Programming For Embedded System

Unit - 2

Overview of AVR

- **What is the difference between a microprocessor and a microcontroller?**
  - By microprocessor it is meant the general-purpose microprocessors such as Intel's x86 family (8086, 80286, 80386, 80486, and the Pentium) or Motorola's PowerPC family. These microprocessors contain no RAM, no ROM, and no I/O ports on the chip itself.
  - This is not the case with microcontrollers. A microcontroller has a CPU (a microprocessor) in addition to a fixed amount of RAM, ROM, I/O ports, and a timer all on a single chip.
  - The fixed amount of on-chip ROM, RAM, and number of I/O ports in microcontrollers makes them ideal for many applications in which cost and space are critical.

Overview of AVR

- Microprocessors and microcontrollers are widely used in embedded system products. An embedded system is controlled by its own internal microprocessor (or micro controller) as opposed to an external controller.
- Typically, in an embedded system, the microcontroller's ROM **is burned with a purpose for specific functions needed for the system.**
- A printer is an example of an embedded system because the processor inside it performs one task only; namely, getting the data and printing it.
- In an embedded system, typically only one application software is burned into ROM.
- A PC contains or is connected to various embedded products such as the keyboard, printer, modem, disk controller, sound card, CD-ROM driver, mouse, and so on.
- **For example,** inside every mouse a microcontroller performs the task of finding the mouse's position and sending it to the PC.

Overview of AVR - A Brief Introduction

- The basic architecture of AVR was designed by two students of Norwegian Institute of Technology (NTH), **Alf-Egil Bogen and Vegard Wollan**, and then was bought and developed by Atmel in 1996.
- It might stand for Advanced Virtual RISC, or Alf and Vegard RISC (the names of the AVR designers).
- AVRs are all 8-bit micro processors, meaning that the CPU can work on only 8 bits of data at a time.
- Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU.
- Except for AVR32, which is a 32-bit microcontroller
- AVRs are generally classified into four broad groups: **Mega, Tiny, Special purpose, and Classic.**

AVR Family

- AVR can be classified into four groups: **Classic, Mega, Tiny, and special purpose**

Classic AVR (AT90Sxxxx)

This is the original AVR chip, which has been replaced by newer AVR chips.

Table 2: Some Members of the Classic Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
AT90S2313	2K	128	128	15	0	2	SOIC20, PDIP20
AT90S2323	2K	128	128	3	0	1	SOIC8, PDIP8
AT90S4433	4K	128	256	20	6	2	TQFP32, PDIP28

Notes:
1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.

Mega AVR (ATmegaxxxx)

- These are powerful microcontrollers with more than 120 instructions
- Some of their characteristics are as follows:
  - **Program memory:** 4K to 256K bytes
  - **Package:** 28 to 100 pins
  - Extensive peripheral set
  - **Extended instruction set:** They have rich instruction sets.

Table 3: Some Members of the ATmega Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATmega8	8K	1K	0.5K	23	8	3	TQFP32, PDIP28
ATmega16	16K	1K	0.5K	32	8	3	TQFP44, PDIP40
ATmega32	32K	2K	1K	32	8	3	TQFP44, PDIP40
ATmega64	64K	4K	2K	54	8	4	TQFP64, MLF64
ATmega1280	128K	8K	4K	86	16	6	TQFP100, CBGA

Notes:
1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.
3. All the above chips have USART for serial data transfer.

Tiny AVR (ATtinyxxxx)

- As its name indicates, the microcontrollers in this group have less instructions and smaller packages in comparison to mega family. You can design systems with low costs and power consumptions using the Tiny AVRs.
- Some of their characteristics are as follows: •
  - **Program memory:** 1K to 8K bytes
  - **Package:** 8 to 28 pins
  - **Limited peripheral set**
  - **Limited instruction set:** The instruction sets are limited. For example, some of them do not have the multiply instruction.

