MODULE 2
AVR PROGRAMMING IN C

SYLLABUS

I/O port programming in AVR – I/O bit Manipulation Programming. Data types and Time Delays in C - I/O programming in C – Logic Operations in C- Data Conversion Programs in C – Data Serialization in C

I/O PORT PROGRAMMING IN AVR

In the AVR family, there are many ports for I/O operations, depending on which family member you choose. Examine Figure 4-1 for the ATmega32 40-pin chip. A total of 32 pins are set aside for the four ports PORTA, PORTB, PORTC, and PORTD. The rest of the pins are designated as VCC, GND, XTALI, XTAL2, RESET, AREF, AGND, and AVCC.

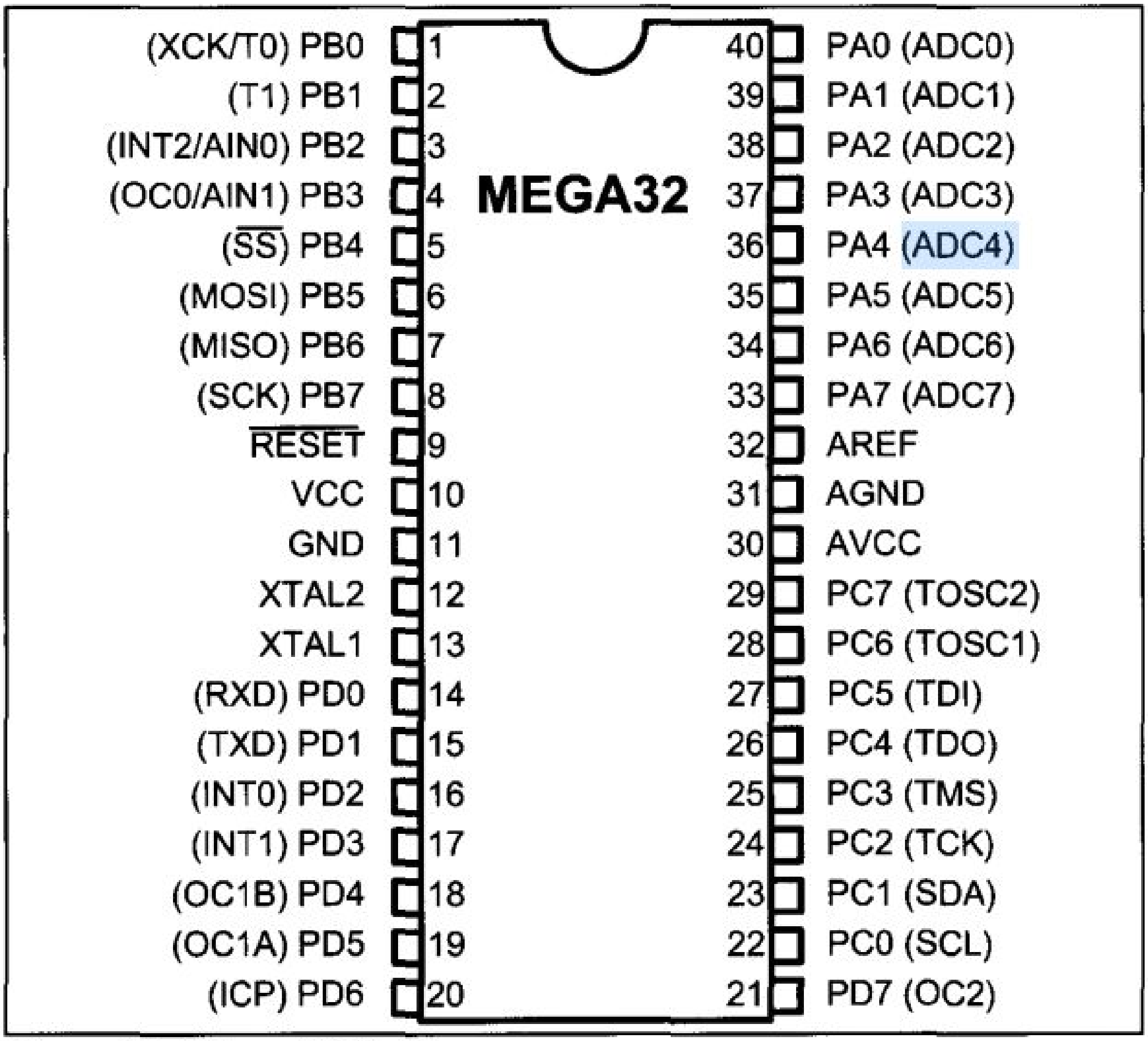


Figure 4-1. ATmega32 Pin Diagram

I/O port pins and their functions

The number of ports in the AVR family varies depending on the number of pins on the chip. The 8-pin AVR has port B only, while the 64-pin version has ports A through F, and the 100-pin AVR has ports A through L, as shown in Table 4-1.The 40-pin AVR has four ports. They are PORTA, PORTB, PORTC, and PORTD.

