Non-Portable C-Language for Microcontroller Applications

Stephanie Goldberg
Department of Technology
Buffalo State College

Abstract

A previous goal of the microprocessor/microcontroller class in the Buffalo State College Engineering Technology Program was to develop proficiency with an assembly language in order that students could write assembly language code for various microprocessors and microcontrollers. The goal has been modified such that students become familiar with assembly language programming as well as understanding the role of a high-level language such as C in microcontroller applications.

Concepts of portability, variable storage space, and hardware registers are presented to help students understand the strengths and weaknesses of programming a microcontroller with high-level language such as C. A high-level language brings features like loops, arrays, and decision-making capability to the very rudimentary assembly language. Standard C languages such as ANSI C are portable, meaning they are independent of the microcontroller that will ultimately be used to execute the code. However, to best utilize the microcontroller for digital I/O and timing delays as well as many other tasks, read and write access to the specific hardware registers of that microcontroller are needed and therefore portability must be sacrificed. In this case, a "special compiler" is required that recognizes the specific hardware of the microcontroller. An example of such a compiler is the Rigel Corporation 8051 C compiler, which provides two methods for communicating with specific hardware in the 8051

Note: Microcontroller will refer to both microcontrollers and microprocessors in this paper.

1. Introduction

The direction of the Microcontroller course in the Bachelor of Electrical Engineering Technology program at Buffalo State College has been transitioning from an intensive assembly language focus to one that incorporates the important role of high-level languages in the microcontroller environment. In past semesters, we prepared students for proficiency in any assembly language. Students were provided with a detailed account of assembly language, focusing on the complete instruction set of a particular microcontroller. Students developed code for "simple" hardware-oriented tasks such as output of digital data from a register as well as more intensive type of programming (tasks that could be done more efficiently with a high-level language), like looping, counting, and decision-based jumps.

The present curriculum provides students with an overview of assembly language in the first part of the course. Students study a portion of the instruction set and then modify existing programs. We focus on the "simple" tasks that relate to the microcontroller hardware, such as input/output from Port registers and time delays using the internal timer/counter. The term Special Function Registers (SFRs) refers to the specific hardware registers in the 8051 microcontroller. We leave the higher-level tasks, such as the setting of a time delay based on input data, which take great effort in assembly language to the high-level language.

In the second part of the course, students examine the same three hardware-oriented tasks of data input, data output, and time delays using C language. Standard C languages such as ANSI C (the latest standard set in 1999) are portable. Portability means the source code is independent of the particular microcontroller that will ultimately execute the code. It is pointed out with great emphasis that a portable language like C does not recognize specific hardware of the system it is being compiled for.

The intimate correspondence between assembly language and the microcontroller hardware must be incorporated in the C environment to achieve the benefits of speed and hardware resource control associated with assembly code. The solution is an enhanced C language compiler, which compiles portable Standard C and, additionally, has provisions (often called extensions) to allow handling of the microcontroller hardware resources. The inclusion of hardware-specific code makes the code non-portable meaning that at the source code stage, it is already microcontroller-specific; it can only be compiled for the targeted microcontroller. C Compilers with extensions can generally be found for a given microcontroller. The compiler may permit all or a subset of a Standard C language and offer extensions to take advantage of the specific microcontroller. It should be noted that the compiler methods used to access the hardware could vary between different 8051 C compilers. Students work with the Rigel Corporation C Compiler[1], which offers two methods for handling 8051 hardware. These methods are discussed in Sections 3.1 and 3.2. 8051 circuit boards and code development software are also provided by Rigel.

Students study an application utilizing LEGO Robots in the last third of the course. LEGO Mindstorms RIS[2] robots are used to provide an introduction to Robotics along with a C-language application. Our school purchased several of the kits and we use a free C-like code called NQC (Not Quite C) to program the robots. Students construct a robot following directions in David Baum's book "Extreme Mindstorms"[3]. They examine the NQC programs that are provided by Baum. Modifications are then made to the code. The programs exercise the sensor input and motor control features of the robot. The NQC code supports a Hitachi microcontroller in the robot.

The paper will discuss the three portions of the current curriculum. In the next section we look at assembly code exercises. Section 3 follows with experiments using the Rigel Corporation C compiler to bridge assembly and C. A C-like application using Lego Mindstorms Robots is presented in Section 4.

