CHAPTER 6

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

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

- >> **List all the addressing modes of the AVR microcontroller**
- >> **Contrast and compare the addressing modes**
- >> **Code AVR Assembly language instructions using each addressing mode**
- >> **Access the data RAM file register using various addressing modes**
- >> **Code AVR instructions to manipulate a look-up table**
- >> **Access fixed data residing in the program Flash ROM space**
- >> **Discuss how to create macros**
- >> **Explain how to write data to EEPROM memory of the AVR**
- >> **Explain how to read data from EEPROM memory of the AVR**
- >> **Code AVR programs to create and test the checksum byte**
- >> **Code AVR programs for ASCII data conversion**

In Section 6.1, you learn some new assembler directives that are used throughout this chapter. In Sections 6.2 through 6.4 we see the different ways in which we can access program and data memories in the AVR.

Section 6.5 explains the bit-addressability of the data memory space. In Section 6.6 we discuss how to access EEPROM in the AVR. Checksum generation and BCD-ASCII conversions are covered in Section 6.7. Macros are examined in Section 6.8.

SECTION 6.1: INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

In Chapter 2, we introduced the assembler directives .ORG, .SET, and .INCLUDE. In this section, you will learn some other useful directives.

Arithmetic and logic expressions with constant values

As you saw in Chapter 2, we can define constant values using .EQU. The AVR Studio IDE supports arithmetic operations between expressions. See Table 6-1. For example, in the following program R24 is loaded with 29, which is the result of the arithmetic expression "`((ALFA-BETA)*2)+9`".

```
.EQU ALFA = 50
.EQU BETA = 40
LDI R23,ALFA                    ;R23 = ALFA = 50
LDI R24,((ALFA-BETA)*2)+9       ;R24 = ((50-40)*2)+9 = 29
```

The AVR Studio IDE supports logic operations between expressions as well. See Table 6-2. For example, in the following program R21 is loaded with 0x14:

```
.EQU C1 = 0x50
.EQU C2 = 0x10
.EQU C3 = 0x04
LDI R21,(C1&C2)|C3   ;R21=(0x10&0x50)|0x04 = 0x10|0x04= 0x14
```

In Table 6-3 you see the shift operators, which are very useful. They shift left and right a constant value. For example, the following instruction loads the R20 register with 0b00001110:

```
LDI R16,0b00000111<<1 ;R16 = 0b00001110
```

One of the uses of shift operators is for initializing the registers. For exam-

Table 6-1: Arithmetic Operators

Symbol	Action
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Table 6-2: Logic Operators

Symbol	Action
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT

Table 6-3: Shift Operators

Symbol	Action	Example
<<	Shifts left the left expression by the number of places given by the right expression	`LDI R20,0b101<<2 ;R20=0b10100`
>>	Shifts right the left expression by the number of places given by the right expression	`LDI R20,0b100>>1 ;R20=0b010`

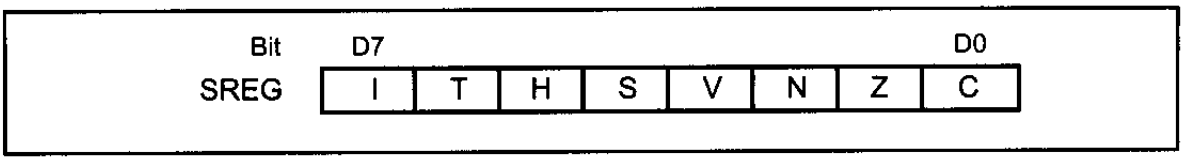


Figure 6-1. Bits of the Status Register

ple, suppose we want to set the Z and C bits of the SREG (Status Register) register and clear the others. Look at Figure 6-1. If we load 0b00000011 to SREG the task will be done:

```
LDI R20, 0b00000011   ;Z = 1, C = 1
OUT SREG,R20
```

In this example, we calculated the 0b00000011 number by looking at Figure 6-1. But imagine you are writing a program and you want to do the same task; you have to open the datasheet or a reference book to see the structure of the SREG register. To make the task simpler, the names of the register bits are defined in the header files of each AVR microcontroller. For example, in M32DEF.INC there are the following lines of code:

```
...
; SREG - Status Register
.equ  SREG_C      = 0   ;carry flag
.equ  SREG_Z      = 1   ;zero flag
.equ  SREG_N      = 2   ;negative flag
.equ  SREG_V      = 3   ;2's complement overflow flag
.equ  SREG_S      = 4   ;sign bit
.equ  SREG_H      = 5   ;half carry flag
.equ  SREG_T      = 6   ;bit copy storage
.equ  SREG_I      = 7   ;global interrupt enable
...
```

So, we can use the names of the bits instead of remembering the structure of the registers or finding them in the datasheet. For example, the following program sets the Z flag of the SREG register and clears the other bits:

```
LDI   R16, 1<<SREG_Z ;R16= 1 << 1 = 0b00000010
OUT   SREG,R16       ;SREG = 0b00000010 (set Z and clear others)
```

As another example, the following program sets the V and S flags of SREG:

```
LDI   R16,(1<<SREG_V)|(1<<SREG_S)     ;R16=0b1000|0b10000=0b11000
OUT   SREG,R16    ;SREG = 0b00011000 (set V and S, clear others)
```

In Example 6-1, you see the usage of the directives in I/O port programming.

Example 6-1

Write codes to set PB2 and PB4 of PORTB to 1 and clear the other pins
(a) without the directives, and
(b) using the directives.

Solution:

(a)
```
LDI   R20,0x14          ;R20 = 0x14
OUT   PORTB,R20         ;PORTB = R20
```

To make the code more readable, we can write the number in binary as well:

```
LDI   R20,0b00010100    ;R20 = 0x14
OUT   PORTB,R20         ;PORTB = 0x14
```

(b)
```
LDI   R20,(1<<4)|(1<<2) ;R20 = (0b10000 | 0b00100) = 0b10100
OUT   PORTB,R20         ;PORTB = R20
```

As we mentioned before, the names of the register bits are defined in the header files of each AVR microcontroller. PB2 and PB4 are defined equal to 2 and 4, as well. Therefore, we can write the code as shown below:

```
LDI   R20,(1<<PB4)|(1<<PB2) ;set the PB4 and PB2 bits
OUT   PORTB,R20             ;PORTB = R20
```

Notice that when the assembler wants to convert a code to machine language it substitutes all of the assembler directives with their equivalent values. Thus, using the directives has no side effects on the performance of our code but rather makes our code more readable. See Examples 6-2 and 6-3.

Example 6-2

What does the AVR assembler do while assembling the following program?

```
.equ  C1 = 2
.equ  C2 = 3
LDI R20,C1|(1<<C2) ;R20= 2|(1<<3)= 0b00000010|0b00001000= 0b00001010
```

Solution:

.equ is an assembler directive. When assembling ".equ C1 = 2", the assembler assigns value 2 to C1. Similarly, while assembling the ".equ C2 = 3" instruction, it assigns the value 3 to C2.

When the assembler converts the "`LDI  R20,C1|(1<<C2)`" instruction to machine language, it knows the values of C1 and C2. Thus it calculates the value of "`C1|(1<<C2)`", and then replaces the expression with its value. Therefore, "`LDI R20,C1|(1<<C2)`" will be converted to "`LDI  R20,0b00001010`". Then the assembler converts the instruction to machine language.

Example 6-3

What does the AVR assembler do while assembling the following program?

```
.INCLUDE "M32DEF.INC"
      LDI   R20,(1<<PB4)|(1<<PB2);set the PB4 and PB2 bits
      OUT   DDRB,R20           ;DDRB = R20
HERE: RJMP  HERE
```

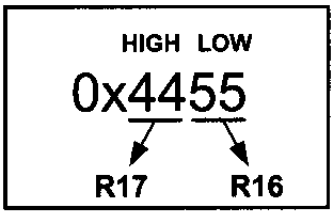
Solution:

Including a header file at the beginning of a program is similar to copying all the contents of the header file to the beginning of the program. Thus, the assembler, first assembles the contents of M32DEF.INC. The header file contains some ".equ" instructions, such as "`.equ PB4 = 4`". Thus, after reading the header file the assembler learns that PB4 is equal to 4, PB2 is equal to 2, and so on. Thus, when it wants to assemble instructions such as "`LDI R20,(1<<PB4)|(1<<PB2)`", it knows the values of PB2 and PB4. It calculates the value of "`(1<<PB4)|(1<<PB2)`" and substitutes it.

It is highly recommended that you take a look at the M32DEF.INC file. The file is located in the following path, if you did not change it while installing the AVR Studio software:
Program Files\Atmel\AVR Tools\AvrAssembler2\Appnotes\m32def.inc

HIGH( ) and LOW( ) functions

The HIGH( ) and LOW( ) functions give the higher and the lower bytes of a 16-bit value. For example, in the following program 0x55 and 0x44 are loaded into R16 and R17, respectively:




```
LDI   R16,LOW(0x4455)  ;R16 = 0x55
LDI   R17,HIGH(0x4455) ;R17 = 0x44
```

In Chapter 2, we used the following instructions to make the stack pointer refer to the last location of the memory:

```
LDI   R16,HIGH(RAMEND) ;R16 = 0x08 (for ATmega32)
OUT   SPH,R16          ;SPH = the high byte of address
LDI   R16,LOW(RAMEND)  ;R16 = 0x5f
OUT   SPL,R16          ;SPL = the low byte of address
```

But how do the instructions work? In the AVR header files (e.g., M32DEF.INC) RAMEND is defined equal to the address of the last location of the memory. For example, in *M32DEF.INC* there is the following line:

```
.equ  RAMEND      = 0x085f
```

The HIGH( ) and LOW( ) functions split the RAMEND into two bytes, $08 and $5F. They go to SPH and SPL, respectively.

You can see the list of the different directives available in the AVR by using the help feature of AVR Studio. (Choose the *assembler Help* option from the *Help* menu and then click on the *Expressions* topic.)

Review Questions

1. Indicate the value loaded into the registers in the following program:

```
.EQU CONST1 = 0x10
.EQU CONST2 = 0x91
.EQU CONST3 = 0x14
.EQU ADDR = (0x91 << 1)+1
LDI   R20,CONST1&CONST2
LDI   R21,CONST2|CONST3
LDI   R30,LOW(ADDR)
LDI   R31,HIGH(ADDR)
```

2. What does the following code do?

```
LDI   R16,(1<<SREG_V)|(1<<SREG_Z)
OUT   SREG,R16     ;SREG = 0b00011000
```

3. Using the assembler directives write a program that sets the Z and C flags and clears the other flags.

4. Calculate the values that are loaded into the TCNT1L and TCNT1H I/O registers.

```
LDI   R16,HIGH(15900)
OUT   TCNT1H,R16          ;TCNT1H = HIGH(15900)
LDI   R16,LOW(15900)
OUT   TCNT1L,R16          ;TCNT1L = LOW(15900)
```

SECTION 6.2: REGISTER AND DIRECT ADDRESSING MODES

The CPU can access data in various ways. The data could be in a register, or in memory, or provided as an immediate value. These various ways of accessing data are called *addressing modes*. In Sections 6.2 through 6.6 we discuss AVR addressing modes in the context of some examples.

The various addressing modes of a microprocessor are determined when it is designed, and therefore cannot be changed by the programmer. The AVR provides a total of 13 distinct addressing modes, which can be categorized into the following groups:

1. Single-Register (Immediate)
2. Register
3. Direct
4. Register indirect
5. Flash Direct
6. Flash Indirect

In this section we look at immediate, two-register, and direct addressing modes. In Section 6.3 we cover accessing RAM data memory using the register indirect mode. Section 6.4 explains how to access fixed data and look-up tables stored in program ROM.

Single-register (immediate) addressing mode

In this addressing mode, the operand is a register. See the examples below.

```
NEG   R18              ;negate the contents of R18
COM   R19              ;complement the contents of R19
INC   R20              ;increment R20
DEC   R21              ;decrement R21
ROR   R22              ;rotate right R22
```

In some of the instructions there is also a constant value with the register operand. See the examples below.

```
LDI   R19,0x25         ;load 0x25 into R19
SUBI  R19,0x6          ;subtract 0x6 from R19
ANDI  R19,0b01000000   ;AND R19 with 0x40
```

The constant value is sometimes referred to as *immediate data* since the operand comes immediately after the opcode when the instruction is assembled; and the addressing mode is referred to as *immediate addressing mode* in some microcontrollers. But the AVR datasheet refers to this mode as a subset of the single-register addressing mode. This addressing mode can be used to load data into any of the R16–R31 general purpose registers. The immediate addressing mode is also used for arithmetic and logic instructions. Note that the letter "I" in instructions such as LDI, ANDI, and SUBI means "immediate." See Figures 6-2a and 6-2b.

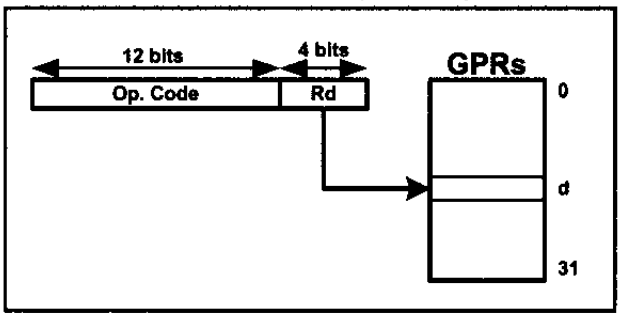


Figure 6-2a. Single-Register Addressing

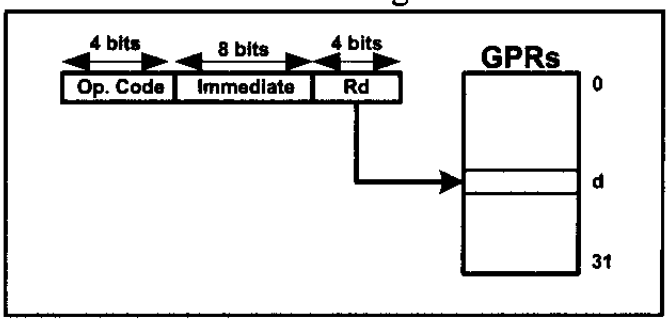


Figure 6-2b. Single-Register (with immediate)

We can use the .EQU directive to access immediate data, as shown below.

```
.EQU COUNT = 0x30
...   ..
LDI   R16,COUNT        ;R16 = 0x30
```

Two-register addressing mode

Two-register addressing mode involves the use of two registers to hold the data to be manipulated. See Figure 6-3.

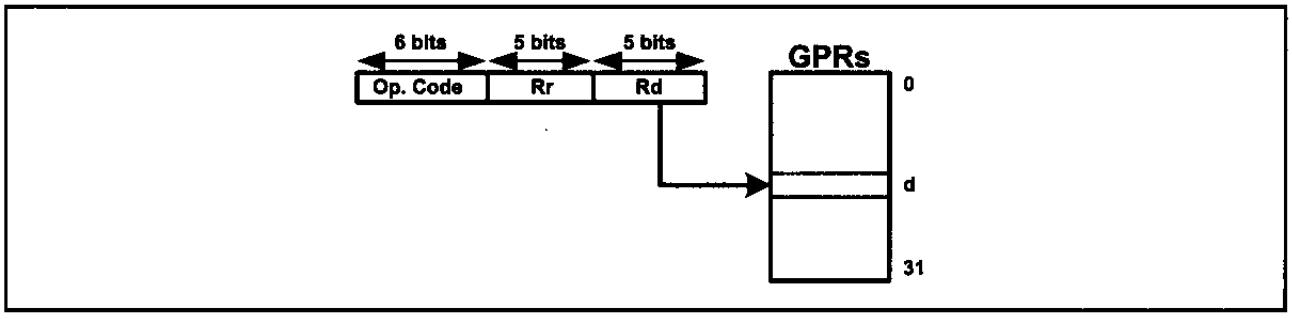


Figure 6-3. Two-Register Addressing

Examples of two-register addressing mode are as follows:

```
ADD   R20,R23          ;add R23 to R20
SUB   R29,R20          ;subtract R20 from R29
AND   R16,R17          ;AND R16 with 0x40
MOV   R23,R19          ;copy the contents of R19 to R23
```

Direct addressing mode

The entire data memory can be accessed using either direct or register indirect addressing modes. The register indirect addressing mode will be discussed in the next section. In direct addressing mode, the operand data is in a RAM memory location whose address is known, and this address is given as a part of the instruction. Contrast this with immediate addressing mode in which the operand data itself is provided with the instruction. Examine the following instructions:

```
LDS   R19,0x560  ;load R19 with the contents of memory loc $560
STS   0x40,R19    ;store R19 to data space location 0x40
```

The two instructions use direct addressing mode. If we dissect the opcode we see that the addresses are embedded in the instruction, as shown in Figure 6-4.

