AVR Microcontroller

Microprocessor Course

Chapter 7

AVR PROGRAMMING IN C

Esfand 1397 (Version 1.2)

Why program the AVR in C?

Compilers produce hex files that we download into the Flash of the micro-controller. The size of the hex file produced by the compiler is one of the main concerns of microcontroller programmers because microcontrollers have limited on-chip Flash. For example, the Flash space for the Atmega16 is 16K bytes.

How does the choice of programming language affect the compiled program size? While Assembly language produces a hex file that is much smaller than C, programming in Assembly language is often tedious and time consuming. On the other hand, C programming is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

The following are some of the major reasons for writing programs in C instead of Assembly:

1. It is easier and less time consuming to write in C than in Assembly.
2. C is easier to modify and update.
3. You can use code available in function libraries.
4. C code is portable to other microcontrollers with little or no modification.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

Several third-party companies develop C compilers for the AVR microcontroller. Our goal is not to recommend one over another, but to provide you with the fundamentals of C programming for the AVR. You can use the compiler of your choice for the chapter examples and programs. For this book we have chosen AVR GCC compiler to integrate with AVR Studio. At the time of the writing of this book AVR GCC and AVR Studio are available as a free download from the Web. See http://www.MicroDigitalEd.com for tutorials on AVR Studio and the AVR GCC compiler.

C programming for the AVR is the main topic of this chapter. In Section

7.1 data types, and time delays

7.2 I/O programming

7.3 The logic operations AND, OR, XOR, inverter, and shift

7.4 ASCII and BCD conversions and checksums

7.5, data serialization for the AVR

7.6, memory allocation in C .

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

In this section we first discuss C data types for the AVR and then provide code for time delay functions. C data types for the AVR C One of the goals of AVR programmers is to create smaller hex files, so it is worthwhile to re-examine C data types.

In other words, a good understanding of C data types for the AVR can help programmers to create smaller hex files. In this section we focus on the specific C data types that are most common and widely used in AVR C compilers.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

Table 7-1 shows data types and sizes, but these may vary from one compiler to another.

Table 7-1: Some Data Types Widely Used by C Compilers

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
char	8-bit	–128 to +127
unsigned int	16-bit	0 to 65,535
int	16-bit	–32,768 to +32,767
unsigned long	32-bit	0 to 4,294,967,295
long	32-bit	–2,147,483,648 to +2,147,483,648
float	32-bit	±1.175e-38 to ±3.402e38
double	32-bit	±1.175e-38 to ±3.402e38

Unsigned char

Because the AVR is an 8-bit microcontroller, the character data type is the most natural choice for many applications. The unsigned char is an 8-bit data type that takes a value in the range of 0-255 (00--FFH). It is one of the most widely used data types for the AVR. In many situations, such as setting a counter value, where there is no need for signed data, we should use the unsigned char instead of the signed char.

In declaring variables, we must pay careful attention to the size of the data and try to use unsigned char instead of int if possible. Because the AVR microcontroller has a limited number of registers and data RAM locations, using int in place of char can lead to the need for more memory space.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

7.1 : DATA TYPES AND TIME DELAYS IN C

Remember that C compilers use the signed char as the default unless we put the keyword unsigned in front of the char.

Example 7-1

Write an AVR C program to send values 00-FF to Port B.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2
3   int main(void)
4   {
5           unsigned char z;
6           DDRB = 0xFF
7           for(z=0; z <= 255; z++)
8           PORTB = z;                     //PORTB is output
9           return 0;
10  }
11  //Notice that the program never exits the for loop because if you
12  //increment an unsigned char variable when it is OxFF, it will
13  //become zero.
```

Example 7-2

Write an AVR C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to Port B.

Solution:

```
1  #include <avr/io.h>                    //standard AVR header
2  int main(void)                         //the code starts from here
3  {
4          unsigned char myList[]= "012345ABCD";
5          unsigned char z;
6          DDRB = 0xFF;                   //PORTB is output
7          for(z=0; z<10; z++)            //repeat 10 times and increment z
8          PORTB = myList[z];             //send the character to PORTB
9          while(1);                      //needed if running on a trainer
10         return 0;
11 }
```

Example 7-3

Write an AVR C program to toggle all the bits of PortB 200 times.

Solution:

```
1  //toggle PB 200 times
2  #include <avr/io.h>                  //standard AVR header
3  int main(void)                       //the code starts from here
4  {
5          DDRB = 0xFF;                 //PORTB is output
6          PORTB = 0xAA;                //PORTB is 10101010
7          unsigned char z;
8          for(z=0; z < 200; z++)       //run the next line 200 times
9              PORTB = ~ PORTB;         //toggle PORTB
10         while(1);                    //stay here forever
11         return 0;
12 }
```

signed char

The signed char is an 8-bit data type that uses the most significant bit (D7 of D7-D0) to represent the - or + value. As a result, we have only 7 bits for the magnitude of the signed number, giving us values from -128 to +127.

In situations where + and - are needed to represent a given quantity such as temperature, the use of the signed char data type is necessary (see Example 7-4).

Again, notice that if we do not use the keyword unsigned, the default is the signed value. For that reason we should stick with the unsigned char unless the data needs to be represented as signed numbers.

Example 7-4

Write an AVR C program to send values of -4 to +4 to Port B.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4       char mynurr[1 = { -4,-3, -21-1, 0,+1, +2, +3, +4} ;
5       unsigned char z;
6       DDRB = 0xFF;                       //PORTB is output
7       for (z=0; z<=8; z++)
8           PORTS = mynum[z] ;
9       while(1);                          //stay here forever
10      return 0;
11  }
```

Run the above program on your simulator to see how PORTS displays values of FCH, FDH, FEH , FFH, 00H, 01H, 02H, 03H, and 04H (the hex values for -4, -3, -2, -1, 0, 1, etc.).