2. Assembly Language

We focus on three hardware-oriented tasks: inputting digital data, outputting digital data, and creating a time delay. Each task is intimately related to the specific registers and counters of the 8051 microcontroller. Note these tasks are essential in microcontroller applications, and each microcontroller has unique hardware and software to efficiently perform them.

The assembly language code and hardware involved in these three tasks are presented in class lecture. (Figure 1 shows the assembly language code for the three tasks. Note the code may not be complete). The students study the 8051 instructions that are employed in the three tasks. Students follow up in laboratory by entering the code with a text editor and assembling and downloading the code using the Rigel RJ 31P 8051 board and READS51 software development environment. They construct a hardware interface board consisting of drivers and LEDs to verify Digital Output data and DIP switches to provide User Digital Input data. The time delay is verified by delaying between different output patterns sent to the LEDs.

In the Rigel environment students can exercise their assembly code in single-step fashion while examining the contents of various registers and memory locations in the 8051. After students become familiar with these tasks they are asked to modify the programs by sending different patterns and coding for different delay times.

Figure 1: Assembly Code for three hardware-oriented tasks (below)

Code 1: Output data from the 8051 using PORT 1

```
#include <sfr51.inc>        ; 8051 ports are defined here
cseg at 8000H               ; Start code at 8000H

MOV A,#55H                  ; Put the 8-bit number 55H into the A reg
MOV P1,A                    ; Move contents of A reg into Port 1

MOV A,#0AAH                 ; Put the 8-bit number 55H into the A reg
MOV P1,A                    ; Move contents of A reg into Port 1
```

Code 2: Input data from DIP switches to the 8051 using Port 3 bits 2-5

```
#include <sfr51.inc>        ; 8051 ports are defined here
cseg at 8000H               ; Start code at 8000H

 MOV  A, P3                 ; Read PORT3 into Register A
ANL  A, #03CH               ; Isolate PORT3 bits 2-5
MOV  P1, A                  ; Move contents of A into Port 1 to verify
```

Code 3: TIMING DELAY

```
        #include <sfr51.inc>      ; 8051 SFRs are defined here
        cseg at 8000H             ; Start code at 8000H

         MOV  R1,#0FFH            ; initialize register R1
        ORL  TMOD, #01H           ; T0 as a MODE 1 Timer
yyy:    CLR TCON.5                ; make sure overflow flag is cleared
        CLR TCON.4                ; init T0 off
        MOV  TH0,#0H              ; init high byte of counter to 0
        MOV  TL0,#0H              ; init low byte of counter to 0
xxx:    SETB  TCON.4              ;turn counter T0 on
        JNB    TCON.5,xxx         ; Go back to xxx if no overflow
        DJNZ  R1,yyy              ; Run down counter again unless R1 = 0
```

In each of the three programs in Figure 1, the statement include <sfr51.inc> is specific to the Rigel compiler and allows memory locations to be referred to by their Special Function Register (SFR) names, for example to access Port 1, "P1" can be used instead of its memory location "90H". The cseg statement is also compiler-specific and directs the compiler to place the starting instruction at a specific memory location.

Code 1 of Figure 1 consists of two instructions MOV A, #55H, which places the constant hex number 55 in a register called A. Next, MOV P1, A moves the contents of A out to Port 1 which can then be observed on the LEDs of the student's interface board. Observe that both Register A and Register Port 1 are specific memory locations in the 8051. It is crucial to note that there is no easy way to read and write to specific hardware registers with a standard portable C language. In Section 3, we will see how microcontroller-specific (non-portable) compilers handle this.

In Code 2 of Figure 1, the instruction MOV A, P3 reads the digital data that is at the pins of the microcontroller that connect to the Port 3 latch register. Recall the DIP switches on the student interface boards connect to Port 3 pins. The ANL A, #03CH is code that places zeros in port 3 bits 7,6,1 and 0 which are not accessible to the user. Bits 2 through 5 should reflect the user settings of the DIP switches located on the interface board. MOV P1,A takes the data read from Port 3 and sends it out to PORT 1 where the user can observe and verify the input data on the output data LEDs.

It can be observed that the time delay program shown as Code 3 in Figure 1 contains several references to Special Function Registers (TCON and TMOD which must be specified to control the timer and THO and TLO which determine the delay time.) THO and TL0 are each set to 0 in order that the register pair counts up to a full FFFFH when it flags the system. Register R1 is used as a loop counter to perform the FFFFH count a number of times.

