CHAPTER 7

AVR PROGRAMMING IN C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Examine C data types for the AVR**
- **Code C programs for time delay and I/O operations**
- **Code C programs for I/O bit manipulation**
- **Code C programs for logic and arithmetic operations**
- **Code C programs for ASCII and BCD data conversion**
- **Code C programs for binary (hex) to decimal conversion**
- **Code C programs for data serialization**
- **Code C programs for EEPROM access**

Why program the AVR in C?

Compilers produce hex files that we download into the Flash of the microcontroller. 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 ATmega328 is 32K 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 larger than if we used Assembly language. 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.

Several third-party companies develop C compilers for the AVR microcontroller. We use the Atmel Studio IDE in this book and provide you with the fundamentals of C programming for the AVR. But you can use the compiler of your choice for the chapter examples and programs. At the time of the writing of this book Atmel Studio is available for free on the Web. See www.MicroDigitalEd.com for tutorials on the Atmel Studio IDE.

C programming for the AVR is the main topic of this chapter. In Section 7.1, we discuss data types, and time delays. I/O programming is shown in Section 7.2. The logic operations AND, OR, XOR, inverter, and shift are discussed in Section 7.3. Section 7.4 describes ASCII and BCD conversions and checksums. In Section 7.5, data serialization for the AVR is shown. In Section 7.6, memory allocation in C is discussed.

SECTION 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. 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. Such misuse of data types in compilers such as Microsoft Visual C++ for x86 IBM PCs is not a significant issue.

Remember that C compilers use the signed char as the default unless we put the keyword *unsigned* in front of the char (see Example 7-1). We can also use the unsigned char data type for a string of ASCII characters, including extended ASCII characters. Example 7-2 shows a string of ASCII characters. See Example 7-3 for toggling a port 200 times.

Example 7-1

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

Solution:

```
#include <avr/io.h>                    //standard AVR header

int main(void)
{
  unsigned char z;
  DDRB = 0xFF;                         //PORTB is output
  for(z = 0; z <= 255; z++)
    PORTB = z;

  return 0;
}
//Notice that the program never exits the for loop because if you
//increment an unsigned char variable when it is 0xFF, it will
//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:

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

Example 7-3

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

Solution:

```
//toggle PB 200 times
#include <avr/io.h>                     //standard AVR header

int main(void)                          //the code starts from here
{
      DDRB = 0xFF;                      //PORTB is output
      PORTB = 0xAA;                     //PORTB is 10101010
      unsigned char z;

      for(z=0; z < 200; z++)            //run the next line 200 times
            PORTB = ~ PORTB;            //toggle PORTB

      while(1);
      return 0;                         //stay here forever
}
```

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:

```
#include <avr/io.h>                          //standard AVR header
int main(void)
{
  char mynum[] = {-4,-3,-2,-1,0,+1,+2,+3,+4};
  unsigned char z;

  DDRB = 0xFF;                               //PORTB is output
  for(z = 0; z <= 8; z++)
    PORTB = mynum[z];

  while(1);                                  //stay here forever
  return 0;
}
```

Run the above program on your simulator to see how PORTB displays values of 0xFC, 0xFD, 0xFE , 0xFF, 0x00, 0x01, 0x02, 0x03, and 0x04 (the hex values for –4, –3, –2, –1, 0, 1, etc.). See Chapter 5 for discussion of signed numbers.

unsigned int

The *unsigned int* is a 16-bit data type that takes a value in the range of 0 to 65,535 (0000–$FFFF). 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. Such misuse is not a problem in PCs with 512 megabytes of memory, the 32-bit Pentium's registers and memory accesses, and a bus speed of 133 MHz. 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. Also, in situations where there is no need for signed data (such as setting counter values), we should use unsigned int instead of signed int. This gives a much wider range for data declaration. Again, remember that the C compiler uses signed int as the default unless we specify the keyword *unsigned*.

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. See Examples 7-5 and 7-6. 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 Port B 50,000 times.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  unsigned int z;
  DDRB = 0xFF;                         //PORTB is output

  for(z=0; z<50000; z++)
  {
    PORTB = 0x55;
    PORTB = 0xAA;
  }

  while(1);                            //stay here forever
  return 0;
}
```

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:

```
//toggle PB 100,00 times
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  unsigned long z;                     //long is used because it should
                                       //store more than 65535.
  DDRB = 0xFF;                         //PORTB is output

  for(z = 0; z < 100000; z++){
    PORTB = 0x55;
    PORTB = 0xAA;
  }

  while(1);
  return 0;                            //stay here forever
}
```

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 produce different code. In other words, if we compile a given C program with different compilers, each compiler produces different hex code.

