The Forest and the Trees: Using an Appropriate Level of Library Abstraction for Microcontroller Instruction

Dr. Bryan A. Jones, Mississippi State University

Bryan A. Jones received the B.S.E.E. and M.S degrees in electrical engineering from Rice University in 1995 and 2002, and a Ph.D. in electrical engineering from Clemson University in 2005. From 1996 to 2000, he worked as a hardware design engineer for Compaq, specializing in board layout for high-availability RAID controllers. He is currently an assistant professor at Mississippi State University. His research interests include engineering education, robotics, real-time control system implementation, rapid prototyping for real-time systems, and modeling and analysis of mechatronic systems.

Dr. Robert B. Reese, Mississippi State University

Dr. Robert B. Reese is an associate professor in the Electrical and Computer Engineering Department at Mississippi State University.

Dr. M. Jean Mohammadi-Aragh, Mississippi State University

Dr. M. Jean Mohammadi-Aragh is a visiting assistant professor with a joint appointment in the Department of Electrical and Computer Engineering and the Bagley College of Engineering Dean's Office at Mississippi State University. Through her role in the Hearin Engineering First-Year Experiences Program, she is assessing the college's current first-year engineering efforts, conducting rigorous engineering education research to improve first-year experiences, and promoting the adoption of evidence-based instructional practices. In addition to research in first year engineering, Dr. Mohammadi-Aragh investigates technology-supported classroom learning and using scientific visualization to improve understanding of complex phenomena. She earned her Ph.D. (2013) in Engineering Education from Virginia Tech, and both her M.S. (2004) and B.S. (2002) in Computer Engineering from Mississippi State. In 2013, Dr. Mohammadi-Aragh was honored as a promising new engineering education researcher when she was selected as an ASEE Educational Research and Methods Division Apprentice Faculty.

The Forest and the Trees: Using an Appropriate Level of Library Abstraction for Microcontroller Instruction

1. Introduction

The relentless advances seen in the microcontroller market now challenge educators with, in the memorable phrase of the 1960s-era cartoon character Pogo, "an insurmountable opportunity." For less than \$5, modern microcontrollers provide a staggering variety of peripherals coupled with unprecedented processing power. This complexity leads to data sheets that easily exceed 1000 pages. Instructors face the problem of providing students ready access to microcontroller features without overwhelming them with the detailed knowledge that professional engineers often spend years acquiring.

In terms of microcontroller hardware abstraction, vendor libraries typically provide a more human-readable interface to control and status registers (`openUART1(BAUD_RATE_9600 | PARITY_NONE…`), for example), but this is of little assistance with higher-level abstractions (write a byte over the I^2C bus, for example, which requires a series of blocking reads and writes). Many real-time operating system (RTOS) products do provide helpful high-level abstractions, but often at the price of significant complexity better covered in a later course in embedded systems. Open-source libraries such as Arduino[1] and mbed[2] provide an easy-to-use interface for all peripherals, but often obscure the underlying implementation concepts students must master. There is a need for a low-complexity library that provides some abstraction while exposing low-level details.

This paper presents the design philosophy behind an open-source library[3] first released in 2008 and updated until the present, which complements a textbook[4] on microcontrollers adopted by nine universities. This library that provides a minimalist hardware abstraction layer for the 16-bit family of Microchip's microcontrollers that we believe will address the need for a better balance between abstraction and exposing students to underlying details. The library was created to enable educators to focus on the core concepts common to a microcontroller (such as general-purpose I/O, timers, and serial protocols such as I^2C and SPI) while simultaneously exposing some low-level details for students to gain a deeper understanding of the operation of a microcontroller. Expected students outcomes resulting from use of this library include the ability to solve problems using a variety of on-chip peripherals (digital I/O, analog I/O), the ability to communicate with off-chip devices using protocols such as SPI, I2C, and serial (UART), and the ability to create complex designs employing multiple peripherals in a capstone design context.

In addition to its pedagogical role, this library also serves a practical role in providing students hands-on experience using microcontroller peripherals in the course of their laboratory exercises and also scales to enable students to employ multiple microcontroller peripherals in the course of their capstone design experience. This library has been used since late 2008, and Google Analytics numbers for the library's website give 84,000 visits since January 2009, with approximately 52% of these visits originating from within the United States.