Table 4: Some Members of the Tiny Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATtiny13	1K	64	64	6	4	1	SOIC8, PDIP8
ATtiny25	2K	128	128	6	4	2	SOIC8, PDIP8
ATtiny44	4K	256	256	12	8	2	SOIC14, PDIP14
ATtiny84	8K	512	512	12	8	2	SOIC14, PDIP14

Special Purpose AVR

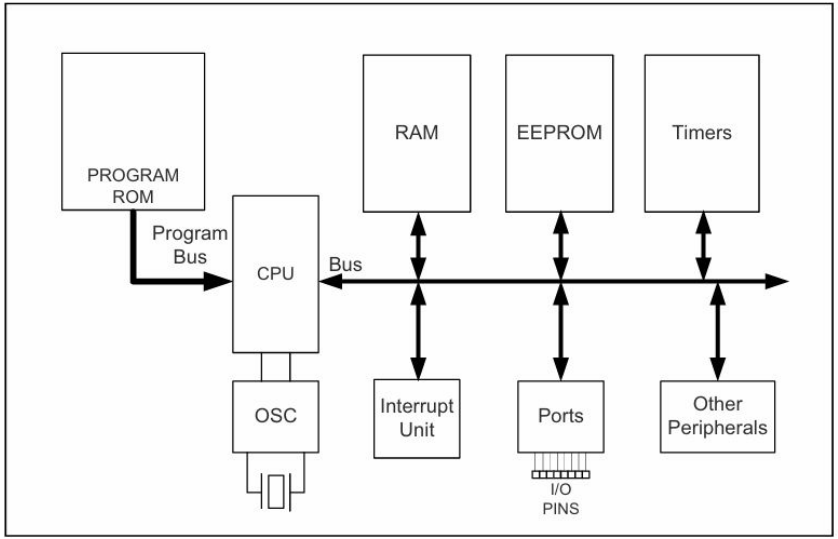
- The ICs of this group can be considered as a subset of other groups, but their special capabilities are made for designing specific applications.
- Some of the special capabilities are:
  - USB controller, CAN controller, LCD controller, Zigbee, Ethernet controller, FPGA, and advanced PWM.

Table 5: Some Members of the Special Purpose Family

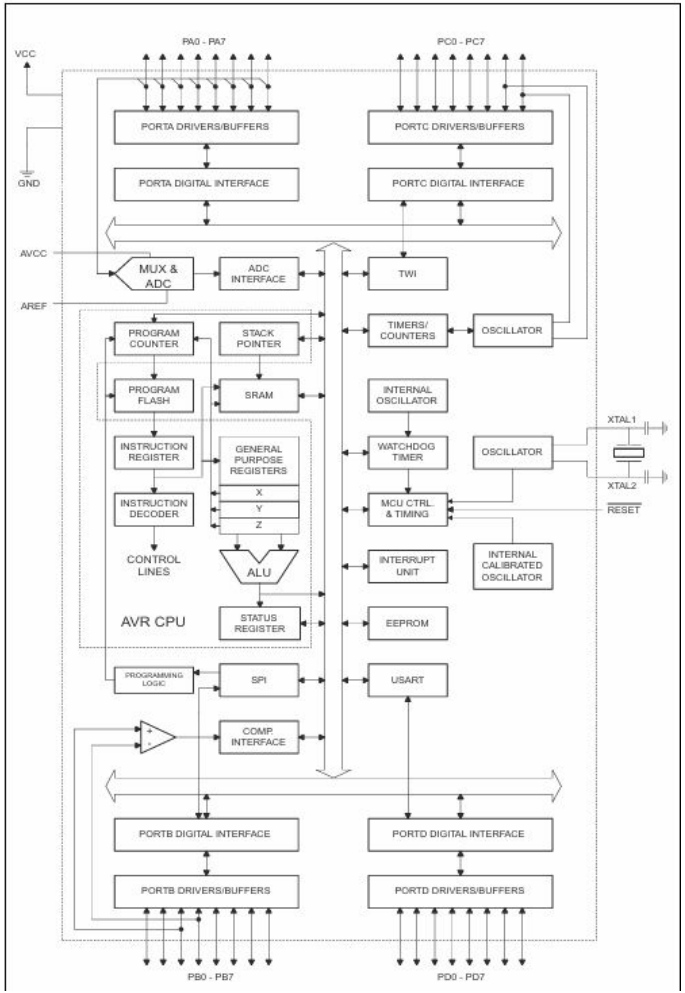
Part Num.	Code ROM	Data RAM	Data EEPROM	Max I/O pins	Special Capabilities	Timers	Pin numbers & Package
AT90CAN128	128K	4K	4K	53	CAN	4	LQFP64
AT90USB1287	128K	8K	4K	48	USB Host	4	TQFP64
AT90PWM216	16K	1K	0.5K	19	Advanced PWM	2	SOIC24
ATmega169	16K	1K	0.5K	54	LCD	3	TQFP64, MLF64

ATmega32 - Architecture

- We will focus on ATmega32 -> it ideal for educational purposes.
- The AVR is an 8-bit RISC single-chip microcontroller with Harvard architecture
- Comes with some standard features such as on-chip program (code) ROM, data RAM, data EEPROM, timers and I/O ports.