These three tasks are intimately related to the microcontroller hardware. Each involves specific registers in the microcontroller, especially in the time delay code, which contains four Special Function Registers. Since C is a portable language, it cannot reference registers unique to a specific microcontroller. In the next section we will see how the Rigel C compiler makes provisions to include hardware-specific references.

3. Accessing SFRs in the C Language Environment

In the second part of the course we take advantage of the Rigel C compiler which not only compiles SmallC (a subset of Standard C useful for simple controller applications) but also permits handling of 8051 hardware resources. The Rigel compiler has two methods of addressing the specific hardware. The first method uses "in-line assembly language" which permits chunks of assembly language code within a C program and is detailed in Section 3.1.

In the second method, SFRs are accessed by defining them as a special variable type that is added to the collection of existing C types (integer, float, etc.) corresponding to the special function registers of the 8051. This method allows SFRs to be treated as C variables and is highlighted in Section 3.2.

To study these two methods, students develop C programs to control the speed of a stepper motor. Digital output is used to step the motor (energize motor phases), time delays control the time between steps, and digital input allows user selection of motor speed.

3.1 Method of In-Line Assembly Code

The method of inline assembly language permits the inclusion of assembly language code in a C program. The assembly portions of the code can be used to read and write to specific hardware registers. We place the three assembly language tasks from Section 2 into individual functions, where #asm and #endasm denote the beginning and end of assembly code portions (see the code in Figure 2). We will see how parameters are passed to the inline assembly functions.

Figure 2: C-Language Stepper Motor Speed Controller using In-Line Assembly Code (below)

```
/* Stepper Motor Controller using In-Line Assembly */

 #include <sfr51.h>
 #include <cSio51.h>

  char PORT3, ndelays;

   void SendToPort1(char value1)     /* Routine to write byte out to PORT 1  */
{
   #asm
   DPTRAST          ; for parameter passing value 1 to A
   movx a, @dptr    ; parameter value1 to A
   MOV P1, A        ; contents of A out to Port 1
   #endasm
}
   char GetP3(void)                  /* Routine to read byte on PORT 3  */
{
```

```
   #asm
   MOV A, P3;          ; Read PORT3 into Register A
   ANL A, #03CH;       ; Isolate PORT3 bits 2-5
     RET_BA            ; for parameter passing, A to variable
   #endasm
}

   void Delay1(char n1delays)              /* Routine for time delay  */
{
        #asm
        DPTRAST             ;for parameter passing ndelays to A
        movx a, @dptr                ;parameter ndelays to A

        MOV  R1, A          ;A to R1
        ORL  TMOD, #01H              ; T0 as a MODE 1 Timer
yyy:    CLR TCON.5                   ; make sure overflow flag is cleared
        CLR TCON.4          ; init T0 off
        MOV  TH0,#0H                 ;init high byte of counter to 0
        MOV  TL0,#0H                 ;init low byte of counter to 0
xxx:    SETB  TCON.4                 ;turn counter T0 on
        JNB    TCON.5,xxx   ; Go back to xxx if no overflow
        DJNZ  R1,yyy         ;Run down counter again unless R1 = 0
     #endasm
}

   void main(void)
{

   PORT3 = GetP3();                     /*  Read in User Data              */
   PORT3 = PORT3 & 0x3C;                /*  Isolate Bits 2-5               */

    switch (PORT3)                      /*   Examine User Inputs           */
    {
        case 0x20: ndelays = 1;         /* Switch 1 high: 80 msec step     */
        break;
        case 0x10: ndelays = 2;         /* Switch 2 high: 160 msec step    */
        break;
        case 0x08: ndelays= 4;          /* Switch 3 high: 320 msec step    */
        break;
        case 0x04: ndelays= 8;          /* Switch 4 high: 640 msec step    */
        break;

        default: ndelays = 0x01;
    }
        SendToPort1(0x0C);          /*  Step the motor                     */
        Delay1(ndelays);            /*  Call Delay                         */
```

```
    SendToPort1(0x06);      /*  Step the motor      */
    Delay1(ndelays);        /*  Call Delay          */
    SendToPort1(0x03);      /*  Step the motor      */
    Delay1(ndelays);        /*  Call Delay          */
    SendToPort1(0x9);       /*  Step the motor      */
    Delay1(ndelays);        /*  Call Delay          */

}
```