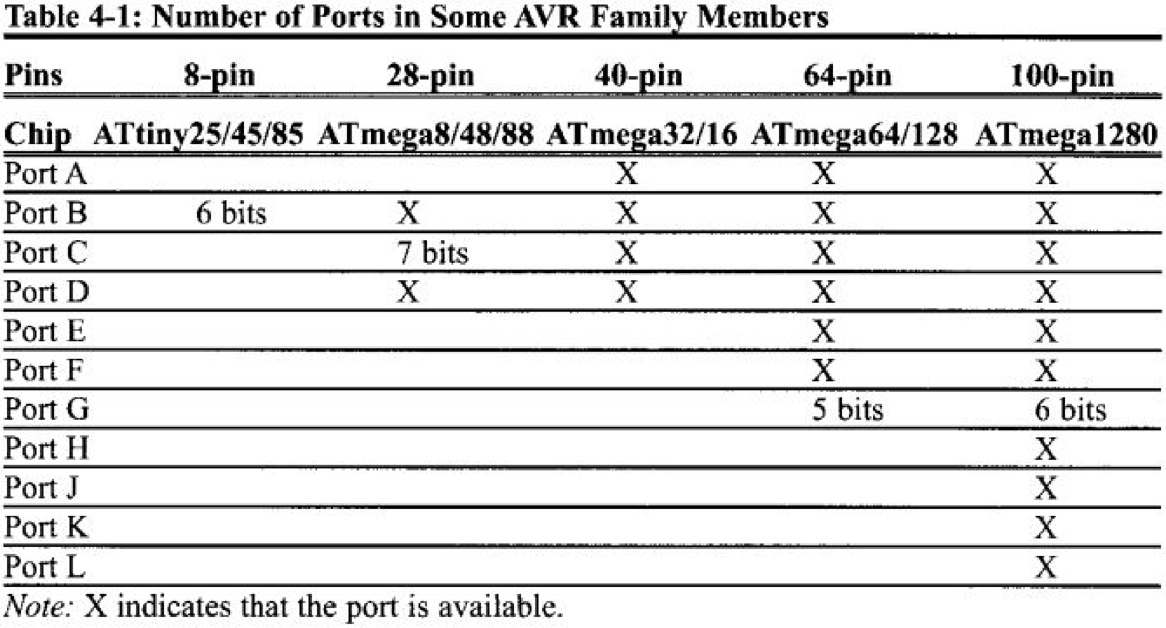


Table 4-1: Number of Ports in Some AVR Family Members

Pins	**8-pin**	**28-pin**	**40-pin**	**64-pin**	**100-pin**
Chip	**ATtiny25/45/85**	**ATmega8/48/88**	**ATmega32/16**	**ATmega64/128**	**ATmega1280**
Port A			X	X	X
Port B	6 bits	X	X	X	X
Port C		7 bits	X	X	X
Port D		X	X	X	X
Port E				X	X
Port F				X	X
Port G				5 bits	6 bits
Port H					X
Port J					X
Port K					X
Port L					X

Note: X indicates that the port is available.

Each port has three I/O registers associated with it. They are designated as PORTx, DDRx, and PINx. For example, for Port B we have PORTB, DDRB, and PINB. Notice that DDR stands for Data Direction Register, and PIN stands for Port INput pins. Also notice that each of the I/O registers is 8 bits wide, and each port has a maximum of 8 pins; therefore each bit of the I/O registers affects one of the pins

DDRx register role in outputting data

Each of the ports A-D in the ATmega32 can be used for input or output. The DDRx I/O register is used solely for the purpose of making a given port an input or output port. For example, to make a port an output, we write 1 s to the DD Rx register. In other words, to output data to all of the pins of the Port B, we must first put Ob 11111111 into the DDRB register to make all of the pins output. The following code will toggle all 8 bits of Port B forever with some time delay between "on" and "off" states:

```
LDI R16,0xFF ;R16 = OxFF = Obllllllll
OUT DDRB,Rl6 ;make Port B an output port (1111 1111)
Ll: LDI R16,0x55 ;R16 = Ox55 = Ob01010101
OUT PORTB, Rl6 ;put Ox55 on port B pins
CALL DELAY
LDI R16,0xAA ;R16 = OxAA = Obl0101010
OUT PORTB, Rl 6 ;put OxAA on port B pins
```

CALL DELAY
RJMP Ll

DDR register role in inputting data

To make a port an input port, we must first put Os into the DDRx register for that port, and then bring in (read) the data present at the pins. As an aid for remembering that the port is input when the DDR bits are Os, imagine a person who has 0 dollars. The person can only get money, not give it. Similarly, when DDR contains Os, the port gets data.