ATmega32 Block Diagram



Program ROM

- The ROM is used to store programs and for that reason it is called program or code ROM.
- **ATmega32 has 32K Program ROM.** The program ROM size can vary from 1K to 256K at the time of this writing, depending on the family member.
- The AVR was one of the first microcontrollers to use **on-chip Flash memory** for program storage.

Data RAM and EEPROM

- While ROM is used to store program (code), the RAM space is for data storage.
- **ATmega32 has 2K SRAM.** *The AVR has a maximum of 64K bytes of data RAM space.*
- The data RAM space has three components: **general-purpose registers, I/O memory, and internal SRAM.**
- There are 32 general-purpose registers in all of the AVRs, but the SRAM's size and the I/O memory size varies from chip to chip.
- The internal SRAM space is used for a read/write scratch pad.
- In AVR, we also have a small amount of EEPROM to store critical data that does not need to be changed very often

I/O Pins

- **ATMega32 has 40 Pins.** The ATmega32 typically has 40 pins in the PDIP (Plastic Dual In-line Package) package.
- Out of these, 32 pins are available for General Purpose Input/Output (GPIO).
- The number of pins for the AVR package goes from 8 to 100 at this time.
- In the case of the 8-pin AT90S2323, we have 3 pins for I/O, while in the case of the 100-pin ATmega1280, we can use up to 86 pins for I/O.

Microcontroller Peripherals

- Most of the AVRs come with ADC (analog-to-digital converter), timers, and USART (Universal Synchronous Asynchronous Receiver Transmitter) as standard peripherals.
- The ADC is 10-bit and the number of ADC channels in AVR chips varies and can be up to 16, depending on the number of pins in the package.
- The AVR can have up to 6 timers besides the watchdog timer.
- The USART peripheral allows us to connect the AVR-based system to serial ports such as the COM port of the x86 IBM PC.
- Most of the AVR family members come with the I2C and SPI buses and some of them have USB or CAN bus as well.

Review Questions (AVR)

1. Name three features of the AVR.
2. The AVR is a(n) _____-bit microprocessor.
3. Name the different groups of the AVR chips.
4. Which group of AVR has smaller packages?
5. Give the size of RAM in each of the following: (a) ATmega32 (b) ATtiny25
6. Give the size of the on-chip program ROM in each of the following: (a) ATtiny84 (b) ATmega32 (c) ATtiny25

C Programming Introduction

- 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 pro- gram 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:**
    - It is easier and less time consuming to write in C than in Assembly.
    - C is easier to modify and update.
    - You can use code available in function libraries.
    - C code is portable to other microcontrollers with little or no modification.
- Several third-party companies develop C compilers for the AVR microcon- troller.

Data Types

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

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 microcon- troller 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

Unsigned Char (Example)

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.
```

Unsigned Char (Example 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;
}
```

Signed Char

- The signed char is an 8-bit data type that uses the most significant bit (D7 of D7-DO0) 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.
- 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.

Signed Char (Example)

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.
- 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.

Unsigned int (Example)

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.

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.
- We have only 15 bits for the magnitude of the number, or values from -32,768 to +32,767.

Signed Int (Example)

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 in AVR C

- 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**

Timer - I

Time Delay in AVR C

In AVR C programming, creating precise time delays is crucial for various applications. Here are three common methods to achieve this:

1. Simple For Loop:

- **How it works:** A **“for loop”** iterates a specific number of times, consuming CPU cycles. By carefully calculating the number of iterations and the time taken per iteration, we can approximate a desired delay.
- **Factors affecting accuracy:**
    - **Crystal Frequency:** The speed of the microcontroller's clock directly influences the duration of each loop iteration. A higher frequency results in faster execution and thus shorter delays.
    - **Compiler Optimization:** Compilers often optimize code to reduce execution time. If the compiler detects that a loop doesn't perform any meaningful task, it might remove or modify it, affecting the intended delay.

Time Delay in C

2. Predefined C Functions:

- **How it works:** C provides functions like delay() or sleep() that can be used to pause execution for a specified duration. These functions often rely on system timers or other hardware mechanisms to achieve accurate delays.

3. AVR Timers:

- **How it works:** AVR microcontrollers have built-in timers that can be configured to generate interrupts at specific intervals. By setting up a timer to count a certain number of clock cycles, we can create precise delays without consuming CPU cycles.
- **Advantages:**
  - **Accuracy:** Timers offer highly accurate delays, as they are directly tied to the system clock.
  - **Efficiency:** They don't waste CPU cycles, allowing the microcontroller to perform other tasks while the delay is active.