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

Unsigned int

The unsigned int is a 16-bit data type that takes a value in the range of 0 to 65,535 (0000-FFFFH). In the AVR, unsigned int is used to define 16-bit variables such as memory addresses. It is also used to set counter values of more than 256. Because the AVR is an 8-bit microcontroller and the int data type takes two bytes of RAM, we must not use the int data type unless we have to.

Because registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in

- larger hex files,
- slower execution of program, and
- more memory usage.

For AVR programming, however, do not use signed int in places where unsigned char will do the job. Of course, the compiler will not generate an error for this misuse, but the overhead in hex file size will be noticeable.

Signed int

Signed int is a 16-bit data type that uses the most significant bit (D15 of D15-D0) to represent the - or + value. As a result, we have only 15 bits for the magnitude of the number, or values from -32,768 to +32,767.

Other data types

The unsigned int is limited to values 0-65,535 (0000-FFFFH). The AVR C compiler supports long data types, if we want values greater than 16-bit. Also, to deal with fractional numbers, most AVR C compilers support float and double data types.

Example 7-5

Write an AVR C program to toggle all bits of PortB 50,000 times.

Solution:

```
1   #include <avr/io.h>                 //standard AVR header
2   int main(void)
3   {
4       unsigned int z;
5       DDRB = 0xFF;                    //PORTB is output
6       for(z=0; z<50000; z++)
7       {
8           PORTB = 0x55;
9           PORTB = 0xAA;
10      }
11      while (1);                      //stay here forever
12      return 0;
13  }
```

Run the above program on your simulator to see how Port B toggles continuously. Notice that the maximum value for unsigned int is 65,535.

Example 7-6

Write an AVR C program to toggle all bits of Port B 100,000 times.

Solution:

```
1   //toggle PE 100,00 times
2   #include <avr/io.h>                    //standard AVR header
3   int main(void)
4   {
5       unsigned long z;                   //long is used because it should
6                                          //store more than 65535.
7       DDRB = 0xFF;                       //PORTB is output
8       for(z=0; z<100000; z++)
9       {
10          PORTS = 0x55;
11          PORTS = 0xAA;
12      }
13      while(1);                          //stay here forever
14      return 0;
15  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.1 : DATA TYPES AND TIME DELAYS IN C

Time delay

There are three ways to create a time delay in AVR C

1. Using a simple for loop
2. Using predefined C functions
3. Using AVR timers

In creating a time delay using a for loop, we must be mindful of two factors that can affect the accuracy of the delay:

1. The crystal frequency connected to the XTAL1-XTAL2 input pins is the most important factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.
2. The second factor that affects the time delay is the compiler used to compile the C program. When we program in Assembly language, we can control the exact instructions and their sequences used in the delay subroutine. In the case of C programs, it is the C compiler that converts the C statements and functions to Assembly language instructions. As a result, different compilers pro-duce different code.

7.1 : DATA TYPES AND TIME DELAYS IN C

For the above reasons, when we use a loop to write time delays for C, we must use the oscilloscope to measure the exact duration. Notice that most compilers do some code optimization before generating a .hex file. In these compilers, you have to set the level of optimization to zero (none).

Example 7-7

Write an AVR C program to toggle all the bits of Port B continuously with a 100 ms delay. Assume that the system is ATmega 32 with XTAL = 8 MHz.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   void delay100ms(void)
3   {
4           unsigned int i;
5           for(i=0; i<42150; i++);        //try different numbers on your
6   }                                      //compiler and examine the result
7   int main(void)
8   {
9           DDRB = 0xFF;                   //PORTB is output
10          while (1)
11          {
12              PORTB = 0xAA;
13              delay100ms();
14              PORTB = 0x55;
15              delay100ms();
16          }
17          return 0;
18  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

7.1 : DATA TYPES AND TIME DELAYS IN C

Mother way of generating time delay is to use predefined functions such as _delay_ms() and _delay_us() defined in delay.h in WinAVR or delay_ms() and delay_us() defined in delay.h in CodeVision.

The only drawback of using these functions is the portability problem. Because different compilers do not use the same name for delay functions, you have to change every place in which the delay functions are used, if you want to compile your program on another compiler.

The use of the AVR timer to create time delays will be discussed in Chapter 9.

Example 7-8

Write an AVR C program to toggle all the pins of Port C continuously with a 10 ms delay. Use a predefined delay function in Win AVR.

Solution:

```
1   #include <util/delay.h>                    //delay loop functions
2   #include <avr/io.h>                        //standard AVR header
3   int main(void)
4   {
5       void delay_ms(int d)                   //delay in d microseconds
6       {
7           _delay_ms(d);
8       }
9       DDRB = 0xFF;                           //PORTB is output
10      while (1)
11      {
12          PORTB = 0xFF;
13          delay_ms(10);
14          PORTB = 0x55;
15          delay_ms(10);
16      }
17      return 0;
18  }
```

Byte size I/O

To access a PORT register as a byte, we use the PORTx label where x indicates the name of the port. We access the data direction registers in the same way, using DDRx to indicate the data direction of port x. To access a PIN register as a byte, we use the PINx label where x indicates the name of the port.

Example 7-9

LEDs are connected to pins of Port B. Write an AVR C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