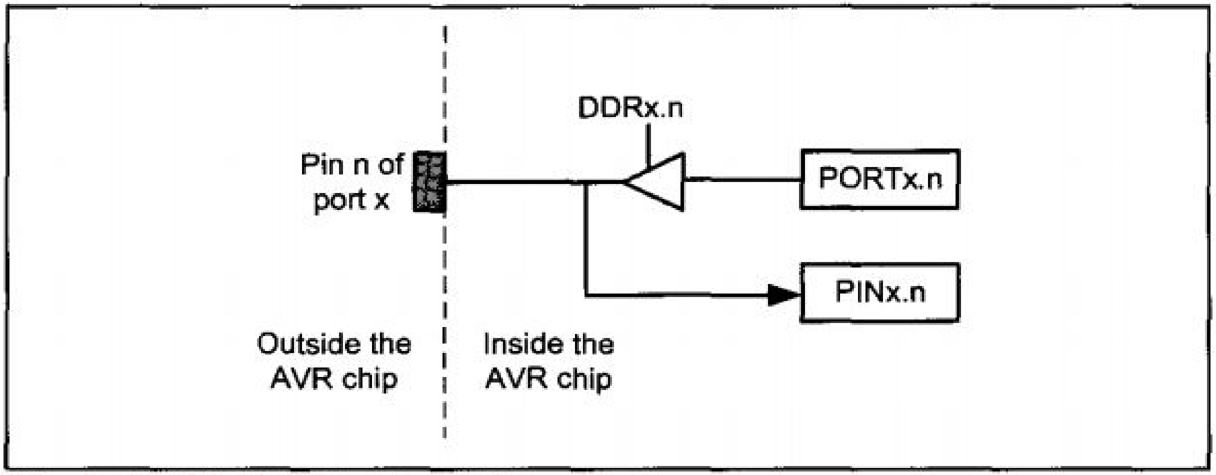


Figure 4-3. The I/O Port in AVR

PIN register role in inputting data

To read the data present at the pins, we should read the PIN register. It must be noted that to bring data into the CPU from pins we read the contents of the PINx register, whereas to send data out to pins we use the PORTx register.

PORT register role in inputting data

There is a pull-up resistor for each of the AVR pins. If we put 1s into bits of the PORTx register, the pull-up resistors are activated. In cases in which nothing is connected to the pin or the connected devices have high impedance, the resistor pull-up the pin. See Figure 4-4. If we put Os into the bits of the PORTx register, the pull-up resistor is inactive.

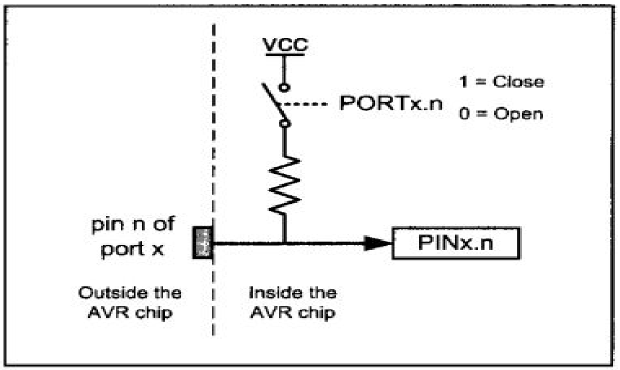


Figure 4-4. The Pull-up Resistor

I/O BIT MANIPULATION PROGRAMMING

I/O ports and bit-addressability

Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8 bits. A powerful feature of AVR 1/0 ports is their capability to access individual bits of the port without altering the rest of the bits in that port. For all AVR ports, we can access either all 8 bits or any single bit without altering the rest.

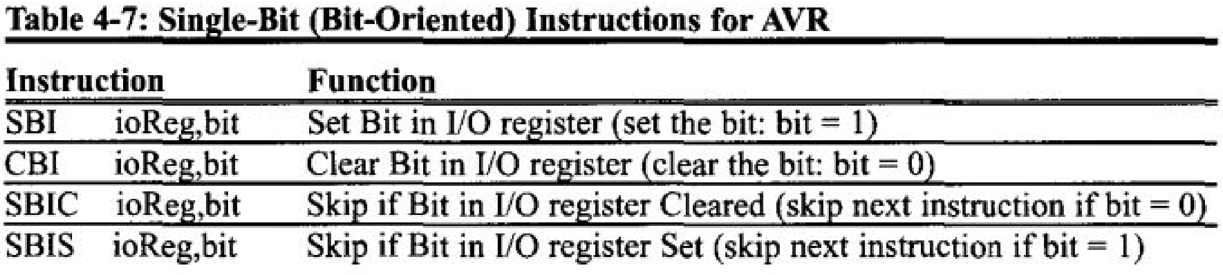


Table 4-7: Single-Bit (Bit-Oriented) Instructions for AVR

Instruction		Function
SBI	ioReg,bit	Set Bit in I/O register (set the bit: bit = 1)
CBI	ioReg,bit	Clear Bit in I/O register (clear the bit: bit = 0)
SBIC	ioReg,bit	Skip if Bit in I/O register Cleared (skip next instruction if bit = 0)
SBIS	ioReg,bit	Skip if Bit in I/O register Set (skip next instruction if bit = 1)

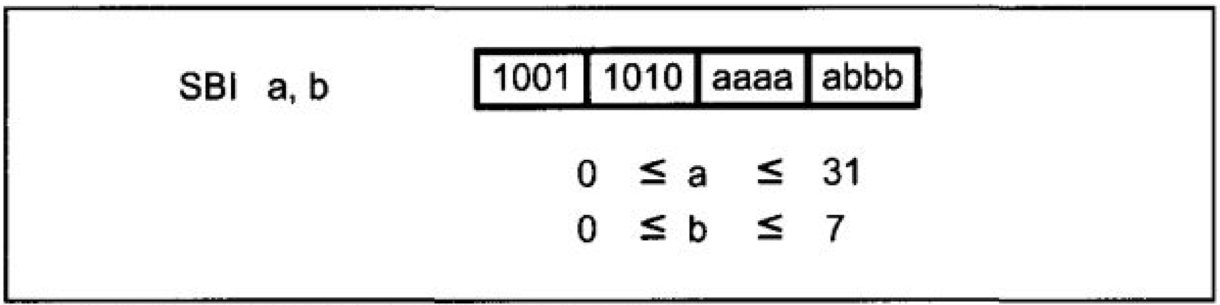
SBI (set bit in 1/0 register)

To set HIGH a single bit of a given I/O register, we use the following syntax:

