CHAPTER 3

AVR Peripherals Programming in C

Objectives ...

- To learn timers of AVR and its Programming.
- To understand AVR serial Programming.
- To know the details of I2C and SPI used in AVR.
- To study AVR on-chip ADC.

INTRODUCTION

AVR microcontrollers include built-in timers/counters which are used for time-based operation like generating delays, measuring pulse widths, creating PWM signals or counting events. AVR also has serial port which is used for serial programming and ADC to convert analog data into digital data.

AVR also use I2C (Inter-Integrated Circuit) which is a two-wire, half-duplex, multi-master, multi-slave serial communication protocol and SPI (Serial Peripheral Interface) which is a four-wire, full-duplex, single-master, multiple-slave serial protocol.

The details of all these peripherals used in AVR are discussed in this chapter.

3.1 AVR TIMER PROGRAMMING

- Basically, timer is a simple counter. The main advantage of timer is that, the input clock and operation of the timer is independent of the program execution. The clock makes it possible to measure time by counting the elapsed cycles and take the input frequency of the timer into account. Thus, timers can be used to generate the delays.
- AVR microcontrollers typically have 8-bit and 16-bit timers, such as Timer0, Timer1 and Timer2. Because of the flexibility of the AVR timers, they can be used for different purposes. Timers can also be used to monitor several events.
- Timers can be used as Internal timer, External counter or PWM (Pulse width Modulation) Generator.
- Internal timer ticks on the oscillator frequency which can be pre-scaled. In this mode, timer generates precise delays.
- In External Counter mode, the timer is used to count events on a specific external pin on a microcontroller.
- PWM is used in speed control of motors and various other applications.
- Atmega32 has 3 timer units, timer0 (8-bit), timer1 (16-bit) and timer2 (8-bit).

3.2 DIFFERENCE BETWEEN TIMER AND COUNTER OPERATION

The main difference between Timer and Counter operations in AVR microcontrollers lies in, what drives the increment of the register, internal clock vs external signal.

Table 3.1: Difference between Timer and Counter

Sr. No.	Aspect	Timer	Counter
1.	Clock Source	Internal clock (derived from CPU clock)	External clock (signal on Tn pin like T0 or T1)
2.	Use Case	Measure time intervals (delays, timeouts, scheduling)	Count external events (pulses, button presses, etc.)
3.	Increment Trigger	Increments on each internal clock tick (after prescaler)	Increments on each external edge (rising/falling)
4.	Prescaler	Prescaler available to slow down the internal clock	Usually no prescaler (or limited edge selection)
5.	Register Used	TCNTn (Timer/Counter Register)	Same TCNTn register used
6.	Configuration	Set clock source via CSn bits in TCCRn	Set external edge trigger (falling/rising) using CSn bits
7.	Example	Delays, PWM generation, scheduling	Event counting, RPM measurement, frequency counting

3.3 SPECIAL FUNCTION REGISTERS (SFRs)

Types of SFRs used in AVR:

SFRs used in AVR are listed below:

1. **TCCRn - Timer/Counter Control Register:** It is used to select mode, prescaler, waveform generation and compare output.

   For Timer0, TCCR0 is used, for Timer1, TCCR1A, TCCR1B are used and for Timer2, TCCR2 is used.

2. **TCNTn - Timer/Counter Register:** It holds the current timer count value. TCNT0, TCNT1, TCNT2 are the registers for Timer 0, 1 and 2 respectively. It is set to zero to start counting fresh. From this register, current count can be read.

3. **OCRn - Output Compare Register:** It is used in CTC or PWM modes to trigger an action when the counter matches this value.

   OCR0, OCR1A, OCR1B, OCR2 are the registers for Timer 0, 1 and 2 respectively.

4. **TIMSK - Timer Interrupt Mask Register:** It enables specific timer interrupts.

5. **TIFR - Timer Interrupt Flag Register:** It indicates if a timer interrupt has occurred. It is cleared by software.

6. **ICR1 - Input Capture Register (Timer1 only):** It is used to capture a timer value on an external event.