2. Background

Sweller's Cognitive Load Theory[5, 6, 7] (CLT) provides a framework for understanding why students learn (or do not learn) new concepts and problem solving strategies. CLT focuses on cognitive load requirements and the limited capacity of working memory. CLT includes three forms of cognitive load that are additive and must remain below cognitive load limits. First, *intrinsic cognitive load* can be thought of as the internal processing requirement that is inherent in the material being learned. Intrinsic cognitive load is static for a specific learning task, and higher for learning tasks with higher concept complexity. Second, *extraneous cognitive load* is the unnecessary cognitive load that has been introduced by the instructional techniques used to present material and the presentation of nonessential, supplementary information. Extraneous cognitive load interferes with learning and should be minimized. Third, *germane cognitive load* is the productive cognitive load associated with processing information and automating tasks. Germane cognitive load is modified by student characteristics (e.g., intrinsic motivation) and positive learning activities.

For improved student learning, cognitive load theorists encourage instructional designers to focus on the reduction of extraneous cognitive load. This is particularly important in cases where the material or learning task has high intrinsic cognitive load. Students' ability to acquire new skills and knowledge, particularly within the confines of a 3-hour, single-semester course, is quite limited. The goal of instructors is to present exactly what the students need to know -- no more, no less -- to maximize their learning. In other words, instructors need to reduce the extraneous cognitive load.

In student's first microcontroller course, the sheer volume of information accompanying a microcontroller, as discussed in the previous section, produces high cognitive loading. Educators must therefore judiciously choose what topics to cover, relying on a library to provide functionality where the details distract from the core concepts in order to minimize extraneous cognitive load. The library chosen, then, has a significant impact on the learning environment and should be selected carefully. Traditional library options are discussed in the following section.

3. Traditional Microcontroller Libraries

3.1 Vendor libraries

At the lowest level of abstraction, vendors often provide a library which gives human-readable descriptions of the hundreds of registers and thousands of control and status bits used by a microcontroller. For example, the 21,964-line header file for Microchip's dsPIC33EP128GP502 microcontroller defines the zero bit in the status register as `_Z`. This allows use of the statement `if (_Z) ...` to execute code if the zero flag was set. This library design presents no level of abstraction, resulting in high extraneous cognitive loading, but provides the clearest view of the hardware.

Vendor libraries also typically provide an additional level of abstraction; for example, the Microchip PIC24H family's library contains `void putsUART1(unsigned int *buffer)`, which writes a string to a serial port. These routines often provide a good level of abstraction, but require intimate knowledge of the underlying hardware. For example, `putsUART1` treats `buffer` as a collection of 16-bit (int) values when UART1 is configured for 9-bit operation, but casts `buffer` to a `char*` and transmits it as series of 8-bit values in 8-bit mode, a subtle point that could cause confusion for students, producing extraneous cognitive loading.

3.2 Arduino and mbed

Arduino[8] and mbed[9] provide a hardware-software combination: the hardware platform is accompanied by a rich set of libraries which gives access not only to the on-board microcontroller peripherals, but to a wide variety of common components (SD cards, Ethernet communication, LCD displays, etc.). These libraries provide a very high level of abstraction; for example, Arduino's `tone(pin, frequency)` outputs a square wave using pulse width modulation (PWM). The `pin` parameter refers to a silkscreen designation, not a physical pin on the underlying microcontroller; `frequency` is in Hertz, rather than in processor clocks. This enables a wide range of users, from K-12 students to students from a variety of majors with little training in electrical or computer engineering, to create amazing designs. However, it also hides the details essential to microcontroller instruction: what is PWM? How is a PWM frequency specified using processor clocks? How are digital I/O pins selected?

3.3 RTOS

A number of both free and commercial real-time operating system (RTOS) products exist, supporting a wide variety of microcontrollers. These products provide a middle ground in abstraction between vendor libraries and Arduino/mbed designs, providing platform-neutral methods for performing common tasks such as writing to a serial port. However, they also add significant complexity and functionality which leads to very high extraneous cognitive loading; for example, the QNX Neutrino product[10] is POSIX-certified, meaning that it provides hundreds of command-line utilities and thousands of functions. Most RTOS products provide threading, semaphores, message passing, and a host of other complex features best covered in a course dedicated to embedded systems, rather than an introductory course in microprocessors.

4. Designing Microcontroller Libraries for Education

Traditional approaches fill needs defined by industry (vendor libraries, RTOS products) or aim for a broad audience by abstracting away essential microcontroller features (Arduino and mbed). The need for an educationally-focused library is clear. In contrast to the traditional approach of designing a library by optimizing for performance, flexibility, or features, the design of this educationally-focused library is based on criteria of: 1) a high-level language, 2) clarity, 3) simplicity, 4) diagnostic error reporting, 5) detailed documentation, and 6) agility and availability.