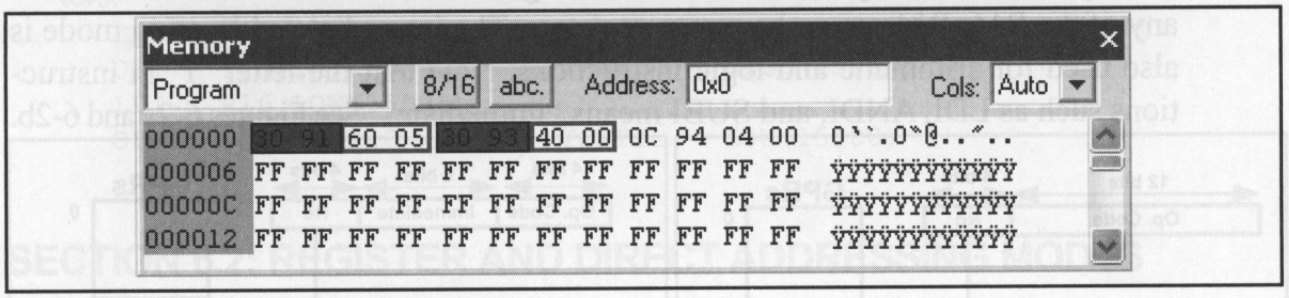


Figure 6-4. Direct Addressing Opcode

As shown in Figure 6-5a, the address field is a 16-bit address and can take values from $0000–$FFFF. Of course, it is much easier to use names instead of addresses in the program, and we have seen many examples of them in the last few chapters. It must be noted that data memory does not support immediate addressing mode. In other words, to move data into internal RAM or to I/O registers, we must first move it to a GPR (R16–R31), and then move it from the GPR to the data memory space using the STS instruction. For example, if we want to store 0x95 in memory location 0x520 we should write the following program, as there is no

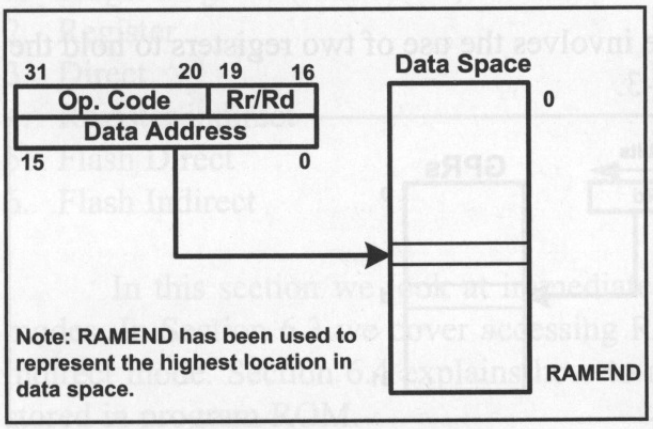


Figure 6-5a. Direct Data Addressing

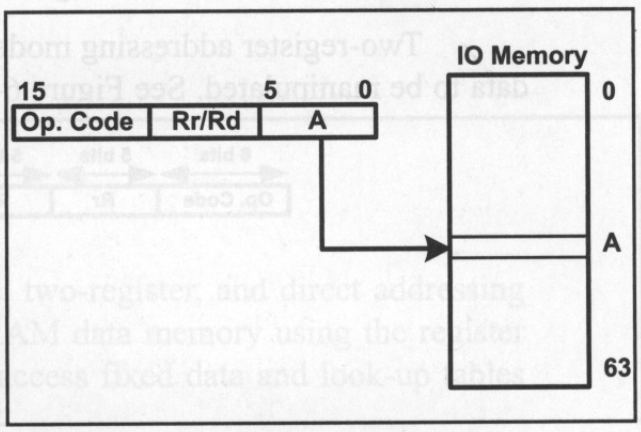


Figure 6-5b. I/O Direct Addressing

instruction for storing immediate values in memory locations:

```
LDI   R19,0x95    ;load 0x95 into R19
STS   0x520,R19   ;store R19 into data location 0x520
```

I/O direct addressing mode

To access the I/O registers there is a special mode called *I/O direct addressing mode*. The I/O direct addressing mode can address only the standard I/O registers. The IN and OUT instructions use this addressing mode. Examine the following instruction, which copies the contents of PINB to PORTC:

```
IN    R18,0x16    ;R18 = contents of location $16 (PINB)
OUT   0x15,R18    ;PORTC (location $15) = R18
```

As shown in Figure 6-5b, the address field is a 6-bit address and can take values from $00 to $3F, which is from 00 to 63 in decimal. So, it can address the entire standard I/O register memory space.

The AVR registers for Ports A, B, and so on are part of the group of registers commonly referred to as *I/O registers*. There are many I/O registers and they are widely used, as we will discuss in future chapters. The I/O registers can be accessed by their names (which is much easier) or by their addresses. For example, PINB has address 0x16, and PORTC the address $15, as shown in Table 6-4. Notice how the following pairs of instructions mean the same thing:

```
OUT 0x15,R19      ;is the same as the next instruction
OUT PORTC,R19     ;which means copy R19 into Port C

IN R26,0x16       ;is the same as the next instruction
IN R26,PINB       ;which means copy PINB into R26
```

Table 6-4: Selected ATmega32 I/O Register Addresses

Symbol	Name	I/O Address	Data Memory Addr.
PIND	Port D input pins	$10	$30
DDRD	Data Direction, Port D	$11	$31
PORTD	Port D data register	$12	$32
PINC	Port C input pins	$13	$33
DDRC	Data Direction, Port C	$14	$34
PORTC	Port C data register	$15	$35
PINB	Port B input pins	$16	$36
DDRB	Data Direction, Port B	$17	$37
PORTB	Port B data register	$18	$38
PINA	Port A input pins	$19	$39
DDRA	Data Direction, Port A	$1A	$3A
PORTA	Port A data register	$1B	$3B
SPL	Stack Pointer, Low byte	$3D	$5D
SPH	Stack Pointer, High byte	$3E	$5E

Table 6-4 lists some of the AVR I/O registers and their addresses. The following points should be noted about the addresses of I/O registers:

1. As shown in Figures 2-3 and 2-4, the addresses between $20 and $5F of the data space have been assigned to standard I/O registers in all of the AVRs. These I/O registers have two addresses: I/O address and data memory address. The I/O address is used when we use the I/O direct addressing mode, while the data memory address is used when we use the direct addressing mode; in other words, the standard I/O registers can be accessed using both the direct addressing and the I/O addressing modes. For example, the following pairs of instructions do the same thing, but the IN and OUT instructions are more efficient, as mentioned in Section 2-3:

```
OUT 0x15,R19 ;PORTC=R19 (0x15 is the I/O addr. of PORTC)
STS 0x35,R19 ;PORTC=R19 (0x35 is the data memory addr. of PORTC)

IN  R19,0x16 ;R19=PINB (0x16 is the I/O addr. of PINB)
LDS R19,0x36 ;R19=PINB (0x36 is the data memory addr. of PINB)
```

2. Some AVRs have less than 64 I/O registers. So, some locations of the standard I/O memory are not used by the I/O registers. The unused locations are reserved and must not be used by the AVR programmer.
3. Some AVRs have more than 64 I/O registers. The extra I/O registers are located above the data memory address $5F. The data memory allocated to the extra I/O registers is called *extended I/O memory*. As shown in Figure 6-2b, in the I/O direct addressing mode, the address field is a 6-bit address and can take values from $00–$3F, which is from 00 to 63 in decimal. So, it can address only the standard I/O register memory, and it cannot be used for addressing the extended I/O memory. For example, the following instruction causes an error, since the I/O address must be between 0 and $3F:

```
OUT 0x65,R19        ;illegal as the address is above $3F
```

   To access the extended I/O registers we can use the direct addressing mode. For example, in ATmega128, PORTF has the memory address of 0x62. So, the following instruction stores the contents of R20 in PORTF.

```
STS 0x62,R20        ;PORTF = R20
```

4. The I/O registers can have different addresses in different AVR microcontrollers. For example, the I/O address $2 is assigned to TWAR in the ATmega32, while the same address is assigned to DDRE in ATmega128. This means that in ATmega32, the instruction "`OUT 0x2,R20`" copies the contents of R20 to TWAR, while the same instruction, in ATmega128, copies the contents of R20 to DDRE. In other words, the same instruction can have different meanings in different AVR microcontrollers. This can cause problems if you want to run programs written for one AVR on another AVR. For example, if you have written a code for ATmega32 and you want to run it on an ATmega128, it might be necessary to change some register locations before loading it into the ATmega128.

The best way to solve this problem is to use the names of the registers instead of their addresses. For example, the instruction "OUT TWAR,R20" has the same meaning on all the AVRs. Therefore, using the names of the registers instead of their addresses makes our code more portable. See Example 6-4.

Example 6-4

Write code to send $55 to Port B. Include
(a) the register name,
(b) the I/O address, and
(c) the data memory address.

Solution:

(a)
```
LDI   R20,0xFF    ;R20 = 0xFF
OUT   DDRB,R20    ;DDRB = R20 (Port B output)
LDI   R20,0x55    ;R20 = $55
OUT   PORTB,R20   ;Port B = 0x55
```

(b) From Table 6-4, DDRB I/O address = $17 and PORTB I/O address = $18.

```
LDI   R20,0xFF    ;R20 = 0xFF
OUT   0x17,R20    ;DDRB = R20 (Port B output)
LDI   R20,0x55    ;R20 = $55
OUT   0x18,R20    ;Port B = 0x55
```

(c) From Table 6-4, DDRB data memory address = $37 and PORTB data memory address = $38.

```
LDI   R20,0xFF    ;R20 = 0xFF
STS   0x37,R20    ;DDRB = R20 (Port B output)
LDI   R20,0x55    ;R20 = $55
STS   0x38,R20    ;Port B = 0x55
```

Review Problems

1. Can the programmer of a microcontroller make up new addressing modes?
2. Show the instructions to load 1000 0000 (binary) into register SPL.
3. True or false. In immediate addressing the value comes immediately after the opcode.
4. True or false. We can access the extended I/O registers using the I/O direct addressing mode.
5. True or false. SPL is an I/O register.
6. True or false. Using the names of the registers makes the code more portable.

SECTION 6.3: REGISTER INDIRECT ADDRESSING MODE

We can use direct or register indirect addressing modes to access data stored in the data memory. In the previous section we showed how to use direct addressing mode. The register indirect addressing mode is a very important addressing mode in the AVR. This topic will be discussed thoroughly in this section.

Register indirect addressing mode

In the register indirect addressing mode, a register is used as a pointer to the data memory location. In the AVR, three registers are used for this purpose: X, Y, and Z. These are 16-bit registers allowing access to the entire 65,536 bytes of data memory space in the AVR.

Each of the registers is made by combining two specific GPRs; for example, combining R26 and R27 makes the X register. In this case R26 is the lower byte of X, and R27 is the higher byte. The Y and Z registers are made by combining R29:R28 and R31:R30, respectively. See Figure 6-6. The R26, R27, R28, R29, R30, and R31 GPRs can be referred to as XL, XH, YL,YH, ZL, and ZH, respectively. For example, "LDI XL,0x31" is the same as "LDI R26,0x31" since XL is another name for R26.

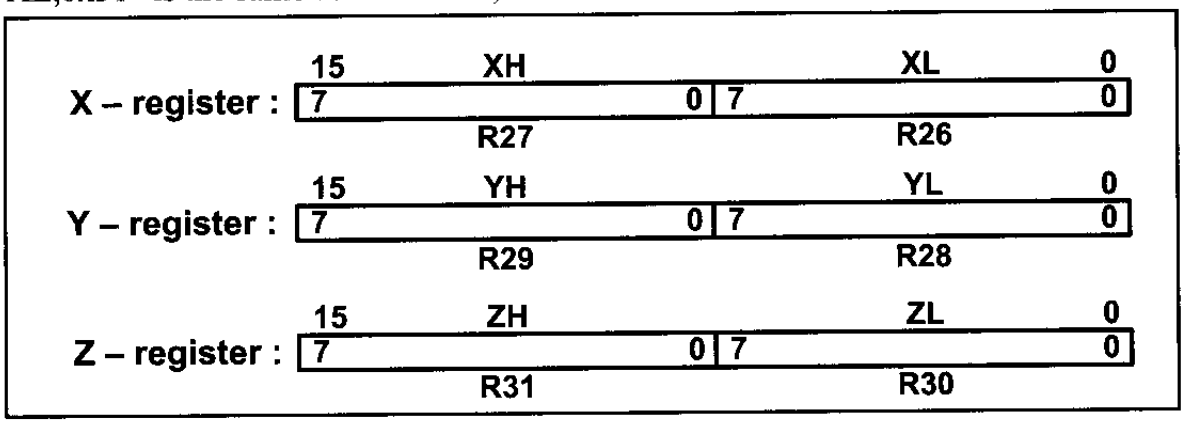


Figure 6-6. Registers X, Y, and Z

The 16-bit registers X, Y, and Z are widely used as pointers. We can use them with the LD instruction to read the value of a location pointed to by these registers. For example, the following instruction reads the value of the location pointed to by the X pointer.

```
LD    R24, X      ;load into R24 from location pointed to by X
```

For instance, the following program loads the contents of location 0x130 into R18:

```
LDI   XL, 0x30    ;load R26 (the low byte of X) with 0x30
LDI   XH, 0x01    ;load R27 (the high byte of X) with 0x1
LD    R18, X      ;copy the contents of location 0x130 to R18
```

The above program loads 0x130 into the X register; this is done by loading 0x30 into R26 (the low byte of X) and 0x1 into R27 (the high byte of X). Then it loads R18 with the contents of the location to which X points. See Figure 6-7.

The ST instruction can be used to write a value to a location to which any of the X, Y, and Z registers points. For example, the following program stores the contents of R23 into location 0x139F:

```
LDI   ZL, 0x9F    ;load 0x9F into the low byte of Z
LDI   ZH, 0x13    ;load 0x13 into the high byte of Z (Z=0x139F)
ST    X, R23      ;store the contents of location 0x139F in R23
```

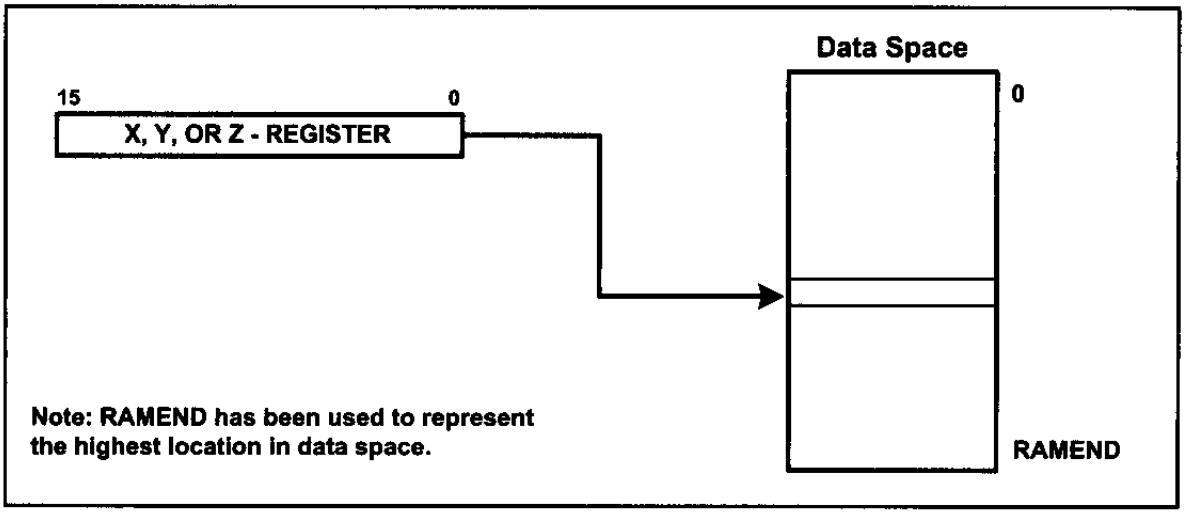


Figure 6-7. Register Indirect Addressing Mode

Advantages of register indirect addressing mode

One of the advantages of register indirect addressing mode is that it makes accessing data dynamic rather than static, as with direct addressing mode. Example 6-5 shows three cases of copying $55 into RAM locations $140 through $144 . Notice in solution (b) that two instructions are repeated numerous times. We can create a loop with those two instructions as shown in solution (c). Solution (c) is the most efficient and is possible only because of the register indirect addressing mode.

Example 6-5

Write a program to copy the value $55 into memory locations $140 through $144 using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) a loop.

Solution:

(a)
```
LDI   R17,0x55          ;load R17 with value 0x55
STS   0x140,R17         ;copy R17 to memory location 0x140
STS   0x141,R17         ;copy R17 to memory location 0x141
STS   0x142,R17         ;copy R17 to memory location 0x142
STS   0x143,R17         ;copy R17 to memory location 0x143
STS   0x144,R17         ;copy R17 to memory location 0x144
```

(b)
```
LDI   R16,0x55    ;load R16 with value 0x55
LDI   YL,0x40     ;load R28 with value 0x40 (low byte of addr.)
LDI   YH,0x1      ;load R29 with value 0x1 (high byte of addr.)
ST    Y,R16       ;copy R16 to memory location 0x140
INC   YL          ;increment the low byte of Y
ST    Y,R16       ;copy R16 to memory location 0x141
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x142
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x143
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x144
```

Example 6-5 (Cont.)

```
(c)     LDI   R16,0x5      ;R16 = 5 (R16 for counter)
        LDI   R20,0x55     ;load R20 with value 0x55 (value to be copied)
        LDI   YL,0x40      ;load YL with value 0x40
        LDI   YH,0x1       ;load YH with value 0x1
L1:     ST    Y,R20        ;copy R20 to memory pointed to by Y
        INC   YL           ;increment the pointer
        DEC   R16          ;decrement the counter
        BRNE  L1           ;loop while counter is not zero
```

Use the AVR Studio simulator to examine memory contents after the above program is run.

$140 = ($55) $141 = ($55) $142 = ($55) $143 = ($55) 144 = ($55)

In Example 6-5, we must use "INC YL" to increment the pointer because there is no such instruction as "INC Y". Looping is not possible in direct addressing mode, and that is the main difference between the direct and register indirect addressing modes. For example, trying to copy a string of data located in consecutive locations of data RAM is much more efficient and dynamic using register indirect addressing mode than using direct addressing mode. See Example 6-6.

Auto-increment and auto-decrement options for pointer registers

Because the pointer registers (X, Y, and Z) are 16-bit registers, they can go from $0000 to $FFFF, which covers the entire 64K memory space of the AVR. Using the "INC ZL" instruction to increment the pointer can cause a problem when an address such as $5FF is incremented. The instruction "INC ZL" will not propagate the carry into the ZH register. The AVR gives us the options of auto-increment and auto-decrement for pointer registers to overcome this problem. The syntax used for the LD instruction in such cases is shown in Table 6-5.

Table 6-5: AVR Auto-Increment/Decrement of Pointer Registers for LD Instruction

Instruction		Function
LD	Rn,X	After loading location pointed to by X, the X stays the same.
LD	Rn,X+	After loading location pointed to by X, the X is incremented.
LD	Rn,-X	The X is decremented, then the location pointed to by X is loaded.
LD	Rn,Y	After loading location pointed to by Y, the Y stays the same.
LD	Rn,Y+	After loading location pointed to by Y, the Y is incremented.
LD	Rn,-Y	The Y is decremented, then the location pointed to by Y is loaded.
LDD	Rn,Y+q	After loading location pointed to by Y+q, the Y stays the same.
LD	Rn,Z	After loading location pointed to by Z, the Z stays the same.
LD	Rn,Z+	After loading location pointed to by Z, the Z is incremented.
LD	Rn,-Z	The Z is decremented, then the location pointed to by Z is loaded.
LDD	Rn,Z+q	After loading location pointed to by Z+q, the Z stays the same.