7. **ASSR - Asynchronous Status Register (Timer2):** It is used for controlling Timer2 with an external crystal or asynchronous clock.

Modes of Timers:

Timers can operate in various modes as:

- **Normal mode:** Timer counts up and overflows at maximum value.
- **CTC (Clear Timer on Compare Match) mode:** Timer resets when it matches a specified value.
- **PWM modes:** It is used for pulse-width modulation output.
- **Fast PWM and Phase Correct PWM:** It generates precise PWM signals.

All three timers and their special function registers are discussed in detail below.

1. **Timer0:**

Timer0 is an 8-bit timer i.e. it can count from 0 to 255. It uses TCNT0 and TCCR0 registers.

(i) **TCNT0: Timer / Counter Register 0:** It is an 8-bit register. It is continuously incremented by each pulse. It holds the current timer count value.

TCNT0:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TCNT0[7]	TCNT0[6]	TCNT0[5]	TCNT0[4]	TCNT0[3]	TCNT0[2]	TCNT0[1]	TCNT0[0]

- Each bit represents part of the 8-bit counter. Its value range is from 0x00 to 0xFF (0 to 255).
- Reading TCNT0 gives the current count value.
- Writing to TCNT0 sets the counter to a specific value.
- To start the timer from a specific count, a count value has to be loaded in TCNT0. A value can be set in the Output Compare Register (OCR0) and whenever TCNT0 reaches that value, the Output Compare Flag (OCF0) flag is Set. Thus, exact clock pulses can be counted.
- TCNT0 only counts when the timer is enabled through TCCR0.
- It overflows after 255 and wraps back to 0, setting TOV0 in TIFR.

(ii) **TCCR0: Timer/Counter Control Register 0:** TCCR0 register is used to configure Timer/Counter mode of operation including waveform generation, clock source selection, prescaling.

The configuration of the Timer can be set using the TCCR0 register. The Frequency of the Clock Source can be selected by CS02, CS01, CS00 bits.

TCCR0:

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

Description of each bit is:

Bit	Name	Description
7	FOC0	Force Output Compare (used in non-PWM modes to force compare match)
6	WGM00	Waveform Generation Mode bit 0 (along with WGM01 for setting timer mode)
5	COM01	Compare Match Output Mode bit 1 (controls OC0 pin behavior)
4	COM00	Compare Match Output Mode bit 0
3	WGM01	Waveform Generation Mode bit 1
2	CS02	Clock Select bit 2 (used to set prescaler)
1	CS01	Clock Select bit 1
0	CS00	Clock Select bit 0

Clock Select bits in the TCCR0 register (CS02, CS01, CS00) determine the clock source and prescaler for Timer/Counter0 as shown in below table:

CS02	CS01	CS00	Timer Clock Source	Description
0	0	0	No clock source	Timer is stopped
0	0	1	Clock (No prescaling)	Timer clock = fCPU
0	1	0	Clock / 8	Prescaler = 8
0	1	1	Clock / 64	Prescaler = 64
1	0	0	Clock / 256	Prescaler = 256
1	0	1	Clock / 1024	Prescaler = 1024
1	1	0	External Clock on T0 (falling edge)	Timer counts on falling edge
1	1	1	External Clock on T0 (rising edge)	Timer counts on rising edge

WGM01 and WGM00 bits in the TCCR0 register (Timer/Counter Control Register) define the waveform generation mode of Timer/Counter0 as shown below:

WGM01	WGM00	Mode	Description
0	0	Normal Mode	Timer counts from 0 to 255 and overflows to 0.
0	1	PWM, Phase Correct	Timer counts up and down (0→255→0), used for phase-correct PWM.
1	0	CTC (Clear Timer on Compare Match)	Timer resets when it matches OCR0; used for precise timing.
1	1	Fast PWM	Timer counts from 0 to 255 continuously; generates high-speed PWM signals.