4.1 High-level language

In the past, introductory computer science courses employed C or C++, the only feasible choice for instruction at the time. However, the modern explosion of many highly capable and expressive languages resulted in a switch to higher-level languages, such as Java or Python. Likewise, microcontrollers now posses the power to execute some of these languages; for example, one of the authors has ported Python-on-a-chip to the Microchip PIC24[11]. Likewise, the Java ME embedded platform supports several ARM processors[12]. In addition, many traditional microprocessor textbooks employ assembly language[13, 14].

The choice of C for this library therefore represents both a selection of an appropriate level of abstraction to enhance student learning by reducing extraneous cognitive loading and a recognition of C's dominant position as the only widely-supported high-level language for microcontrollers. Assembly, a low-level language, exposes every detail of a microprocessor's operation to the student, making it an ideal platform to expose students to the principles of microprocessor operation. Accordingly, the first half of the textbook accompanying this library[15] and its second edition which will be available in Fall 2014 employ assembly language. However, a microcontroller consists of more than just the microprocessor at its core; it includes a wide variety of on-chip peripherals whose operation must be covered in an introductory course. The choice of C both for this library and for the second half of the textbook shifts the focus from the low-level details of the core processor to higher-level concepts underlying topics such as digital I/O, interrupts, and communications protocols such as SPI, I^{2}C, and serial communication using UARTs. Higher-level languages, such as Java and Python, run on a virtual machine and therefore lack a direct connection to the underlying hardware of on-chip peripherals, which abstracts away some of the essential details (interrupts, control and status register bits) of on-chip peripherals. In contrast, C's ability to directly access hardware and its rich set of bit-manipulation operators makes it a better choice for presenting on-chip peripherals.

4.2 Clarity

Most C libraries are optimized for size, rather than simplicity. Per the C language specification[16], the entire contents of single source file (e.g. a translation unit) must be included when linking an executable. Therefore, most C libraries define one function per file, which produces the smallest-possible executable by linking only the required functions and variables from the library; as a result, most C libraries consist of hundreds of files. This approach significantly impedes student understanding by fragmenting a concept, such as serial communication over a UART (a universal asynchronous receiver/transmitter), over tens of files. In contrast, this library groups all the related functions and variables for a concept into two files (a header and a source file, as required by C); for example, the files pic24_uart.h/.c[17] provides routines which cover all low-level aspects of UART communication. This choice results in executables which often contain unused functions, but provides students with a cohesive view of a given topic.

Likewise, traditional approaches to library design often use complex, multi-statement macros to produce efficient executable code. This choice of optimizing for speed produces code that is often more difficult to understand; in particular, there are many unintuitive subtleties of macro usage[18]. This library, designed for clarity and readability, employs a minimum of multi-statement macros, while making extensive use of simple macros: `#define LED1 (_LATB13)` allows clear statements such as `LED = 1`, which turns an LED on.

4.3 Simplicity

Many libraries encourage extensive use of multi-threaded programming. Traditionally, the `main()` function performs foreground processing, while interrupt service routines (ISRs) handle device I/O. Even worse, traditional approaches to RTOS rely on multiple threads that interact via semaphores. This approach leads to complex, difficult to debug designs with subtle problems, including livelock, deadlock, starvation, and data corruption when shared variables are not accessed properly. Regardless of these perils, many courses expect students to successfully write multi-threaded programs in the form of interrupt service routines (ISRs) when interfacing with on-chip peripherals.

A modern approach espoused by Samek[19] views interrupts as events, and provides a state-machine driven framework for processing these events using message passing to communicate between state machines. This text encourages the use of a cooperative multi-tasking environment, which is inherently single-threaded, for many embedded designs. It provides excellent low-power capabilities; when the event queue is empty, the processor can be put to sleep until an interrupt generates an event for the state machine(s) to process.

Inspired by this approach, this library employs state machines to handle interrupts in a single-threaded manner. This significantly enhances the ability of students to both understand the interrupt-driven laboratory exercises they must complete and to debug relatively complex designs, due to the single-threaded nature of the exercise. For example, consider a classic LED/switch I/O problem in which a pushbutton should toggle the LED between a blinking and off states. A traditional solution would be to sample the pushbutton in a timer interrupt, then rely on a flag shared with `main()` to instruct `main()` to blink the LED, since blinking requires long delays unsuitable for ISRs. In contrast, this library's `main()` routine puts the processor in a low-power state, while an edge-triggered interrupt coupled with a timer interrupt provides a single-threaded approach to detect pushbutton events and schedule blinks using the timer. Further implementation details are given in the code which implements this approach.[20]