Note: This table shows the syntax for the LD instruction, but it works for all such instructions. The auto-decrement or auto-increment affects the entire 16 bits of the pointer register and has no effect on the status register. This means that pointer register going from FFFF to 0000 will not raise any flag.

Example 6-6

Assume that RAM locations $90–$94 have a string of ASCII data, as shown below.
$90 = ('H') $91 = ('E') $92 = ('L') $93 = ('L') $94 = ('O')

Write a program to get each character and send it to Port B one byte at a time. Show the program using:
(a) Direct addressing mode.
(b) Register indirect addressing mode.

Solution:

(a) Using direct addressing mode

```
      LDI   R20,0xFF
      OUT   DDRB,R20          ;make Port B an output
      LDS   R20,0x90          ;R20 = contents of location 0x90
      OUT   PORTB,R20         ;PORTB = R20
      LDS   R20,0x91          ;R20 = contents of location 0x91
      OUT   PORTB,R20         ;PORTB = R20
      LDS   R20,0x92          ;R20 = contents of location 0x92
      OUT   PORTB,R20         ;PORTB = R20
      LDS   R20,0x93          ;R20 = contents of location 0x93
      OUT   PORTB,R20         ;PORTB = R20
      LDS   R20,0x94          ;R20 = contents of location 0x94
      OUT   PORTB,R20         ;PORTB = R20
```

(b) Using register indirect addressing mode

```
      LDI   R16,0x5           ;R16=0x5 (R16 for counter)
      LDI   R20,0xFF
      OUT   DDRB,R20          ;make Port B an output
      LDI   ZL,0x90           ;the low byte of address (ZL = 0x90)
      LDI   ZH,0x0            ;the high byte of address (ZH = 0x0)
L1:   LD    R20,Z             ;read from location pointed to by Z
      INC   ZL                ;increment pointer
      OUT   PORTB,R20         ;send to PortB the contents of R20
      DEC   R16               ;decrement counter
      BRNE  L1                ;if R16 is not zero go to L1
```

When simulating the above program on the AVR Studio, make sure that memory locations $90–$94 have the message "HELLO".

See Figures 6-8 and 6-9. Then, see Examples 6-7 through 6-9.

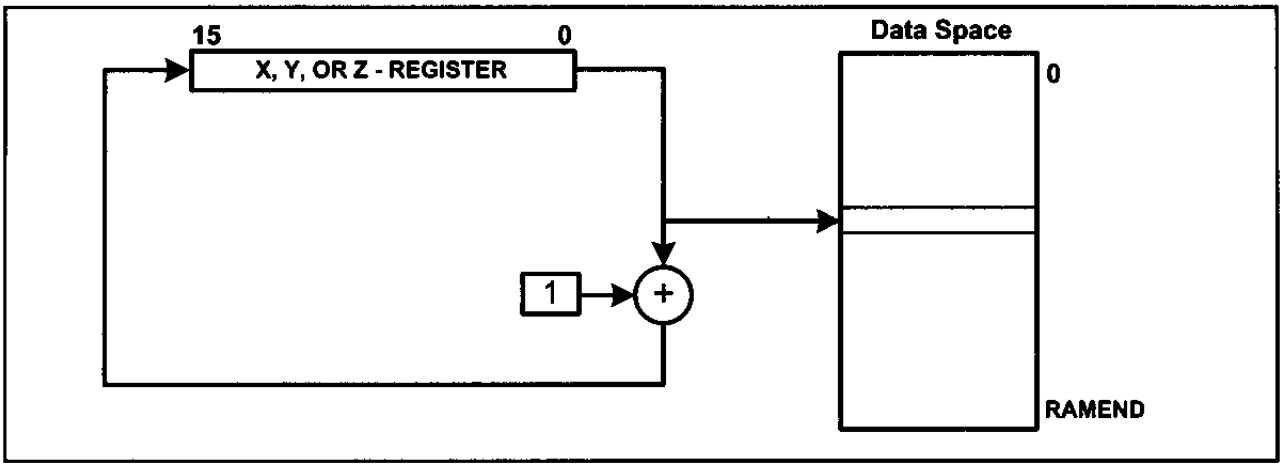


Figure 6-8. Register Indirect Addressing with Post-increment

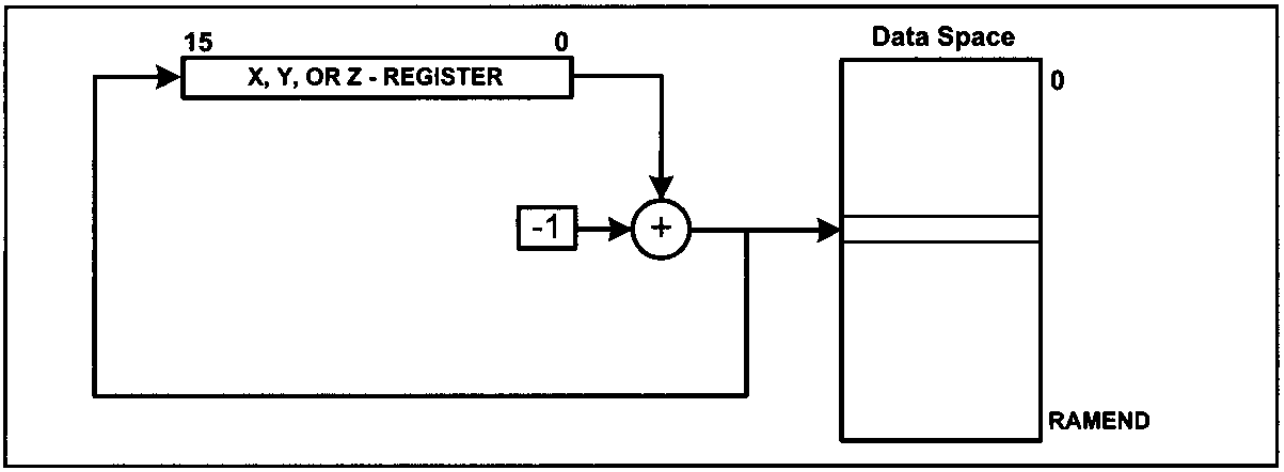


Figure 6-9. Register Indirect Addressing with Pre-decrement

Example 6-7

Write a program to clear 16 memory locations starting at data memory address $60. Use the following:
(a) INC Rn
(b) Auto-increment

Solution:

```
(a)    LDI   R16, 16      ;R16 = 16 (counter value)
       LDI   XL, 0x60     ;XL = the low byte of address
       LDI   XH, 0x00     ;XH = the high byte of address
       LDI   R20, 0x0     ;R20 = 0
L1:    ST    X, R20       ;clear location X points to
       INC   XL           ;increment pointer
       DEC   R16          ;decrement counter
       BRNE  L1           ;loop until counter = zero

(b)    LDI   R16, 16      ;R16 = 16 (counter value)
       LDI   XL, 0x60     ;the low byte of X = 0x60
       LDI   XH, 0x00     ;the high byte of X = 0
       LDI   R20, 0x0     ;R20 = 0
L1:    ST    X+, R20      ;clear location X points to
       DEC   R16          ;decrement counter
       BRNE  L1           ;loop until counter = zero
```

Example 6-8

Assume that data memory locations $240–$243 have the following hex data. Write a program to add them together and place the result in locations $220 and $221.

$240 = ($7D) $241 = ($EB) $242 = ($C5) $243 = ($5B)

Solution:

```
      .INCLUDE "M32DEF.INC"
      .EQU L_BYTE = 0x220     ;RAM loc for L_Byte
      .EQU H_BYTE = 0x221     ;RAM loc for H_Byte
      LDI   R16,4
      LDI   R20,0
      LDI   R21,0
      LDI   XL, 0x40    ;the low byte of X = 0x40
      LDI   XH, 0x02    ;the high byte of X = 02
L1:   LD    R22, X+     ;read contents of location where X points to
      ADD   R20, R22
      BRCC  L2          ;branch if C = 0
      INC   R21         ;increment R21
L2:   DEC   R16         ;decrement counter
      BRNE  L1          ;loop until counter is zero
      ST    L_BYTE, R20 ;store the low byte of the result in $220
      ST    H_BYTE, R21 ;store the high byte of the result in $221
```

Example 6-9

Write a program to copy a block of 5 bytes of data from data memory locations starting at $130 to RAM locations starting at $60.

Solution:

```
      LDI   R16, 16     ;R16 = 16 (counter value)
      LDI   XL, 0x30    ;the low byte of address
      LDI   XH, 0x01    ;the high byte of address
      LDI   YL, 0x60    ;the low byte of address
      LDI   YH, 0x00    ;the high byte of address
L1:   LD    R20, X+     ;read where X points to
      ST    Y+, R20     ;store R20 where Y points to
      DEC   R16         ;decrement counter
      BRNE  L1          ;loop until counter = zero
```

Before we run the above program.

```
130 = ('H') 131 = ('E') 132 = ('L') 133 = ('L') 134 = ('O')
```

After the program is run, the addresses $60–$64 have the same data as $130–$134.

```
130 = ('H') 131 = ('E') 132 = ('L') 133 = ('L') 134 = ('O')
60 = ('H')  61 = ('E')  62 = ('L')  63 = ('L')  64 = ('O')
```

To see an example of how to use all three pointer registers, study and simulate Example 6-10.

Example 6-10

Two multibyte numbers are stored in locations $130–$133 and $150–$153. Write a program to add the multibyte numbers and save the result in address $160–$163.

```
      $C7659812
+     $2978742A
```

Solution:

```
      .INCLUDE "M32DEF.INC"
      LDI   R16, 4            ;R16 = 4 (counter value)
      LDI   XL, 0x30
      LDI   XH, 0x1           ;load pointer. X = $130
      LDI   YL, 0x50
      LDI   YH, 0x1           ;load pointer. Y = $150
      LDI   ZL, 0x60
      LDI   ZH, 0x1           ;load pointer. Z = $160
      CLC                     ;clear carry
L1:   LD    R18, X+           ;copy memory to R18 and INC X
      LD    R19, Y+           ;copy memory to R19 and INC Y
      ADC   R18,R19           ;R18 = R18 + R19 + carry
      ST    Z+,R18            ;store R18 in memory and INC Z
      DEC   R16               ;decrement R16 (counter)
      BRNE  L1                ;loop until counter = zero
```

Before the addition we have:

```
      MSByte                                  LSByte
      133 = ($C7) 132 = ($65) 131 = ($98)  130 = ($12)
      153 = ($29) 152 = ($78) 151 = ($74)  150 = ($2A)
```

After the addition we have:

```
      163 = ($F0) 162 = ($DE) 161 = (0C)  160 = (3C)
```

Notice that we are using the little endian convention of storing a low byte to a low address, and a high byte to a high address. Single-step the program in AVR Studio and examine the pointer registers and memory contents to gain insight into register indirect addressing mode.

Register indirect with displacement

Suppose we want to read a byte that is a few bytes higher than where the Z register points to. To do so we can increment the Z register so that it points to the desired location and then read it. But there is an easier way; we can use the register indirect with displacement. In this addressing mode a fixed number is added to the Z register. For example, if we want to read from the location that is 5 bytes after the location to which Z points, we can write the following instruction:

```
LDD    R20, Z+5      ;load from Z+5 into R20
```

The general format of the instruction is as follows:

```
LDD    Rd, Z+q       ;load from Z+q into Rd
```

where q is a number between 0 to 63, and Rd is any of the general purpose registers. See Figure 6-10.

To store a byte of data in a data memory location using the register indirect

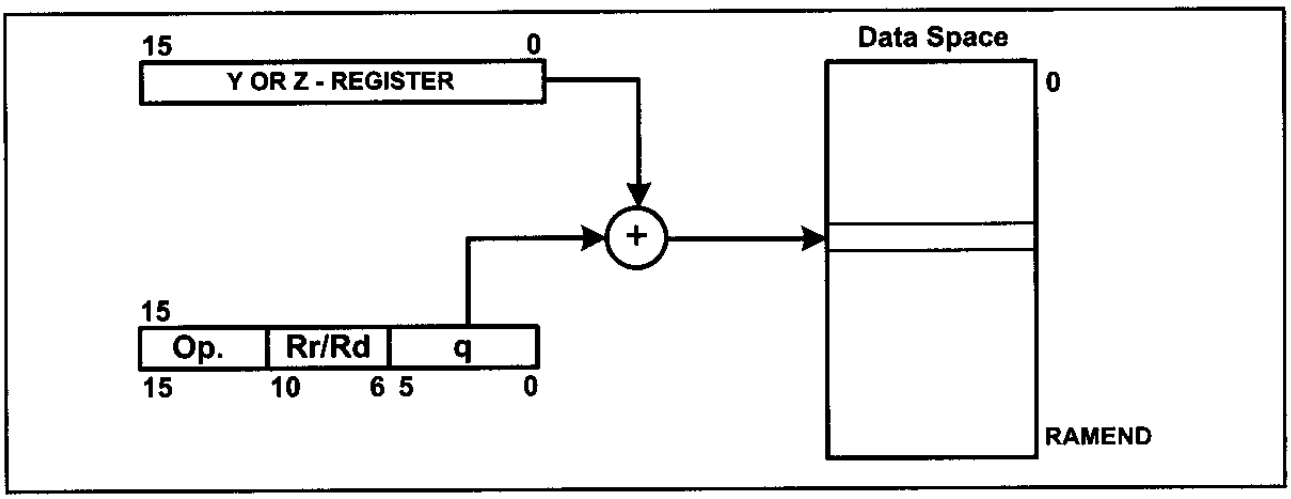


Figure 6-10. Register Indirect with Displacement

with displacement addressing mode we can use STD (Store with Displacement). The instruction is as follows:

```
STD    Z+q,Rr        ;store Rr into location Z+q
```

For example, the following instruction writes the contents of R20 into the location that is five bytes away from where Z points to:

```
STD    Z+5,R20       ;store R20 into location Z+5
```

To see an example of how to use the addressing mode, see Example 6-11.

Example 6-11

Write a function that adds the contents of three continuous locations of data space and stores the result in the first location. The Z register should point to the first location before the function is called.

Solution:

```
.INCLUDE "M32DEF.INC"
      LDI   R16,HIGH(RAMEND)  ;initialize the stack pointer
      OUT   SPH,R16
      LDI   R16,LOW(RAMEND)
      OUT   SPL,R16
      LDI   ZL,0x00           ;initialize the Z register
      LDI   ZH,2
      CALL  ADD3LOC           ;call add3loc
HERE: JMP   HERE              ;loop forever
ADD3LOC:
      LDI   R21,0             ;R21 = 0
      LD    R20,Z             ;R20 = contents of location  Z
      LDD   R16,Z+1           ;R16 = contents of location Z+1
      ADD   R20,R16           ;R20 = R20 + R16
      BRCC  L1                ;branch if carry cleared
      INC   R21               ;increment R21 if carry occurred
L1:   LDD   R16,Z+2           ;R16 = contents of location Z+2
      ADD   R20,R16           ;R20 = R20 + R16
      BRCC  L2                ;branch if carry cleared
      INC   R21               ;increment R21
L2:   ST    Z,R20             ;store R20 into location Z
      STD   Z+1,R21           ;store R21 into location Z+1
      RET
```

Review Questions

1. The instruction "`LD R19,0x95`" uses _____ addressing mode.
2. Which register is the low byte of the X register?
3. The pointer registers are _____-bit registers.
4. Write a program that adds 2 to the contents of locations $90–$9A and stores the results in locations $200–$20A.
5. Which registers may be used for register indirect addressing mode if the data is in the data memory?

SECTION 6.4: LOOK-UP TABLE AND TABLE PROCESSING

So far, we have seen that the AVR has a maximum of 8M bytes of code (program) space and 64K of data memory space. We can use the code space to store fixed data. In this section we discuss how to access fixed data residing in the program ROM space of the AVR. First we examine how to store fixed data in the program ROM space using the `.DB` (define byte) directive.

.DB (define byte) and fixed data in program ROM

The `.DB` data directive is widely used to allocate ROM program (code) memory in byte-sized chunks. In other words, `.DB` is used to define an 8-bit fixed data. When `.DB` is used to define fixed data, the numbers can be in decimal, binary, hex, or ASCII formats. The `.DB` directive is widely used to define ASCII strings.

See Example 6-12. In Example 6-12 notice that each location of program memory is 2 bytes, whereas the `.DB` directive allocates byte-sized chunks. If we allocate a few bytes of data using the `.DB` directive, the first byte goes to the low byte of ROM location; the second byte goes to the high byte of ROM location; the third byte goes to the low byte of the next location of program ROM; and so on. In the cases in which we allocate an odd number of ROM locations using `.DB`, the assembler will automatically make the number of allocated locations even by placing a zero into the high byte of the last location. In other words, even if we allocate a fraction of a program ROM location, the assembler will allocate the whole location and load the unused part of it with zero. In Example 6-12 notice also that we must use single quotes (') for a single character and double quotes (") for a string.

AVR assembly also allows the use of `.DW` in place of `.DB` to define values greater than 255 (0xFF) but not larger than 65,535 (0xFFFF). See Example 6-13.