(iii) TIFR (Timer/Counter Interrupt Flag Register): It is used to indicate the status of interrupt flags for all timers (Timer0, Timer1, Timer2). Each bit in the TIFR register represents a specific interrupt condition, such as overflow or compare match. The bit structure of TIFR of ATmega16/32 is given below:

D7	D6	D5	D4	D3	D2	D1	D0
–	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	TOV0

The description of each bit is given below:

Bit	Name	Description
7	–	Reserved (unused)
6	OCF2	Output Compare Flag for Timer2 – set when TCNT2 matches OCR2
5	TOV2	Timer2 Overflow Flag – set on overflow (from 0xFF to 0x00)
4	ICF1	Input Capture Flag for Timer1 – set when input capture occurs
3	OCF1A	Output Compare A Match Flag for Timer1 – set on compare match with OCR1A
2	OCF1B	Output Compare B Match Flag for Timer1 – set on compare match with OCR1B
1	TOV1	Timer1 Overflow Flag – set on overflow
0	TOV0	Timer0 Overflow Flag – set on overflow (from 0xFF to 0x00)

In AVR, interrupt flags are cleared by writing a '1' to the respective bit (not '0').

2. Timer1:

The Timer 1 is 16-bit timer, so it will count from 0 to 655356 (2^{16}).

TCNT1 (Timer/Counter1 Register): Timer/Counter 1 is a 16-bit timer having TCNT1H and TCNT1L as shown below:

TCNT1H								TCNT1L							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

TCNT1H holds higher byte (bits 15 to 8) and TCNT1L holds lower byte (bits 7 to 0).

TCCR1 (Timer/Counter1 Control Registers): It consists of TCCR1A and TCCR1B.

(i) TCCR1A – Timer/Counter1 Control Register A:

Bit No.	7	6	5	4	3	2	1	0
Bit Name	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10

Bit description:

COM1A1 and COM1A0 : Compare Output Mode for channel A

COM1B1 and COM1B0 : Compare Output Mode for channel B

WGM11 and WGM10 : Waveform Generation Mode (along with WGM12 & WGM13 from TCCR1B)

Bits 3 and 2 are reserved or unused in many configurations.

(ii) TCCR1B – Timer/Counter1 Control Register B:

Bit No.	7	6	5	4	3	2	1	0
Bit Name	ICNC1	ICES1	–	–	–	WGM13	WGM12	CS12

Bit description:

ICNC1 : Input Capture Noise Canceler

ICES1 : Input Capture Edge Select

WGM13 and WGM12 : Waveform Generation Mode bits (with WGM11,WGM10 from TCCR1A)

CS12, CS11, CS10 : Clock Select bits are for prescaler selection (similar to CS00,01,02)

Bits 3,4,5 are unused.

3. **Timer2:**

Timer2 is an 8-bit timer, similar to above timers. So, same logic can be used to program Timer2.

3.3.1 Timer Modes

There are 4 timer modes which are listed below:

1. **Normal Mode:**

In normal mode, timer counts from 0 to max count, 255 for 8-bit counter and 65535 for 16-bit and then overflows to 0. On overflow, TOVn flag in TIFRn is set. This mode is used for delay and time measurement.

Steps to Program Timer0 in Normal Mode:

To generate a delay using timer0 in Normal mode, steps to be followed as:

(i) Load TCNT0 register with the initial count value.

(ii) Load the value into the TCCR0 register which will indicate the mode (8-bit or 16-bit) and the prescaler option. When the clock source is selected, the timer/counter starts to count and each tick will increment the timer/counter by 1.

(iii) Keep monitoring the timer overflow flag (TOV0) to check whether it is raised. Come out of the loop, when it becomes high.

(iv) Stop the timer by discontinuing the clock source.

Steps to find the value to be loaded into the timer:

(i) Calculate the period of Timer clock as:

$$T_{clock} = \frac{1}{F_{timer}}$$

F_{timer} is the frequency of the clock used for timer.

(For no prescaler mode, $T_{clock} = F_{timer}$)

T_{clock} gives the period at which timer increments.