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. Look at Example 7-7. Notice that most compilers do some code optimization before generating a .hex file and they omit the delay loop and any parts of the code which do nothing other than wasting CPU time. But in the delay100ms, the *volatile* keyword tells the compiler that the ***i*** variable is used by other parts of the program and the compiler does not optimize the parts of the code which use the variable.

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 ATmega328 with XTAL = 16 MHz.

Solution:

```
#include <avr/io.h>                    //standard AVR header
void delay100ms(void)
{
  volatile unsigned long i;
  for(i=0; i < 47060; i++);            //try different numbers on your
}                                      //compiler and examine the result.

int main(void)
{
  DDRB = 0xFF;                         //PORTB is output
  while (1)
  {
    PORTB = 0xAA;
    delay100ms();
    PORTB = 0x55;
    delay100ms();
  }
  return 0;
}
```

Another way of generating time delay is to use predefined functions such as _delay_ms() and _delay_us() defined in delay.h in Atmel Studio or delay_ms() and delay_us() defined in delay.h in CodeVision. See Example 7-8.

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 Atmel Studio.

Solution:

```
#define F_CPU 16000000UL                    //XTAL = 16MHz
#include <util/delay.h>                     //delay loop functions
#include <avr/io.h>                         //standard AVR header
int main(void)
{
      DDRB = 0xFF;                          //PORTB is output
      while (1){
            PORTB = 0xFF;
            delay_ms(10);
            PORTB = 0x55;
            delay_ms(10);
      }
      return 0;
}
```

Note: In Atmel Studio, define F_CPU before using delay.h to inform the compiler the crystal frequency.

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. To overcome this problem, programmers use macro or wrapper function. Wrapper functions do nothing more than calling the predefined function. If you use a function in your code and decide to change your compiler, instead of changing all instances of predefined functions, you simply define a wrapper function and use the previous code. For example, if you want to migrate from Code Vision to Atmel Studio, instead of replacing all delay_ms functions with _delay_ms, you can define the following wrapper function:

```
void delay_ms(int d) {          //delay in d microseconds
      _delay_ms(d);
}
```

Notice that calling a wrapper function may take some microseconds.

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

Review Questions

1. Give the magnitude of the unsigned char and signed char data types.
2. Give the magnitude of the unsigned int and signed int data types.
3. If we are declaring a variable for a person's age, we should use the ___ data type.
4. Give two factors that can affect the delay size.

SECTION 7.2: I/O PROGRAMMING IN C

As we stated in Chapter 4, all port registers of the AVR are both byte accessible and bit accessible. In this section we look at C programming of the I/O ports for the AVR. We look at both byte and bit I/O programming.

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. See Examples 7-9, 7-10, and 7-11.

Example 7-9

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

Solution:

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
int main(void)
{
  DDRB = 0xFF;                    //Port B is output
  while (1)
  {
    PORTB = PORTB + 1;
    _delay_ms(500);             //wait 0.5 sec.
  }
}
```

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:

```
#include <avr/io.h>                 //standard AVR header
int main(void)
{
  unsigned char temp;

  DDRB = 0x00;                      //Port B is input
  DDRC = 0xFF;                      //Port C is output

  while(1)
  {
    temp = PINB;
    PORTC = temp;
  }
  return 0;
}
```

Note: In the above program, instead of using temp, you can write "PORTC = PINB;".

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:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  DDRC = 0;                            //Port C is input
  DDRB = 0xFF;                         //Port B is output
  DDRD = 0xFF;                         //Port D is output
  unsigned char temp;
  while(1)
  {
    temp = PINC;                       //read from PINC
    if ( temp < 100 )
      PORTB = temp;
    else
      PORTD = temp;
  }
  return 0;
}
```

Bit size I/O

The I/O ports of ATmega328 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:

```
PORTB.0 = 1;
```

but it cannot be used in other compilers such as Atmel Studio.