Reading table elements in the AVR

Example 6-12 showed how to place fixed data into program ROM. Now, we need to have a register pointer to point to the data to be fetched from the code space. The Z register can be used for this purpose. For this reason we can call it register indirect flash addressing mode. This is an addressing mode widely used to access data elements located in the program space of the AVR. In AVR terminology, there are two register indirect flash addressing modes: *program memory constant addressing* and *program memory addressing with post-increment*. In the pro-

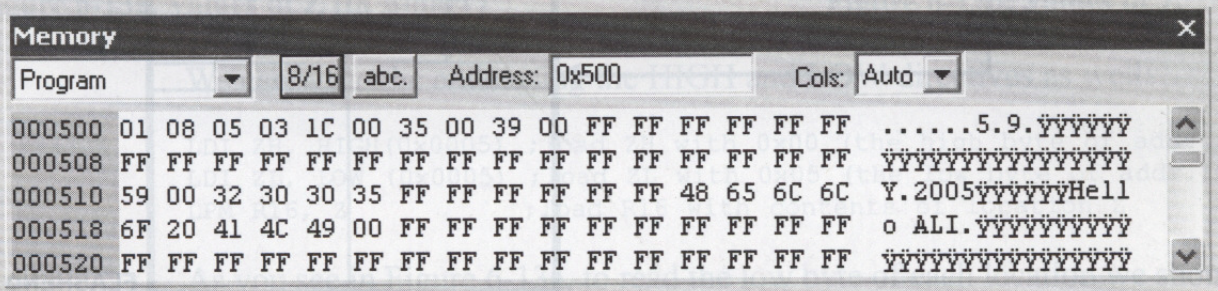
Example 6-12

Assume that we have burned the following fixed data into the program ROM of an AVR chip. Give the contents of each ROM location starting at $500. See Appendix F for the hex values of the ASCII characters.

```
;MY DATA IN FLASH ROM
      .ORG $500
DATA1: .DB 1,8,5,3
DATA2: .DB 28              ;DECIMAL(1C in hex)
DATA3: .DB 0b00110101      ;BINARY (35 in hex)
DATA4: .DB 0x39            ;HEX
      .ORG 0x510
DATA4: .DB 'Y'             ;single ASCII char
DATA5: .DB '2','0','0','5';ASCII numbers
      .ORG $516
DATA6: .DB "Hello ALI"  ;ASCII string
```

Solution:

DATA1 has four bytes of data. The "`.ORG $500`" directive causes the assembler to put the first byte of DATA1 in the low byte of location $500. The second byte of DATA1, which is 8, goes to the high byte of location $500; the third byte goes to the low byte of location $501, and the fourth byte goes to the high byte of location $501.
DATA2 will be located after DATA1, in location $502 of memory. As DATA2 has one byte of data and each location of program is 2 bytes wide, the assembler puts zero in the high byte of location $502.



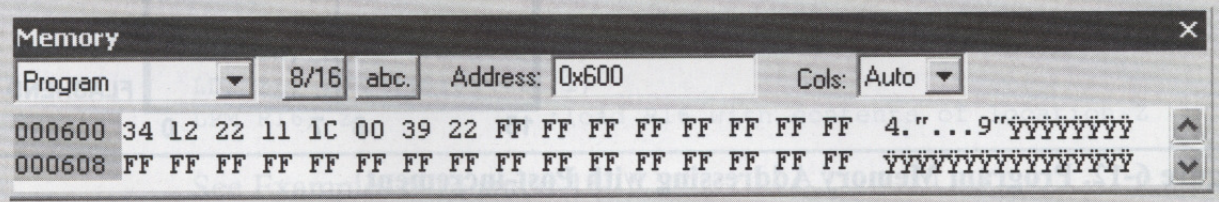
Example 6-13

Give the contents of each ROM location starting at $600.

```
      .ORG $600
DATA1: .DW 0x1234,0x1122
DATA2: .DW 28              ;DECIMAL (001C in hex)
DATA3: .DW 0x2239          ;HEX
```

Solution:

Since AVR is little endian, the low byte of 0x1234, which is 0x34, goes to the low byte of location $600, and its high byte goes to the high byte.



gram memory constant addressing mode, the content of Z does not change when the instruction is executed, which is why it is called *constant addressing*; whereas in the program memory addressing *with post-increment*, the content of Z increments after each execution.

"LPM Rn, Z" uses program memory constant addressing mode, while "LPM Rn,Z+" uses program memory addressing with post-increment. (See Table 6-6.) In Figures 6-11 and 6-12 you see the addressing modes.

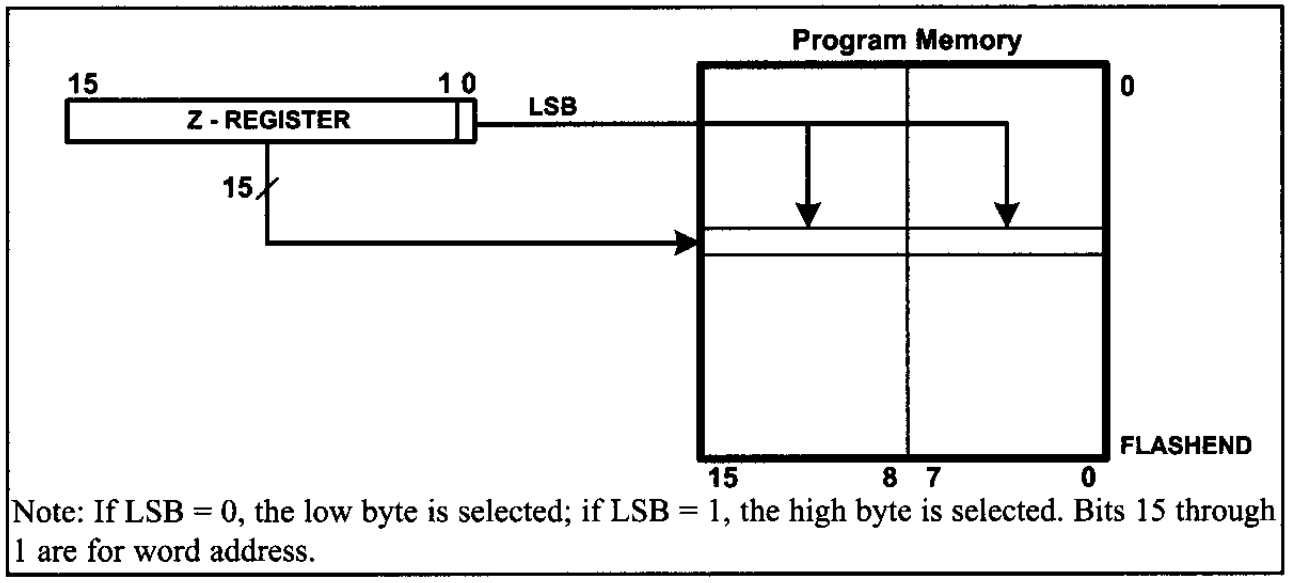
There is a group of AVR instructions designed for table processing. Table 6-6 shows the instructions for table reading in the AVR.

The "LPM Rn, Z" instruction loads the byte pointed to by Z into the Rn. As you know, in the AVR, each location of the program memory is 2 bytes. So, we should mention if we want to read the low byte or the high byte. The least significant bit (LSB) of the Z register indicates whether the low byte or the high byte should be read. If LSB = 0, then the low byte will be read; otherwise, the high byte

Table 6-6: AVR Table Read Instructions

Instruction	Function	Description
LPM Rn,Z	Load from Program Memory	After read, Z stays the same
LPM Rn,Z+	Load from Program Memory with post-inc.	Reads and increments Z

Note: The byte of data is read into the Rn register from code space pointed to by Z.



Note: If LSB = 0, the low byte is selected; if LSB = 1, the high byte is selected. Bits 15 through 1 are for word address.

Figure 6-11. Program Memory Constant Addressing

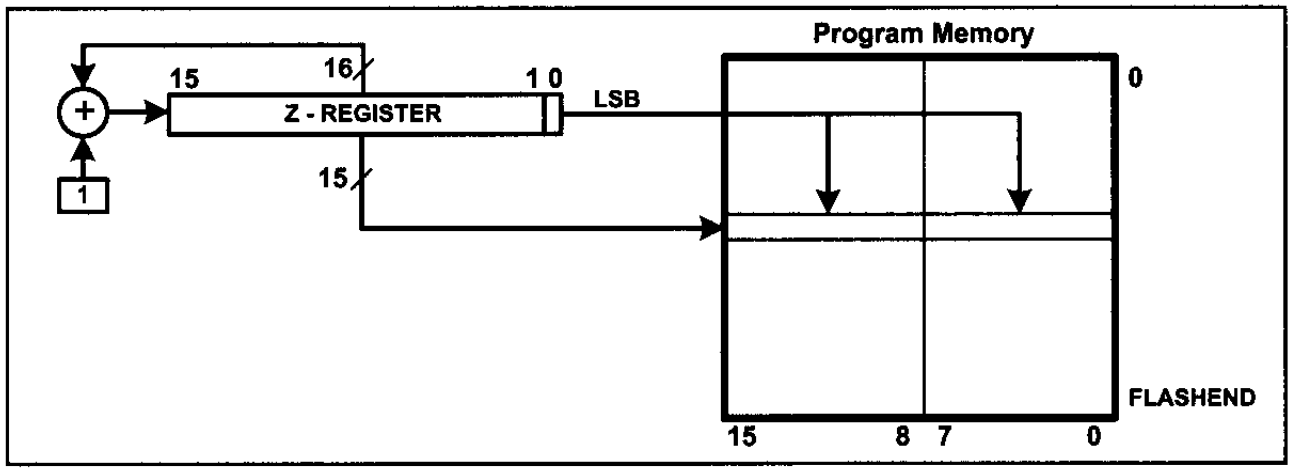


Figure 6-12. Program Memory Addressing with Post-increment

will be read. The other bits of the Z register (bit 1 to bit 15) represent the address of the location that should be read. See Figure 6-11.

Figure 6-13b shows the value that should be loaded into the Z register in order to address each byte of the program memory. For example, to address the low byte of location $0002, we should load the Z register with $0005, as shown below:

```
LDI ZH, 0x00        ;load ZH with 0x00 (the high byte of addr.)
LDI ZL, 0x05        ;load ZL with 0x05 (the low byte of addr.)
LPM R16, Z          ;load R16 with contents of location Z
```

Low	High	Address
0000 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 0000
0000 0000 0000 0010	0000 0000 0000 0011	0000 0000 0000 0001
0000 0000 0000 0100	0000 0000 0000 0101	0000 0000 0000 0010
0000 0000 0000 0110	0000 0000 0000 0111	0000 0000 0000 0011
0000 0000 0000 1000	0000 0000 0000 1001	0000 0000 0000 0100
0000 0000 0000 1010	0000 0000 0000 1011	0000 0000 0000 0101
⋮	⋮	
1111 1111 1111 1100	1111 1111 1111 1101	0111 1111 1111 1110
1111 1111 1111 1110	1111 1111 1111 1111	0111 1111 1111 1111

Figure 6-13a. Values of Z (in Binary)

Low	High	Address
$0000	$0001	$0000
$0002	$0003	$0001
$0004	$0005	$0002
$0006	$0007	$0003
$0008	$0009	$0004
$000A	$000B	$0005
$FFFC	$FFFD	$7FFE
$FFFE	$FFFF	$7FFF

Figure 6-13b. Values of Z

We can write the code using the HIGH and LOW directives as well:

```
LDI ZH, HIGH(0x0005) ;load ZH with 0x00 (the high byte of addr.)
LDI ZL, LOW (0x0005) ;load ZL with 0x05 (the low byte of addr.)
LPM R16, Z           ;load R16 with contents of location Z
```

As you see in Figure 6-13a, to read the low byte of each location we should shift the address of that location one bit to the left. For instance, to access the low byte of location 0b00000101, we should load Z with 0b**000001010**. To read the high byte, we shift the address to the left and we set bit 0 to one.

We can shift the address using the << directive as well. For example, the following program reads the low byte of location $100:

```
LDI ZH, HIGH($100<<1) ;load ZH with the high byte of addr.
LDI ZL, LOW ($100<<1) ;load ZL with the low byte of addr.
LPM R16, Z            ;load R16 with contents of location Z
```

If we *OR* a number with 1, its bit 0 will be set. Thus, the following program reads the high byte of location $100.

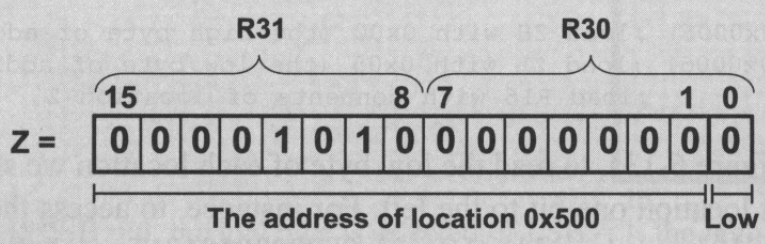
```
LDI ZH, HIGH(($100<<1)|1)
LDI ZL, LOW (($100<<1)|1)
LPM R16, Z            ;load R16 with contents of location Z
```

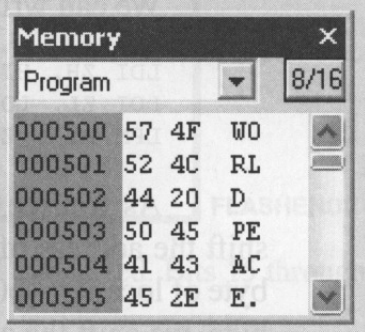
See Examples 6-14 and 6-15.

Example 6-14

In this program, assume that the phrase "WORLD PEACE." is burned into ROM locations starting at $500, and the program is burned into ROM locations starting at 0. Analyze how the program works and state where "WORLD PEACE." is stored after this program is run.

```
        .ORG   $0000                ;burn into ROM starting at 0
        LDI    R20,0xFF
        OUT    DDRB,R20             ;make PB an output
        LDI    ZL,LOW(MYDATA<<1)          ;ZL = 0x00 (low byte of address)
        LDI    ZH,HIGH(MYDATA<<1)         ;ZH = 0x05 (high byte of address)
        LPM    R20,Z
        OUT    PORTB,R20     ;send it to port B
        INC    ZL            ;ZL = 01 pointing to next byte (A01)
        LPM    R20,Z         ;load R20 with 'W' (char pointed to by Z)
        OUT    PORTB,R20     ;send it to port B
        INC    ZL            ;ZL = 02 pointing to next byte (A02)
        LPM    R20,Z         ;load R20 with 'O' (char pointed to by Z)
        OUT    PORTB,R20     ;send it to port B
        INC    ZL            ;ZL = 03 pointing to next byte (A03)
        LPM    R20,Z         ;load R20 with 'R' (char pointed to by Z)
        OUT    PORTB,R20     ;send it to port B
        INC    ZL            ;ZL = 04 pointing to next byte (A04)
        LPM    R20,Z         ;load R20 with 'L' (char pointed to by Z)
        OUT    PORTB,R20     ;send it to port B
        INC    ZL            ;ZL = 05 pointing to next byte (A05)
        LPM    R20,Z         ;load R20 with 'D' (char pointed to by Z)
        OUT    PORTB,R20     ;send it to port B
HERE:   RJMP   HERE          ;stay here forever
        .ORG   $500  ;data is burned into program space starting at $500
MYDATA: .DB "WORLD PEACE."
```





Solution:

In the above program, ROM locations $500–$505 have the following contents.

```
$500 (Low byte) = ('W')        $500 (High byte) = ('O')
$501 (Low byte) = ('R')        $501 (High byte) = ('L')
$502 (Low byte) = ('D')        $502 (High byte) = (' ')
$503 (Low byte) = ('P')        $503 (High byte) = ('E')
$504 (Low byte) = ('A')        $504 (High byte) = ('C')
$505 (Low byte) = ('E')        $505 (High byte) = ('.')
```

We start with Z = $0A00 (R31:R30 = $A00). The instruction "`LPM R20,Z`" loads R20 with the contents of the low byte of ROM location $500. Register R20 contains $57, the ASCII value for 'W'. This is loaded to Port B. Next, ZL is incremented to make Z = $A01. The LPM instruction will get the contents of the high byte of ROM location $500, which is character 'O'. After this program is run, we send the ASCII values for the characters 'W', 'O', 'R', 'L', and 'D' to Port B one character at a time. The loop version of this program is given in the next example.

Example 6-15

Assuming that program ROM space starting at $500 contains "WORLD PEACE.". write a program to send all the characters to Port B one byte at a time.

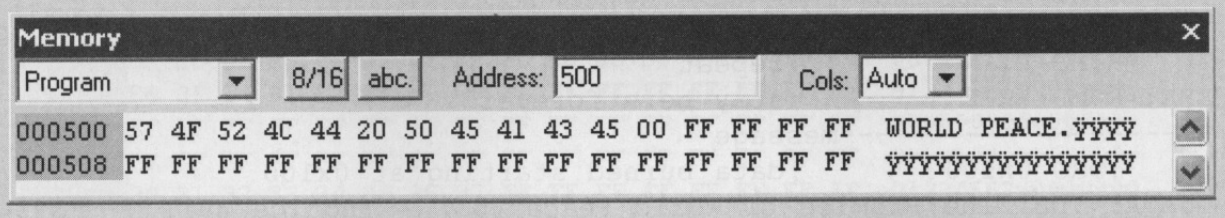
Solution:

```
(a) This method uses a counter

.ORG   $0000                    ;burn into ROM starting at 0
.INCLUDE "M32DEF.INC"
       LDI    R16,11
       LDI    R20,0xFF
       OUT    DDRB,R20          ;make PB an output
       LDI    ZH,HIGH(MYDATA<<1);ZH = high byte of addr.
       LDI    ZL,LOW(MYDATA<<1) ;ZL = low byte of addr.
L1:    LPM    R20,Z
       OUT    PORTB,R20         ;send it to Port B
       INC    ZL                ;pointing to next byte
       DEC    R16               ;decrement counter
       BRNE   L1                ;repeat if counter not zero
HERE:  RJMP   HERE              ;stay here forever
;-----------------------------------------------
;data is burned into code (program) space starting at $500
       .ORG   0x500
MYDATA DB     "WORLD PEACE."


(b) This method uses null char for end of string

.ORG   $0000                    ;burn into ROM starting at 0
.INCLUDE "M32DEF.INC"
       LDI    R20,0xFF
       OUT    DDRB,R20          ;make PB an output
       LDI    ZH,HIGH(MYDATA<<1);ZH = high byte of addr.
       LDI    ZL,LOW(MYDATA<<1) ;ZL = low byte of addr.
L1:    LPM    R20,Z             ;bring in next byte
       CPI    R20,0             ;compare R20 with 0
       BREQ   HERE              ;branch if equal
       OUT    PORTB,R20         ;send it to Port B
       INC    ZL                ;pointing to next byte
       RJMP   L1                ;repeat
HERE:  RJMP   HERE              ;stay here forever
;-----------------------------------------------
;data is burned into code (program) space starting at $500
       .ORG 0x500
MYDATA: .DB "WORLD PEACE",0     ;notice null
```



Auto-increment option for Z

Using the "INC ZL" instruction to increment the pointer can cause a problem when an address such as $5FF is incremented. The carry will not propagate into ZH. The AVR gives us the option of LPM Rn, Z+ (load program memory with post-increment) as shown in Table 6-6. See Examples 6-16 and 6-17.