```
1   #include <avr/io.h>          //standard AVR header
2   int main(void)
3   {
4       DDRB = 0xFF;             //Port B is output
5       while (1)
6       {
7           PORTB = PORTB + 1;
8       }
9       return 0;
10  }
```

Example 7-10

Write an AVR C program to get a byte of data from Port B, and then send it to Port C.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4       unsigned char temp;
5       DDRB = 0x00;                       //Port B is input
6       DDRC = 0xFF;                       //Port C is output
7       while(1)
8       {
9           temp = PINB;
10          PORTC = temp;
11      }
12      return 0;
13
```

Example 7-11

Write an AVR C program to get a byte of data from Port C. If it is less than 100, send it to Port B; otherwise, send it to Port D.

Solution:

```
1  #include <avr/io.h>                  //standard AVR header
2  int main(void)
3  {
4          DDRC = 0;                    //Port C is input
5          DDRB = 0xFF;                 //Port B is output
6          DDRD = 0xFF;                 //Port D is output
7          unsigned char temp;
8          while (1)
9          {
10             temp = PINC;             //read from PINB
11             if ( temp < 100 )
12                 PORTB = temp;
13             else
14                 PORTD = temp;
15         }
16         return 0;
17 }
```

Bit size I/O

The I/O ports of ATmega32 are bit-accessible. But some AVR C compilers do not support this feature, and the others do not have a standard way of using it. For example, the following line of code can be used in CodeVision to set the first pin of Port B to one:

```
PORTE.0 = 1;
```

but it cannot be used in other compilers such as WinAVR. To write portable code that can be compiled on different compilers, we must use AND and OR bit-wise operations to access a single bit of a given register.

So, you can access a single bit without disturbing the rest of the byte.

Bit-wise operators in C

While every C programmer is familiar with the logical operators AND (&&), OR (||), and NOT (!), many C programmers are less familiar with the bit-wise operators AND (&) OR (|), EX-OR (^), inverter (~), shift right (>>), and shift left (<<). These bit-wise operators are widely used in software engineering for embedded systems and control; consequently, their understanding and mastery are critical in microcontroller-based system design and interfacing.

Table 7-2: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

7.3 : LOGIC OPERATIONS IN C

The following shows some examples using the C bit-wise operators:

1. 0x35 & 0x0F = 0x05 /* ANDing */
2. 0x04 | 0x68 = 0x6C /* ORing */
3. 0x54 ^ 0x78 = 0x2C /* XORing */
4. ~0x55 = 0xAA /* Inverting 55H */

Examples 7-12 through 7-20 show how the bit-wise operators are used in C. Run these programs on your simulator and examine the results.

Example 7-12

Run the following program on your simulator and examine the results.

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4       DDRB = 0xFF;                       //make Port B output
5       DDRC = 0xFF;                       //make Port C output
6       DDRD = 0xFF;                       //make Port D output
7       PORTB = 0x35 & 0x0F;               //ANDing
8       PORTC = 0x04 | 0x68;               //ORing
9       PORTD = 0x54 ^ 0x78;               //XORing
10      PORTB = ~0x55;                     //inverting
11      while (1);
12      return 0;
13  }
```

Example 7-13

Write an AVR C program to toggle only bit 4 of Port B continuously without disturbing the rest of the pins of Port B.

Solution:

```
1  #include <avr/io.h>                    //standard AVR header
2  int main (void)
3  {
4          DDRB = 0xFF;
5          while(1)                       //PORTB is output
6          PORTB = PORTB | 0b00010000; //set bit 4 (5th bit) of PORTS
7          PORTB = PORTB & 0b11101111; //clear bit 4 (5th bit) of PORTB
8          return 0;
9  }
```

Example 7-14

Write an AVR C program to monitor bit 5 of port C. If it is HIGH, send 55H to Port B; otherwise, send AAH to Port B.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main (void)
3   {
4           DDRB = 0xFF;                   //PORTB is output
5           DDRC = 0x00;                   //PORTC is input
6           DDRD = 0xFF;                   //PORTD is output
7           while (1)
8           {
9           if (PINC & 0b00100000)         //check bit 5 (6th bit) of PINC
10              PORTB = 0x55;
11          else
12              PORTB = 0xAA;
13          }
14          return 0;
15  }
```

Example 7-15

A door sensor is connected to bit 1 of Port B, and an LED is connected to bit 7 of Port C. Write an AVR C program to monitor the door sensor and, when it opens, turn on the LED.

Solution:

```
1   #include <avr/io.h>                          //standard AVR header
2   int main(void)
3   {
4       DDRB = DDRB & 0b11111101;                //pin 1 of Port B is input
5       DDRC = DDRC | 0b10000000;                //pin 7 of Port C is output
6       while (1)
7       {
8           if (PINB & 0b00000010)               //check pin 1 (2nd pin) of PINB
9               PORTC = PORTC | 0b10000000;      //set pin 7 (8th pin) of PORTC
10          else
11              PORTC = PORTC & 0b01111111;      //clear pin 7 (8th pin) of PORTC
12      }
13      return 0;
14  }
```

Example 7-16

The data pins of an LCD are connected to Port B. The information is latched into the LCD whenever its Enable pin goes from HIGH to LOW. The enable pin is connected to pin 5 of Port C (6th pin). Write a C program to send "The Earth is but One Country" to this LCD.

