AC 2008-2938: EXPERIENCE OF TEACHING EMBEDDED SYSTEM DESIGN

Han-Way Huang, Minnesota State University-Mankato

Experience of Teaching Embedded System Design

Abstract
This paper reports our experience of teaching embedded system design. Embedded system design is the second microcontroller course in our microcontroller course sequence. In the first microcontroller course, we taught assembly language programming and basic microcontroller interface concepts. In the embedded system design course, the focus was on effective system design and development.

The course started with the definition and features of embedded systems and then moved on to system development methodology. C language was chosen as the language for programming embedded applications. Top-down design and hierarchical refinement were taught as the system development methodology. Students learned to think in blocks rather than in a single C language statement.

Systematic software and hardware debugging were taught. With a well-understood algorithm, students would know what the program execution result should be at certain point of their programs. This helped students to pinpoint the software bugs. For the hardware bugs, the students were taught to first make sure the circuit connection and the expected signal waveform during the program execution process. From there, students learn to locate the source of errors.

Programming style is another area that deserves a lot of attention. By following certain guidelines, many syntax errors can be avoided while at the same time programs become more readable and extensible. Students were taught to write reusable code. They were taught to convert common operations into functions and group them into files so that they could be included in other projects and be reused.

The choice of microcontroller to teach in an embedded system course is also important. There are several microcontroller families with good development tools and demo boards support. We have taught the Freescale HCS12, Microchip PIC18, and Silicon Laboratory C8051 in the embedded system class in the last few years. We felt that they are all very suitable for such an endeavor. All of these three families of microcontrollers have inexpensive software development tools and feature-rich demo boards for lab use. We have taught most of the peripheral functions available in the microcontroller including I/O ports, timers, compare/capture/PWM, UART, SPI, I^2C, A/D, D/A, CAN, and on-chip flash memory programming. Interrupts have been used extensively to allow multiple operations to be overlapped.

Introduction
In our first microcontroller course, we taught students assembly language programming and basic microcontroller interfacing concepts. Learning assembly language programming gave students an intimate feeling about how microcontroller hardware functions. On the other hand, they discovered that assembly language was not a productive language. In the embedded system design course, C language is chosen as the programming language. C language enables us to cover topics in a pace much faster than the assembly language. Our goal is to teach students to become an effective embedded system developer. The following topics became the focus of this course:

- Programming style
- System development methodology
- Debugging skills
- Software reuse

Programming Style

Programming style refers to a set of rules or guidelines used when writing the source code for a computer program. Many people believe that following a particular programming style will help programmers quickly read and understand source code and avoid introducing faults. However, students coming to the embedded system class seem to be ignorant of this issue. Therefore, we put out a set of guidelines for students to follow:

- Program header: Add a block of comments before the program. This block of comments indicates the author, the date of creation, and the purpose of the program.
- Function comment: Add a block of comments before the function declaration including the operations performed by the function, parameters, the caller of the function, and side effect.
- Naming of variables, constants, and functions: A name should spell out the purpose or function of the variable, constant, and function. We also recommended students to capitalize the first letter of the word when a name consists of several words. For example, the function name GetsUart indicates that the purpose of this function is to read a string from the UART port.
- Code appearance: Students were told to pay attention to program indentation, vertical alignment, and spacing. Proper indentation makes the logical relationship between statements stand out. This is especially important when there are several layers of if-else-if. Align similar elements vertically in a table allows the user to quickly discover whether there is any missing element. Proper spacing makes the program easier to read.
- Use program loop whenever the same operation need to be performed more than once. Using program loops may shorten the program and also makes the program modification easier.
- Enter matching parentheses before entering statements to avoid syntax errors. Mismatching of parenthesis could also totally change the program logic.