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.

In next section you will see how to mask a bit of a byte. You can use masking for both bit-accessible and byte-accessible ports and registers.

Review Questions

1. Write a short program that toggles all bits of Port C.
2. True or false. All bits of Port B are bit addressable.
3. True or false. To access the data direction register of Port B, we use DDRB.

SECTION 7.3: LOGIC OPERATIONS IN C

One of the most important and powerful features of the C language is its ability to perform bit manipulation. Because many books on C do not cover this important topic, it is appropriate to discuss it in this section. This section describes the action of bit-wise logic operators and provides some examples of how they are used.

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. See Table 7-2.

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	

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 0x55 */

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.

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

Setting and Clearing Bits

OR, AND, and EXOR can be used to set, clear, and toggle bits:

Anything ORed with a 1 results in a 1; anything ORed with a 0 results in no change.

Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.

Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change. See Examples 7-13 through 7-20.

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:

```
#include <avr/io.h>                    //standard AVR header

int main(void)
{
  DDRB = 0xFF;                         //PORTB is output

  while(1)
  {
    PORTB = PORTB | 0b00010000;        //set bit 4 (5th bit) of PORTB
    PORTB = PORTB & 0b11101111;        //clear bit 4 (5th bit) of PORTB
  }

  return 0;
}
```

Example 7-14

Write an AVR C program to monitor bit 5 of port C. If it is HIGH, send 0x55 to Port B; otherwise, send 0xAA to Port B.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  DDRB = 0xFF;                         //PORTB is output
  DDRC = 0x00;                         //PORTC is input
  DDRD = 0xFF;                         //PORTB is output

  while(1)
  {
    if((PINC & 0b00100000) != 0)       //check bit 5 (6th bit) of PINC
      PORTB = 0x55;
    else
      PORTB = 0xAA;
  }

  return 0;
}
```

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:

```
#include <avr/io.h>                          //standard AVR header

int main(void)
{
  DDRB = DDRB & 0b11111101;                  //pin 1 of Port B is input
  DDRC = DDRC | 0b10000000;                  //pin 7 of Port C is output

  while(1)
  {
    if((PINB & 0b00000010) != 0)             //check pin 1 (2nd pin) of PINB
      PORTC = PORTC | 0b10000000;            //set pin 7 (8th pin) of PORTC
    else
      PORTC = PORTC & 0b01111111;            //clear pin 7 (8th pin) of PORTC
  }
  return 0;
}
```

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:

```
#include <avr/io.h>                          //standard AVR header

int main(void)
{
  unsigned char message[] = "The Earth is but One Country";
  unsigned char z;

  DDRB = 0xFF;                               //Port B is output
  DDRC = DDRC | 0b00100000;                  //PC5 is output

  for ( z = 0; z < 28; z++)
  {
    PORTB = message[z];
    PORTC = PORTC | 0b00100000;              //pin LCD_EN of Port C is 1
    PORTC = PORTC & 0b11011111;              //pin LCD_EN of Port C is 0
  }
  while (1);
  return 0;
}
//In Chapter 12 we will study more about LCD interfacing
```

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:

```
#include <avr/io.h>              //standard AVR header

int main(void)
{
  unsigned char z;

  DDRB = 0;                      //make Port B an input
  DDRD = 0xFF;                   //make Port D an output

  while(1)                       //repeat forever
  {
    z = PINB;                    //read PORTB
    z = z & 0b00000011;          //mask the unused bits

    switch(z)                    //make decision
    {
      case 0:
        PORTD = '0';             //issue ASCII 0
        break;

      case 1:
        PORTD = '1';             //issue ASCII 1
        break;

      case 2:
        PORTD = '2';             //issue ASCII 2
        break;

      case 3:
        PORTD = '3';             //issue ASCII 3
        break;
    }
  }

  return 0;
}
```

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:

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

  return 0;
}
```