Example 6-16

Repeat Example 6-15, using auto-increment.

Solution:

```
.ORG   $0000              ;burn into ROM starting at 0
.INCLUDE "M32DEF.INC"
       LDI    R20,0xFF
       OUT    DDRB,R20           ;make PB an output
       LDI    ZH,HIGH(MYDATA<<1)       ;ZH = high byte of addr.
       LDI    ZL,LOW(MYDATA<<1)        ;ZL = low byte of addr.
L1:    LPM    R20,Z+             ;bring in next byte and inc. Z
       CPI    R20,0        ;compare R20 with 0
       BREQ   HERE         ;branch if equal
       OUT    PORTB,R20    ;send it to Port B
       RJMP   L1           ;repeat
HERE:  RJMP   HERE         ;stay here forever
;data is burned into code (program) space starting at $500
       .ORG   0x500
MYDATA: .DB "WORLD PEACE",0          ;notice null
```

Example 6-17

Assume that ROM space starting at $100 contains the message "The Promise of World Peace". Write a program to bring this message into the CPU one byte at a time and place the bytes in RAM locations starting at $140.

Solution:

```
.EQU  RAM_BUF = 0x140
.ORG $0000                      ;burn into ROM starting at 0
.INCLUDE "M32DEF.INC"
       LDI R20,0xFF
       OUT DDRB,R20             ;make PB an output
       LDI ZH,HIGH(MYDATA<<1)   ;ZH = high byte of addr.
       LDI ZL,LOW(MYDATA<<1)    ;ZL = low byte of addr.
       LDI XH,HIGH(RAM_BUF)     ;XH = $1, high byte of RAM addr.
       LDI XL,LOW(RAM_BUF)      ;XL = $40, low byte of RAM addr.
L1:    LPM   R20,Z+             ;bring in next byte and increment Z
       CPI   R20,0        ;compare R20 with 0
       BREQ HERE          ;branch if end of string
       ST    X+,R20       ;store R20 in RAM and increment X
       RJMP L1            ;repeat
HERE:  RJMP HERE          ;stay here forever
;----------------------message
       .ORG 0x100         ;data burned starting at 0x100
MYDATA: .DB "The Promise of World Peace",0     ;notice null
```

Look-up table

The look-up table is a widely used concept in microcontroller programming. It allows access to elements of a frequently used table with minimum operations. As an example, assume that for a certain application we need $4 + x^2$ values in the range of 0 to 9. We can use a look-up table instead of calculating the values, which takes some time. In the AVR, to get the table element we add the index to the address of the look-up table. This is shown in Examples 6-18 through 6-20.

Example 6-18

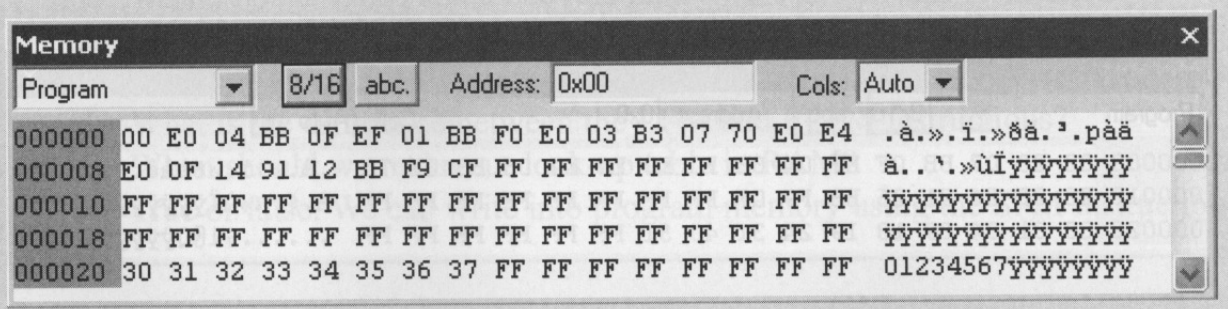
Assume that the lower three bits of Port C are connected to three switches. Write a program to send the following ASCII characters to Port D based on the status of the switches.

000	'0'
001	'1'
010	'2'
011	'3'
100	'4'
101	'5'
110	'6'
111	'7'

Solution:

```
.ORG 0
.INCLUDE "M32DEF.INC"
      LDI   R16,0x0
      OUT   DDRC,R16                ;DDRC = 0x00 (port C as input)
      LDI   R16,0xFF
      OUT   DDRD,R16                ;DDRD = 0xFF (port D as output)
      LDI   ZH,HIGH(ASCII_TABLE<<1) ;ZH = high byte of addr.
BEGIN:IN    R16,PINC                ;read from port C into R16
      ANDI  R16,0b00000111          ;mask upper 5 bits
      LDI   ZL,LOW(ASCII_TABLE<<1)  ;ZL = the low byte of addr.
      ADD   ZL,R16                  ;add PINC to the addr
      LPM   R17,Z                   ;get ASCII from look-up table
      OUT   PORTD,R17
      RJMP  BEGIN

;look-up table for ASCII numbers 0-7
.ORG 0x20
ASCII_TABLE:
      .DB '0','1','2','3','4','5','6','7'
```



Example 6-19

Write a program to get the x value from Port B and send x^2 to Port C. Assume that PB3–PB0 has the x value of 0–9. Use a look-up table instead of a multiply instruction.

What is the value of Port C if we have 9 at Port B?

Solution:

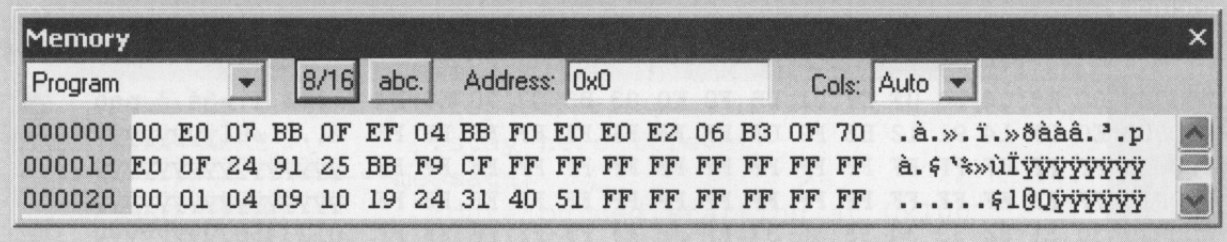
```
.INCLUDE "M32DEF.INC"
      .ORG 0
      LDI   R16,0x00
      OUT   DDRB,R16     ;DDRB = 0x00 (Port B as input)
      LDI   R16,0xFF
      OUT   DDRC,R16     ;DDRC = 0xFF (Port C as output)

      LDI   ZH,HIGH(XSQR_TABLE<<1)  ;ZH = high byte of addr.
L1:   LDI   ZL,LOW(XSQR_TABLE<<1)   ;ZL = low byte of addr.
      IN    R16,PINB     ;read from Port B into R16
      ANDI  R16,0x0F     ;mask upper bits
      ADD   ZL,R16
      LPM   R18,Z        ;get x^2 from the look-up table
      OUT   PORTC,R18
      RJMP  L1

;look-up table for square of numbers 0-9
.ORG 0x10
XSQR_TABLE:
      .DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

From the screenshot below, notice that location 0020 has 0, the square of 0. Location 0021 has 01, the square of 1. Location 0022 has 04, the square of 2. Location 0023 has 09, the square of 3. Location 0024 has $10, the square of 4 ($4 \times 4 = 16 = \$10$) and so on. Notice that the Memory window shows the low bytes and the high bytes of each program memory location separately, and the locations are addressed the same way as the Z register. This simplifies debugging since we usually use the *Memory* window to examine data. If we want to examine the instruction, we would better use the *Disassembly* window.

If we have 9 at Port B, then Port C will have $51, which is the hex value of decimal 81 ($9^2 = 81$).



Example 6-20

Write a program to get the x value from Port B and send $x^2 + 2x + 3$ to Port C. Assume PB3–PB0 has the x value of 0–9. Use a look-up table instead of a multiply instruction.

Solution:

```
      .ORG 0
      .INCLUDE "M32DEF.INC"
      LDI R16,0x00
      OUT   DDRB,R16     ;DDRB = 0x00 (Port B as input)
      LDI   R16,0xFF
      OUT   DDRC,R16     ;DDRC = 0xFF (Port C as output)

      LDI   ZH,HIGH(TABLE<<1) ;ZH = high byte of addr.
L1:   LDI   ZL,LOW(TABLE<<1)  ;ZL = low byte of addr.
      IN    R16,PINB     ;read from Port B into R16
      ANDI  R16,0x0F     ;mask upper bits
      ADD   ZL,R16
      LPM   R18,Z        ;get x2 + 2x + 3 from the look-up table
      OUT   PORTC,R18
      RJMP L1

.ORG 0x10
TABLE:
      .DB 3, 6, 11, 18, 27, 38, 51, 66, 83, 102
```

Accessing a look-up table in RAM

The look-up table elements can also be in RAM instead of ROM. Sometimes we need to bring in the elements of the look-up table from RAM because the elements are dynamic and can change. In the AVR, we can do that using the pointers.

Writing table elements in AVR

In AVR we also have the SPM instruction, which allows us to write (store) data into program memory. See the AVR datasheets to see how to write to Flash ROM.

Review Questions

1. The instruction "LPM" uses register _____ as the address pointer.
2. What register holds data once it is read by the LPM Rd,Z instruction?
3. What is the size of Z? How much ROM space does it cover?
4. What register is incremented upon execution of the LPM Rd,Z+ instruction?
5. What is the difference between the LPM and ELPM instructions?
6. When should we make our look-up table in RAM?
7. True or false. We can write into program memory using the SPM instruction.

SECTION 6.5: BIT-ADDRESSABILITY

Many microprocessors such as the 386 or Pentium allow programs to access registers and I/O ports in byte size only. In other words, if you need to check a single bit of an I/O port, you must read the entire byte first and then manipulate the whole byte with some logic instructions to get hold of the desired single bit. This is not the case with the AVR as we saw in Chapter 4. In this section, we provide more programming examples of bit manipulation using the bit-addressable and byte-addressable options of the AVR family.

In Table 6-7, some of the bit-oriented instructions are given. Notice that the bit-oriented instructions use only one addressing mode, the direct addressing mode. In the previous sections of this chapter we showed various addressing modes of byte-addressable space of the AVR, among them register indirect addressing mode for both data RAM and program (code) ROM. Note that there is no register indirect addressing mode for bit-oriented instructions in the AVR, nor are there any bit-oriented instructions for program memory.

Manipulating the bits of general purpose registers

In this part we discuss how to set, clear, or copy the bits of a GPR.

Setting the bits

The SBR (Set Bits in Register) instruction sets the specified bits in the general purpose register. It has the following format:

```
SBR Rd,K    ;set bits in register Rd
```

K is an 8-bit value that can be 00–FF in hex, and Rd is R16 to R31 (any of the 16 general purpose registers). The SBR instruction is just another name for the ORI instruction and it sets any of the bits of the general purpose register whose bit in the K variable is 1. For example, in the following program the SBR instruction sets the bits 2, 5, and 6 regardless of their previous values.

```
LDI R17,0b01011001    ;R17 = 0x59
SBR R17,0b01100100    ;set bits 2, 5, and 6 in register R17
```

When execution of the above instructions is finished, R17 contains 0x7D. Notice that the SBR instruction is a byte-oriented instruction as it manipulates the whole byte at one time.

Clearing the bits

The CBR (Clear Bits in Register) instruction clears the specified bits in the general purpose register. It has the following format:

```
CBR Rd,K    ;clear bits in register Rd
```

K is an 8-bit value that can be 00–FF in hex, and Rd is R16 to R31 (any of the 16 general purpose registers). The CBR instruction clears any of the bits of the general purpose register whose bit in the K variable is 1. For example, in the following program the CBR instruction clears the bits 2, 5, and 6 regardless of their

Table 6-7: Single-Bit (Bit-Oriented) Instructions for AVR

Instruction	Function
SBI A,b	Set Bit b in I/O register
CBI A,b	Clear Bit b in I/O register
SBIC A,b	Skip next instruction if Bit b in I/O register is Cleared
SBIS A,b	Skip next instruction if Bit b in I/O register is Set
BST Rr,b	Bit store from register Rr to T
BLD Rd,b	Bit load from T to Rd
SBRC Rr,b	Skip next instruction if Bit b in Register is Cleared
SBRS Rr,b	Skip next instruction if Bit b in Register is Set
BRBS s,k	Branch if Bit s in status register is Set
BRBC s,k	Branch if Bit s in status register is Cleared

Note: A can be any location of the I/O register.

previous values.

```
LDI R17,0b01011001  ;R17 = 0x59
CBR R17,0b01100100  ;clear bits 2, 5, and 6 in register R17
```

After the execution of the above instructions, R17 contains 0x19.

Copying a bit

As we saw in Chapter 2, one of the bits in the SREG (status register) is named T (temporary), which is used when we want to copy a bit of data from one GPR to another GPR. The BST (Bit Store from register to T) and BLD (Bit Load from T to register) instructions can be used to copy a bit of a register to a specific bit of another register. The "`BST Rd,b`" instruction stores bit b from Rd to the T flag, while the "`BLD Rr,b`" instruction copies the T flag to bit b in register Rr.

For example, the following program copies bit 3 from R17 to bit 5 in register R19:

```
BST R17,3   ;store bit 3 from R17 to the T flag
BLD R19,5   ;copy the T flag to bit 5 in R19
```

See Example 6-21.

Example 6-21

A switch is connected to pin PB4. Write a program to get the status of the switch and save it in D0 of internal RAM location 0x200.

Solution:

```
.EQU MYREG = 0x200      ;set aside loc 0x200
      CBI   DDRB,0      ;make PB0 an input
      IN    R17,PINB    ;R17 = PINB
      BST   R17,4       ;T = PINB.4
      LDI   R16,0x00    ;R16 = 0
      BLD   R16,0       ;R16.0 = T
      STS   MYREG,R16   ;copy R16 to location $200
HERE: JMP   HERE
```

Checking a bit

To see if a bit of a general purpose register is set or cleared, we can use the SBRS (Skip next instruction if Bit in Register is Set) and SBRC (Skip next instruction if Bit in register is Cleared) instructions.

The SBRS instruction tests a bit of a register and skips the instruction right below it if the bit is HIGH. The format of the SBRS instruction is as follows:

```
SBRS Rd,b   ;skip next instruction if Bit b in Rd is set
```

For example, in the following program the "`LDI  R20,0x55`" instruction will not be executed since bit 3 of R17 is set.

```
LDI  R17,0b00001010
SBRS R17,3   ;skip next instruction if Bit 3 in R17 is set
LDI  R20,0x55
LDI  R30,0x33
```

The SBRC instruction skips the next instruction if a bit of a GPR is cleared. It has the following format:

```
SBRC Rd,b ;skip next instruction if Bit b in Rd is cleared
```

For example, in the following program the "`LDI  R20,0x55`" instruction will not be executed since bit 2 of R16 is cleared.

```
LDI  R16,0b00001010
SBRC R16,2 ;skip next instruction if Bit 2 in R16 is cleared
LDI  R20,0x55
LDI  R30,0x33
```

See Example 6-22.

Example 6-22

A switch is connected to pin PC7. Using the SBRS instruction, write a program to check the status of the switch and perform the following:
(a) If switch = 0, send letter 'N' to Port D.
(b) If switch = 1, send letter 'Y' to Port D.

Solution:

```
.INCLUDE "M32DEF.INC"   ;include a file according to the IC you use
      CBI   DDRC,7      ;make PC7 an input
      LDI   R16,0xFF
      OUT   DDRD,R16    ;make Port D an output port
AGAIN:IN    R20,PINC    ;R20 = PINC
      SBRS  R20,7       ;skip next line if Bit PC7 is set
      RJMP  OVER        ;it must be LOW
      LDI   R16,'Y'     ;R16 = 'Y' ASCII letter Y
      OUT   PORTD,R16   ;issue R16 to PD
      RJMP  AGAIN       ;we could use JMP instead
OVER: LDI   R16,'N'     ;R16 = 'N' ASCII letter N
      OUT   PORTD,R16   ;issue R16 to PORTD
      RJMP  AGAIN       ;we can use JMP too
```

Manipulating the bits of I/O registers

As we discussed in Chapter 4, we can set and clear the lower 32 I/O registers (addresses 0 to 31) using the SBI (Set bit in I/O register) and CBI (Clear bit in I/O register) instructions. For example, the following two instructions set the PORTA.1 and clear the PORTB.4, respectively:

```
SBI PORTA,1       ;set Bit 1 in PORTA
CBI PORTB,4       ;clear Bit 4 in PORTB
```

See Example 6-23.

Example 6-23

Write a program to toggle PB2 a total of 200 times.

Solution:

```
      LDI   R16,200     ;load the count into R16
      SBI   DDRB,2      ;DDRB.1 = 1, make RB1 an output
AGAIN:SBI   PORTB,2     ;set bit PB2 (toggle PB2)
      CBI   PORTB,2     ;clear bit PB2 (toggle PB2)
      DEC   R16         ;decrement R16
      BRNE  AGAIN       ;continue until counter is zero
```

In Chapter 4 we mentioned that we can test a bit in the lower 32 I/O registers using the SBIS (Skip if Bit in I/O register is Set) and SBIC (Skip if Bit in I/O register is Cleared) instructions. See Examples 6-24 and 6-25.

Example 6-24

Rewrite the program of Example 6-22 using the SBIC instruction.

Solution:

```
.INCLUDE "M32DEF.INC"   ;include a proper file
      CBI   DDRC,7      ;make PC7 an input
      LDI   R16,0xFF
      OUT   DDRD,R16    ;make Port D an output port
AGAIN:IN    R20,PINC    ;R20 = PINC
      SBRC  R20,7       ;skip next line if Bit PC7 is cleared
      RJMP  OVER        ;it must be HIGH
      LDI   R16,'N'     ;R16 = 'N' ASCII letter N
      OUT   PORTD,R16   ;issue R16 to PD
      RJMP  AGAIN       ;we could use JMP instead
OVER: LDI   R16,'Y'     ;R16 = 'Y' ASCII letter Y
      OUT   PORTD,R16   ;issue R16 to PORTD
      RJMP  AGAIN       ;we can use JMP too
```

Example 6-25

Rewrite the program of Example 6-22 using the SBIS instruction.

Solution:

```
.INCLUDE "M32DEF.INC"   ;include a proper file
      CBI   DDRC,7      ;make PC7 an input
      LDI   R16,0xFF
      OUT   DDRD,R16    ;make Port D an output port
AGAIN:SBIS  PINC,7      ;skip next line if Bit PC7 is set
      RJMP  OVER        ;it must be LOW
      LDI   R16,'Y'     ;R16 = 'Y' ASCII letter Y
      OUT   PORTD,R16   ;issue R16 to PD
      RJMP  AGAIN       ;we could use JMP instead
OVER: LDI   R16,'N'     ;R16 = 'N' ASCII letter N
      OUT   PORTD,R16   ;issue R16 to PORTD
      RJMP  AGAIN       ;we can use JMP too
```

Status register bit-addressability

Now let's see how we can use bit-addressability of the status register. As we discussed in Chapter 2, the bits of the status register are used for the flags C, Z, N, V, S, H, T, and I. The status register is shown in Figure 6-14.

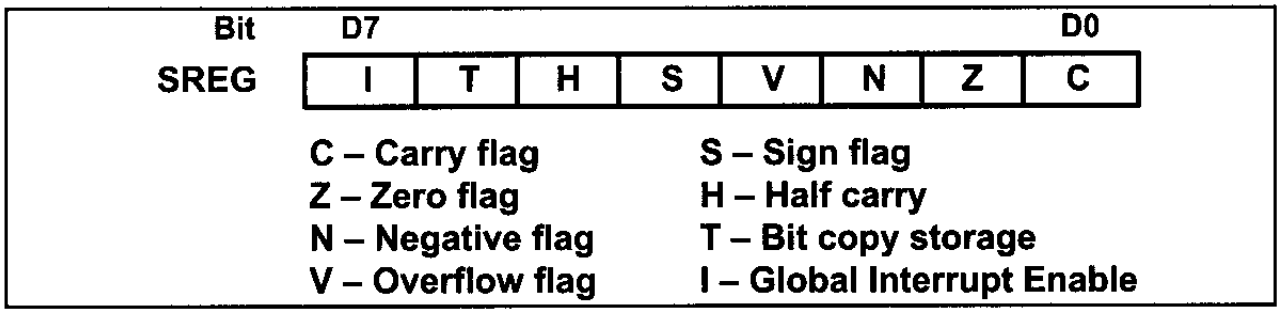


Figure 6-14. Bits of the Status Register

Checking a flag bit

There are some instructions for checking the bits in the status register, as shown in Table 6-8. All of the instructions are derived from two instructions: BRBS (Branch if status flag is Set) and BRBC (Branch if status flag is Cleared). The instructions are as follows:

```
BRBS s,k          ;branch if status flag bit is set
BRBC s,k          ;branch if status flag bit is cleared
```

where s is a number between 0 and 7, and represents the bit in the status register, and k is the relative address of the target location to which the instruction branches when the condition is true.

For example, in the following program the LDI instruction is not executed when the carry flag is set:

```
      BRBS  0,L1        ;branch if status flag bit 0 is set
      LDI   R20,3
L1:
```

Table 6-8: AVR Conditional Branch (Jump) Instructions

Instruction	Action	Instruction	Action
BRCS	Branch if C = 1	BRCC	Branch if C = 0
BRLO	Branch if C = 1	BRSH	Branch if C = 0
BREQ	Branch if Z = 1	BRNE	Branch if Z = 0
BRMI	Branch if N = 1	BRPL	Branch if N = 0
BRVS	Branch if V = 1	BRVC	Branch if V = 0
BRLT	Branch if S = 1	BRGE	Branch if S = 0
BRHS	Branch if H = 1	BRHC	Branch if H = 0
BRTS	Branch if T = 1	BRTC	Branch if T = 0
BRIE	Branch if I = 1	BRID	Branch if I = 0

We can write the same program using the "BRCS L1" instruction as follows:

```
      BRCS  L1          ;branch if carry flag is set
      LDI   R20,3
L1:
```

Since it is hard to memorize the bits of the status register and use the BRBC and BRBS instructions, we can use the instructions in Table 6-8 to simplify checking the bits of the status register.

Manipulating a bit

To set a flag we can use the BSET instruction.

```
BSET  s           ;flag bit set
```

where *s* is a number between 0 and 7, and represents the bit to be set in the status register.

For example, the following instruction sets the carry flag.

```
BSET  0           ;set bit 0 (carry flag)
```

As another example, the instruction "BSET 2" sets the N (Negative) flag.

To clear a flag we can use the BCLR (flag bit clear) instruction.

```
BCLR  s           ;flag bit clear
```

where *s* is a number between 0 and 7, and represents the bit to be cleared in the status register.

For example, the following instruction clears the carry flag.

```
BCLR  0           ;clear bit 0 (carry flag)
```

As another example, the instruction "BCLR 1" clears the Z (Zero) flag.

A more convenient way is to use the CLZ instruction, as shown in Table 6-9.

Table 6-9: Manipulating the Flags of the Status Register

Instruction	Action		Instruction	Action	
SEC	Set Carry	C = 1	CLC	Clear Carry	C = 0
SEZ	Set Zero	Z = 1	CLZ	Clear Zero	Z = 0
SEN	Set Negative	N = 1	CLN	Clear Negative	N = 0
SEV	Set overflow	V = 1	CLV	Clear overflow	V = 0
SES	Set Sign	S = 1	CLS	Clear Sign	S = 0
SEH	Set Half carry	H = 1	CLH	Clear Half carry	H = 0
SET	Set Temporary	T = 1	CLT	Clear Temporary	T = 0
SEI	Set Interrupt	I = 1	CLI	Clear Interrupt	I = 0

Internal RAM bit-addressability

The internal RAM is not bit-addressable. So, in order to manipulate a bit of the internal RAM location, you should bring it into the general purpose register and then manipulate it, as shown in Examples 6-26 and 6-27.

Example 6-26

Write a program to see if the internal RAM location $195 contains an even value. If so, send it to Port B. If not, make it even and then send it to Port B.

Solution 1:

```
.EQU  MYREG = 0x195          ;set aside loc 0x195
      LDI   R16,0xFF
      OUT   DDRB,R16         ;make Port B an output port
AGAIN:LDS   R16,MYREG
      SBRS  R16,0            ;bit test D0, skip if set
      RJMP  OVER             ;it must be LOW
      CBR   R16,0b00000001   ;clear bit D0 = 0
OVER: OUT   PORTB,R16        ;copy it to Port B
      JMP   AGAIN            ;we can use RJMP too
```

Solution 2:

```
.EQU  MYREG = 0x195          ;set aside loc 0x195
      LDI   R16,0xFF
      OUT   DDRB,R16         ;make Port B an output port
AGAIN:LDS   R16,MYREG
      CBR   R16,0b00000001   ;clear bit D0 = 0
OVER: OUT   PORTB,R16        ;copy it to Port B
      JMP   AGAIN            ;we can use RJMP too
```

Example 6-27

Write a program to see if the internal RAM location $137 contains an even value. If so, write 0x55 into location $200. If not, write 0x63 into location $200.

Solution:

```
.EQU  MYREG = 0x137     ;set aside location 0x137
.EQU  RESULT= 0x200
      LDS   R16,MYREG
      SBRC  R16,0       ;skip if clear Bit D0 of R16 register is clr
      RJMP  OVER        ;it is odd
      LDI   R16,0x55
      STS   RESULT,R16
      RJMP  HERE
OVER: LDI   R16,0x63
      STS   RESULT,R16
HERE: RJMP  HERE
```

Review Questions

1. True or false. All registers of the AVR are bit-addressable.
2. True or false. The status register of the AVR is bit-addressable.
3. Indicate which of the following registers are bit-addressable.
   (a) Port A (b) Port B (c) R19 (d) status register (e) PC register
4. How would you check to see whether bit D1 of R23 is HIGH or LOW?
5. Show how to clear the carry flag.
6. State what each instruction does.
   (a) `SBR R16,0x1` (b) `CBR R30,0x7` (c) `BST R19,2`
   (d) `SBI PORTB,4` (e) `CBI SREG,1` (f) `CLI`

SECTION 6.6: ACCESSING EEPROM IN AVR

Every member of the AVR microcontrollers has some amount of on-chip EEPROM. In Table 6-10 you can see the amount of EEPROM memory in each member of the ATmega family. As we mentioned in Chapter 0, the data in SRAM will be lost if the power is disconnected. However, we need a place to save our data to protect them against power failure. EEPROM memory can save stored data even when the power is cut off. In this section we will show how to write to EEPROM memory and how to access it.

Table 6-10: Size of EEPROM Memory in ATmega Family

Chip	Bytes	Chip	Bytes	Chip	Bytes
ATmega8	512	ATmega16	512	ATmega32	1024
ATmega64	2048	ATmega128	4096	ATmega256RZ	4096
ATmega640	4096	ATmega1280	4096	ATmega2560	4096

EEPROM registers

There are three I/O registers that are directly related to EEPROM. These are EECR (EEPROM Control Register), EEDR (EEPROM Data Register), and EEARH-EEARL (EEPROM Address Register High-Low). Each of these registers is discussed in detail in this section.

EEPROM Data Register (EEDR)

To write data to EEPROM, you have to write it to the EEDR register and then transfer it to EEPROM. Also, if you want to read from EEPROM you have to read from EEDR. In other words, EEDR is a bridge between EEPROM and CPU.

EEPROM Address Register (EEARH and EEARL)

The EEARH:EEARL registers together make a 16-bit register to address each location in EEPROM memory space. When you want to read from or write to EEPROM, you should load the EEPROM location address in EEARs. As you see in Figure 6-15, only 10 bits of the EEAR registers are used in ATmega32. Because ATmega32 has 1024-byte EEPROM locations, we need 10 bits to address each location in EEPROM space. In ATmega16, 9 bits of the EEAR registers are used because ATmega16 has 512 bytes of EEPROM, and to address 512 bytes we need a 9-bit address.

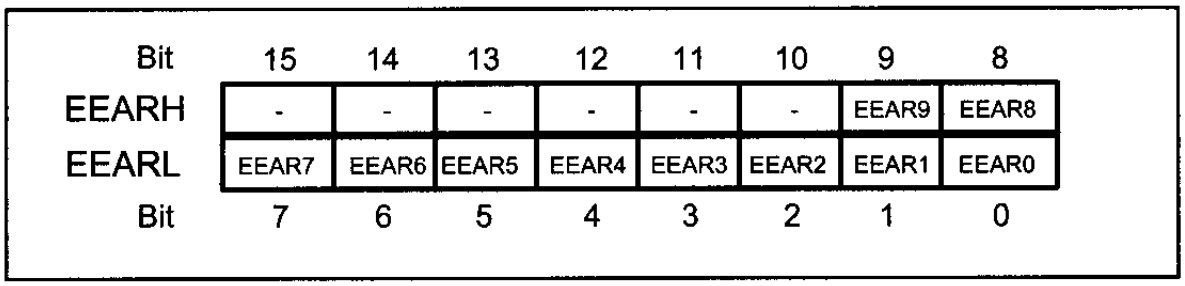


Figure 6-15. EEPROM Address Registers

EEPROM Control Register (EECR)

The EECR register is used to select the kind of operation to perform on. The operation can be start, read, and write. In Figure 6-16 you see the bits of the EECR register. The bits are as follows:

EEPROM Read Enable (EERE): Setting this bit to one will cause a read operation if EEWE is zero. When a read operation starts, one byte of EEPROM will be read into the EEPROM Data Register (EEDR). The EEAR register specifies the address of the desired byte.

EEPROM Write Enable (EEWE) and EEPROM Master Write Enable (EEMWE): When EEMWE is set, setting EEWE within four clock cycles will start a write operation. If EEMWE is zero, setting EEWE to one will have no effect.

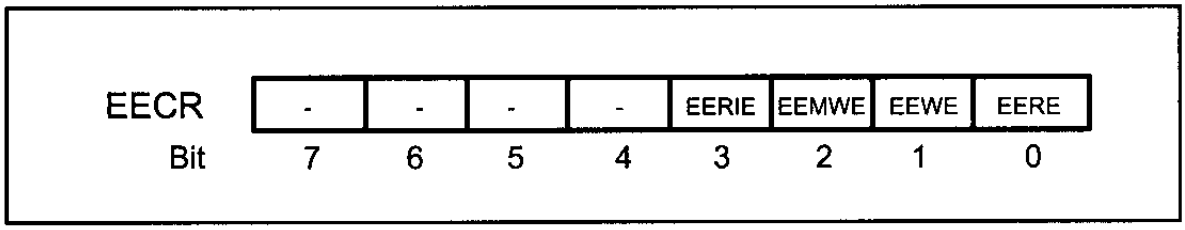


Figure 6-16. EEPROM Control Registers

When you set EEMWE to one, the hardware clears the bit to zero after four clock cycles. This prevents unwanted write operations on EEPROM contents. Notice that you cannot start read or write operations before the last write operation is finished. You can check for this by polling the EEWE bit. If EEWE is zero it means that EEPROM is ready to start a new read or write operation.

EEPROM Ready Interrupt Enable (EERIE): In Chapter 10 you will learn about interrupts in AVR. As you see in Figure 6-16, bits 4 to 7 of EECR are unused at the present time and are reserved.

Programming the AVR to write on EEPROM

To write on EEPROM the following steps should be followed. Notice that steps 2 and 3 are optional, and the order of the steps is not important. Also note that you cannot do anything between step 4 and step 5 because the hardware clears the EEMWE bit to zero after four clock cycles.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Write new EEPROM data to EEDR (optional).
4. Set the EEMWE bit to one (in EECR register).
5. Within four clock cycles after setting EEMWE, set EEWE to one.

See Example 6-28 to see how we write a byte on EEPROM.

Example 6-28

Write an AVR program to store 'G' into location 0x005F of EEPROM .

Solution:

```
.INCLUDE "M16DEF.INC"

WAIT:              ;wait for last write to finish
SBIC  EECR,EEWE    ;check EEWE to see if last write is finished
RJMP  WAIT         ;wait more
LDI   R18,0        ;load high byte of address to R18
LDI   R17,0x5F     ;load low byte of address to R17
OUT   EEARH, R18   ;load high byte of address to EEARH
OUT   EEARL, R17   ;load low byte of address to EEARL
LDI   R16,'G'      ;load 'G' to R16
OUT   EEDR,R16     ;load R16 to EEPROM Data Register
SBI   EECR,EEMWE   ;set Master Write Enable to one
SBI   EECR,EEWE    ;set Write Enable to one
```

Run and simulate the code on AVR Studio to see how the content of the EEPROM changes after the last line of code. Enter four NOP instructions before the last line, change the 'G' to 'H', and run the code again. Explain why the code doesn't store 'H' at location 0x005F of EEPROM.

Programming the AVR to read from EEPROM

To read from EEPROM the following steps should be taken. Note that step 2 is optional.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Set the EERE bit to one.
4. Read EEPROM data from EEDR.

See Example 6-29 to see how we read a byte from EEPROM.

Example 6-29

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

Solution:

```
.INCLUDE "M16DEF.INC"
      LDI   R16,0xFF
      OUT   DDRB,R16
WAIT:                    ;wait for last write to finish
      SBIC  EECR,EEWE    ;check EEWE to see if last write is finished
      RJMP  WAIT         ;wait more
      LDI   R18,0        ;load high byte of address to R18
      LDI   R17,0x5F     ;load low byte of address to R17
      OUT   EEARH, R18   ;load high byte of address to EEARH
      OUT   EEARL, R17   ;load low byte of address to EEARL
      SBI   EECR,EERE    ;set Read Enable to one
      IN    R16,EEDR     ;load EEPROM Data Register to R16
      OUT   PORTB,R16    ;out R16 to PORTB
```

Initializing EEPROM

In Section 6-4, you saw how to allocate program memory using the .DB directive. We can also allocate and initialize the EEPROM using the .DB directive. If we write .ESEG before a definition, the variable will be located in the EEPROM, whereas .CSEG before a definition causes the variable to be allocated in the code (program) memory. By default the variables are located in the program memory. For example, the following code allocates locations $10 and $11 of EEPROM for DATA1 and DATA2, and initializes them with $95 and $19, respectively:

```
      .ESEG
      .ORG $10
DATA1:      .DB   $95
DATA2:      .DB   $19
```

The following code allocates DATA1 and DATA3 in program memory and DATA2 in EEPROM:

```
DATA1:      .DB   $10   ;by default it is located in code memory
      .ESEG
DATA2:      .DB   $20   ;it is located in EEPROM
DATA3:      .DB   $35   ;it is located in EEPROM
      .CSEG
DATA4:      .DB   $45   ;it is located in code memory
```

See Example 6-30.

Example 6-30

Write a program that counts how many times a system has been powered up.

Solution:

```
.INCLUDE "M32DEF.INC"
      LDI   R20,HIGH(RAMEND)
      OUT   SPH,R20
      LDI   R20,LOW(RAMEND)
      OUT   SPL,R20            ;initialize stack pointer

      LDI   XH,HIGH(COUNTER)
      LDI   XL,LOW(COUNTER)    ;X points to COUNTER
      CALL  LOAD_FROM_EEPROM   ;load R20 with value of COUNTER
      INC   R20                ;increment R20
      CALL  STORE_IN_EEPROM    ;store R20 in EEPROM
HERE: RJMP  HERE

;-----Load R20 with contents of location X of EEPROM
LOAD_FROM_EEPROM:
      SBIC  EECR, EEWE
      RJMP  LOAD_FROM_EEPROM   ;wait while EEPROM is busy
      OUT   EEARH,XH
      OUT   EEARL,XL           ;EEAR = X
      SBI   EECR,EERE          ;set Read Enable to one
      IN    R20,EEDR           ;load EEPROM Data Register to r20
      RET

;-----Store R20 into location X of EEPROM
STORE_IN_EEPROM:
      SBIC  EECR, EEWE
      RJMP  STORE_IN_EEPROM    ;wait while EEPROM is busy
      OUT   EEARH,XH
      OUT   EEARL,XL           ;EEAR = X
      OUT   EEDR,R20
      SBI   EECR,EEMWE         ;set Master Write Enable to one
      SBI   EECR,EEWE          ;write EEDR into EEPROM
      RET
;-------EEPROM
.ESEG
.ORG 0
COUNTER:    .DB   0
```

COUNTER is initialized with $0. Then, it is incremented on each power-up.

Review Questions

1. True or false. The AVR EEPROM memory is used for both program code and data.
2. True or false. The ATmega32 has 1,024 bytes of EEPROM memory.
3. True or false. In the AVR, EEPROM contents are lost when power to the chip is cut off.
4. True or false. In the AVR, EEPROM memory is read and write memory.
5. True or false. Every AVR chip comes with 1 KB of EEPROM.

SECTION 6.7: CHECKSUM AND ASCII SUBROUTINES

In this section we look at some widely used subroutines: checksum byte, BCD, and ASCII conversion.

Checksum byte in EEPROM

To ensure the integrity of ROM contents, every system must perform a checksum calculation. The checksum will detect any corruption of the contents of ROM. One cause of ROM corruption is current surge, either when the system is turned on, or during operation. To ensure data integrity in ROM, 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, and that is the checksum byte, which becomes the last byte of the series.

To perform a 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). To clarify these important concepts, see Example 6-31.

Checksum program

The checksum generation and testing program is given in subroutine form. Five subroutines perform the following operations:

1. Retrieve the data from EEPROM.
2. Test the checksum byte for any data error.
3. Initialize variables if the checksum byte is corrupted.
4. Calculate the checksum byte.
5. Store the data in EEPROM.

Each of these subroutines can be used in other applications. Example 6-31 shows how to manually calculate the checksum for a list of values. Also, see Program 6-1.