Solution:

```
1  #include <avr/io.h>                        //standard AVR header
2  int main (void)
3  {
4          unsigned char message() = "The Earth is but One Country";
5          unsigned char z;
6          DDRB = 0xFF;                       //Port B is output
7          DDRC = DDRC | 0b00100000;          //pin 5 of Port C is output
8          for (z=0;z<28;z++)
9          {
10             PORTB = message[z];
11             PORTC = PORTC | 0b00100000; //pin LCD EN of Port C is 1
12             PORTC = PORTC & 0b11011111; //pin LCD EN of Port C is 0
13         }
14         while(1);
15         return 0;
16 }
```

Example 7-17

Write an AVR C program to read pins 1 and 0 of Port B and issue an ASCII character to Port D according to the following table:

pin1	pin0	
0	0	send '0' to Port D (notice ASCII '0' is 0x30)
0	1	send '1' to Port D
1	0	send '2' to Port D
1	1	send '3' to Port D

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4       unsigned char z;
5       DDRB = 0;                          //make Port B an input
6       DDRD = 0xFF;                       //make Port D an output
7       while (1)                          //repeat forever
8       {
9           z = PINB;                      //read PORTB
10          z = z & 0b00000011;            //mask the unused bits
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

7.3 : LOGIC OPERATIONS IN C

```
11          switch(z)                       //make decision
12          {
13              case(0):
14              {
15                  PORTD =  '0';           //issue ASCII 0
16                  break;
17              }
18              case(1):
19              {
20                  PORTD = '1';            //issue ASCII 1
21                  break;
22              }
23              case(2):
24              {
25                  PORTD = '2';            //issue ASCII 2
26                  break;
27              }
28              case(3):
29              {
30                  PORTD = '3';            //issue ASCII 3
31                  break;
32              }
33          }
34      }
35  return 0;
36  }
```

Example 7-18

Write an AVR C program to monitor bit 7 of Port B. If it is 1, make bit 4 of Port B input; otherwise, change pin 4 of Port B to output.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4           DDRB = DDRB & 0b01111111;      //bit 7 of Port B is input
5           while (1)
6           {
7               if(PINB & 10000000)
8                   DDRB = DDRB & 0b11101111;    //bit 4 of Port B is input
9               else
10                  DDRB = DDRB | 0b00010000;    //bit 4 of Port B is output
11          }
12          return 0;
13  }
```

7.3 : LOGIC OPERATIONS IN C

Example 7-19

Write an AVR C program to get the status of bit 5 of Port B and send it to bit 7 of port C continuously.

Solution:

```
1   #include <avr/io.h>                          //standard AVR header
2   int main(void)
3   {
4       DDRB = DDRB & 0b11011111;                //bit 5 of Port B is input
5       DDRC = DDRC | 0b10000000;                //bit 7 of Port C is output
6       while (1)
7       {
8           if(PINB & 0b00100000)
9               PORTC = PORTC | 0b10000000;      //set bit 7 of Port C to 1
10          else
11              PORTC = PORTC & 0b01111111;      //clear bit 7 of Port C to 0
12      }
13      return 0;
14  }
```

Example 7-20

Write an AVR C program to toggle all the pins of Port B continuously.
(a) Use the inverting operator. (b) Use the EX-OR operator.

Solution:

```
1   (a)
2   #include <avr/io.h>                    //standard AVR header
3   int main(void)
4   {
5       DDRB = 0xFF;                       //Port B is output
6       PORTB = 0xAA;
7       while (1)
8           PORTB = ~ PORTB;               //toggle PORTB
9       return 0;
10  }
11
12  (b)
13  #include <avr/io.h>                    //standard AVR header
14  int main(void)
15  {
16      DDRB = 0xFF;                       //Port B is output
17      PORTB = 0xAA;
18      while(1)
19          PORTB = PORTB ^ 0xFF;
20      return 0;
21  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.3 : LOGIC OPERATIONS IN C

```
4:                      {
+00000049:    EF8F      SER     R24             Set Register
+0000004A:    BB87      OUT     0x17,R24        Out to I/O location
6:                      PORTB = 0xAA;
+0000004B:    EA8A      LDI     R24,0xAA        Load immediate
+0000004C:    BB88      OUT     0x18,R24        Out to I/O location
9:                      PORTB =~ PORTB ;
+0000004D:    B388      IN      R24,0x18        In from I/O location
+0000004E:    9580      COM     R24             One's complement
+0000004F:    BB88      OUT     0x18,R24        Out to I/O location
+00000050:    CFFC      RJMP    PC-0x0003       Relative jump
+00000051:    94F8      CLI                     Global Interrupt Disab
+00000052:    CFFF      RJMP    PC-0x0000       Relative jump
```

Disassembly of Example 7-20 Part a

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.3 : LOGIC OPERATIONS IN C

```
4:                  {
+00000049:   EF8F       SER     R24             Set Register
+0000004A:   BB87       OUT     0x17,R24        Out to I/O location
6:                  PORTB = 0xAA;
+0000004B:   EA8A       LDI     R24,0xAA        Load immediate
+0000004C:   BB88       OUT     0x18,R24        Out to I/O location
9:                  PORTB = PORTB ^ 0xFF ;
+0000004D:   B388       IN      R24,0x18        In from I/O location
+0000004E:   9580       COM     R24             One's complement
+0000004F:   BB88       OUT     0x18,R24        Out to I/O location
+00000050:   CFFC       RJMP    PC-0x0003       Relative jump
+00000051:   94F8       CLI                     Global Interrupt Disab
+00000052:   CFFF       RJMP    PC-0x0000       Relative jump
```

Disassembly of Example 7-20 Part b

Compound assignment operator in C

To reduce coding (typing) we can use compound statements for bit-wise operators in C.