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:

```
#include <avr/io.h>                     //standard AVR header

int main(void)
{
  DDRB = DDRB & 0b11011111;             //bit 5 of Port B is input
  DDRC = DDRC | 0b10000000;             //bit 7 of Port C is output

  while (1)
  {
    if(PINB & 0b00100000 )
      PORTC = PORTC | 0b10000000;       //set bit 7 of Port C to 1
    else
      PORTC = PORTC & 0b01111111;       //clear bit 7 of Port C to 0
  }
  return 0;
}
```

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:

(a)

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
      DDRB = 0xFF;                     //Port B is output
      PORTB = 0xAA;
      while (1)
            PORTB = ~ PORTB;           //toggle PORTB
      return 0;
}
```

(b)

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
      DDRB = 0xFF;                     //Port B is output
      PORTB = 0xAA;
      while (1)
            PORTB = PORTB ^ 0xFF;
      return 0;
}
```

```
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

```
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

Examine the Assembly output for parts (a) and (b) of Example 7-20. You will notice that the generated codes are the same because they do exactly the same thing.

Compound assignment operators in C

To reduce coding (typing) we can use compound statements for bit-wise operators in C. See Table 7-3 and Example 7-21.

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
Ex-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:

(a)

```
#include <avr/io.h>            //standard AVR header
int main(void)
{
  DDRB &= 0b11011111;   //bit 5 of Port B is input
  while (1)
  {
    if(PINB & 0b00100000)
      DDRB &= 0b11101111;     //bit 4 of Port B is input
    else
      DDRB |= 0b00010000;     //bit 4 of Port B is output
  }
  return 0;
}
```

(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 shift operators in C. See Table 7-4.

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). See Example 7-22.

Examples 7-23 and 7-24 are the same as Examples 7-18 and 7-19, but they use shift operation.

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)`

As we mentioned before, bit-wise shift operation can be used to increase code clarity. See Example 7-25.

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:

```
#include <avr/io.h>                          //standard AVR header

int main(void)
{
  DDRB = DDRB & ~(1<<7);                     //bit 7 of Port B is input

  while (1)
  {
    if(PINB & (1<<7))
      DDRB = DDRB & ~(1<<4);                 //bit 4 of Port B is input
    else
      DDRB = DDRB | (1<<4);                  //bit 4 of Port B is output
  }

  return 0;
}
```

Note: Using compound assignment, we can write "DDRB &= ~(1<<4);" and "DDRB |= (1<<4);".

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:

```
#include <avr/io.h>                          //standard AVR header

int main(void)
{
  DDRB &= ~(1<<5);                           //bit 5 of Port B is input
  DDRC |= (1<<7);                            //bit 7 of Port C is output

  while (1)
  {
    if(PINB & (1<<5) )
      PORTC |= (1<<7); //set bit 7 of Port C to 1
    else
      PORTC &= ~(1<<7);//clear bit 7 of Port C to 0
  }
  return 0;
}
```

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). In future chapters you will see how we use this method.

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:

```
#include <avr/io.h>                    //standard AVR header
#define LED 7
#define SENSOR 1

int main(void)
{
   DDRB &= ~(1<<SENSOR);          //SENSOR pin is input
   DDRC |= (1<< LED);             //LED pin is output

   while(1)
   {
     if (PINB & (1 << SENSOR))        //check SENSOR pin of PINB
       PORTC |= (1<<LED);         //set LED pin of Port C
     else
       PORTC &= ~(1<<LED);        //clear LED pin of Port C
   }
   return 0;
}
```

Review Questions

1. Find the content of PORTB after the following C code in each case:
   (a) `PORTB=0x37&0xCA;`
   (b) `PORTB=0x37|0xCA;`
   (c) `PORTB=0x37^0xCA;`
2. To mask certain bits we must AND them with _____.
3. To set high certain bits we must OR them with _____.
4. EX-ORing a value with itself results in _____.
5. Find the contents of PORTC after execution of the following code:
   ```
   PORTC = 0;
   PORTC = PORTC | 0x99;
   PORTC = ~PORTC;
   ```
6. Find the contents of PORTC after execution of the following code:
   ```
   PORTC = ~(0<<3);
   ```

SECTION 7.4: DATA CONVERSION PROGRAMS IN C

Recall that BCD numbers were discussed in Chapters 5 and 6. As stated there, many newer microcontrollers have a real-time clock (RTC) where the time and date are kept even when the power is off. Very often the RTC provides the time and date in packed BCD. To display them, however, we must convert them to ASCII. In this section we show the application of logic and rotate instructions in the conversion of BCD and ASCII.