Software Development Methodology

In order to be productive, students need to learn a good system development methodology. We taught students to use the “top down design with hierarchical refinement” approach to deal with difficult problems. This approach was considered to be a very efficient, if not the best, system development methodology. Several examples were used to demonstrate this methodology. One of the examples that we used is “generation of a three-tone siren with frequencies equal 250 Hz, 500 Hz, and 1 kHz, and each tone lasts for half of a second”.

The top-down design with hierarchical refinement approach will go through this problem several iterations:

The first iteration may go like this:
Repeat
1. Generate the 250-Hz tone for half a second.
2. Generate the 500-Hz tone for half a second.
3. Generate the 1-kHz tone for half a second.

Forever

The second iteration deals with the generation of each tone. For example, the 250-Hz tone can be generated by:
Repeat
1. Pull the selected output pin to high.
2. Wait for 2 ms.
3. Pull the same pin to low.
4. Wait for 2 ms.

for 125 times

In the third iteration, the only issue that needs to be dealt with is how to generate 2-ms, 1-ms, and 500-μs time delays. A straightforward solution is to use one of the timer functions to create a time delay that is a multiple of 500 μs. By setting the required multiples, the desired delays are created.

After iteration 3, all the program details have been worked out and hence the algorithm can be converted into C program.

Throughout the whole course, students were reminded to practice this methodology in lab assignments.

Debugging Skills
Almost all the program assignments in this course involved both hardware and software. When an algorithm is first converted into a program, syntax errors may occur. Many syntax errors are not difficult to fix. However, there are some syntax errors that are not obvious for inexperienced students to figure out. A short list is as follows:

- Calling a predefined function but misspelling the case of one or a few letters of the function name.
- Calling functions that are in a different file but the file was not included in the same project
- Calling functions that are in a different file but the header file that contains the function prototypes is not included in the same program
- Mixing variable declarations with the executable statements

After all syntax errors have been removed, the project can be built successfully and downloaded into the target hardware or demo board for execution.

If the program did not work, the first thing to do is locating the source of the problem: hardware or software or both.

In our lab environment, students used integrated development environment (IDE) to enter and debug their programs and used debug adapter to download program onto the demo boards for execution. The IDE contains a text editor, assembler, linker, compiler, simulator, device drivers, and project manager in one package and allows the user to stay in the same environment when performing different tasks. The IDE interacts with the debug adapter to perform real-time debug activities such as monitoring the contents of memory locations and registers, stepping the program, setting breakpoints, displaying the contents of memory locations and registers at the breakpoint. In most of the assignments, students were required to wire peripheral chips and/or other I/O devices such as motors, seven-segment displays, keypad, and so on. The most common hardware errors that students made include missing power, missing ground, loose wires, and wrong signal connection. Occasionally, problems were caused by bad demo boards, bad I/O devices, and so on. Students were told to check power and ground connection before checking other wiring errors. If signals were wired correctly and the program still didn't work as expected, then check the program.

There could be any kind of software errors. We suggested students to read their program first to find out obvious errors such as wrong expression, incorrect configuration in peripheral modules, and so on.

We have taught students how to identify the following software errors:

- *Mismatch of data types*. Data type mismatch has often been ignored by students. For example, assigning an expression that contains character typed source variables to a destination variable of integer type may generate correct result at sometime but incorrect result at other time. This type of error can be fixed by type-casting the source variables to the same type of the destination variable.
- *Getting stuck at some point of the program*. Embedded software often contains statements that wait for a flag to be set (or cleared). If the flag never gets set (or cleared), then the program gets stuck. This type of errors can be discovered by setting a breakpoint after the statement that waits for the flag to be set (or cleared).
- *A finite program loop became an infinite loop*. There could be many reasons for this to happen. This type of errors can be identified by using breakpoints. Once this error is identified, the exact cause is not hard to discover.
- *Negligence to the precedence of operators*. If the programmer did not pay attention to the precedence of operators, the computation result could be very wrong. We taught students to use parenthesis whenever they are not sure about the precedence of operators.
- *Incorrect program algorithm*. After making sure the previous errors did not exist but the program still did not run correctly, then the most likely cause would be the algorithm. Students were told to reevaluate their algorithm to fix errors.

