported coding productivities that exceed the industry values, but that may be simply a difference of definition or interpretation of metrics. Students reported defect densities that correspond nearly perfectly with industrial practice. Finally, qualitative feedback from students was mostly positive apart from concerns about workload. Mostly student identified that the experience was “realistic” and “help them to know what to expect upon employment”.

References