```
;PROG 6-1: CHECKSUM
.INCLUDE "M32DEF.INC"
.EQU  OPTION_SIZE = 0x4
.EQU  RAM_OPTIONS = 0x100
;-------------main program
.ORG  0
      LDI   R16,HIGH(RAMEND)
      OUT   SPH,R16
      LDI   R16,LOW(RAMEND)
      OUT   SPL,R16            ;SP points to RAMEND
      RCALL LOAD_OPTIONS       ;load options
      RCALL TEST_CHKSUM        ;test checksum
      TST   R20
      BREQ  L1                 ;if data is not corrupted go to L1
```

Example 6-31

Assume that we have 4 bytes of hexadecimal data: $25, $62, $3F, and $52.

(a) Find the checksum byte.
(b) Perform the checksum operation to ensure data integrity.
(c) If the second byte, $62, has been changed to $22, show how the checksum method detects the error.

Solution:

(a) Find the checksum byte.

```
  $25
+ $62
+ $3F
+ $52
 $118
```

(Dropping the carry of 1, we have $18. Its 2's complement is $E8. Therefore, the checksum byte is $E8.)

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

```
  $25
+ $62
+ $3F
+ $52
+ $E8
 $200
```

(Dropping the carries, we see 00, indicating that the data is not corrupted.)

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

```
  $25
+ $22
+ $3F
+ $52
+ $E8
 $1C0
```

(Dropping the carry, we get $C0, which is not 00. This means that the data is corrupted.)

```
      RCALL INIT_OPTIONS      ;initialize options
L1:   ;Here you can use the options

      RCALL CAL_CHKSUM        ;calculating checksum
      RCALL STORE_OPTIONS     ;storing options in EEPROM
HERE: RJMP  HERE

;-----Load R20 with contents of location X of EEPROM
LOAD_FROM_EEPROM:
      SBIC  EECR, EEWE
      RJMP  LOAD_FROM_EEPROM  ;wait while EEPROM is busy
      OUT   EEARH,XH
      OUT   EEARL,XL          ;EEAR = X
      SBI   EECR,EERE         ;set Read Enable to one
      IN    R20,EEDR          ;load EEPROM Data Register to R20
      RET
;-----Store R20 into location X of EEPROM
```

```
STORE_IN_EEPROM:
      SBIC  EECR, EEWE
      RJMP  STORE_IN_EEPROM   ;wait while EEPROM is busy
      OUT   EEARH,XH
      OUT   EEARL,XL          ;EEAR = X
      OUT   EEDR,R20
      SBI   EECR,EEMWE        ;set Master Write Enable to one
      SBI   EECR,EEWE         ;write EEDR into EEPROM
      RET
;-----copying the data from EEPROM to internal SRAM
LOAD_OPTIONS:
      LDI   XL,LOW(E_OPTIONS)
      LDI   XH,HIGH(E_OPTIONS)      ;X points to E_OPTIONS
      LDI   YL,LOW(RAM_OPTIONS)
      LDI   YH,HIGH(RAM_OPTIONS)    ;Y points to RAM_OPTIONS
      LDI   R16,OPTION_SIZE+1       ;COUNTER = OPTION_SIZE+1
LL1:  CALL  LOAD_FROM_EEPROM        ;load R20 with EEPROM loc X
      ST    Y+,R20                  ;store R20 in RAM loc Y
      INC   XL                ;increment XL
      BRNE  LL2               ;if not carry go to LL2
      INC   XH
LL2:  DEC   R16               ;decrement COUNTER
      BRNE  LL1               ;if COUNTER not zero go to LL1
      RET                     ;return
;-------copying data from code ROM to data RAM
INIT_OPTIONS:
      LDI   ZL,LOW(FLASH_OPTIONS<<1);Z points to FLASH_OPTIONS
      LDI   ZH,HIGH(FLASH_OPTIONS<<1)
      LDI   YL,LOW(RAM_OPTIONS)
      LDI   YH,HIGH(RAM_OPTIONS)    ;Y points to RAM_OPTIONS
      LDI   R16,OPTION_SIZE         ;COUNTER = OPTION_SIZE
H1:   LPM   R18,Z+        ;load R18 with program mem. location Z
      ST    Y+,R18              ;store R18 in loc Y of RAM
      DEC   R16                 ;decrement COUNTER
      BRNE  H1                  ;if COUNTER is not zero go to H1
      RET                       ;return
;-------calculating checksum byte
CAL_CHKSUM:
      LDI   YL,LOW(RAM_OPTIONS)
      LDI   YH,HIGH(RAM_OPTIONS)    ;Y points to RAM_OPTIONS
      LDI   R16,OPTION_SIZE   ;COUNTER = OPTION_SIZE
      LDI   R20,0             ;SUM = 0
CL1:  LD    R17,Y+            ;load R17 with contents of loc Y
      ADD   R20,R17           ;SUM = SUM + R17
      DEC   R16               ;decrement COUNTER
      BRNE  CL1               ;if COUNTER is not zero go to CL1
      NEG   R20               ;two's complement SUM
      ST    Y,R20             ;store checksum in loc Y of RAM
      RET                     ;return
;-------testing checksum byte
TEST_CHKSUM:
      LDI   YL,LOW(RAM_OPTIONS)
      LDI   YH,HIGH(RAM_OPTIONS)    ;Y points to RAM_OPTIONS
      LDI   R16,OPTION_SIZE+1
      LDI   R20,0             ;SUM = 0
TL1:  LD    R17,Y+            ;load R17 with contents of loc Y
      ADD   R20,R17           ;SUM = SUM + R17
```

```
      DEC   R16                ;decrement COUNTER
      BRNE  TL1                ;loop while COUNTER is not zero
      RET
;-----copying the data from internal SRAM to EEPROM
STORE_OPTIONS:
      LDI   XL,LOW(E_OPTIONS)
      LDI   XH,HIGH(E_OPTIONS)      ;X points to E_OPTIONS
      LDI   YL,LOW(RAM_OPTIONS)
      LDI   YH,HIGH(RAM_OPTIONS)    ;Y points to RAM_OPTIONS
      LDI   R16,OPTION_SIZE+1       ;COUNTER = OPTION_SIZE+1
SL1:  LD    R20, Y+
      CALL  STORE_IN_EEPROM   ;store R20 in loc X
      INC   XL                ;increment XL
      BRNE  SL2               ;if not carry go to SL2
      INC   XH
SL2:  DEC   R16               ;decrement COUNTER
      BRNE  SL1               ;loop while COUNTER is not zero
      RET                     ;return
;-------initial values in program ROM
FLASH_OPTIONS: .DB      0x25,0x62,0x3F,0x52
;-------EEPROM
.ESEG
.ORG  $0
E_OPTIONS: .DB    0x25,0x62,0x3F,0x52
```

BCD to ASCII conversion program

Many RTCs (real-time clocks) provide time and date in BCD format. To display the BCD data on an LCD or a PC screen, we need to convert it to ASCII. Program 6-2 (a) transfers packed BCD data from program ROM to data RAM, (b) converts packed BCD to ASCII, and (c) sends the ASCII to port B for display. The displaying of data on an LCD will be shown in Chapter 12. See Chapter 5 for the BCD to ASCII conversion algorithm.

```
;PROG 6-2: CONVERTING PACKED BCD TO ASCII
.INCLUDE "M32DEF.INC"
.EQU RAM_ADDR =   0x80
      LDI   R16,HIGH(RAMEND)
      OUT   SPH,R16
      LDI   R16,LOW(RAMEND)
      OUT   SPL,R16                 ;SP = RAMEND
      CALL  BCD_ASCII_COV
HERE: RJMP  HERE
;-----convert packed BCD to ASCII
BCD_ASCII_COV:
      LDI   ZL,LOW(MYBYTE<<1)
      LDI   ZH,HIGH(MYBYTE<<1)      ;Z = MYBYTE
      LDI   XL,LOW(RAM_ADDR)
      LDI   XH,HIGH(RAM_ADDR)       ;X = RAM_ADDR
      LDI   R16,4                   ;COUNTER = 4
L1:   LPM   R20,Z+                  ;
      MOV   R21,R20                 ;R21 = R20
      ANDI  R21,0x0F                ;mask the upper nibble
      ORI   R21,0x30                ;make it an ASCII
      ST    X+,R21
```

```
      SWAP  R20          ;swap the nibbles
      ANDI  R20,0x0F     ;mask the upper nibble
      ORI   R20,0x30     ;make it an ASCII
      ST    X+,R20
      DEC   R16          ;decrement COUNTER
      BRNE  L1           ;loop while COUNTER is not zero
      RET                ;return
;-----send ASCII to Port B
SEND_TO_PORTB:
      LDI   XH,HIGH(RAM_ADDR)
      LDI   XL,LOW(RAM_ADDR)  ;X = RAM_ADDR
      LDI   R16,8             ;COUNTER = 8
L2:   LD    R20,X+            ;
      OUT   PORTB,R20         ;PORTB = R20
      DEC   R16               ;decrement counter
      BRNE  L2                ;loop while counter is not zero
      RET
MYBYTE: .DB 0x25, 0x67, 0x39, 0x52
```

Binary (hex) to ASCII conversion program

Many ADC (analog-to-digital converter) chips provide output data in binary (hex). To display the data on an LCD or PC screen, we need to convert it to ASCII. The code for the binary-to-ASCII conversion is shown in Program 6-3. Notice that the subroutine gets a byte of 8-bit binary (hex) data from Port B and converts it to decimal digits, and the second subroutine converts the decimal digits to ASCII digits and saves them. We are saving the low digit in the lower address location and the high digit in higher address location. This is referred to as the little-endian convention (i.e., low byte to low location, and high byte to high location). All AVR products use the little-endian convention. For the binary-to-ASCII conversion algorithm see Chapter 5.

```
;PROG 6-3: CONVERTING BINARY TO ASCII
.INCLUDE "M32DEF.INC"
.DEF  NUM = R20
.DEF  DENOMINATOR = R21
.DEF  QUOTIENT = R22
.EQU  RAM_ADDR = 0x200
.EQU  ASCII_RESULT = 0x210
;--------------main program
.ORG 0
      LDI   R18,HIGH(RAMEND)
      OUT   SPH,R18
      LDI   R18,LOW(RAMEND)
      OUT   SPL,R18
      LDI   R16,0x00
      OUT   DDRA,R16
      RCALL BIN_DEC_CONVRT
      RCALL DEC_ASCI_CONVRT
END:  RJMP  END

;-----------Converting BIN(HEX) TO DEC (00-FF TO 000-255)
BIN_DEC_CONVRT:
      LDI   XL,LOW(RAM_ADDR)  ;save DEC digits in these locations
      LDI   XH,HIGH(RAM_ADDR)
```

```
        IN    NUM,PINA          ;read data from PORT A
        LDI   DENOMINATOR,10
        RCALL DIVIDE            ;QUOTIENT=PINA/10 NUM=PINA%10
        ST    X+,NUM            ;save lower digit
        MOV   NUM,QUOTIENT
        RCALL DIVIDE            ;divide by 10 once more
        ST    X+,NUM            ;save the next digit
        ST    X+,QUOTIENT       ;save the last digit
        RET
DEC_ASCI_CONVRT:
        LDI   XL,LOW(RAM_ADDR)        ;addr. of DEC data
        LDI   XH,HIGH(RAM_ADDR)
        LDI   YL,LOW(ASCII_RESULT)    ;addr. of ASCII data
        LDI   YH,HIGH(ASCII_RESULT)
        LDI   R16,3                   ;count
BACK:   LD    R20,X+                  ;get DEC digit
        ORI   R20,0x30                ;make it an ASCII digit
        ST    Y+,R20                  ;store it
        DEC   R16                     ;decrement counter
        BRNE  BACK                    ;repeat until the last one
        RET
;---------------------------------------------------------------
DIVIDE:
        LDI   QUOTIENT,0
L1:     INC   QUOTIENT
        SUB   NUM, DENOMINATOR
        BRCC  L1
        DEC   QUOTIENT
        ADD   NUM, DENOMINATOR
        RET
```

We can write a function that directly converts binary to ASCII as shown below:

```
.INCLUDE "M32DEF.INC"
.DEF  NUM = R20
.DEF  DENOMINATOR = R21
.DEF  QUOTIENT = R22
.EQU  ASCII_RESULT = 0x210
;--------------main program
.ORG 0
        LDI   R18,HIGH(RAMEND)
        OUT   SPH,R18
        LDI   R18,LOW(RAMEND)
        OUT   SPL,R18           ;initialize stack pointer
        LDI   R16,0x00
        OUT   DDRA,R16
        RCALL BIN_ASCII_CONVRT
HERE:   RJMP  HERE
;-----------Converting BIN(HEX) TO DEC (00-FF TO 000-255)
BIN_ASCII_CONVRT:
        LDI   XL,LOW(ASCII_RESULT)    ;save results in these loc.
        LDI   XH,HIGH(ASCII_RESULT)
        IN    NUM,PINA          ;read data from PORT A
        LDI   DENOMINATOR,10
        RCALL DIVIDE            ;QUOTIENT=PINA/10    NUM=PINA%10
```

```
ORI    NUM,0x30          ;make it an ASCII digit
ST     X+,NUM            ;save lower digit
MOV    NUM,QUOTIENT
RCALL  DIVIDE            ;divide by 10 once more
ORI    NUM,0x30          ;make it an ASCII digit
ST     X+,NUM            ;save the next digit
ORI    QUOTIENT,0x30     ;make it an ASCII digit
ST     X+,QUOTIENT       ;save the last digit
RET
```

SECTION 6.8: MACROS

In this section we explore macros and their use in Assembly language programming. The format and usage of macros are defined and many examples of their applications are examined.

What is a macro and how is it used?

There are applications in Assembly language programming in which a group of instructions performs a task that is used repeatedly. For example, moving data into a RAM location is done repeatedly in the same program. It does not make sense to rewrite this code every time it is needed. Therefore, to reduce the time that it takes to write code and reduce the possibility of errors, the concept of macros was born. Macros allow the programmer to write the task (code to perform a specific job) once only, and to invoke it whenever it is needed.

Macro definition

Every macro definition must have three parts, as follows:

```
.MACRO      name
.....
.ENDMACRO
```

The .MACRO directive indicates the beginning of the macro definition and the .ENDMACRO directive signals the end. What goes between the .MACRO and .ENDMACRO directives is called the *body* of the macro. The name must be unique and must follow Assembly language naming conventions. A macro can take up to 10 parameters. The parameters can be referred to as @0 to @9 in the body of the macro. After the macro has been written, it can be invoked (or called) by its name, and appropriate values are substituted for parameters.

For example, moving immediate data into I/O register data RAM is a widely used service, but there is no instruction for that. We can use a macro to do the job as shown in the following code:

```
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO
```

The above is the macro definition. Note that parameters @0 and @1 are

mentioned in the body of the macro.

The following are three examples of how to use the above macro:

```
1. LOADIO   PORTA, 0x20        ;send value 0x20 to PORTA

2. .EQU     VAL_1 = 0xFF
   LOADIO   DDRC, VAL_1
3. LOADIO   SPL, 0x55          ;send value $55 to SPL
```

Now examine Program 6-4 to see how to use a macro in a program.

```
;Program 6-4: toggling Port B using macros
;----------------------------------------
.INCLUDE "M32DEF.INC"
.MACRO LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO

;---------------------------time delay macro
.MACRO DELAY
      LDI @0,@1
BACK:
      NOP
      NOP
      NOP
      NOP
      DEC   @0
      BRNE BACK
.ENDMACRO

;---------------------------program starts
      .ORG  0
      LOADIO   DDRB,0xFF       ;make PORTB output
L1:   LOADIO   PORTB,0x55      ;PORTB = 0x55
      DELAY    R18,0x70        ;delay
      LOADIO   PORTB,0xAA      ;PB = 0xAA
      DELAY    R18,0x70        ;delay
      RJMP     L1
```

.INCLUDE directive

Assume that several macros are used in every program. Must they be rewritten every time? The answer is no, if the concept of the .INCLUDE directive is known. The .INCLUDE directive allows a programmer to write macros and save them in a file, and later bring them into any program file. For example, assume that the following widely used macros were written and then saved under the filename "MYMACRO1.MAC".

Assuming that the LOADIO and DELAY macros are saved on a disk under the filename "MYMACRO1.MAC", the .INCLUDE directive can be used to bring this file into any ".asm" file and then the program can call upon any of the macros as many times as needed. When a file includes all macros, the macros are listed at the beginning of the ".lst" file and, as they are expanded, will be part of the program.

To understand this, see Program 6-5.