(ii) Divide the desired time delay by T_{clock}. It gives how many clocks (n) are needed.

(iii) Calculate 256 – n, where n is the value as per step 2

(iv) Convert the result of step 3 into hex (xx) which is to be loaded into the timer's register.

(v) Set TCNT0 = xx

2. **CTC Mode (Clear Timer on Compare Match):**

In this mode, Timer resets when it reaches a value in OCRn. It generates precise time delays or frequencies. It is useful for wave generation.

In CTC mode, the timer is incremented with a clock. It will count up until the content of TCNT0 will become equal to the content of OCR0 (compare math occurs) then timer will be cleared and OCF0 flag will be set when the next clock occurs. OCF0 is located in TIFR register.

3. **PWM Modes:**

There are two PWM modes; Fast PWM and Phase correct PWM mode.

(a) **Fast PWM Mode:** In this mode, Timer counts from 0 to TOP and resets. It is high-frequency PWM with adjustable duty cycle. It is used in motor control, LED dimming.

(b) **Phase Correct PWM Mode:** In this mode, Timer counts up to TOP and then counts down. It generates symmetrical PWM waveform. It generates lower frequency than Fast PWM but avoids glitches.

WGM00 and WGM01 sets the waveform generation mode as discussed in pervious section.

3.4 C PROGRAMS FOR DELAY GENERATION

To generate delays, calculations have to be done and steps have to be followed as mentioned in previous section.

Examples:

1. **Write a C program to toggle LED connected to PD4 every 100 msec using Timer0 with 1024 prescalar in normal mode.**

Calculations:

Calculations of Max delay Timer0 overflow:

Assume, the highest pre-scalar of 1024, crystal frequency 16 MHz.

$$\text{Timer frequency} = \frac{16\,\text{MHz}}{1024} = 15{,}625\ \text{Hz}.$$

Timer0 is 8-bit timer, so it will count from 0 to 256 (2^8).

$$\text{Timer overflow time} = \frac{256}{15{,}625} = 16.384\ \text{ms}$$ (as timer 0 is 8 bit timer, total count 256).

Calculation shows, it can generate a delay of 16 ms every time when timer zero overflows.

16 ms is not enough, so number of times, it has to be overflowed to generate 100 ms delay.

So, Overflow Count = $\frac{100\ \text{ms}}{16.384\ \text{ms}} \approx 6.1$, so 6 overflows gives ~ 98.3 ms, approximately 100 msec.

A simple program which will toggle a port pin (PD4) after the timer 0 overflows 6 times can be written using flowing steps:

1. Load TCNT0 with 0x00.
2. Set CS00 and CS02 bits in TCCR0 register. This will start the timer.
3. Monitor the TOV0 flag in the TIFR0 register to check if the timer has over-flowed, keep a timerOverFlowCount.
4. If timerOverFlowCount>= 6, toggle the LED on PD4 and reset the count.

AVR C Program:

```
#include <avr/io.h>
#define LED PD4
int main ()
{
   uint8_t timerOverflowCount=0;
   DDRD=0xff;          //configure PORTD as output
   TCNT0=0x00;
   TCCR0 = (1<<CS00) | (1<<CS02);
   while (1)
   {
      while ((TIFR & 0x01) == 0);
      TCNT0 = 0x00;
      TIFR=0x01;       //clear timer1 overflow flag
      timerOverflowCount++;
      if (timerOverflowCount>=6)
      {
         PORTD ^= (0x01 << LED);
         timerOverflowCount=0;
      }
   }
}
```

2. **Write a C program to toggle LED connected to PD4 every 100 msec. Use Timer1 (16-bit timer), crystal frequency 16 MHz, prescaler 1024.**

Assume,

Timer: Timer1 (16-bit)

Mode: CTC (Clear Timer on Compare Match)

Prescaler: 1024

Clock Frequency: 16 MHz

Calculations:

$$\text{Timer frequency} = \frac{16{,}000{,}000}{1024} = 15625 \text{ Hz}$$

$$\text{Timer Tick Duration} = \frac{1}{\text{F timer}} = \frac{1}{15625} = 64 \ \mu s$$

A 16-bit timer (Timer1) counts from 0 to 65535 (2^{16}).