4.4 Diagnostic capability

As much as possible, a library should provide support for detecting and reporting errors in a program. For example, most libraries provide some type of assertion statement, which halts execution if the assertion fails. The library discussed in this paper does so, reporting the failed assertion, line, and file before resetting the processor. When an I/O operation fails, most libraries return an error code to the calling function; it is then the developer's responsibility to check every return code to insure the calls succeeded. This gives developers flexibility to handle errors in an application-specific way, representing a design choice for flexibility; students, along with many programmers, fail to check these results then become confused by the resulting errors which later calls produce. In contrast, this library does not return a error code, but uses an assert statement to reset the processor, representing a design choice for diagnostic feedback. This provides immediate and guaranteed feedback to students, helping them to isolate the source of a bug.

A unique feature lacking from many libraries but explicitly included in this library and invoked by default involves reporting diagnostic information on every reset; an example is given below.

```
Reset cause: Power-on.
Device ID = 0x00001E4D (dsPIC33EP128GP502), revision 0x00004003 (A3)
Fast RC Osc with PLL
../ledsw1.c, built on Jan  3 2014 at 10:35:25
```

The reset cause helps explain unexpected resets, by reporting brown-outs (possibly due to insufficient decoupling of V_{DD}), unhandled interrupts (typically due to a misspelled ISR name), misaligned accesses, and several other sources which help pinpoint the programming error; without this information, students (and even experienced developers) struggle to understand the unexpected behavior of their program.

Printing out the device ID also includes a check to verify that the chip the program was compiled for is actually in use, printing out a prominent error message if not. Next, specifying the operating clock source (the built-in RC oscillator multiplied by a phase-locked loop, or PLL) reminds students of their time base. If an external crystal is expected but not present, the library will print a diagnostic message stating that the switch from the internal RC oscillator to the crystal failed, helping pinpoint another common source of student confusion.

Finally, printing the time, date, and source file being run provides a quick sanity check: did the student actually flash the chip with the correct program, or did they accidentally program it with the last lab assignment? Did the student remember to recompile after editing? If so, the time and date should be within a minute or so of the current time. This information helps catch additional mistakes commonly made in the laboratory.

4.5 Expressive documentation

Traditional documentation generators, such as doxygen, javadoc, or Sphinx with the autodoc extension, transform structured comments into a hyperlinked application programming interface (API) reference; the homepage[21] of this library provides an example of doxygen's output. While this is a powerful, expressive tool which helps keep documentation synchronized with changes to the code by embedded the documentation into the source code as comments, this class of tool focuses solely on API documentation and lacks the ability to produce beautiful documentation of the implementation of a specific function. This ability is critical for educational use, particularly when explaining the operation of example code.

The literate programming paradigm introduced by Knuth[22] and implemented in the WEB (for the Pascal language) and CWEB[23] (for C) applications provides this needed ability, albeit with several significant drawbacks making it unsuitable for this library. First, CWEB's input is not source code, but a document containing fragments of code mixed with troff/nroff and CWEB markup, making it difficult to read. CWEB then transforms this input into source code, stripping out much of the markup and formatting, producing source code that cannot be understood apart from referring to the CWEB document it came from. While many other literate programming packages exist, most share the same weakness: they take a document as input and produce source code as output, producing relatively difficult to read code that cannot be directly edited, because it will be overwritten by the next document to code transformation.

Examples demonstrating concepts from the library therefore employ a novel new system which supports most of the concepts behind literate programming, termed CodeChat.[24] This system takes source code as input and produces an HTML document as output, relying on structured comments to guide the transformation. This helps keep documentation synchronized with changes to the code. The use of reStructuredText (ReST) for structured comments makes the source code readable to those with no knowledge of ReST; for example, the word `*italics*` would be rendered in an italic font using ReST.

4.6 Agility and availability

Finally, the design of this library has followed several common software engineering techniques to make it easier to develop and easier to share. First, all code is publicly available under a permissive open-source license using the Mercurial distributed version control system hosted by Bitbucket, providing a social coding platform which encourages collaboration and disseminates code readily. Second, a single invocation of scons, a build system, builds the libraries across many devices and clock configurations, builds the bootloader for many devices, and builds the documentation, making it relatively safe to test additions and verify contributions.

5. Conclusions and future work

Design of a library based on educationally-focused criteria, such as clarity and simplicity, produces a surprisingly unique result, particularly when compared to traditional library offerings. Commercially-available libraries are typically designed for rapid execution, flexibility, and a plethora of features. In contrast, this approaches helps produce a student-centric library to designed to maximize learning by reducing extraneous cognitive loading though abstracting unnecessary complexity, providing a readable and well-documented code base, focusing on modern single-threaded techniques, and providing unique and powerful debug capability.