ASCII numbers

On ASCII keyboards, when the "0" key is activated, "0011 0000" (30H) is provided to the computer. Similarly, 31H (0011 0001) is provided for the "1" key, and so on, as shown in Table 7-5.

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	01000111 or 47H
7	37	00000111	

See Example 7-27.

Example 7-26

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

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  unsigned char x, y;
  unsigned char mybyte = 0x29;

  DDRB = DDRC = 0xFF;                  //make Ports B and C output
  x = mybyte & 0x0F;                   //mask upper 4 bits
  PORTB = x | 0x30;                    //make it ASCII
  y = mybyte & 0xF0;                   //mask lower 4 bits
  y = y >> 4;                          //shift it to lower 4 bits
  PORTC = y | 0x30;                    //make it ASCII

  return 0;
}
```

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:

```
#include <avr/io.h>                    //standard AVR header

int main(void)
{
  unsigned char bcdbyte;
  unsigned char w = '4';
  unsigned char z = '7';
  DDRB = 0xFF;                         //make Port B an output
  w = w & 0x0F;                        //mask 3
  w = w << 4;                          //shift left to make upper BCD digit
  z = z & 0x0F;                        //mask 3
  bcdbyte = w | z;                     //combine to make packed BCD
  PORTB = bcdbyte;

  return 0;
}
```

Checksum byte in ROM

To ensure the integrity of data, every system must perform the checksum calculation. When you transmit data from one device to another or when you save and restore data to a storage device you should perform the checksum calculation to ensure the integrity of the data. The checksum will detect any corruption of data. 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 checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted). See Examples 7-28 through 7-30.

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 checksum detects the error.

Solution:

(a) Find the checksum byte.

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

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

```
   25H
+  62H
+  3FH
+  52H
+  E8H
   00H  (dropping the carries 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
   C0H  (dropping the 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.

Solution:

```
#include <avr/io.h>                              //standard AVR header
int main(void)
{
      unsigned char mydata[] = {0x25,0x62,0x3F,0x52};
      unsigned char sum = 0;
      unsigned char x;
      unsigned char chksumbyte;

      DDRD = 0xFF;                               //make Port D output
      DDRB = 0xFF;                               //make Port B output
      DDRC = 0xFF;                               //make Port C output

      for(x=0; x<4; x++)
      {
            PORTD = mydata[x];                   //issue each byte to PORTD
            sum = sum + mydata[x];               //add them together
            PORTB = sum;                         //issue the sum to PORTB
      }
      chksumbyte = ~sum + 1;                     //make 2's complement (invert +1)
      PORTC = chksumbyte;                        //show the checksum byte
      while(1);              //wait here
}
//Note:  To make 2's complement you can also write "chksumbyte = -sum;"
```

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 'B' to PORTD.

Solution:

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

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. 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 shows the C program for the above algorithm.

Example 7-31

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

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
    unsigned char x, binbyte, d1, d2, d3;
    DDRB = DDRC = DDRD =0xFF;          //Ports B, C, and D output
    binbyte = 0xFD;                    //binary (hex) byte
    x = binbyte / 10;                  //divide by 10
    d1 = binbyte % 10;                 //find remainder (LSD)
    d2 = x % 10;                       //middle digit
    d3 = x / 10;                       //most-significant digit (MSD)
    PORTB = d1;
    PORTC = d2;
    PORTD = d3;

    return 0;
}
```

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.

Review Questions

1. For the following decimal numbers, give the packed BCD and unpacked BCD representations:
   (a) 15 (b) 99
2. Show the binary and hex for "76".
3. 67H in BCD when converted to ASCII is ____H and ____H.
4. Does the following convert unpacked BCD to ASCII?
   ```
   mydata=0x09+0x30;
   ```
5. Why is the use of packed BCD preferable to ASCII?
6. Which takes more memory space to store numbers: packed BCD or ASCII?
7. In Question 6, which is more universal?
8. Find the checksum byte for the following values: 22H, 76H, 5FH, 8CH, 99H.
9. To test data integrity, we add the bytes together, including the checksum byte. The result must be equal to ____ if the data is not corrupted.
10. An ADC provides an output of 0010 0110. How do we display that on the screen?

SECTION 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 I²C, SPI, and CAN, not all devices support such standards. For this reason we need to be familiar with data serialization using the C language.

Examine the next four examples to see how data serialization is done 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:

```
#include <avr/io.h>
#define serPin 3

int main(void)
{
    unsigned char conbyte = 0x44;
    unsigned char regALSB;
    unsigned char x;
    regALSB = conbyte;
    DDRC |= (1<<serPin);

   for(x=0;x<8;x++)
      {
        if(regALSB & 0x01)
            PORTC |= (1<<serPin);
        else
            PORTC &= ~(1<<serPin);
        regALSB = regALSB >> 1;
      }
     return 0;
}
```

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:

```
#include <avr/io.h>
#define serPin 3
int main(void)
{
    unsigned char conbyte = 0x44;
    unsigned char regALSB;
    unsigned char x;
    regALSB = conbyte;
    DDRC |= (1<<serPin);
    for(x=0;x<8;x++)
      {
        if(regALSB & 0x80)
            PORTC |= (1<<serPin);
        else
            PORTC &= ~(1<<serPin);
        regALSB = regALSB << 1;
      }
     return 0;
}
```

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:

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

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:

```
#include <avr/io.h>                    //standard AVR header
#define serPin 3

int main(void)
{
    unsigned char x;
    unsigned char REGA=0;
    DDRC &= ~(1<<serPin);              //serPin as input
    for(x=0; x<8; x++)                 //repeat for each bit of REGA
    {
       REGA = REGA << 1;               //shift REGA to left one bit
       REGA |= (PINC &(1<<serPin))>> serPin; //copy bit serPin of
    }                                  //PORT C to LSB of REGA.
    while(1);
}
```

SECTION 7.6: 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.

The 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.

In Chapter 6 we saw how to read from or write to EEPROM and Flash

memory. In this section we will show the same concept using C programming.

Using Flash memory to store data

In order to define and access constant variables in the Flash memory, the Atmel Studio provides the `pgmspace` header file.

To define a variable in the Flash memory, the `PROGMEM` keyword is used:

```
const char PROGMEM txtHelloWorld[]="Hello World!";
```

In the above example, `"Hello World!"` is stored in the Flash memory at compile time. The contents of `txtHelloWorld` can be read from the Flash memory using the `pgm_read_byte(unsigned int addr)` function. Program 7-1 puts the contents of `txtHelloWorld` on `PORTC`.

```
#include "avr/io.h"
#include "avr/pgmspace.h"

const char PROGMEM txtHelloWorld[] = "Hello World!";

int main(void)
{
  DDRC = 0xFF;
  for (unsigned char i = 0; i < 10; i++)
  {
      //read txtHelloWorld[i] from Flash memory
     c = pgm_read_byte(&txtHelloWorld[i]);
     PORTC = c;
  }
  while(1);
}
```

Program 7-1: Using Flash Memory in Atmel Studio

In the above program, `pgm_read_byte( )` gets the address of the Flash memory location to be read and returns its contents. To get the address of `txtHelloWorld[i]`, an "&" is put before it.

pgm_read_byte reads a byte of data from Flash memory. The following table lists the functions which are available for reading the different types of data.

Table 7-7: Functions to Access Flash Memory Const Variables in C

Function signature	Data Type
int pgm_read_byte(char *str)	char
pgm_read_word	int
pgm_read_dword	long
pgm_read_float	float

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. 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 0x005F of EEPROM.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  while(EECR & (1<<EEWE));             //wait for last write to finish
  EEAR = 0x5F;                         //write 0x5F to address register
  EEDR = 'G';                          //write 'G' to data register
  EECR |= (1<<EEMWE);                  //write one to EEMWE
  EECR |= (1<<EEWE);                   //start EEPROM write
  while(1);
}
```

Example 7-37

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

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
  DDRB = 0xFF;                         //make PORTB an output
  while(EECR & (1<<EEWE));             //wait for last write to finish
  EEAR = 0x5F;                         //write 0x5F to address register
  EECR |= (1<<EERE);                   //start EEPROM read by writing EERE
  PORTB = EEDR;                        //move data from data register to PORTB
  while(1);
}
```

EEPROM access in Atmel Studio

To access the EEPROM, Atmel Studio provides the eeprom header file. Using the EEPROM variables are similar to the Flash variables. To define a variable in the EEPROM memory, the EEMEM keyword is used, and the eeprom_read_byte function can be used to read from eeprom. The following program defines an EEPROM variable. It stores 'G' into the variable if PB0 is high, and then reads the variable from memory.