Table 7-3: Compound Assignment Operator in C

Operation	Abbreviated Expression	Equal C Expression
And assignment	a &= b	a = a & b
OR assignment	a |= b	a = a | b

Example 7-21

Using bitwise compound assignment operators

(a) Rewrite Example 7-18 (b) Rewrite Example 7-19

Solution:

```
1   (a)
2   #include <avr/io.h>                    //standard AVR header
3   int main (void)
4   {
5       DDRB &= DDRB & 0b11011111;
6       while (1)                          //bit 5 of Port B is input
7       {
8           if(PINB & 0b00100000)
9               DDRB &=  0b11101111;       //bit 4 of Port B is input
10          else
11              DDRB |= 0b00010000;        //bit 4 of Port B is output
12      }
13      return 0;
14  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.3 : LOGIC OPERATIONS IN C

```
(b)
#include <avr/io.h>                          //standard AVR header
int main (void)
{
        DDRB &= 0b11011111;                  //bit 5 of Port B is input
        DDRC |= 0b10000000;                  //bit 7 of Port C is output
        while(1)
        {
            if(PINB & 0b00100000)
                PORTC |= 0b10000000;         //set bit 7 of Port C to 1
            else
                PORTC &= 0b01111111;         //clear bit 7 of Port C to 0
        }
        return 0;
}
```

Bit-wise shift operation in C

There are two bit-wise operator in C

Table 7-4: Bit-wise Shift Operators for C

Operation	Symbol	Format of Shift Operation
Shift right	>>	data >> number of bits to be shifted right
Shift left	<<	data << number of bits to be shifted left

The following shows some examples of shift operators in C:

```
1. 0b00010000 >> 3 =  0b00000010    /* shifting right 3 times */
2. 0b00010000 << 3 =  0b10000000    /* shifting left 3 times */
3. 1 << 3  =  0b00001000            /* shifting left 3 times */
```

Bit-wise shift operation and bit manipulation

Reexamine the last 10 examples. To do bit-wise I/O operation in C, we need numbers like 0b00100000 in which there are seven zeroes and one one. Only the position of the one varies in different programs. To leave the generation of ones and zeros to the compiler and improve the code clarity, we use shift operations. For example, instead of writing “0b00100000" we can write “0b00000001 << 5" or we can write simply "1<<5".

Sometimes we need numbers like 0b11101111. To generate such a number, we do the shifting first and then invert it. For example, to generate 0b11101111 we can write ~ (1<<5).

Example 7-22

Write code to generate the following numbers:

(a) A number that has only a one in position D7
(b) A number that has only a one in position D2
(c) A number that has only a one in position D4
(d) A number that has only a zero in position D5
(e) A number that has only a zero in position D3
(f) A number that has only a zero in position D1

Solution:

(a) `(1<<7)`
(b) `(1<<2)`
(c) `(1<<4)`
(d) `~(1<<5)`
(e) `~(1<<3)`
(f) `~(1<<1)`

Example 7-23

Write an AVR C program to monitor bit 7 of Port B. If it is 1, make bit 4 of Port B input; else, change pin 4 of Port B to output.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main(void)
3   {
4       DDRB = DDRB & ~(1<<7);             //bit 7 of Port B is input
5       while (1)
6       {
7           if (PING & (1<<7))
8               DDRB = DDRB & ~(1<<4);     //bit 4 of Port B is input
9           else
10              DDRB = DDRB | (1<<4);      //bit 4 of Port B is output
11      }
12      return 0;
13  }
```

Example 7-24

Write an AVR C program to get the status of bit 5 of Port B and send it to bit 7 of port C continuously.

Solution:

```
1   #include <avr/io.h>                          //standard AVR header
2   int main(void)
3   {
4           DDRB = DDRB & ~(1<<5);               //bit 5 of Port B is input
5           DDRC = DDRC | (1<<7);                //bit 7 of Port C is output
6           while (1)
7           {
8               if(PINB & (1<<5))
9                   PORTC = PORTC | (1<<7); //set bit 7 of Port C to 1
10              else
11                  PORTC = PORTC & ~(1<<7); //clear bit 7 of Port C to 0
12          }
13          return 0;
14  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.3 : LOGIC OPERATIONS IN C

As we mentioned before, bit-wise shift operation can be used to increase code clarity.

Example 7-25

A door sensor is connected to the port B pin 1, and an LED is connected to port C pin 7. Write an AVR C program to monitor the door sensor and, when it opens, turn on the LED.

Solution:

```
1  #include <avr/io.h>                          //standard AVR header
2  #define     LED     7
3  #define     SENSOR  1
4  int main(void)
5  {
6      DDRB = DDRB & ~(1<<SENSOR);              //SENSOR pin is input
7      DDRC = DDRC | (1<<LED);                  //LED pin is output
8      while (1)
9      {
10         if (PINB & (1<<SENSOR))              //check SENSOR pin of PINB
11             PORTC = PORTC | (1<<LED);        //set LED pin of Port C
12         else
13             PORTC = PORTC & ~(1<<LED);       //clear LED pin of Port C
14     }
15     return 0;
16 }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.3 : LOGIC OPERATIONS IN C

Notice that to generate more complicated numbers, we can OR two simpler numbers. For example, to generate a number that has a one in position D7 and another one in position D4, we can OR a number that has only a one in position D7 with a number that has only a one in position D4. So we can simply write (1<<7)|(1<<4).

ASCII numbers

On ASCII keyboards, when the "0" key is activated, "0011 0000" (30H) is provided to the computer. Similarly, 31 H (0011 0001) is provided for the "1" key, and so on.

Table 7-5: ASCII Code for Digits 0–9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Packed BCD to ASCII conversion

The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off. This data is provided in packed BCD. To convert packed BCD to ASCII, you must first convert it to unpacked BCD. Then the unpacked BCD is tagged with 011 0000 (30H). The following demonstrates converting from packed BCD to ASCII. See also Example 7-26.

Packed BCD	Unpacked BCD	ASCII
0x29	0x02, 0x09	0x32, 0x39
00101001	00000010, 00001001	00110010, 00111001

ASCII to packed BCD conversion

To convert ASCII to packed BCD, you first convert it to unpacked BCD (to get rid of the 3), and then combine the numbers to make packed BCD. For example, 4 and 7 on the keyboard give 34H and 37H, respectively. The goal is to produce 47H or "0100 0111", which is packed BCD.

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

Example 7-26

Write an AVR C program to convert packed BCD 0x29 to ASCII and display the bytes on PORTB and PORTC.

Solution:

```
1  #include <avr/io.h>                    //standard AVR header
2  int main (void)
3  {
4          unsigned char x, y;
5          unsigned char mybyte = 0x29;
6          DDRB = DDRC = 0xFF;            //make Ports B and C output
7          x = mybyte & 0x0F;             //mask upper 4 bits
8          PORTB = x | 0x30;              //make it ASCII
9          y = mybyte & 0xF0;             //mask lower 4 bits
10         y = y >> 4;                        //shift it to lower 4 bits
11         PORTC = y | 0x30;              //make it ASCII
12
13         return 0;
14 }
```

Example 7-27

Write an AVR C program to convert ASCII digits of '4' and '7' to packed BCD and display them on PORTB.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2   int main (void)
3   {
4           unsigned char bcdbyte;
5           unsigned char w = '4';
6           unsigned char z = '7';
7           DDRB = 0xFF;                   //make Port B an output
8           w = w & 0x0F;                  //mask 3
9           w = w << 4;                    //shift left to make upper BCD digit
10          z = z & 0x0F;                  //mask 3
11          bcdbyte = w | z;               //combine to make packed BCD
12          PORTB = bcdbyte;
13
14          return 0;
15  }
```

Checksum byte in ROM

To ensure data integrity, the checksum process uses what is called a checksum byte. The checksum byte is an extra byte that is tagged to the end of a series of bytes of data. To calculate the checksum byte of a series of bytes of data, the following steps can be taken:

1. Add the bytes together and drop the carries.
2. Take the 2's complement of the total sum.

This is the checksum byte, which becomes the last byte of the series.

To perform the checksum operation, add all the bytes, including the check-sum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted).

Example 7-28

Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H.
(a) Find the checksum byte,
(b) perform the checksum operation to ensure data integrity, and
(c) if the second byte, 62H, has been changed to 22H, show how check-sum detects the error.

Solution:

(a) Find the checksum byte.

```
  25H
+ 62H
+ 3FH
+ 52H
------------
1 18H (dropping carry of 1 and taking 2's complement, we get E8H)
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.4 : DATA CONVERSION PROGRAMS IN C

(b) Perform the checksum operation to ensure data integrity.

```
  25H
+ 62H
+ 3FH
+ 52H
+ E8H
------------
2 00H (dropping carry we get 00,which means data is not corrupted)
```

(c) If the second byte, 62H has been changed to 22H show how checksum detects the error.

```
  25H
+ 22H
+ 3FH
+ 52H
+ E8H
------------
2 C0H (dropping carry we get C0H,which means data is corrupted)
```

Example 7-29

Write an AVR C program to calculate the checksum byte for the data given in Example 7-28.

```
1   #include <avr/io.h>                          //standard AVR header
2   int main(void)
3   {
4       unsigned char   mydata1l = (0x25,0x62,0x3F,0x52);
5       unsigned char   sum = 0;
6       unsigned char   x;
7       unsigned char   chksumbyte;
8
9       DDRA    =   0xFF;                        //make  Port    A   output
10      DDRB    =   0xFF;                        //make  Port    B   output
11      DDRC    =   0xFF;                        //make  Port    C   output
12
13      for(x=0; x<4; x++) f
14      {
15          PORTA = mydata[x] ;                  //issue each byte to PORTA
16          sum = sum + mydata[ x] ;             //add them together
17          PORTB = sum;                         //issue the sum to PORTB
18
19      }
20      chksumbyte = -sum + 1;                   //make 2's complement (invert +1)
21      PORTC = chksumbyte;                      //show the checksum byte
22      return 0;
23  }
```

Example 7-30

Write a C program to perform step (b) of Example 7-28. If the data is good, send ASCII character `G.' to PORTD. Otherwise, send '13' to PORTD.