The most important next step consists of measuring the impact of these design choices on student learning, then using the results gained to further refine the library. We are currently designing a survey to gauge student perceptions of the library, and a pairwise comparison of students who used a traditional library and students who used our educational library. Through those assessments have not been completed, we anticipate positive results based on anecdotal evidence from instructors and students.

In particular, instructors report that by using the library, students have demonstrated several important outcomes: the ability to solve relatively complex problems, such as generating an analog waveform from a DAC, communicating over the I^2C bus with a peripheral, and generating various frequencies using a ISR-driven timer during a single 3-hour lab session in a test-like practicum environment, where students may not work with others and are only given the problem at the beginning of the lab session. Senior capstone teams have both used the library in a number of complex designs and have used their familiarity with a library to employ microcontrollers and their libraries not taught in the classroom.

Students are capable of creating amazing designs with the proper instructional support from educators. A library designed for pedagogical use provides instructors with the ability to explain fundamental concepts behind microcontroller operation while challenging their students with laboratory exercises which explore the many communication protocols and capabilities of a modern microcontroller.

References

1 (2006). Arduino - HomePage. Retrieved January 5, 2014, from http://arduino.cc/.

2 (2009). Development Platform for Devices | mbed. Retrieved January 5, 2014, from https://mbed.org/.

3 (2014). PIC24 Support Library - Electrical and Computer Engineering. Retrieved January 5, 2014, from http://www.ece.msstate.edu/courses/ece3724/main_pic24/docs/_p_i_c24_support.html.

4 Reese, R. B., Bruce, J. W., & Jones, B. A. (2009). *Microcontrollers: From Assembly Language to C Using the PIC24 Family*. Cengage Learning.

5 Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, *12*(2), 257-285.

6 Sweller, J., & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and instruction*, *12*(3), 185-233.

7 Sweller, J., Van Merriënboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. *Educational psychology review*, *10*(3), 251-296.

8 (2006). Arduino - HomePage. Retrieved January 5, 2014, from http://arduino.cc/.

9 (2009). Development Platform for Devices | mbed. Retrieved January 5, 2014, from https://mbed.org/.

10 (2010). QNX Neutrino RTOS - QNX Software Systems. Retrieved January 5, 2014, from http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html#POSIX.

11 (2009). python-on-a-chip - p14p for short - Google Code. Retrieved January 5, 2014, from http://code.google.com/p/python-on-a-chip/.

12 (2012). Oracle Java ME Embedded Data Sheet. Retrieved January 5, 2014, from http://www.oracle.com/us/technologies/java/java-me-embedded-ds-1851546.pdf.

13 Peatman, J. B. (2002, August 1). *Embedded Design with the PIC18F452*. Pearson Education.

14 Kumar, N. S., Saravanan, M., & Jeevananthan, S. (2011, January 17). *Microprocessors and Microcontrollers*. Oxford University Press, Inc.

15 Reese, R. B., Bruce, J. W., & Jones, B. A. (2009). *Microcontrollers: From Assembly Language to C Using the PIC24 Family*. Cengage Learning.

16 ISO/IEC 9899:TC3 - Committee Draft of the C99 Standard, Section 5.1.1.1.

17 (2014). pic24_uart.h File Reference. Retrieved January 5, 2014, from http://www.ece.msstate.edu/courses/ece3724/main_pic24/docs/pic24__uart_8h.html..

18 (2002). 3.10 Macro Pitfalls - GCC. Retrieved January 5, 2014, from http://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html.

19 Samek, M. (2009). *Practical UML statecharts in C/C++: event-driven programming for embedded systems*. Taylor & Francis US.

20 ledsw1_cn_revised.c - Example of implementing a FSM in an interrupt. Retrieved January 5, 2014, from http://www.ece.msstate.edu/courses/ece3724/main_pic24/docs/sphinx/chap09/ledsw1_cn_revised.c.html.

21 (2014). PIC24 Support Library - Electrical and Computer Engineering. Retrieved January 5, 2014, from http://www.ece.msstate.edu/courses/ece3724/main_pic24/docs/_p_i_c24_support.html.

[22] Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97-111.

[23] Thimbleby, H. (1986). Experiences of ‘Literate Programming’ using cweb (a variant of Knuth's WEB). *The Computer Journal*, *29*(3), 201-211.

[24] (2014). CodeChat -- a conversational coding system. Retrieved January 5, 2014, from https://bitbucket.org/bjones/documentation/overview.