Observe the global character variables PORT3 and ndelays declared at the beginning of the program. Note Character variable types work well since they are 8 bits wide, which is the size of 8051 registers. Three functions follow, the first named SendToPort1, which takes the value sent to it from the main program (for example, the constant 0C hex is sent to it in the line SendToPort1(0x0C)) and in turn sends that out to Port 1. Function GetP3 reads the digital data on the pins of Port 3, modified by logically ANDING with 3C hex and returning the value to char variable PORT3 (note line *PORT3 = GetP3()* in the main program*)*. The third function Delay1 receives a parameter, which is passed to it from the main program, for example in line *Delay1(ndelays).* The parameter turns up in the A register (ACC) via the first two instructions of the function. It is then put in Register R1 that loops the delay "ndelays" number of times.

Examining the Main program we see that GetP3 is called to read Port 3 data into variable PORT3. The powerful C language switch statement is used to perform different actions based on the value of PORT3; in this case it corresponds to the position of four DIP switches on the student's interface board and will set variable ndelays to one of four values which ultimately winds up in Register R1, the delay loop count.

Finally, the four patterns are sent out to Port 1 which drive the stepper motor, and the delay between the patterns is based on variable ndelays. Note a four-phase unipolar stepper motor is used in a two-phase-on scheme so there are 4 step patterns. It can be observed that the four patterns 0C, 06, 03, 09 (shown in hex) will energize pairs of adjacent motor phases.

3.2 Method of Special Function Register Data Types in C

Figure 3 shows the code for the stepper motor speed controller using a method where SFRs can be treated as C variables. The first line of code (#include <sfr51.h>) includes program sfr51.h which contains a list of C variables of type SFR (not a standard C variable type) which defines all the 8051 SFRs (their specific memory locations) as variables of this type. An example of a line in sfr51.h is

```
sfr ACC (0xE0)
```

which makes 8051 Register ACC, located at E0 hex, a C variable called ACC. The 8051 registers can then be read from and written to by dealing with them as C variables.

There is a function called inittimer, which initializes the 8051 timer. The three functions that follow correspond to the functions in the above in-line assembly code method. Note, however, there are no assembly language instructions in the functions; instead, SFRs are handles as C variables. For example, the instruction in function Delay1

```
TH0 = 0x00;   /*  init high byte of counter to 0          */
```

is treating the SFR THO as an 8-bit C variable that can be set equal to 0.

.

Figure 3: C-Language Stepper Motor Speed Controller using SFR Date Types as C Variables Code (below)

```
/* Stepper Motor Controller using Variable type SFRs */

   #include <sfr51.h>
   #include <cSio51.h>

   char PORT3, ndelays;

   void inittimer()          /*  Initialize the timer                    */
{
   TMOD = TMOD  0x01;
   TF0 = 0;                  /*  make sure overflow flag is cleared      */
   TR0 = 0;                  /*  init T0 off                             */
}

   void Delay1(char n1delays)
{
        char nloops;
        for (nloops =n1delays; nloops > 0; nloops = nloops -1)
   {
        TF0 = 0;       /*  make sure overflow flag is cleared      */
        TR0 = 0;       /*  init T0 off                             */
        TH0 = 0x00;    /*  init high byte of counter to 0          */
        TL0 = 0x00;    /*  init low byte of counter to 0           */
        TR0 = 1;       /*  turn counter T0 on                      */

        while (TF0 == 0)
        {
        }
   }

}

   void main(void)
{
```

```
  inittimer();

  while (1)
 {
  PORT3 = P3 & 0x3C;

   switch (PORT3)                    /*   Examine User Inputs          */
    {
        case 0x20: ndelays = 1;      /*  Switch 1 high: 80 msec step   */
        break;
        case 0x10: ndelays = 2;      /* Switch 2 high: 160 msec step   */
        break;
        case 0x08: ndelays= 4;       /* Switch 3 high: 320 msec step   */
        break;
        case 0x04: ndelays= 8;       /* Switch 4 high: 640 msec step   */
        break;

        default: ndelays = 0x01;
    }

        P1 = 0x0C;                   /*  Step the motor               */
        Delay1(ndelays);             /*  Call Delay                   */
        P1 = 0x06;                   /*  Step the motor               */
        Delay1(ndelays);             /*  Call Delay                   */
        P1 = 0x03 ;                  /*  Step the motor               */
        Delay1(ndelays);             /*  Call Delay                   */
        P1 = 0x9;                    /*  Step the motor               */
        Delay1(ndelays);             /*  Call Delay                   */
 }
}
```

The code above performs the same task as the code of the previous section. Port 3 data is read into C variable PORT3 and examined using the switch statement. The switch statement sets variable ndelays equal to one of four values which controls the time between steps.

