Teaching an Embedded System Course to Electrical Engineering and Technology Students

Kalyan Mondal
Gildart Haase School of Computer Sciences and Engineering
Fairleigh Dickinson University
Teaneck, NJ 07666

Introduction

A rapid growth in the application of embedded programmable processors in systems from simple household machines (e.g., washers and dryers) to complex real-time control in automobiles has been seen over last three decades. Microprocessors, microcontrollers, and digital signal processors (DSPs) have been at the forefront such development. This has warranted training electrical and computer engineers in this important area of embedded system design using a multi-disciplinary approach. Electrical and computer engineering undergraduate programs require at least one embedded system design or programming course to train the future workforce in this important field.

Many interesting embedded system teaching paradigms have been presented in [1-3]. In this paper, we present our experience with teaching a microcontroller based system design course to the junior undergraduate students. In this course, our focus mostly has been in teaching input / output interface design through proper programming techniques. We used the technique of compare and contrast of multiple solutions for such interface designs as outlined in the rest of the paper. We believe that students become well prepared for the real world by learning the tradeoffs between different approaches to the design and mastering them in a hands-on laboratory environment.

Microcontroller Architecture and Assembly Language Programming

Our students take a course on microcontroller based design as the sixth-semester course after they have learnt about the architectures of Intel 80xxx series microprocessors and their programming in assembly language. They also learned in their fifth-semester microprocessor design course the disassembly, memory, and register viewing processes. So in this course we introduce students with microcontrollers that get used in the design of many embedded systems. A hands-on laboratory is an integral part of this course and the instructor and students try out various concepts on the available hardware and software platforms. Specifically we focus on the Freescale HCS12 microcontroller and point out major differences (e.g., a plethora of parallel ports, built-in interfaces, etc.) between microprocessors and microcontrollers. We introduce the architecture and assembly language of HCS12 which allows students to appreciate their differences from those of the Intel microprocessors. We provide a few simple examples of

assembly language programming and do not dwell on this topic for long since one objective of this course is for students to use a high level language for programming. By the time we introduce interfacing input / output devices (e.g., 7-segment displays, pushbutton switches, etc.) through parallel ports, the students realize why programmable microcontrollers are preferred in embedded system applications.

Programming in C Language: Integrated Development Environment

We quickly move to using C language for programming HCS12 microcontrollers since that allows students to get a perspective on embedded system programming in the industry. We introduce a few code fragments showing how simple C language constructs can be coded in assembly language showing the usefulness of high level language programming. One such example is shown in Fig. 1.

```
movb    #$FF,DDRB       ; configure PORT B for output
ldaa    DDRJ
oraa    #$03            ; perform logical OR operation on DDRJ data
staa    DDRJ            ; configure PJ1 ~ PJ0 pins for output
movb    #$FF,PORTB      ; output a high
ldaa    PTJ
anda    #$FE            ; perform logical AND operation on PTJ data
staa    PTJ             ; send data to DAC and start it
```

Fig. 1 (a): An example of an assembly language code fragment

```
DDRB = 0xFF;            // configure PORT B for output
DDRJ |= 0x03;           // configure PJ1 ~ PJ0 pins for output
PORTB = 0xFF;           // output a high
PTJ &= 0xFE;            // send data to DAC and start it
```

Fig. 1 (b): Equivalent C language code fragment

Students use CodeWarrior integrated development environment (IDE) to code, compile, simulate, debug, and run it on the Wytec Dragon-12 Plus development board. Students very quickly notice the availability of various external input / output devices interfaced to the HCS12 on the Dragon-12 Plus board and anticipate programming the microcontroller for generating audio-visual effects.

Code Debugging without "printf"

Most students in the course come with some exposure to the C/C++ language and programming on Microsoft Visual Studio. However, a considerable amount of time is spent in reviewing the important language features to improve students' programming skills. The CodeWarrior IDE is

introduced and students are encouraged to code and debug simple C functions and programs on it. At this point, the students realize that high level language program debugging cannot always be done by formatted print out of various intermediate variables and the final result. We show techniques to modify “printf” oriented programs for debug on CodeWarrior. An example of such code modification is included in Fig. 2. Students realize that program inputs / outputs can be realized in many different ways and thus they do have to know the specifications for input / output prior to developing embedded system programs.

```
// Find first five integers not divisible by 2, 3, 4, 5, and 6 but divisible by 7 [4]
#include "hcs12.h"
#include <stdio.h>
int main (void)
{
        int cnt = 0, i = 1;
        printf(“\nThe 1st five qualified integers are: ");
        while (cnt < 5) {
                if ((i % 2 == 1) && (i % 3 == 1) && (i % 4 == 1) && (i % 5 == 1))
                        if ((i % 6 == 1) && (i % 7 == 0)) {
                                cnt ++;
                                printf(“   %d”,i);
                        }
                i++;
        }
        printf(“\n”);
        asm("swi");   /* software interrupt so that processor halts at this breakpoint */
        return 0;
}
```