SBI ioReg, bit_num

where ioReg can be the lower 32 I/O registers (addresses 0 to 31) and bit_ num is the desired bit number from 0 to 7. For example the following instruction sets HIGH bit 5 of Port B:

SBI PORTB, 5



SBI a, b

1001	1010	aaaa	abbb

$0 \leq a \leq 31$

$0 \leq b \leq 7$

Figure 4-6. SBI (Set Bit) Instruction Format

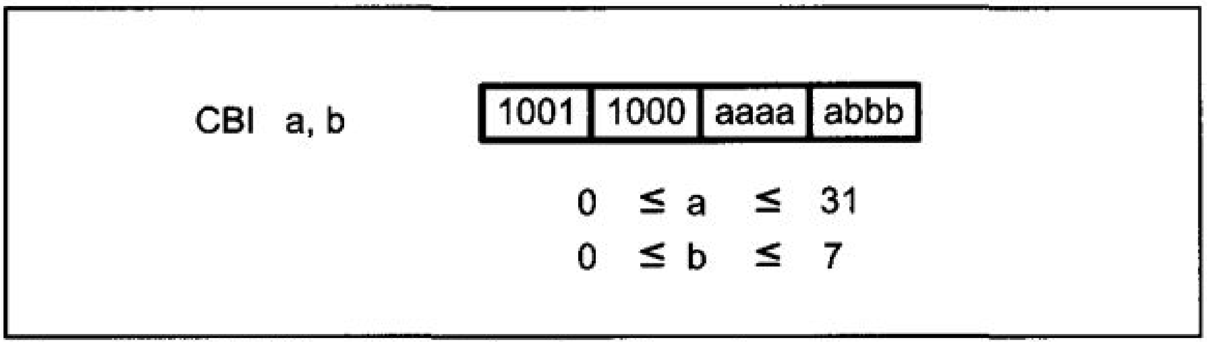
CBI (Clear Bit in 1/0 register)

To clear a single bit of a given I/O register, we use the following syntax:

CBI ioReg, bit_number

For example, the following code toggles pin PB2 continuously:

```
SBI DDRB, 2
AGAIN:SBI PORTB, 2
CALL DELAY
CBI PORTB, 2
CALL DELAY
RJMP AGAIN
```



CBI a, b

1001	1000	aaaa	abbb

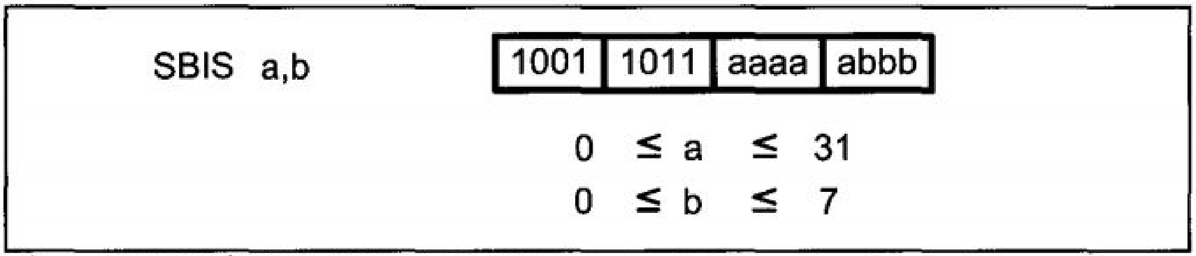
$0 \leq a \leq 31$

$0 \leq b \leq 7$

Figure 4-7. CBI (Clear Bit) Instruction Format

SBIS (Skip if Bit in 1/0 register Set)

To monitor the status of a single bit for HIGH, we use the SBIS instruction. This instruction tests the bit and skips the next instruction if it is HIGH.



SBIS a,b

1001	1011	aaaa	abbb

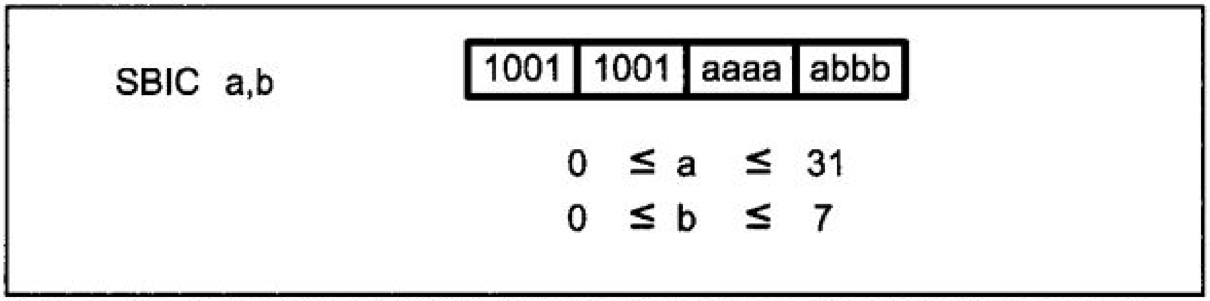
$0 \leq a \leq 31$

$0 \leq b \leq 7$

Figure 4-8. SBIS (Skip If Bit in I/O Register Set) Instruction Format

SBIC (Skip if Bit in 1/0 register Cleared)

To monitor the status of a single bit for LOW, we use the SBIC instruction. This instruction tests the bit and skips the instruction right below it if the bit is LOW.



SBIC a,b

1001	1001	aaaa	abbb

$0 \leq a \leq 31$

$0 \leq b \leq 7$

Figure 4-9. SBIC (Skip if Bit in I/O Register Cleared) Instruction Format

DATA TYPES AND TIME DELAYS IN C

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.