```
#include <avr/io.h>
#include <avr/eeprom.h>

unsigned char EEMEM myVar;

int main()
{
    DDRC = 0xFF;
    if((PINB&(1<<0)) != 0) //if PB0 is high
        eeprom_write_byte(&myVar,'G'); //init myVar with 'G'
    char c = eeprom_read_byte(&myVar); //read myVar from eeprom
    PORTC = c;
    while(1);
}
```

Program 7-2: Using EEPROM in Atmel Studio

Tables 7-8 and 7-9 list the functions which are available for reading and writing the different types of data.

Table 7-8: Functions for Writing into EEPROM

Data Type	Function signature
char	void eeprom_write_byte (unsigned char *p, unsigned char value)
int	void eeprom_write_word (unsigned int *p, unsigned int value)
long	void eeprom_write_dword (unsigned long *p, unsigned long alue)
float	void eeprom_write_float (float *p, float value)

Table 7-9: Functions for Reading from EEPROM

Data Type	Function signature
char	unsigned char eeprom_read_byte (unsigned char *p)
int	unsigned int eeprom_write_word (unsigned int *p)
long	unsigned long eeprom_write_dword (unsigned long *p)
float	float eeprom_write_float (float *p)

Review Questions

1. The AVR family has a maximum of ____ of program ROM space.
2. The ATmega128 has ____ of program ROM.
3. True or false. The program (code) ROM space can be used for data storage, but the data space cannot be used for code.
4. True or false. Using the program ROM space for data means the data is fixed and static.
5. If we have a message string with a size of over 1000 bytes, then we use _____ (program ROM, data RAM) to store it.

SUMMARY

This chapter dealt with AVR C programming, specifically I/O programming and time delays in C. We also showed the logic operators AND, OR, XOR, and complement. In addition, some applications for these operators were discussed. This chapter described BCD and ASCII formats and conversions in C. We also discussed how to access EEPROM in C. The data serialization was also discussed.

PROBLEMS

SECTION 7.1: DATA TYPES AND TIME DELAYS IN C

1. Indicate what data type you would use for the following variables:
   (a) temperature
   (b) the number of days in a week
   (c) the number of days in a year
   (d) the number of months in a year
   (e) a counter to track the number of people getting on a bus

(f) a counter to track the number of people going to a class
(g) an address of 64K RAM space
(h) the age of a person
(i) a string for a message to welcome people to a building

2. Give the hex value that is sent to the port for each of the following C statements:
   (a) PORTB=14; (b) PORTB=0x18; (c) PORTB='A';
   (d) PORTB=7; (e) PORTB=32; (f) PORTB=0x45;
   (g) PORTB=255; (h) PORTB=0x0F;
3. Give two factors that can affect time delay in the AVR microcontroller.
4. Of the two factors in Problem 3, which can be set by the system designer?
5. Can the programmer set the number of clock cycles used to execute an instruction? Explain your answer.
6. Explain why various C compilers produce different hex file sizes.

SECTION 7.2: I/O PROGRAMMING IN C

7. What is the difference between PORTC=0x00 and DDRC=0x00?
8. Write a C program to toggle all bits of Port B every 200 ms.
9. Write a C program to toggle bits 1 and 3 or Port B every 200 ms.
10. Write a time delay function for 100 ms.
11. Write a C program to toggle only bit 3 of PORT C every 200 ms.
12. Write a C program to count up Port B from 0–99 continuously.

SECTION 7.3: LOGIC OPERATIONS IN C

13. Indicate the data on the ports for each of the following:
    Note: The operations are independent of each other.