So, maximum time delay using 16-bit timer is:

$T_{max} = 65536 \times 64 \ \mu s = 4.194$ sec

For 100 ms delay:

$$\frac{100 \text{ ms}}{64 \ \mu s} = \frac{100{,}000 \ \mu s}{64 \ \mu s} = 1562.5$$

To get 100 ms delay, approximately 1562 timer ticks are required.

CTC (Clear Timer on Compare Match) mode is used, set OCR1A = 1561, as counting starts from 0. When the timer reaches 1561, it resets, completing one 100 ms interval.

Final values to be loaded in OCR1A is 1561 (Decimal) = 0 × 0619 (Hex)

(For calculation, 1562 is used (as total ticks), but since timer starts from 0, 1 is subtracted).

To generate the required delay, following steps are used:

1. Load OCR1H with 0x3d and OCR1L with 0x09.
2. Run the timer with pre-scalar of 1024 by setting CS12 and CS10 bits.
3. Monitor OCF flag and if it is set, toggle the LED.
4. Reset the TCNT1L and TCNT1H values to zero and repeat steps 1 to 3.

AVR C Program:

```
#include <avr/io.h>
void delay_100ms_CTCTimer1(void)
{
   TCCR1A = 0x00;        // Normal port operation
   TCCR1B = (1 << WGM12) | (1 << CS12) | (1 << CS10);
                                          // CTC mode, prescaler 1024
   OCR1A = 1561;         // Compare match value for 100 ms
   TCNT1 = 0;            // Reset Timer counter
   while ((TIFR & (1 << OCF1A)) == 0);  // Wait for compare match flag
   TIFR |= (1 << OCF1A);                // Clear the compare match flag
}
int main(void)
{
   DDRD |= (1 << PD4);  // Set PD4 as output
```

```
    while (1)
    {
        PORTD ^= (1 << PD4);          // Toggle PD4
        delay_100ms_CTCTimer1();      // Delay using Timer1
    }
}
```

3. **Write a C program to toggle LED connected to PD4 after every 100 msec. Use timer 2, crystal frequency 16 MHz. Use polling (no interrupts) and Normal Mode.**

Calculations:

Crystal frequency, XTALK = 16 MHz

Prescaler is 1024.

$$\text{Timer tick time} = \frac{1}{\left(\frac{16000{,}000}{1024}\right)} = 64\ \mu s$$

$$\text{Count needed for required delay of 100 msec} = \frac{100{,}000}{64\ \mu s} = 1562 \text{ counts}$$

Since Timer 2 is 8-bit (max count = 256), overflow counting is:

$$\text{Number of overflows} = \frac{1562}{256} = 6 \text{ overflows.}$$

Load TCNT2 for partial overflow:

Partial overflow = 1562 – (256 × 6) = 86

Start TCNT2 from = 256 – 86 = 170 (in decimal) = 0×AA (in Hex)

AVR C Program:

```
#include <avr/io.h>
#include <util/delay.h>
void timer2_delay_100ms(void)
{
   // Configure Timer2 in Normal Mode
   TCCR2 = (1 << CS22) | (1 << CS21) | (1 << CS20);   // Prescaler = 1024
   uint8_t overflow_count = 0;
   // Load TCNT2 for partial count first
   TCNT2 = 170;
   while (overflow_count< 7)
   {
      // Wait until TOV2 is set
      while ((TIFR & (1 << TOV2)) == 0);
      TIFR |= (1 << TOV2);          // Clear TOV2 by writing 1
```



```
      if (overflow_count == 0)
      TCNT2 = 0;
      overflow_count++;              // Full count for remaining 6 overflows
   }
}
int main(void)
{
   DDRD |= (1 << PD4);
   while (1)                         // Set PD4 as output
   {
      PORTD ^= (1 << PD4);
      timer2_delay_100ms();          // Toggle LED
   }                                 // Wait for 100 ms
}
```