Sometimes the circuit wiring appeared to be correct and program also looked correct but the result was wrong. In this situation, we suggested students to change demo boards. Sometimes this approach did solve the problem. This approach made sense because after a period of use, the I/O pin may be damaged due to inappropriate handling of the demo board. The cultivation of debug skills takes time, experience, and patience. We have encountered a case in which students were assigned to control the DC motor speed using one of the timer functions. One student could not get the motor to rotate no matter what he tried. It turned out that the power supply in the demo board did not drive the DC motor adequately. After a separate power supply was added, the motor started to run correctly.

Software Reuse

After programming the microcontroller for sometime, the programmer may discover that there are common operations associated with each peripheral module. These common operations can be converted into functions and may be called by many applications that involved in the associated peripheral modules. The following operations are the most common ones:

- Output a single character via the peripheral function
- Output a string via the peripheral function
- Read a single character from the peripheral function
- Read a string from the peripheral function

Some of the functions can be parameterized and become more useful. For example, the user may write a function to create a constant time delay. The created function would be more useful if it is modified to generate a time delay that is a multiple of certain base length such as 1 ms, 10 ms, etc. The caller of the time delay function can generate a wide variety of time delays by passing a different multiple to the function.

As applications become more and more complicated, more and more common operations can be identified. By making these operations into functions, they can be reused in many other applications. To promote the idea of software reuse, conscious effort is needed right from the beginning of the course. Students need to be reminded from time to time until the idea gets into their instincts. Students were also encouraged to use the library functions provided by the C compiler.

Conclusion

We are teaching embedded system as a second microcontroller course in both engineering and technology programs. The main focus is on effective system development. We thought students can write readable, maintainable programs and avoid many syntax errors by following a good programming style. We taught students to follow the “top down design with hierarchical refinement” approach to make complicate problems more manageable. We also reminded students to write reusable code by extracting common operations into functions so that they could be reused in other applications. Hardware and software debugging is an important part of the system development cycle. Students learned debugging skills by working on a wide range of lab assignments. The key in debugging is to identify the cause of the error. Some errors are

easy to identify whereas others are not. It takes patience and experience to build up the debugging skill.

To teach embedded system design, we need to choose a microcontroller as a target. The main consideration is whether the chosen microcontroller has appropriate software tools, debug adaptors, and demo boards. We have tried the Freescale HCS12, the Microchip PIC18, and the Silabs C8051F040 microcontrollers in this course over the years. These three companies provide free integrated development environment (IDE) to be used by their customers. Freeware software C compilers are also available for these three microcontrollers. These three microcontrollers also have good demo boards to be used in learning their products. The information of the PIC18 and C8051 demo boards that we used can be found at www.evb.com.tw whereas the HCS12 demo board information can be found in www.evbplus.com. Good textbooks will make the teaching of embedded system design course a breeze. There are several good textbooks available for the HCS12 [2,4] and the PIC18 [1,3]. Currently there is no textbook for the SiLabs C8051F040, we prepared a set of lecture notes for it [5].

References

1. "The PIC18 Microcontroller—An Introduction", Han-Way Huang, Delmar Cengage, Clifton Park, New York, 2004.
2. "The HCS12/9S12—An Introduction", Han-Way Huang, Delmar Cengage, Clifton Park, New York, 2005.
3. "Embedded Design with the PIC18F452 Microcontroller", J. Peatman, Prentice Hall, Upper Saddle River, New Jersey, 2003.
4. "Embedded Systems: Design and Applications with the 68HC12 and HCS12", Steven F Barrett, Daniel J Pack, Prentice Hall, Upper Saddle River, New Jersey, 2004.
5. "C8051 Lecture Notes", prepared by the author.