Solution:

```
1   #include <avr/io.h>                          //standard AVR header
2   int main(void)
3   {
4       unsigned    char    mydata[]    = {0x25,0x62,0x3F,0x52,0xE8};
5       unsigned    char    chksum =    0;
6       unsigned    char    x;
7       DDRD = 0xFF;                             //make Port D an output
8       for(x=0;x<5;x++)
9           chksum = chksum + mydata[x];         //add them together
10      if (chksum == 0)
11          PORTD = 'G';
12      else
13      PORTD = 'B';
14      return 0;
15  }
```

Change one or two values in the mydata array and simulate the program to see the results.

Binary (hex) to decimal and ASCII conversion in C

The printf function is part of the standard I/O library in C and can do many things including converting data from binary (hex) to decimal, or vice versa. But printf takes a lot of memory space and increases your hex file substantially. For this reason, in systems based on the AVR microcontroller, it is better to know how to write our own conversion function instead of using printf.

One of the most widely used conversions is binary to decimal conversion. In devices such as ADCs (Analog-to-Digital Converters), the data is provided to the microcontroller in binary. In some RTCs, the time and dates are also provided in binary.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.4 : DATA CONVERSION PROGRAMS IN C

In order to display binary data, we need to convert it to decimal and then to ASCII. Because the hexadecimal format is a convenient way of representing binary data, we refer to the binary data as hex. The binary data 00-FFH converted to decimal will give us 000 to 255. One way to do that is to divide it by 10 and keep the remainder, as was shown in Chapters 5 and 6. For example, 11111101 or FDH is 253 in decimal. The following is one version of an algorithm for conversion of hex (binary) to decimal:

Hex	Quotient	Remainder
FD/0A	19	3 (low digit) LSD
19/0A	2	5 (middle digit)
		2 (high digit) MSD

Example 7-31

Write an AVR C program to convert 11111101 (FD hex) to decimal and display the dig-its on PORTB, PORTC, and PORTD.

Solution:

```
1   #include <avr/io.h>                  //standard AVR header
2   int main(void)
3   {
4       unsigned char x, binbyte, d1, d2, d3;
5       DDRB = DDRC = DDRD =0xFF;        //Ports B, C, and D output
6
7       binbyte = 0xFD;                  //binary (hex) byte
8       x = binbyte / 10;                //divide by 10
9       d1 = binbyte % 10;               //find remainder (LSD)
10      d2 = x % 10;                     //middle digit
11      d3 = x / 10;                     //most-significant digit (MSD)
12      PORTB = d1;
13      PORTC = d2;
14      PORTD = d3;
15
16      return 0;
17  }
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.4 : DATA CONVERSION PROGRAMS IN C

Many compilers have some predefined functions to convert data types. In Table 7-6 you can see some of them. To use these functions, the stdlib.h file should be included. Notice that these functions may vary in different compilers.

Table 7-6: Data Type Conversion Functions in C

Function signature	Description of functions
int atoi(char *str)	Converts the string str to integer.
long atol(char *str)	Converts the string str to long.
void itoa(int n, char *str)	Converts the integer n to characters in string str.
void ltoa(int n, char *str)	Converts the long n to characters in string str.
float atof(char *str)	Converts the characters from string str to float.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.5 : DATA SERIALIZATION IN C

Serializing data is a way of sending a byte of data one bit at a time through a single pin of a microcontroller. There are two ways to transfer a byte of data serially:

1. Using the serial port. In using the serial port, the programmer has very limited control over the sequence of data transfer. The details of serial port data transfer are discussed in Chapter 11.
2. The second method of serializing data is to transfer data one bit a time and control the sequence of data and spaces between them. In many new generations of devices such as LCD, ADC, and EEPROM, the serial versions are becoming popular because they take up less space on a printed circuit board. Although we can use standards such as PC, SPI, and CAN, not all devices support such standards. For this reason we need to be familiar with data serialization using the C language.

7.5 : DATA SERIALIZATION IN C

Example 7-32

Write an AVR C program to send out the value 44H serially one bit at a time via PORTC, pin 3. The LSB should go out first.

Solution:

```
1  #include <avr/io.h>
2  #define serPin 3
3  int main (void)
4  {
5      unsigned char conbyte = 0x44;
6      unsigned char regALSB;
7      unsigned char x;
8      regALSB = conbyte;
9      DDRC |= (1<<serPin);
10
11     for(x=0;x<8;x++)
12     {
13         if(regALSB & 0x01)
14             PORTC |= (1<<serPin);
15         else
16             PORTC &= ~(1<<serPin);
17         regALSB = regALSB >> 1;
18     }
19 return 0;
20 }
21
```

Example 7-33

Write an AVR C program to send out the value 44H serially one bit at a time via PORTC, pin 3. The MSB should go out first.

Solution:

```
1  #include <avr/io.h>
2  #define serPin 3
3  int main(void)
4  {
5      unsigned char conbyte = 0x44;
6      unsigned char regALSB;
7      unsigned char x;
8      regALSB = conbyte;
9      DDRC 1= (1<<serPin);
10     for(x=0;x<8;x++)
11     {
12         if (regALSB & 0x80)
13             PORTC |= (1<<serPin);
14         else
15             PORTC &= ~(1<<serPin);
16         regALSB = regALSB << 1;
17     }
18     return 0;
19 }
```

Example 7-34

Write an AVR C program to bring in a byte of data serially one bit at a time via PORTC, pin 3. The LSB should come in first.

Solution:

```
1  //Bringing in data via PC3 (SHIFTING RIGHT)
2  #include <avr/io.h>                          //standard AVR header
3  #define serPin 3
4  int main (void)
5  {
6      unsigned char x;
7      unsigned char REGA=0;
8      DDRC &= -(1<<serPin);                    //serPin as input
9      for(x=0; x<8; x++)                       //repeat for each bit of REGA
10     {
11     REGA = REGA >> 1;                        //shift REGA to right one bit
12     REGA |= (PINC & (1<<serPin)) << (7-serPin); //copy bit serPin 1
13     }                                        //of PORTC to MSB of REGA.
14     return 0;
15 }
```

Example 7-35

Write an AVR C program to bring in a byte of data serially one bit at a time via PORTC, pin 3. The MSB should come in first.

Solution:

```
1   #include <avr/io.h>                          //standard AVR header
2   #define serPin 3
3
4   int main (void)
5   {
6       unsigned char x;
7       unsigned char REGA=0;
8       DDRC &= ~(1<<serPin);                    //serPin as input
9       for(x=0; x<8; x++)                       //repeat for each bit of REGA
10      {
11        REGA = REGA << 1;                      //shift REGA to left one bit
12        REGA |=(PINC &(1<<serPin))>> serPin;   //copy bit serPin of
13      }                                        //PORT C to LSB of REGA.
14      return 0;
15  }
```

Memory Allocation in C

Using program (code) space for predefined fixed data is a widely used option in the AVR, as we saw in Chapter 6. In that chapter we saw how to use Assembly language programs to access the data stored in ROM. Next, we do the same thing with C language.

Flash, RAM, and EEPROM data space in the AVR

In the AVR we have three spaces in which to store data. They are as follows:

Table 7-7: Memory Size for Some ATmega Family Members(Bytes)

	Flash	SRAM	EEPROM
ATmega 8	8K	256	256
ATmega 16	16K	1K	512
ATmega 32	32K	2K	1K
ATmega 64	64K	4K	2K
ATmega 128	128K	8K	4K

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.6 : MEMORY ALLOCATION IN C

1. The 64K bytes of SRAM space with address range 0000-FFFFH. We also have seen how we can read (from) or write (into) this RAM space directly or indirectly. We store temporary variables in SRAM since the SRAM is the scratch pad.
2. The 2M words (4M bytes) of code (program) space with addresses of 000000-1FFFFFH. This 2M words of on-chip Flash ROM space is used primarily for storing programs (opcodes) and therefore is directly under control of the program counter (PC).
3. EEPROM. As we mentioned before, EEPROM can save data when the power is off. That is why we use EEPROM to save variables that should not be lost when the power is off. For example, the temperature set point of a cooling system should be changed by users and cannot be stored in program space.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
7.6 : MEMORY ALLOCATION IN C

In Chapter 6 we saw how to read from or write to EEPROM. In this chapter we will show the same concept using C programming. Notice that different C compilers may have their built-in functions or directives to access each type of memory.

In CodeVision, to define a const variable in the Flash memory, you only need to put the Flash directive before it. Also, to define a variable in EEPROM, you can put the eeprom directive in front of it:

```
flash unsigned char mynum[] = "Hello";       //use Flash code space
eeprom unsigned char = 7                        //use EEPROM space
```

To learn how you can use the built-in functions or directives of your compiler, you should consult the manual for your compiler. Also, you can download some examples using different compilers from www.MicroDigitalEd.com.

EEPROM access in C

In Chapter 6 we saw how we can access EEPROM using Assembly language. Next, we do the same thing with C language. Notice that as we mentioned before, most compilers have some built-in functions or directives to make the job of accessing the EEPROM memory easier. See Examples 7-36 and 7-37 to learn how we access EEPROM in C.

Example 7-36

Write an AVR C program to store 'G' into location Ox005F of EEPROM.

Solution:

```
1   #include <avr/io.h>                    //standard AVR header
2
3   int main(void)
4   {
5       while (EECR & (1<<EEWE));          //wait for last write to finish
6       EEAR = Ox5f;                       //write Ox5F to address register
7       EEDR = 'G';                        //write 'G' to data register
8       EECR |= (1<<EEMWE);                //write one to EEMWE
9       EECR |= (1<<EEWE);                 //start EEPROM write
10      return 0;
11  }
```

Example 7-37

Write an AVR C program to read the content of location 0x005F of EEPROM into PORTB.

Solution:

```
1   #include <avr/io.h>                  //standard AVR header
2
3   int main(void)
4   {
5       DDRB = 0xFF;                     //make PORTS an output
6       while (EECR & (1<<EEWE));        //wait for last write to finish
7       EEAR = 0x5f;                     //write 0x5F to address register
8       EECR |= (1<<EERE);               //start EEPROM read by writing FERE
9       PORTS = EEDR;                    //move data from data register to PORTE
10  }
```