Teaching Cyber Physical Systems in Layers

Patrick Schaumont
schaum@vt.edu

***Abstract*—Computer Engineering, as a field of knowledge, relies on a large number of abstractions to hide design details in hardware and software. However, the heterogeneous nature of CPS requires that students work *across* abstraction levels (hardware, software, physical). Contemporary technologies are not well adapted to this. This means that design abstractions, generally thought of as an advantage, may become a liability to learning. In a sophomore-level microcontroller-interfacing class, we identified complex and poor technology abstraction as one of the major difficulties faced by students. From that experience, we propose that CPS education places special emphasis on defining proper design abstraction levels. This is the basis for appreciating the importance of efficient modeling and sound methodology. We share some initial ideas on how this can be applied in the context of a microcontroller lab with a roving bot.**

I. Introduction

The design of embedded computing systems is profoundly based on abstractions. They are an essential tool to master complexity. They hide unnecessary detail, and they enable engineers to focus on the essence of the problem. C programs, finite state machines, data-flow process models, memory-maps, and handshake protocols are all examples of abstractions in the toolkit of computer engineers. Abstractions also work as a safety net, and they help engineers to achieve a reasonable level of success towards correct design.

Unfortunately, it's not always possible to keep design details hidden with abstraction. The design of embedded computing systems, and of cyber-physical systems in particular, relies on abstractions that straddle different domains. As a first example, consider a memory-write to a special-function register of a micro-controller peripheral. In software, this looks like a simple memory-access. In hardware, this may trigger a specific operation within the peripheral. As a second example, consider a UART that communicates with a motor controller to set the motor speed. In software, this merely looks like the transmission of a byte using a specific serial protocol. In the physical world, this means that a motor will change its speed.

In our experience with teaching a microcontroller interfacing class, we found that students have great difficulty with these *abstractions that really aren't abstractions*. A memory-write to control a peripheral does not serve the purpose of variable storage; it controls hardware. A UART transmission to set the motor speed does not serve the purpose of communication; it controls motor speed. In our experience, it seems that the translation of design problems across different domains (software, hardware, physical) may require so much effort from students that they tend to forget about the convenience and power offered by abstraction in the first place. When students finally get the hardware peripheral to work using a magic sequence of software commands, they have lost the energy to think about finite state machines, reliability, exception handling, and all the other things that make software cool.

To address this issue, we propose that CPS education places special emphasis on abstractions, and how they connect different domains together. We believe that CPS are very well suited for this because they are partly based on physics. Hence, design concepts can be explained to students in terms of physical-world phenomena (turning wheels, spinning rotors, ..). Abstractions can then be created in a bottom-up fashion to capture the CPS application logic (the 'cyber-part').

The paper is structured as follows. In the next section, we illustrate the complexity of abstractions with an example from our microcontroller interfacing course. We then propose to approach CPS design as a layering of abstractions. We show how design experiments with a roving bot can be formulated as a decomposition of the design over the different layers of abstraction.

II. Complexity of Abstractions: an example

To illustrate how abstractions can become a burden, we discuss an example from a spring 2012 class on microcontroller interfacing. Figure 1 shows a picture of the microcontroller kit that is used in this class: a Cerebot MX7ck board from Digilent [1]. This highly flexible kit comes with multiple additional modules that plug into connectors on the side of the board. The photograph shows an accelerometer module and a display module. In this lab assignment, the students had to implement tap-detection using the accelerometer, and call a specific interrupt service routine upon detection of a tap. In the following, we analyze, in a bottom-up fashion, the number and nature of the abstraction levels that students have to master in order to correctly implement this assignment.

- *Physical*: The 3D-accelerometer requires a specific orientation; the assignment asks for tap detection in the X-direction (towards the main PCB).
- *Platform*: This lab used an accelerometer module and a display module. Each has their own datasheet (40 pages

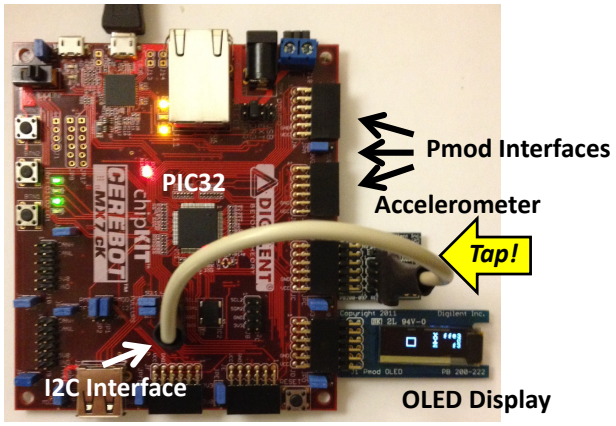

Figure 1. Hardware Setup of a lab using Cerebot MX7ck. Students need to implement a tap-detection mechanism using the accelerometer.

and 65 pages, respectively), specifying its electrical and logical connectivity.