Fig. 2 (a): Example C code not suitable for CodeWarrior debug.

```
// Find first five integers not divisible by 2, 3, 4, 5, and 6 but divisible by 7
#include "c:\cwHCS12\include\hcs12.h"
void main (void)
{
        int cnt = 0, i = 1, arr[5];                     // arr[]: Saves 1st five qualified integers
        while (cnt < 5) {
                if ((i % 2 == 1) && (i % 3 == 1) && (i % 4 == 1) && (i % 5 == 1))
                        if ((i % 6 == 1) && (i % 7 == 0)) {
                                arr[cnt ++] = i;         // store matching values in arr[]
                        }
                i++;
        }
        for (;;) {                                      // infinite loop
        }
```

```
}
```

Fig. 2 (b): Modified code for the example in Fig. 2 (a) suitable for CodeWarrior debugging. Arr[] shows the five integers in the Data2 panel of the CodeWarrior debug window.

Parallel Input / Output Interface Programming: Direct Port Programming versus Using Existing Functions

Next we introduce simple parallel input / output devices (e.g., LEDs, 7-segment displays, DIP and pushbutton switches) interfaced through various ports of the HCS12. We first show how the relevant parallel ports are programmed by bit-level manipulations using C language [4]. Later as prescribed by the authors of [5], we introduce the C functions that encapsulate detailed port programming. Although these functions make interface programming quite easy, the students lose sight of how the interfaces actually work based upon what gets read and/or written to the relevant ports. Some students do initially get a little confused with the two types of solutions to the I/O interface programming and keep on mixing up the two methods. Later we show them examples such as the one shown in Fig. 3 to drive the point that both methods result in the same interface behavior.

```
// LBE [5] Example 1b: Turn ON every other LED after disabling 7-segment displays
#include <hidef.h>                  /* common defines and macros */
#include <mc9s12dg256.h>            /* derivative information */
#pragma LINK_INFO DERIVATIVE "mc9s12dg256b"
#include "main_asm.h"               /* interface to the assembly module */
void main(void) {
        PLL_init();                 /* set system clock frequency to 24 MHz */
        seg7_disable();             // disable 7-segment displays
        led_enable();               // enable leds
        leds_on(0x55);              // turn on every other LED
        for(;;) {}                  /* wait forever */
}
```

Fig. 3 (a): Example C code using functions.

```
// Textbook [4] version: Turn ON every other LED after disabling 7-segment displays
#include "hcs12.h"
void SetClk8(void);
void main (void) {
        SetClk8();                  // set E clock to 24-MHz
        DDRB    = 0xFF;             /* configure Port B for output */
        DDRJ    |= 0x02;            /* configure PJ1 pin for output */
        PTJ     &= 0xFD;            /* enable LEDs to light by setting PJ1 = 0 */
        DDRP    |= 0x0F;            /* configure PP[0:3] pins for output */
```

```
        PTP     |= 0x0F;         /* disable 7-segment displays */
        PTB = 0x55;             // turn on every other LED
        while(1);
}
```

Fig. 3 (b): Example C code in Fig. 3(a) rewritten using direct parallel port programming.

The students learn that the functions in [5] can be readily written from the bit level port programming concepts as shown by an example in Fig. 4. Such a study helps them build up their confidence in embedded system software development in a high level language under different support conditions.

```
void led_enable()               //Function to enable on board LEDs
{
        DDRB = 0xFF:            //set Port B for output; LED anode to be set to a voltage
        DDRJ = 0xFF;            //set Port J for output; LED cathode to a voltage
        PTJ &= 0x02;            //connect LED common cathode to ground
        PORTB = 0x00;           //turn off all 8 LEDs by setting anode voltage low
}
```

Fig. 4: Example of a function in [5] in terms of port programming.