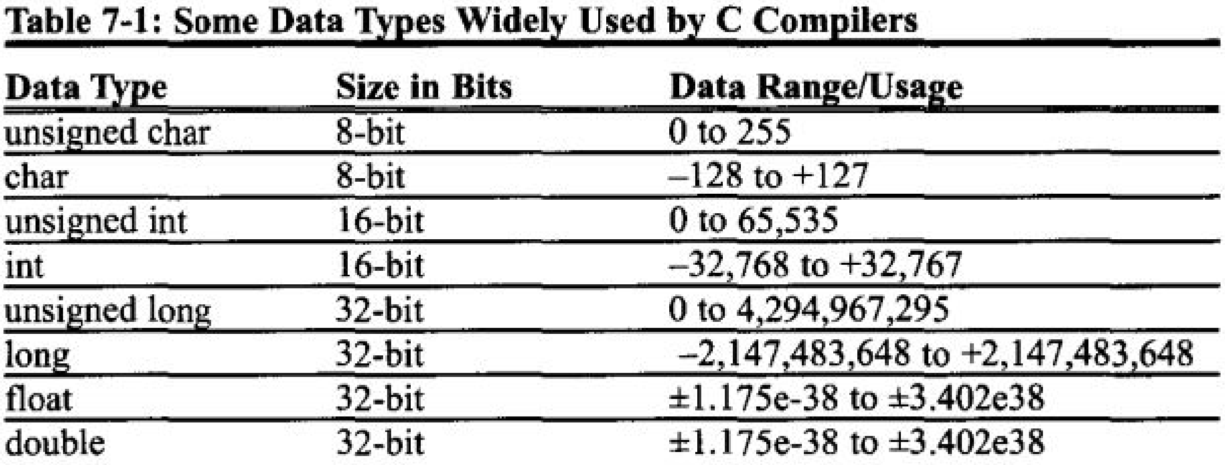


Table 7-1: Some Data Types Widely Used by C Compilers

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.

Signed char

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

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 theAVR, 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.

Signed int

Signed int is a 16-bit data type that uses the most significant bit (D 15 of D15-DO) 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.

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:

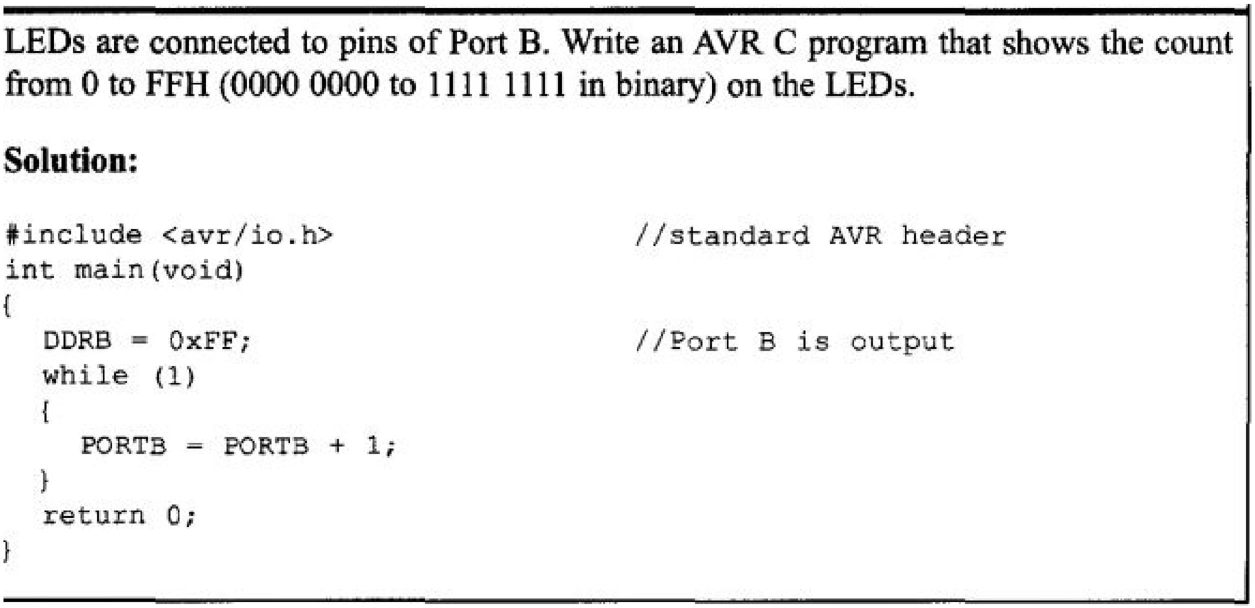
1. The crystal frequency connected to the XTAL l-XTAL2 input pins is the most important factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.
2. The second factor that affects the time delay is the compiler used to compile 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.

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 Code Vision. 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 functions. 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.

I/O PROGRAMMING IN C

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.