- *Board*: The platform needs to be configured by fitting the accelerometer and display modules into header slots ('Pmod interfaces') on the board. The slots in Figure 1 appear to be uniform, but each of them carries a different set of connections (IO-ports, Timer capture/generation pins, interrupt input, analog inputs, ..). A separate 34-page board user manual explains the functionality of each slot. The accelerometer needs an I^2C interface, unavailable on any of the slots. The I^2C connection is therefore jumper-wired to a header on the board.
- *SoC*: The board includes a PIC32MX795F512L microcontroller, which contains multiple peripherals to control the display and the accelerometer. The display is controlled through an SPI port and several general-purpose I/O pins. The accelerometer is controlled through an I^2C port, and a change-notification interrupt input pin. The large variety of PIC microcontrollers has prompted the manufacturer (Microchip) to create a modular documentation format. The PIC32MX795F512L is summarized in a 256-page core manual, while each peripheral has a separate manual to explain its operation. The I^2C and SPI peripherals each have their own 60-page manual.
- *HAL*: The PIC32 is based on the MIPS architecture. The PIC32 peripherals are memory-mapped and integrated into the interrupt vector table. Based on this mapping and the processor application binary interface, driver software can be developed. This process is greatly aided through a compiler; Microchip's XC32 compiler is free and easy to use, although it still comes with a 238-page user manual (needed, in part, to document non-standard-C coding specifics).
- *Firmware*: Microchip provides a peripheral programming library called PLIB. PLIB provides easy C-based API's for each peripheral, but its documentation is limited to a 314-page listing of headers and macro's. The mapping from API to low-level peripheral programming (as documented in the datasheets) is not obvious, and requires studying the PLIB source code. Hence, PLIB is both a boon and a burden for the students.
- *Application Logic*: At the apex of this stack of abstractions sits the C program that controls tap detection. Ideally, at this level all abstractions should converge, and the application should decompose into a suitable set of period tasks and interrupts, captured in clear state-machine logic. In practice, we have observed very few students who are able to achieve this. Instead, they are switching back and forth between abstraction levels, browsing documentation and *hacking* the lab to completion.

As an instructor, one wishes it was possible to hide all of this detail and have the students focus on the essentials. Unfortunately, there is very little hiding available. The point of a class on microcontroller interfacing is precisely to learn about these abstraction levels and their associations.

One could argue that there should be fewer abstraction levels. For example, by assuming a standard computing architecture, it may be possible to merge the levels of *HAL*, *SoC* and *board*. Furthermore, by assuming a standard input/output architecture, we could remove or integrate the *platform* level. But would this really help? In our opinion, it may hurt the purpose of the course. Training a computer engineer with predefined libraries and standard computing platforms is like training a cook with instant-meals. By reducing the number of abstraction levels, the flexibility of the system design reduces as well. The objective of this course - microprocessor interfacing - is not just to train (embedded) computer scientists, but rather, to train engineers that can integrate embedded computing systems into physical platforms. They need to know what it takes to integrate a new chip (eg a gyro) in their embedded computer.

Instead of reducing the abstraction levels, we believe that more educational effort is needed at the level of composition. How can we reduce the heterogeneity and quantity of documentation while still supporting multiple and flexible levels of design abstraction? How can we build meaningful links between abstraction levels? For example, how can we teach design methods to link physical events (tapping an accelerometer) to logical exceptions (interrupts)? And, given a physical problem (detecting taps), how can we teach how to select the right peripherals, peripheral configurations, interrupts, task structures and task logic?

In the following section, we enumerate a few preliminary ideas in the context of the microcontroller interfacing class.

Table I
SEVEN RELEVANT ABSTRACTION LEVELS, ENUMERATED BOTTOM-UP, IN THE DESIGN OF A CPS CONTROLLER.