Interrupts & Their Uses: Elimination of delay Functions

Later we introduce the concepts of interrupt programming. We explain how real time interrupt can be used to delay an event from another. As an example, we show the conventional code (in Fig. 5 (a)) for displaying ten shifting patterns of natural numbers on four 7-segment display digits. This requires a delay of 1600 ms between each shifting pattern display [4].

```
// Display 10 patterns of shifting 0 – 9 digits on 7-segment displays [4]
#include "hcs12.h"
#include "delay.h“
void SetClk8(void);
unsigned char segPat[13]  =   {0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x67,0x3F,0x06,0x5B,0x4F};
unsigned char digit[4]      = {0xFE,0xFD,0xFB,0xF7};
void main(void) {
        int seq, j, ix;
        SetClk8();                                              // set E clock frequency to 24 MHz
        DDRB = 0xFF;                                            //configure Port B for output
        DDRP = 0xFF;                                            //configure Port P for output
        while(1)
                for (seq = 0; seq < 10; seq++) {                // pattern array start index
                        for (j = 0; j < 400; j++) {             // repeat loop for each pattern sequence
                                for (ix = 0; ix < 4; ix++) {    // select the display # to be lighted
                                        PTB     = segPat[seq+ix]; // output segment pattern
```

```
                                        PTP     = digit[ix];        // output digit select value
                                        delayby1ms(1);              // display one digit for 1 ms
                                }
                        }
                }
}
```

Fig. 5 (a): Example C code for displaying shifting patterns without using interrupts.

The run time interrupt (RTI) based program [4] eliminates explicit use of "delay" functions and some of the loops used above as can be seen from Fig. 5 (b). The core display function gets included in the interrupt service routine (ISR). Comparing and contrasting the two coding schemes, pros and cons of interrupt programming are elegantly explained.

```
// Display 10 patterns of shifting 0 – 9 digits on 7-segment displays using interrupts [4]
#include "hcs12.h"
#include "SetClk.h"
int     seq;                    // start index to segPat[] of a sequence of digits (0 to 9)
int     ix;                     // index of digits of a sequence (0 to 3)
int     count;                  // repetition count of a sequence
char segPat[13] = {0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x67,0x3F,0x06,0x5B,0x4F};
char digit[4]   = {0xFE,0xFD,0xFB,0xF7};
void main (void) {
        seq     = 0;            // initialize the start index to segPat[] for the display sequence
        ix      = 0;            // initialize the index of a new sequence
        count   = 400;          // initialize the RTI interrupt count of a sequence
        SetClk8();              // set E clock to 24 MHz from an 8-MHz crystal oscillator
        RTICTL  = 0x40;         // RTI interrupt interval set to 2^13 OSCCLK cycles
        DDRB    = 0xFF;         // configure Port B for output
        DDRP    = 0xFF;         // configure Port P for output
        CRGINT|= RTIE;          // enable RTI interrupt
        asm("CLI");             // enable interrupt globally
        while(1);
}

// RTI interrupt service routine

interrupt void rtiISR(void) {
        CRGFLG = 0x80;          // clear RTIF bit
        PTB     = segPat[seq+ix]; // send out digit segment pattern
        PTP     = digit[ix];    // turn on the display
        if (++ix == 4)          // make sure the index to digits of a sequence is from 0 to 3
                ix = 0;         //       "
        if (--count == 0) {     // is time for the current sequence expired?
                seq++;          // change to a new sequence of digits
```

```
                count = 400;        // reset repetition count
        }
        if(seq == 10)               // is this the last sequence?
                seq = 0;            // reset start index of a sequence
}
```

Fig. 5 (b): Example C code for displaying shifting patterns without using interrupts.

Existing Code Modification: Code Reuse for Faster Design

We emphasize reusing and modifying an existing code for a specific application. Such exercises help students in appreciating the role of design support engineer. Many homework and examination problems are set by providing working codes and asking students to modify the codes appropriately per given specifications. In some cases, the header files in the code provided to students point to nonexistent folders. These forces students to learn how to search and find appropriate header file for inclusion in the project. Another common mistake by students is in not including an interrupt vector table file in programs based on interrupts. In such cases, the students find the code to compile and not run properly. By correcting for such omissions, students learn a valuable lesson of code debugging and fixing.

Outcome Assessment and Analysis

Student homeworks and laboratory exercises have especially been identified to test most of the specific learning that we have described in this paper. Specifically, the following outcome assessments are being developed for use in the course:

1. Code Debugging without “printf”: This outcome is assessed as part of a larger assessment involving testing the proficiency of code debugging. We assess students’ ability to debug “printf” based codes on CodeWarrior by asking them to write an application program in C with a print out of the results. Proficiency in code debugging outcome assessment involves additional tests where either incomplete or buggy code fragments are provided or students have to fix the problems and make the program run with expected results. We consider this learning outcome to be very important since many engineers will be involved with fixing and upgrading existing applications in their jobs.

2. Parallel Port Programming versus Using Library Functions: We assess whether students are capable of interface programming in both ways. This is done through several laboratory exercises. One such exercise involves providing a fully functional program using functions (or direct port programming) and asking students to recode it by direct port programming (resp. using functions). The students get graded on whether the recoded program works or not and how

many functions (resp. direct port program lines) are actually recoded by direct port programming (resp. using functions). We have not mechanized the speed of coding in this assessment yet which is planned for the future.