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;
}
```

<u>Bit size I/O</u>

The 1/0 ports of ATmega32 are a bit accessible. But some AYR 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 Code Vision 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 bitwise operations to access a single bit of a given register. So, you can access a single bit without disturbing the rest of the byte.

LOGIC OPERATIONS IN C

One of the most important and powerful features of the C language is its ability to perform bit manipulation.

<u>Bitwise operators in C</u>

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

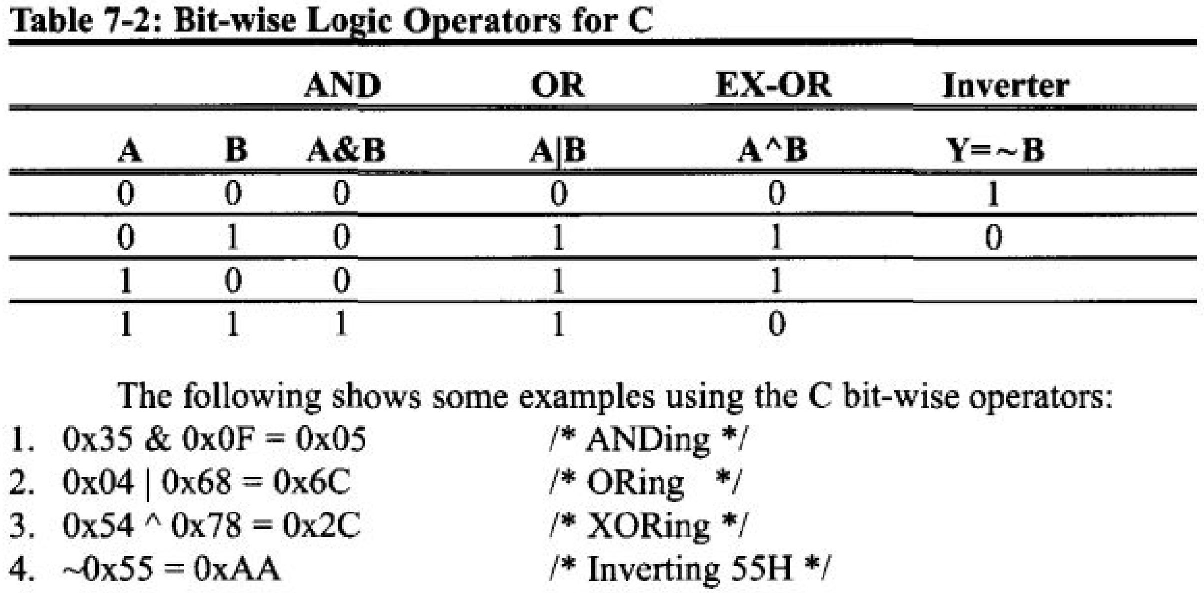


Table 7-2: Bit-wise Logic Operators for C

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

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 */

Compound assignment operators in C

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