Design Abstraction Level	Interfaces and Constraints	Reference Documentation
1. Physical Environment	terrain, obstacles, environment, path	problem specification
2. Platform	encoders, sensors, switches, motors, gyro, accel	datasheet
3. Board	wiring, chip pinout, on-board communications, wireless	board user guide
4. System-on-Chip (SoC)	peripherals for sensor decoding, data conversion, communication (I2C, SPI, UART, CAN, ETH), timing and time capture, ADC	data sheet
5. Hardware Abstraction (HAL)	memory map and organization, interrupt vector table, register map, low-level API (bit-banging) for peripheral access	compiler backend, library source code
6. Firmware	high-level API (application-specific)	manual, library source code
7. Application Logic	computations, Finite State Machines, tasks and task communication	textbook, theory

III. CPS DESIGN IN LAYERS

Table I shows the seven levels of abstraction relevant to the integration of an embedded controller in a CPS. The learning objective for students is to understand the role of each layer in Table I in the overall implementation. There's great value in learning-by-doing. We therefore illustrate layered design by means of a roving-bot lab. This lab has not been used in a class yet, but is meant as an experiment to make CPS education tangible.

We use a Lynxmotion 4WD rover, shown in Figure 2 [2]. This rover has 4 independent motor drives but is controlled with a two-channel Sabertooth controller (left- and right-drive). The Sabertooth controller has a serial input that accepts various control commands for left/right motor power. Each wheel has a quadrature encoder. The Cerebot board, used in our microcontroller class, is serving as the rover controller.

In a roving bot controller design, a designer seeks to achieve a motion objective by writing a feasible program: a program that achieves real-time operation within the implementation constraints of the controller. Let's evaluate the layered design space of this roving bot. Figure 3 illustrates how the design space requires decisions at every layer of abstraction. Some of them, such as the decision to control either wheel speed or else wheel position, have a direct relationship to the motion objective, and they are easy. Other decisions are less straightforward, as they are only indirectly related to the motion objective. For example, consider the processing of encoder inputs. Speed measurement of slow-turning wheels can be done by direct polling of the encoders, or by a change-notification interrupt for each encoder step. Speed measurement of fast-spinning wheels, on the other hand, requires the use of input-capture with a timer, since the software can no longer keep up with the encoder events in real-time. Hence, the motion objective (wheel speed) affects peripheral selection, which in turns affects configuration of the Board, SoC, and HAL layer.

By working in layers, students will learn how some design decisions in the roving bot design will have only local effect, while other design decisions have repercussions across all abstraction levels. Although decisions will often still be ad hoc, they are made within the correct context, and they can

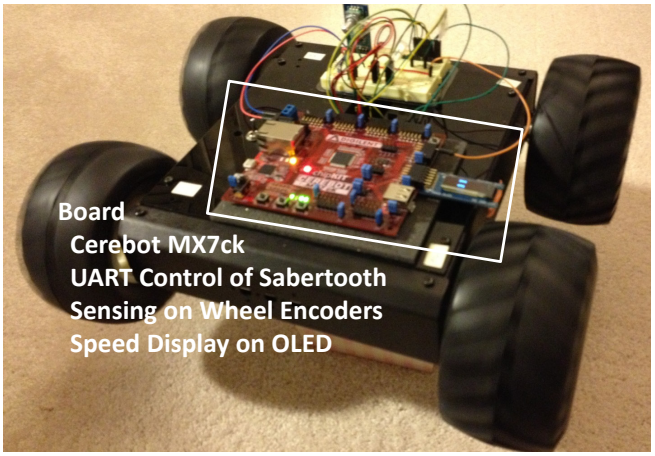

Figure 2. Hardware Setup of a Cerebot MX7ck on a Lynxmotion roving bot.

be related to the overall design. The major expected outcome is that students will learn to keep system-level concerns (such as design of system-level application logic) separate from architectural concerns (such as sensor integration).

A concrete example shows how students could navigate this design space. We'd like to write a program that lets the bot drive in circles, at half speed. Figure 4 shows the design process leading up to the development of the application logic. We use a bottom-up approach as it seems quite natural to start a circle-driver design with the motion objective.

- *Physical:* Driving in circles is easy enough if you can control the wheel speed. Knowing the physical dimensions of the roving bot and the radius of the circle leads to the set-speed for the left and right wheel.
- *Platform:* The left and right motor speeds are independently controlled by the Sabertooth controller, which itself is steered through a UART input. We also measure speed of the left- and right-front wheel.
- *Board:* To define the proper connections from the

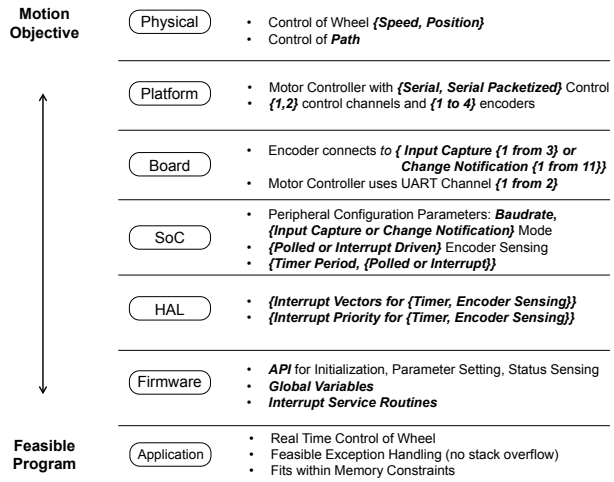

Figure 3. Design Space of the Roving Bot Controller; intermediate design decisions are captured in bold-italics.