3. Conventional Timing versus Interrupt Based Designs: This outcome is assessed by providing students a functional program that uses "delay" functions and conventional looping technique and asking them to convert it using real time interrupt or some other appropriate clock-timer based interrupt mechanism. Many students find this code conversion challenging and first time assessment results in poor score. The score improves by the time of final examination after students have done multiple such laboratory exercises.

In the spring 2010 when the course was first presented, we used EAC-ABET specified outcome C to assess the learning outcome of this course. Outcome C specifies assessing an ability to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability. The rubric developed for this outcome partially measured learning 2 and 3 mentioned above. Basic rubric and assessment result are included herein to outline the process used in 2010.

Rubric Based on the Lab: RTI Driven Display System of a Microcontroller

The student explores the software and hardware input/output interfacing aspects of a practical microcontroller subsystem by designing blinking of lights and/or characters on various display devices. These laboratory exercises include the following elements:

- The use of specific ports and loading of proper register values to enable and/or disable displays (Lab 1).
- The use of real-time-interrupt (RTI) capability of the microcontroller for cost effective display generation.
- Application of library functions to implement the display interface program (Lab 2).
- The use of a commercial integrated development environment (IDE) to permit the development, compilation, debugging, downloading into the development board and running the program.

The outcome C rubric outlines the design elements of these laboratory exercises and how a student's performance of this activity is rated.

Applying the Rubrics:

These rubrics were applied to the Lab Exercises 1 & 2 performed by students during the Spring 2010 semester of the course as part of their homework set #8. An experienced graduate assistant was available in the laboratory to help the students, explain the requirements of the laboratory and observe the correct operation of the programs.

Each Lab Exercise is structured in three parts, and each part is worth 10 points for a total of 30 points.

Lab Exercises 1 & 2 Measurements:

1 = Interpretation of specification and devising a cost effective solution
2 = Using implementation tools/equipment properly
3 = Documenting the design and drawing conclusions

Rating of Sample Lab Exercises Performed during the Spring 2010 Semester

The laboratory exercises were developed for the first time using two different concepts. Lab 1 is based upon programming the uC port registers and other control registers to implement a display function. Lab 2 on the other hand encourages the use of pre-existing input-output interface functions to implement a display driver solution.

	Lab 1			Lab 2			
Measure	1	2	3	1	2	3	Total
Average	4.5	9.5	3.5	6	9	2.5	35

Based on the scores of 10 students, the average was 35 points (58.3%) with a spread of 15 points (25%) to 50 points (83.3%). The minimum acceptable score is 30 points (50%). All students met the objectives of outcome C.

Concluding Remarks

Additional interrupt based programming using timers and other on and off-chip devices are introduced in the course. Overall we found that showing students how to use multiple techniques to program the same input / output interface application is quite essential in the learning process. Learning to develop interface programs using a good set of support functions is easier, but it fails to prepare students to face the varying work place setups. In many cases, a good function package may not be available and the engineer needs to develop the application by bit level interface programming. Thus at the end of our course, the students feel more comfortable in programming with any given set of support functions as they may face in their industrial career. New rubrics to assess the specialized learning outlined in this paper are under development and will be used for the course. The results will be reported in a future publication.

References

[1] Wong, S., Cotofana, S. "On Teaching Embedded Systems Design to Electrical Engineering Students." Retrieved March 13, 2011, from http://ce.et.tudelft.nl/publicationfiles/620_14_s_wong_ES.pdf.

[2] Flynn, A. M. "DSPs in Teaching Embedded Systems." Retrieved March 13, 2011, from http://www.ti.com/sc/docs/general/dsp/festproceedings/fest2000/micro001.pdf.

[3] Berger, A. S. (2001). "A New Perspective on Teaching Embedded Systems Design." 3/20/2001 12:43 PM EST Retrieved March 13, 2011, from http://www.eetimes.com/discussion/guest-editor/4023302/A-New-Perspective-on-Teaching-Embedded-Systems-Design.

[4] Huang, H.-W. (2010). The HCS12/9S12: An Introduction to Software and Hardware Interfacing, Delmar Cengage Learning.

[5] Haskell, R. E., Hanna, D.M. (2008). Learning By Example Using C Programming the Dragon 12-Plus Using CodeWarrior. Rochester, MI, LBE Books.

[6] ABET. (2009). "Criteria for Accrediting Engineering Programs." Engineering Accreditation Commission, ABET Board of Directors, from http://www.abet.org/Linked%20Documents-UPDATE/Criteria%20and%20PP/E001%2010-11%20EAC%20Criteria%201-27-10.pdf .