```
;Program 6-5: toggling Port B using macros
.INCLUDE "M32DEF.INC"
.INCLUDE "MYMACRO1.MAC" ;get macros from macro file
;----------------------------program starts
      .ORG  0
      LOADIO   DDRB,0xFF
L1:   LOADIO   PORTB,0x55
      DELAY    R18,0x70
      LOADIO   PORTB,0xAA
      DELAY    R18,0x70
      RJMP     L1
```

.LISTMAC directive

When viewing the .lst file with macros, the details of the macros are not displayed. This means that the bodies of the macros are not displayed when they are invoked during the code. But when we are debugging the code we might need to see exactly what instructions are executed. Using the .LISTMAC directive we can turn on the display of the bodies of macros in the list file. For example, examine the following code:

```
.INCLUDE "M32DEF.INC"
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO

      LOADIO      PORTA,0x20
      LOADIO      DDRA,0x53
HERE: JMP         HERE
```

The assembler provides the following code in the .lst file:

```
                  .MACRO      LOADIO
                       LDI    R20,@1
                       OUT    @0,R20
                  .ENDMACRO

000000 e240
000001 bb4b       LOADIO      PORTA,0x20
000002 e543
000003 bb4a       LOADIO      DDRA,0x53
000004 940c 0004 HERE:JMP     HERE
```

If we add the .LISTMAC directive to the above code:

```
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO
.LISTMAC
      LOADIO      PORTA,0x20
```

```
      LOADIO      DDRA,0x53
HERE:JMP    HERE
```

The assembler expands the macro by providing the following code in the .lst file:

```
                    .MACRO      LOADIO
                         LDI    R20,@1
                         OUT    @0,R20
                    .ENDMACRO
                    .LISTMAC
                   +
000000 e240        +LDI R20 , 0x20
000001 bb4b        +OUT PORTA , R20
                         LOADIO      PORTA,0x20
                   +
000002 e543        +LDI R20 , 0x53
000003 bb4a        +OUT DDRA , R20
                         LOADIO      DDRA,0x53
000004 940c 0004 HERE:JMP     HERE
```

The + indicates that the code is from the macro.

Macros vs. subroutines

Macros and subroutines are useful in writing assembly programs, but each has limitations. Macros increase code size every time they are invoked. For example, if you call a 10-instruction macro 10 times, the code size is increased by 100 instructions; whereas, if you call the same subroutine 10 times, the code size is only that of the subroutine instructions. On the other hand, a function call takes 3 or 4 clocks and the RET instruction takes 4 clocks to get executed. So, using functions adds around 8 clock cycles. The subroutines use stack space as well when called, while the macros do not.

Review Questions

1. Discuss the benefits of macro programming.
2. List the three parts of a macro.
3. Explain and contrast the macro definition and invoking the macro.

SUMMARY

This chapter described the addressing modes of the AVR. Immediate addressing mode uses a constant for the operand. Direct or register indirect addressing modes can be used to access data stored in data memory of the AVR. Register indirect addressing mode uses a register as a pointer to the data. The advantage of this is that it makes addressing dynamic rather than static. Program memory addressing mode is widely used in accessing data elements of look-up table entries located in the program Flash ROM space of the AVR. The AVR allows

the reading of fixed data stored in program Flash ROM space, in addition to writing to Flash ROM.

The I/O registers can be accessed by six different addressing modes: I/O direct addressing (by their names or their addresses), direct data addressing, data indirect with displacement, data indirect addressing, data indirect addressing with pre-decrement, and data indirect addressing with post-increment.

We also discussed the bit-addressable locations and showed how to use single-bit instructions to access them directly.

We also explained how to access EEPROM and how to use checksum to make sure data is not corrupted.

Macros were also explored and their advantages were discussed.

PROBLEMS

SECTION 6.1: INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

1. Indicate the value loaded into the registers in the following program:

```
.EQU C1 = 0x20
.EQU C2 = 0x6F
.EQU C3 = 0x14
LDI   R20,(C1&C2)|C3
LDI   R21,C2-(C1+C3)
```

2. Indicate the value loaded into R30, R31, and R20 in the following program:

```
.ORG 0x0
.EQU  DATA_ADDR = (OUR_DATA<<1)
      LDI   R30,LOW(DATA_ADDR)
      LDI   R31,HIGH(DATA_ADDR)
      LPM   R20,Z

.ORG 0x100
OUR_DATA: .DB 20,'A','C'
```

SECTION 6.2: REGISTER AND DIRECT ADDRESSING MODES

3. Which of the following are invalid uses of immediate addressing mode?
   (a) LDI R20,0x24 (b) STS 0x70, 0x30 (c) OUT 0x20,0x42
4. Identify the addressing mode for each of the following:
   (a) OUT PORTB,R20 (b) LDI R20, 0x50 (c) LDS 0x40,R20
   (d) ADD R20,R25 (e) MOV R20,R25
5. Indicate the addresses assigned to each of the following:
   (a) PORTB (b) PORTC (c) DDRC
   (d) DDRD (e) SPL (f) SPH
   (g) SREG
6. In accessing the I/O registers, we should use _____ addressing mode.
7. What does the following instruction do? "`STS 0xF0,R20`"
8. What does the following instruction do? "`OUT PORTC,R19`"

9. The byte addresses assigned to the internal SRAM are _____ to _____ in ATmega32. (Hint: To calculate the address of the last location, add the size of SRAM in ATmega32 to the address of the first location of SRAM and decrease the result by one.)
10. The byte addresses assigned to the SRAM are _____ to _____ in ATmega16.
11. Write a program to add the following data and place the result in RAM location $200: The data values are `6, 9, 2, 5, 7`

SECTION 6.3: REGISTER INDIRECT ADDRESSING MODE

12. Which registers are allowed to be used as a pointer for register indirect addressing mode when accessing data RAM? Give their names and show how they are loaded.
13. Write a program to copy $AA into RAM locations $80 to $9F.
14. Write a program to clear RAM locations $90 to $12F.
15. Write a program to copy 10 bytes of data starting at RAM address $80 to RAM locations starting at $90.
16. Write a program to toggle RAM locations $80 to $8F.

SECTION 6.4: LOOK-UP TABLE AND TABLE PROCESSING

17. Compile and state the contents of each ROM location for the following data:

```
                .ORG  0x200
MYDAT_1:        .DB   "Earth"
MYDAT_2:        .DB   "987-65"
MYDAT_3:        .DB   "GABEH 98"
```

18. Compile and state the contents of each ROM location for the following data:

```
                .ORG  0x340
DAT_1: .DB 0x22,0x56, 0b10011001, 32, 0xF6, 0b11111011
```

19. Which register is allowed to be used as a pointer for register indirect addressing mode when accessing data stored in program ROM? Give the name and show how it is loaded.
20. What is the size of the Z register? How much ROM space does the LPM instruction cover?
21. Write a program to read data from the low byte of Flash ROM location 0x200.
22. Write a program to read data from the high byte of Flash ROM location 0x340.
23. Write a program to read the following message from ROM and place it in data RAM starting at 0x60:

```
                .ORG 0x600
MYDATA:         .DB   "1-800-999-9999",0
```

24. Write a program to find y where $y = x^2 + 2x + 5$, and x is between 0 and 9.
25. Write a program to find y where $y = 20x + 5$, and x is between 0 and 9.
26. Write a program to read the following message from ROM and place it in data RAM starting at 40:

```
                .ORG 0x700
MYDATA:         .DB   "The earth is but one country",0
```

27. True or false. In all AVR members we can access the Flash ROM memory.
28. True or false. The ELPM instruction works for all AVR members.
29. Assume that the lower four bits of PORTB are connected to four switches.

Write a program to send the following ASCII characters to a PORTC, based on the status of the switches:

0000	'0'
0001	'1'
0010	'2'
0011	'3'
0100	'4'
0101	'5'
0110	'6'
0111	'7'
1000	'8'
1001	'9'
1010	'A'
1011	'B'
1100	'C'
1101	'D'
1110	'E'
1111	'F'

SECTION 6.5: BIT-ADDRESSABILITY

30. Write a program to generate a square wave with 75% duty cycle on bit PB5.
31. Write a program to generate a square wave with 80% duty cycle on bit PC7.
32. Write a program to monitor PB4. When it goes HIGH, the program will generate a sound (square wave of 50% duty cycle) on pin PB7.
33. Write a program to monitor PC1. When it goes LOW, the program will send the value $55 to PD.
34. What register does the carry flag belong to?
35. What bit address is assigned to the Z flag?
36. Which of the following instructions are valid? If valid, indicate which bit is altered.

    (a) `SBI PORTB,1` (b) `CBI PORTC.3` (c) `SBR SREG,1`
    (d) `SBR R20,1` (e) `BLD PORTD,0` (f) `BST R20,3`
    (g) `CLV R3` (h) `CLN`
37. "`SBI PORTB, 0`" is a(n) _____ (valid, invalid) instruction.
38. Which of the I/O ports of PORTB, PORTC, and PORTD are bit-addressable?
39. Which of the general purpose registers are bit-addressable?
40. Give an instruction to clear the carry flag.
41. Show how would you check whether the C flag is HIGH.
42. Show how would you check whether the Z flag is HIGH.
43. Give the bit locations in the status register assigned to the flag bits C, Z, H, and V.
44. True or false. I/O registers are not bit-addressable.
45. Write instructions to save the C flag bit in bit 4 of location 0x60.
46. Write instructions to save the H flag bit in bit 2 of location 0x160.
47. Write instructions to save the Z flag bit in bit 7 of location 0x120.
48. Write instructions to see whether the D0 and D1 bits of register R20 are LOW.

If so, divide register R20 by 4.

49. Write a program to see whether the D7 bit of register R25 is HIGH. If so, send 0xFF to PORTD.
50. Write a program to set HIGH all the bits of the PORTC I/O register using the following methods:
    (a) byte addresses (b) bit addresses
51. Write a program to see whether the R24 register is divisible by 8.

SECTION 6.6: ACCESSING EEPROM IN AVR

52. Write a program that writes 0 in EEPROM locations $0 to $30.
53. Write a program to copy 10 bytes of data starting at RAM address $80 to EEPROM locations starting at $10.
54. Write a program to copy 10 bytes of data starting at EEPROM address $10 to RAM locations starting at $80.
55. Write a program that calculates the sum of the values of locations $10 to $20 of EEPROM.

SECTION 6.7: CHECKSUM AND ASCII SUBROUTINES

56. Find the checksum byte for the following ASCII message: "Hello"
57. In each of the following cases perform checksum calculation to see if data is corrupted or not.
    (a) Data = $65, $09, and $95; checksum = $23.
    (b) Data = $71, $69, $38, and $81; checksum = $6D.

58. True or false. If we add all bytes, including the checksum byte, and the result is $00, there is no error in the data.
59. Write a program to (a) get the data "Hello, my fellow world citizens" from program ROM, (b) calculate the checksum byte, and (c) test the checksum byte for any data error.
60. To display data on LCD or PC monitors, it must be in _____ (binary, BCD, ASCII).
61. Write a program to convert a series of packed BCD numbers to ASCII. Assume that the packed BCD is located in ROM locations starting at $700. Place the ASCII codes in RAM locations starting at $40.

```
            .ORG $700
MYDATA:     .DB  $76, $87, $98, $43
```

62. Write a program to convert a series of ASCII numbers to packed BCD. Assume that the ASCII data is located in ROM locations starting at $300. Place the BCD data in RAM locations starting at $60.

```
            .ORG $300
MYDATA:     .DB  "87675649"
```

63. Write a program to get an 8-bit binary number from PORTD, convert it to ASCII, and save the result in RAM locations $40, $41, and $42. What is the result if PORTD has 1000 1101 binary as input?

SECTION 6.8: MACROS

64. Give two advantages of macros.
65. Which uses more program Flash ROM space: a macro or a subroutine?

ANSWERS TO REVIEW QUESTIONS

SECTION 6.1: INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

1. R20 = 0x10&0x91 = 0x10
   R21 = 0x91|0x14 = 0x95
   Z = ZH:ZL = 0x123
2. It sets the V and Z flags and clears the other flags.
3.
```
LDI     R16,(1<<SREG_Z)|(1<<SREG_C)
OUT     SREG,R16     ;set Z and C, clear others
```
4. 15900 is $3E1C in hex. Therefore, TCNT1H is loaded with $3E and $1C is loaded into TCNT1L.

SECTION 6.2: REGISTER AND DIRECT ADDRESSING MODES

1. No
2.
```
LDI R20,0b10000000
OUT SPL,R20
```
3. True
4. False
5. True
6. True

SECTION 6.3: REGISTER INDIRECT ADDRESSING MODE

1. Indirect
2. R26
3. 16
4.
```
        .INCLUDE "M32DEF.INC"
        LDI   XL,$90
        LDI   XH,$00
        LDI   YL,$00
        LDI   YH,$2
        LDI   R16,11
        LDI   R22,2
L1:     LD    R20,X+
        ADD   R20,R22
        ST    Y+,R20
        DEC   R16
        BRNE  L1
HERE:   RJMP  HERE
```
5. X, Y, Z

SECTION 6.4: LOOK-UP TABLE AND TABLE PROCESSING

1. Z
2. Rd
3. 16 bits, 32K words

4. Z
5. ELPM can address up to 4M words of Flash memory.
6. When we want to be able to change the look-up table
7. True

SECTION 6.5: BIT-ADDRESSABILITY

1. False
2. True
3. a, b, and d
4. BST R23,1 ;T = R23.1
   BRTS L1 ;branch if T = 1 (branch if R23.1 is high)
   ....
   L1:
5. CLC
6. (a) It sets to HIGH bit 0 of R16.
   (b) It clears bits 0, 1, and 2 of R30.
   (c) It stores bit 2 of R19 to the T flag.
   (d) It sets to HIGH bit 4 of PORTB.
   (e) It clears bit 1 of the status register.
   (f) It clears the I flag of the status register.

SECTION 6.6: ACCESSING EEPROM IN AVR

1. False
2. True
3. False
4. True
5. False

SECTION 6.8: MACROS

1. Macro programming can save the programmer time by allowing a set of frequently repeated instructions to be invoked within the program with a single line. This can also make the code easier to read.
2. The three parts of a macro are the .MACRO directive, the body, and the .ENDMACRO directive.
3. The macro definition is the list of statements the macro will perform. It begins with the .MACRO directive and ends with the .ENDMACRO directive. The macro is invoked whenever it is called from within an Assembly language program. The macro is expanded when the Assembly program replaces the line invoking the macro with the Assembly language code in the body of the macro.

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 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. 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. 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, 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:

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

int main(void)                         //the code starts from here
{
      unsigned char myList[ ]= "012345ABCD";
      unsigned char z;
      DDRB = 0xFF;                     //PORTB is output
      for(z=0; z<10; z++)              //repeat 10 times and increment z
            PORTB = myList[ z] ;       //send the character to PORTB

      while(1);                        //needed if running on a trainer
      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);                        //stay here forever
      return 0;
}
```

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 FCH, FDH, FEH , FFH, 00H, 01H, 02H, 03H, and 04H (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–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. 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. Also, to deal with fractional numbers, most AVR C compilers support float and double data types. See Examples 7-5 and 7-6.

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);                           //stay here forever
   return 0;
}
```

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. In this process they may omit the delay loop because it does not do anything other than wasting CPU time. In these compilers, you have to set the level of optimization to zero (none). To see how you can set the level of optimization for WinAVR and AVR Studio, refer to www.MicroDigitalEd.com.

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:

```
#include <avr/io.h>                    //standard AVR header
void delay100ms(void)
{
   unsigned int i;
   for(i=0; i<42150; 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 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. To overcome this problem, programmers use macro or wrapper function. Wrapper functions do nothing more than call the predefined delay function. If you use wrapper functions and decide to change your compiler, instead of changing all instances of predefined delay functions, you simply change the wrapper function. Look at Example 7-8. 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.

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:

```
#include <util/delay.h>                 //delay loop functions
#include <avr/io.h>                     //standard AVR header

int main(void)
{
       void delay_ms(int d)             //delay in d microseconds
      {
            _delay_ms(d);
      }
      DDRB = 0xFF;                      //PORTA is output
      while (1){
            PORTB = 0xFF;
            delay_ms(10);
            PORTB = 0x55;
            delay_ms(10);
      }
      return 0;
}
```

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. True or false. Using predefined functions of compilers to create a time delay is not recommended if you want your code to be portable to other compilers.
5. 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 FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

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

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;
}
```

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 PINB
      if ( temp < 100 )
        PORTB = temp;
      else
        PORTD = temp;
   }
   return 0;
}
```

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:

```
PORTB.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. 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. Write a short program that toggles bit 2 of Port C using the functions of your compiler.
4. 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 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.

```
#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;
}
```

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 55H to Port B; otherwise, send AAH 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)     //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)           //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;           //pin 5 of Port C 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

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 &= 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.

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)`

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

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;
}
```

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

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

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

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

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

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	
7	37	00000111	01000111 or 47H

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;

      DDRA = 0xFF;                     //make Port A output
      DDRB = 0xFF;                     //make Port B output
      DDRC = 0xFF;                     //make Port C output

      for(x=0; x<4; x++)
      {
            PORTA = mydata[ x] ;       //issue each byte to PORTA
            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
      return 0;
}
```

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;                            //make Port D an output
     for(x=0;x<5;x++)
       chksum = chksum + mydata[ x] ;        //add them together
     if(chksum == 0)
       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 I^2C, 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.
     return 0;
}
```

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.
     return 0;
}
```

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. 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:

1. The 64K bytes of SRAM space with address range 0000–FFFFH. As we have seen in previous chapters, many AVR chips have much less than 64K bytes for the SRAM. 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). As we have seen in the previous chapters, many AVR chips have much less than 2M words of on-chip program ROM (see Table 7-7). We have also seen how to access the program space for the purpose of data storage (see Chapter 6).
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. Also, it should be saved when the power is off, so we place it in EEPROM. Also, when there is not enough code space, we can place permanent variables in EEPROM to save some code space.

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

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.

See www.MicroDigitalEd.com for using Flash data space to store fix data

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 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
     return 0;
}
```

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
}
```

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 widely used technique of 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. True
5. (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. In CodeVision, the code can be:
```
void main()
{
   DDRC.2 = 1;
   PORTC.2 = 0;
}
```
4. 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.

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