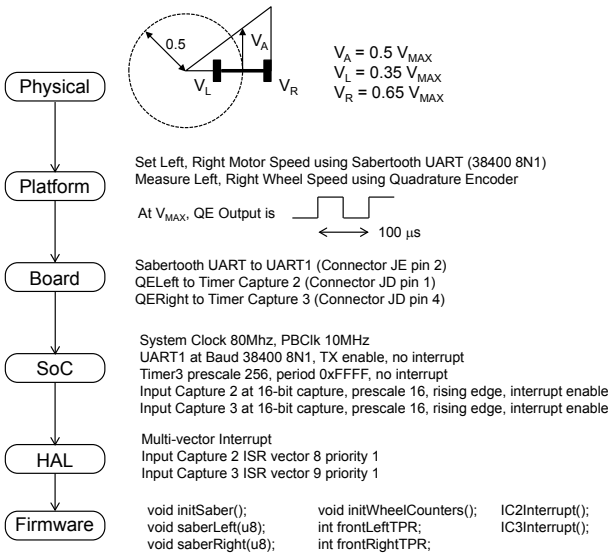

Figure 4. Design of the Circle Driver.

platform to the controller board, we need to select the peripherals that will be used in this application. We select UART1 for the motor control, input-capture 2 for the left wheel, input-capture 3 for the right wheel. This defines the proper board connections.

- *SoC:* The peripherals require appropriate configuration, including the system clock speed, the UART transmission format, the timer period and the capture parameters. The parameters have been chosen such that the fastest wheel speed does not overwhelm the microcontroller with input events. We've also defined interrupts for both input-capture peripherals.
- *HAL:* We've used independent interrupt vectors for each wheel, and we chose interrupt priority levels for each of them.
- *Firmware:* We developed an API to take care of motor control and motor speed measurement. Figure 4 shows the set of functions that were created. The interrupt service routines will update two global variables (`frontLeftTPR` and `frontRightTPR`) with the latest measurement from the encoders. `TPR` stands for *timer ticks per revolution*: a larger number means a slower speed.

After the structured development of this API, we can now write the application logic. For a student, who worked step by step through each of the abstraction layers, we think the program shown on the right should pose no problem to write.

IV. Conclusions

This paper argued for a layered approach to CPS education. It shows students the value of abstractions, while at the same time, it enables them to see across the abstractions in a structured manner. There are many interesting points left to cover, including design methodology, verification, performance analysis, and more. Such issues seem appropriate for a more advanced CPS course. In any case, the advent of CPS in courses brings an incredibly exciting time for students. At very few occasions before, has computer engineering had the opportunity to gain this level of visibility in daily life.

```
void main() {
  int leftspeed, rightspeed;
  initSaber();
  delayInit();
  initWheelCounters();
  enableWheelCounters();
  INTEnableSystemMultiVectoredInt();
  leftspeed  = SETLEFT;
  rightspeed = SETRIGHT;
  while (1) {
    saberLeft(leftspeed);
    saberRight(rightspeed);
    delayMs(100);  // adjust speed 10 times per sec
    if (frontLeftTPR > SETLEFTTPR) // too slow
      leftspeed = (leftspeed < MAX) ? leftspeed+1 :
                                      leftspeed;
    if (frontLeftTPR < SETLEFTTPR) // too fast
      leftspeed = (leftspeed > 0)   ? leftspeed-1 :
                                      leftspeed;
    if (frontRightTPR > SETRIGHTTPR) // too slow
      rightspeed = (rightspeed < MAX)? rightspeed+1 :
                                       rightspeed;
    if (frontRightTPR < SETRIGHTTPR) // too fast
      rightspeed = (rightspeed > 0)  ? rightspeed-1 :
                                       rightspeed;
  }
}
```

References

[1] Cerebot MX7ck, http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,396,986&Prod=CEREBOT-MX7CK.

[2] Lynxmotion 4WD, http://www.lynxmotion.com/c-111-no-electronics-kit.aspx.