4. LEGO Robots and NQC

In the last part of the semester, students explore a Robotics application using the NQC (Not Quite C) language. NQC is very similar to C and it targets the Hitachi H8 microcontroller which is used in the LEGO robot. By studying the programs and corresponding robot constructions from David Baum's book "Extreme Mindstorms"[3], students were given an introduction to Robotics.

A nice feature of NQC is the use of descriptive names for routines and variables. The code is extremely user friendly. The programs in Baum's book are well constructed, easy to follow and yet concise. An NQC program from Baum's book is shown in Figure 4.

The program has the robot moving forward (both right and left motors OUT_A and OUT_C) rotating in the same direction. When the touch sensor detects an object in its path (until BUMPER == 0) both motors spin in the opposite direction so the motor backs up in the reverse direction for the time specified by variable *BUMP_BACK_TIME*. After that, the robot moves right by adjusting the motor directions for a time specified by variable *BUMP_SPIN_TIME*. The robot continues forward after that.

Once everything is working properly, students are asked to modify the program, for example keeping count of the number of obstacles and sounding specific tones when a certain number of obstacles has been found.

Figure 4: NQC Code to Drive a Robot Past Obstacles (below)
(The code is found in David Baum's book Extreme Mindstorms[3])

```
/* bump.nqc
 *
 * Drive forward until hitting an obstacle, then back up,
 * turn right a little, and resume.
 */

// motors and sensors
#define LEFT    OUT_A
#define RIGHT   OUT_C
#define BUMPER  SENSOR_1

// timing
#define BUMP_BACK_TIME  60  // 0.6 seconds
#define BUMP_SPIN_TIME  20  // 0.2 seconds

task main()
{
        // setup sensor and start driving
        SetSensor(BUMPER, SENSOR_TOUCH);
        OnFwd(LEFT+RIGHT);

        while(true)
        {
                // wait for bumper to be activated
                until(BUMPER==0);
```

```
            // back up a bit
            OnRev(LEFT+RIGHT);
            Wait(BUMP_BACK_TIME);

            // spin a bit
            Fwd(LEFT);
            Wait(BUMP_SPIN_TIME);

            // resume
            Fwd(RIGHT);
        }
}
```

5. Conclusion

The microcontroller course outlined above provides students with some coding experiences in assembly language and in C language. By highlighting the hardware-specific capabilities of assembly language, students can appreciate that you can't simply substitute C language for assembly. Students observe the advantage of accessing specific microcontroller hardware provided by assembly language. Two methods are explored in which a Standard C language is enhanced with hardware-specific addressing. From their previous work with the C language, students understand the capabilities presented by a high-level language. Advantages of assembly language programming like code size and control of hardware resources can be appreciated along with understanding the disadvantages like tedious programming and knowledge of the particular instruction set. Students can understand and appreciate the different purposes of the languages and how a non-portable C compiler can make the best use of each.

References

1. Reads51 V410 (IDE, SmallC-compatible 8051 compiler, relative assembler, linker/locator, editor, chip simulator, assembly language debugger, monitor, 95 / 98 / 2000 / NT), Rigel Corporation, PO Box 90040, Gainesville, FL 32607
2. Robotics Invention System (RIS) 1.5, LEGO Mindstorms, The LEGO Group.
3. David Baum, Michael Gasperi, Ralph Hempel, Louis Villa, Extreme Mindstorms An Advanced guide to LEGO Mindstorms, Apress, 2002.

Author

Stephanie Goldberg currently teaches in the Electrical Engineering Technology program. She currently teaches Microcontrollers as well as Digital Systems 1 and Analog Circuits. She received her Ph.D in Electrical and Computer Engineering at the State University of New York at Buffalo.