4. **Write a C program to toggle PB4 continuously every 2 msec. Use Timer1, Normal mode and no prescaler to create delay. Assume crystal frequency XTAL = 8 MHZ.**

XTALK = 8MHZ

For no prescaler mode, $T_{clock} = F_{timer}$

$$T = \frac{1}{8 \text{ MHz}} = 0.125\ \mu s$$

$$\text{Count needed for delay of 2 msec.} = \frac{2 \text{ msec}}{0.125\ \mu s} = 16000 \text{ clocks} = 0 \times 3E80 \text{ clocks}$$

A 16-bit timer (Timer1) counts from 0 to 65535 (2^{16}).

1 + 0FFFF – 0 × 3E80 = 0 × C180

AVR C Program:

```
#include <avr/io.h>
#include <util/delay.h>
Void T1Delay ();
int main ()
{
   DDRB = 0xFF:     //PORT B configured as output port
   while (1)
   {
      PORTB = PORTB^(1<<PB4);     // toggle PB4
      T1Dealy ();                 // some delay
   }
}
```

```
void T1Dealy()
{
   TCNT1H = 0xC1H;
   TCNT1L = 0x80;
   TCCR1A = 0x00:        //normal mode
   TCCR1B = 0x01;        // normal mode, no prescaler
   while ((TIFR &(0x1<<TOV1)) == 0)
                         // wait till TOV roll over
   TCCR1B = 0;
   TIFR = 0x1<<TOV1;     //clear TOV1
}
```

3.5 COUNTER PROGRAMMING

Counters in AVR microcontrollers work similarly to timers but increment their value based on external events (clock pulses) rather than internal clock ticks.

Timer Mode counts internal clock pulses, whereas, Counter Mode counts external events on a specific pin.

Steps to Program AVR Counter for Timer 0 as Counter:

1. Select Counter Mode by setting TCCR0 appropriately, use CS01, CS00, CS02 bits in TCCR0 to select external clock source and external Clock on T0 pin (PD4) for Timer0.

Bits (CS02 – CS00)	Mode
110	External clock on T0 pin, falling edge
111	External clock on T0 pin, rising edge

2. Configure TCCR0 Register:

   `TCCR0 = (1 << CS01) | (1 << CS00);` // Falling edge external clock (for example)

3. Clear or Set the Counter Register (TCNT0):

   `TCNT0 = 0x00;` // Start from zero

4. Read Counter Value:

   `uint8_t count = TCNT0;`

5. Use Polling or Interrupts:

   poll TCNT0 is used regularly or use overflow interrupt if needed.

Example:

1. Count pulses on T0 pin and light LED after 100 counts.

   (Timer 0 uses pin PD4 (T0) to count external pulses)

AVR C Program:

```
#include <avr/io.h>
int main(void)
{
   DDRB |= (1 << PB0);                    // Set PB0 as output (LED)
   TCCR0 = (1 << CS01) | (1 << CS00);     // Counter mode on falling edge
   TCNT0 = 0;                             // Clear counter
   while (1)
   {
      if (TCNT0 >= 100)
      {
         PORTB ^= (1 << PB0);             // Toggle LED
         TCNT0 = 0;                       // Reset counter
      }
   }
}
```

3.6 AVR SERIAL PORT PROGRAMMING

AVR serial programming is the process of uploading code and communicating with the microcontroller using serial communication protocols UART, SPI, or I2C.

3.6.1 Serial vs Parallel Communication

Serial communication sends data bit-by-bit over a single wire, unlike parallel communication (multiple lines), serial communication is simpler and uses fewer pins as shown in below Fig. 3.1.