    (a) `PORTB=0xF0&0x45;` (b) `PORTB=0xF0&0x56;`
    (c) `PORTB=0xF0^0x76;` (d) `PORTC=0xF0&0x90;`
    (e) `PORTC=0xF0^0x90;` (f) `PORTC=0xF0|0x90;`
    (g) `PORTC=0xF0&0xFF;` (h) `PORTC=0xF0|0x99;`
    (i) `PORTC=0xF0^0xEE;` (j) `PORTC=0xF0^0xAA;`
14. Find the contents of the port after each of the following operations:
    (a) `PORTB=0x65&0x76;` (b) `PORTB=0x70|0x6B;`
    (c) `PORTC=0x95^0xAA;` (d) `PORTC=0x5D&0x78;`
    (e) `PORTC=0xC5|0x12;` (f) `PORTD=0x6A^0x6E;`
    (g) `PORTB=0x37|0x26;`
15. Find the port value after each of the following is executed:
    (a) `PORTB=0x65>>2;` (b) `PORTC=0x39<<2;`
    (c) `PORTB=0xD4>>3;` (d) `PORTB=0xA7<<2;`
16. Show the C code to swap 0x95 to make it 0x59.
17. Write a C program that finds the number of zeros in an 8-bit data item.

SECTION 7.4: DATA CONVERSION PROGRAMS IN C

18. Write a C program to convert packed BCD 0x34 to ASCII and display the bytes on PORTB and PORTC.
19. Write a program to convert ASCII digits of '7' and '2' to packed BCD and display them on PORTB.

SECTION 7.5: DATA SERIALIZATION IN C

20. Write a C program to that finds the number of 1s in a given byte.

SECTION 7.6: MEMORY ALLOCATION IN C

21. Indicate what type of memory (data SRAM or code space) you would use for the following variables:
    (a) temperature
    (b) the number of days in a week
    (c) the number of days in a year
    (d) the number of months in a year
22. True or false. When using code space for data, the total size of the array should not exceed 256 bytes.
23. Why do we use the code space for video game characters and shapes?
24. What is the advantage of using code space for data?
25. What is the drawback of using program code space for data?
26. Write a C program to send your first and last names to EEPROM.
27. Indicate what type of memory (data RAM, or code ROM space) you would use for the following variables:
    (a) a counter to track the number of people getting on a bus
    (b) a counter to track the number of people going to a class
    (c) an address of 64K RAM space
    (d) the age of a person
    (e) a string for a message to welcome people to a building

28. Why do we not use the data RAM space for video game characters and shapes?
29. What is the drawback of using RAM data space for fixed data?
30. What is the advantage of using data RAM space for variables?

ANSWERS TO REVIEW QUESTIONS

SECTION 7.1: DATA TYPES AND TIME DELAYS IN C

1. 0 to 255 for unsigned char and –128 to +127 for signed char
2. 0 to 65,535 for unsigned int and –32,768 to +32,767 for signed int
3. Unsigned char
4. (a) Crystal frequency of the AVR system
   (b) Compiler used for C

SECTION 7.2: I/O PROGRAMMING IN C

1.
```
void main()
{
   DDRC = 0xFF;          //PORTC is output
   PORTC = 0x55;         //PORTC is 0101 0101
   PORTC = 0xAA;         //PORTC is 1010 1010
}
```
2. True
3. True

SECTION 7.3: LOGIC OPERATIONS IN C

1. (a) 02H
   (b) FFH
   (c) FDH
2. Zeros
3. One
4. All zeros
5. 66H
6. ~ ((0000 0000) << 3) = ~ (1111 1111) = FFH

SECTION 7.4: DATA CONVERSION PROGRAMS IN C

1. (a) 15H = 0001 0101 packed BCD, 0000 0001,0000 0101 unpacked BCD
   (b) 99H = 1001 1001 packed BCD, 0000 1001,0000 1001 unpacked BCD
2. "76" = 3736H = 00110111 00110110B
3. 36, 37
4. Yes, because mydata = 0x39
5. Space savings
6. ASCII
7. BCD
8. E4H
9. 0
10. First, convert from binary to decimal, then convert to ASCII, and then send the results to the screen and we will see 038.

SECTION 7.6: PROGRAM ROM ALLOCATION IN C

1. 2M words (4M bytes)
2. 128K bytes
3. True
4. True
5. Program ROM