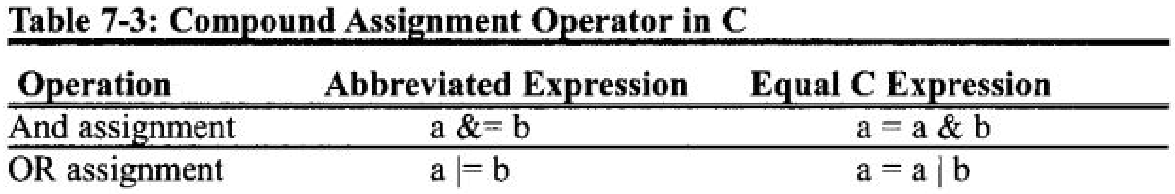


Table 7-3: Compound Assignment Operator in C

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

Bitwise shift operation in C

There are two bit-wise shift operators in C.

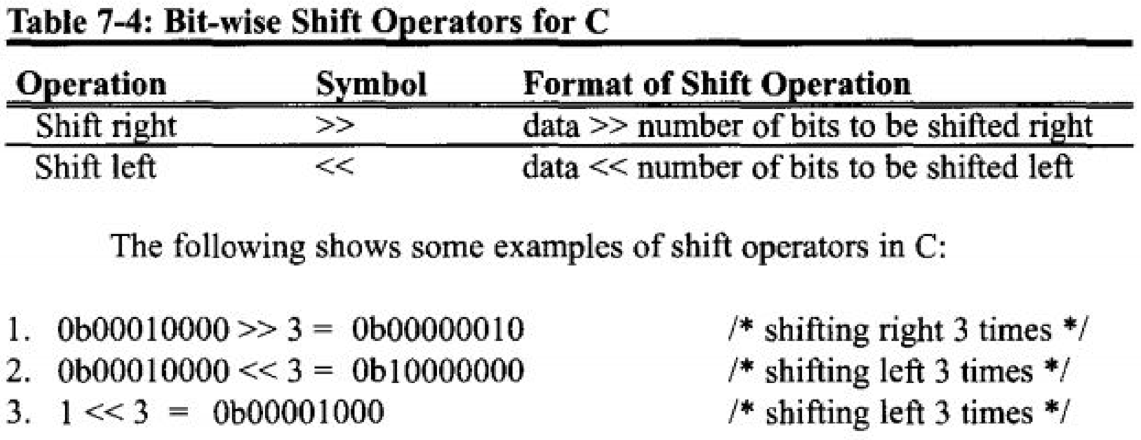


Table 7-4: Bit-wise Shift Operators for C

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

The following shows some examples of shift operators in C:

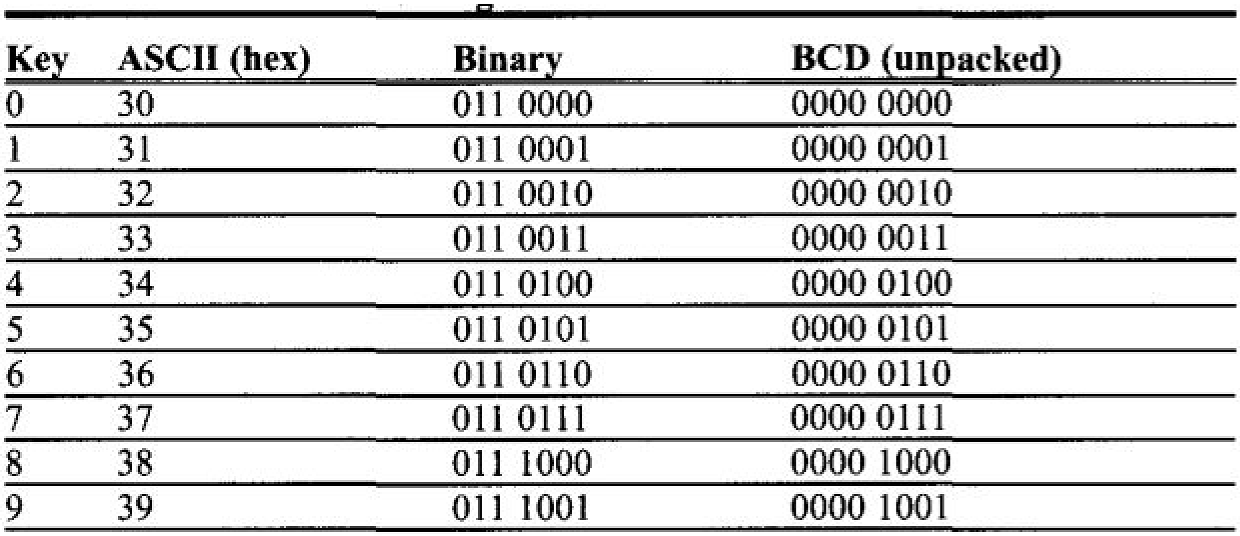
1. 0b00010000 >> 3 = 0b00000010 /* shifting right 3 times */
2. 0b00010000 << 3 = 0b10000000 /* shifting left 3 times */
3. 1 << 3 = 0b00001000 /* shifting left 3 times */

DATA CONVERSION PROGRAMS IN C

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

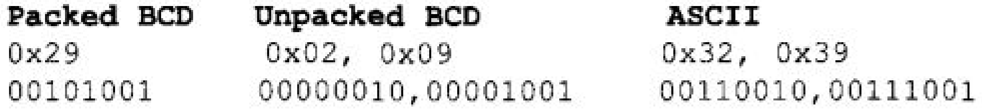
On ASCII keyboards, when the "O" key is activated, "0011 0000" (30H) is provided to the computer. Similarly, 31H (0011 0001) is provided for the"!" key, and so on, as shown in Table