Time Delay in AVR C

Why For Loops Can Be Less Reliable:

While for loops are a simple approach, they have limitations:

- **Dependency on Crystal Frequency:** The accuracy of the delay is directly tied to the crystal frequency. Any deviation in the frequency can lead to inaccurate delays.
- **Compiler Optimization:** As mentioned, compilers might optimize away the loop if they don't see any side effects, rendering the delay ineffective.
- **CPU Usage:** The for loop consumes CPU cycles, which can impact the performance of other tasks, especially in resource-constrained microcontrollers.

To ensure accurate delays using for loops, consider the following:

- **Measure the Actual Delay:** Use an oscilloscope or other timing tools to measure the actual delay produced by your loop.
- **Disable Compiler Optimization:** If necessary, disable compiler optimization for the code section containing the delay loop.
- **Use a More Reliable Method:** For critical timing applications, consider using predefined C functions or AVR timers, which offer greater precision and efficiency.

Using “For Loop” for Time Delay

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;
}
```

Using Predefined Function (delay_ms)

- 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.

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;
}
```

I/O Programming in C

Introduction

- All port registers of the AVR are both byte accessible and bit accessible.
- We look at C programming of the I/O ports for the AVR. We look at both byte and bit I/O programming.

1. 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.**

Byte Size I/O (Example 1)

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;
}
```

Byte Size I/O (Example 2)

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;
}
```

Byte Size I/O (Example 3)

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;
}
```

2. Bitsize I/O

- The I/O ports of ATmega32 are bit-accessible.
- 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.

Logic Operation

Introduction

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

  Bit-wise operators in C

- While every C programmer is familiar with the logical operators AND (&&), OR (I), and NOT (!), many C programmers are less familiar with the bit- wise operators AND (&), OR (1), 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.

Bitwise Logical Operator

		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 */

Example 1

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 2

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 3

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 4

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 5

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;
}
```

Compound Assignment Operator in C

- To reduce coding we can use compound statement for bitwise operation.

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

Compound Assignment Operator in C (Example 1)

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:
(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;
}
```

Compound Assignment Operator in C (Example 2)

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.

```
#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;
}
```

Bitwise Shift Operation and bit manipulation

- Reexamine the last 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 "Ob00100000" we can write "Ob00000001 << 5" or we can write simply "1<<5".
- Sometimes we need numbers like Ob11101111. To generate such a number, we do the shifting first and then invert it. For example, to generate 0b11101111 we can write ~(1<<5).

Example 1

Write code to generate the following numbers:

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

Solution:

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

Example 2

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;
}
```

Serial Communication in AVR

Introduction

- 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:
  - **Using the serial port:** In using the serial port, the programmer has very limited control over the sequence of data transfer.
  - **Serializing Data:** 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 IC, SPI, and CAN, not all devices support such standards. For this reason we need to be familiar with data serialization using the C language.

Serialization in C (Example 1)

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;
}
```

Serialization in C (Example 2)

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;
}
```

Serialization in C (Example 3)

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;
}
```

Serialization in C (Example 4)

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;
}
```

- Initialization:
- conbyte is initialized with the value 0x44.
- regALSB is used to hold the current bit to be sent.
- DDRC |= (1 << serPin); sets PORTC pin 3 as an output pin.
- (1 << serPin): This part shifts the number 1 to the left by serPin positions.
- Shift Operation: 1 << serPin shifts the bit pattern 00000001 left by the number of positions specified by serPin. If serPin is 3, the result is 00001000
- Loop:
- The for loop runs 8 times, once for each bit in the byte.
- Inside the loop, the least significant bit (LSB) of regALSB is checked.
- If the LSB is 1, PORTC pin 3 is set high; otherwise, it is set low.
- regALSB is then right-shifted by 1 bit to prepare for the next iterat

Memory Allocation

Memory Allocation In C

- Using program (code) space for predefined fixed data is a widely used option in the AVR.
- access the data stored in ROM with C language.

Flash, RAM, and EEPROM data space in AVR

- In the AVR we have three spaces in which to store data. They are as follows:
- **SRAM:**
  - The 64K bytes of SRAM space with address range 0000-FFFFH.
  - 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.
- **On-Chip Flash ROM:**
  - 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.
- **EEPROM.**
  - 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.

EEPROM Programming with C

- 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";
  - eeprom unsigned char = 7

Example - 1

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 - 2

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
}
```