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.



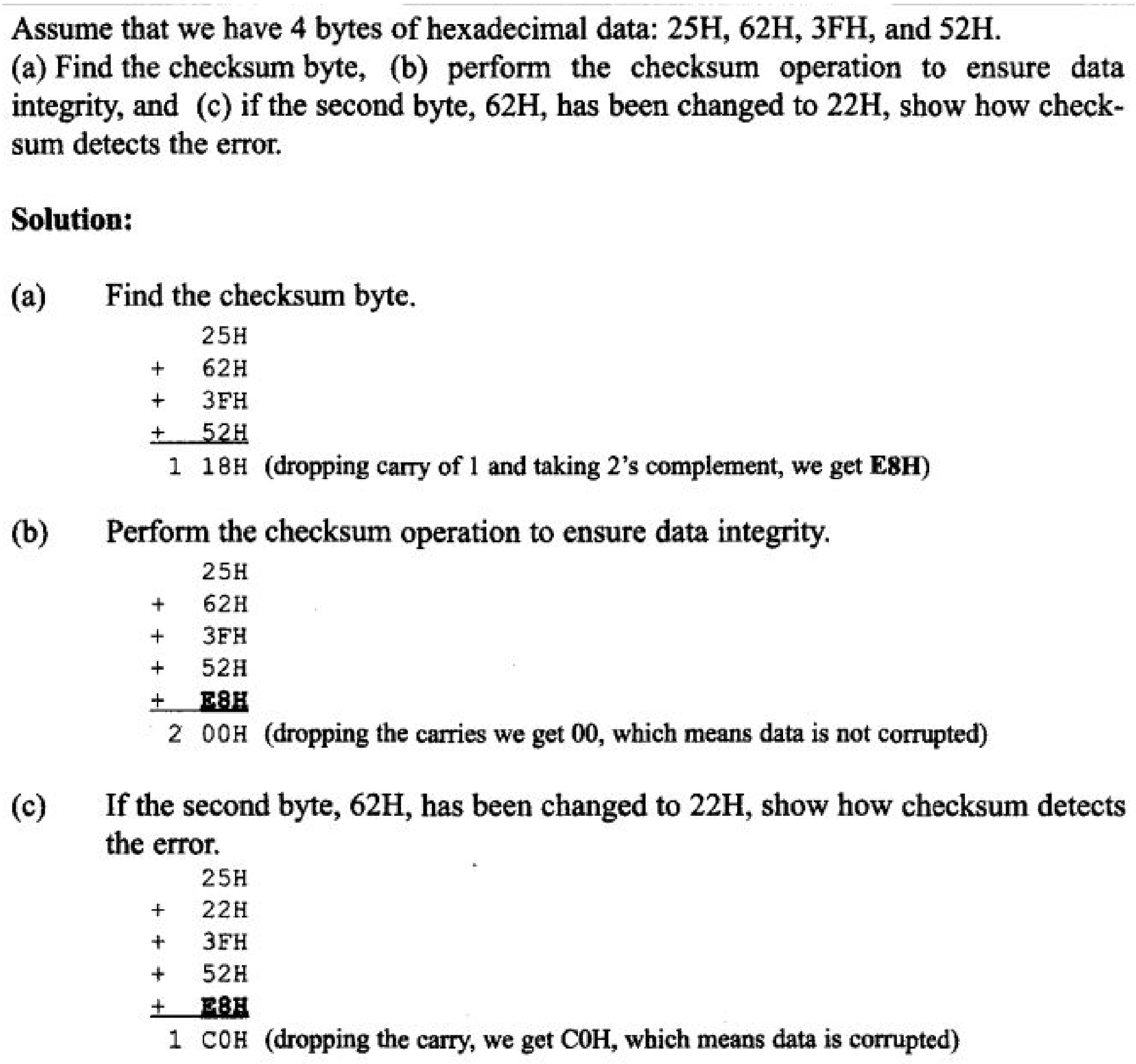
```
Packed BCD   Unpacked BCD         ASCII
0x29          0x02, 0x09          0x32, 0x39
00101001     00000010,00001001    00110010,00111001
```

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:
I. 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).



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
    1 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
    2 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.

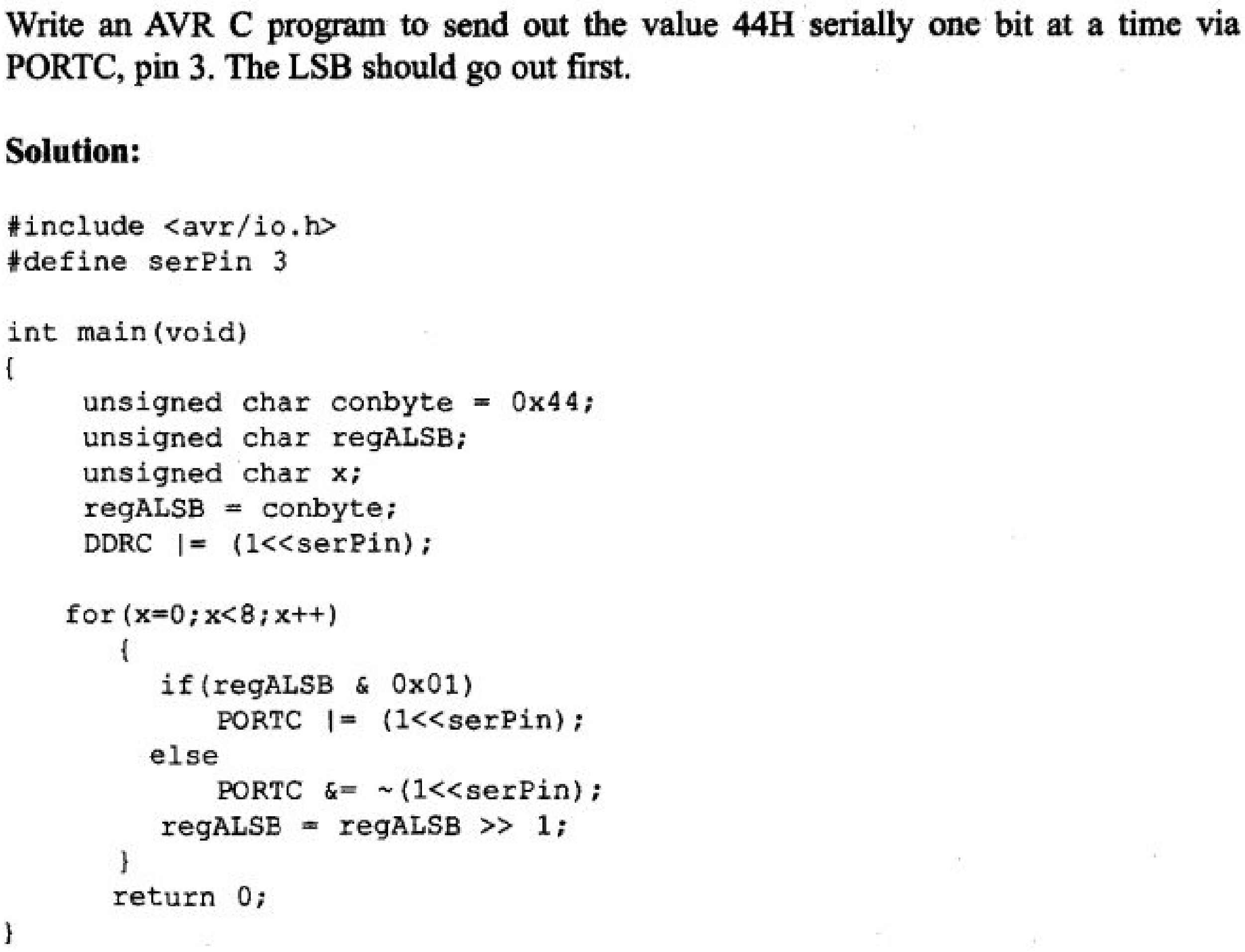
```
      25H
  +   22H
  +   3FH
  +   52H
  +   E8H
    1 C0H
```

(dropping the carry, we get C0H, which means data is corrupted)

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



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;
}
```