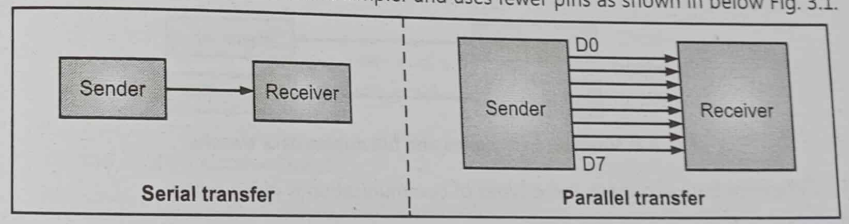

Fig. 3.1: Serial and Parallel Communication

Table 3.2: Difference between Serial and Parallel Communication

Sr. No.	Feature	Serial Communication	Parallel Communication
1.	Data Transmission	Sends data one bit at a time	Sends multiple bits simultaneously
2.	Number of Wires	Fewer wires (usually 1-2)	More wires (1 wire per bit + control lines)

contd. ...

3.	Speed	Slower (but can be faster over long distances)	Faster (over short distances)
4.	Cost	Cheaper (less wiring)	Expensive (more hardware needed)
5.	Distance	Suitable for long distances	Effective only over short distances
6.	Examples	UART, USB, SPI, I2C	Internal data buses, printer ports

3.6.2 Simplex vs Duplex

Data can be sent in three different ways:

1. **Simplex:** Data can be sent in one direction only.
2. **Half-Duplex:** Data can be sent in both directions, but one at a time and not simultaneously.
3. **Full Duplex:** Data can be sent both ways at the same time or simultaneously.

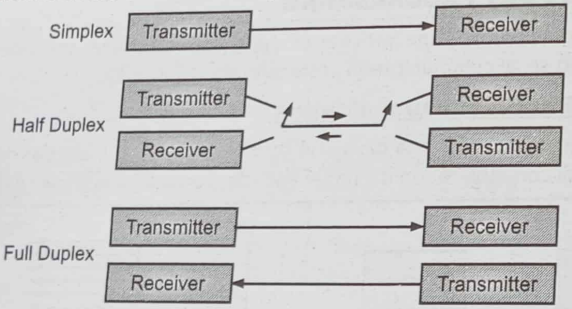

Fig. 3.2: Simplex, half duplex and full duplex data transfer

The Difference between these three types of communication is given below:

Table 3.3

Type	Direction	Description	Example
Simplex	One-way only	Data flows in one direction only	TV Broadcast
Half-Duplex	Both, but one at a time	Data flows in both directions, one at a time	Walkie-Talkie
Full-Duplex	Both simultaneously	Data flows both ways at the same time	Telephone, UART

3.6.3 Asynchronous and Synchronous Serial Communication

There are two types of serial communication; synchronous and asynchronous.

Asynchronous Communication:

- In asynchronous transmission method, each data word is accompanied by a stop and start bits that identifies the beginning and ending of the word.
- When no information is being transmitted, the communication line is usually high or digitally in logic '1' state. In data communication, this is referred to as a mark.
- To understand asynchronous transmission, consider example of a character as shown in below Fig. 3.3.

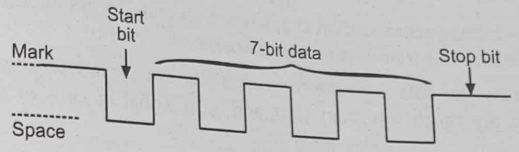

Fig. 3.3: Asynchronous transmission with start and stop bit

- As shown in the Fig. 3.3 a start bit is transmitted to indicate the beginning of the word. It is a binary '0' or space as shown in Fig. 3.3.
- The transmission of the start bit indicates the receiving circuit to prepare themselves for reception of the remainder of the bits.
- Hence after the start bit, other individual 7 bits ASCII code of the word is transmitted along with the stop bit.
- The stop bit is logic '1' level bit, which indicates end of the data.

Synchronous Communication:

- In the synchronous technique, start and stop bits are not transmitted, but the data is transmitted in multiword blocks with synchronization bits along with them.
- A group of synchronization bits is placed at the beginning of the multiword block and at the end of the multiword block.
- This makes synchronization between transmitter and receiver. To understand synchronous data transmission, consider an example as shown in Fig. 3.4.

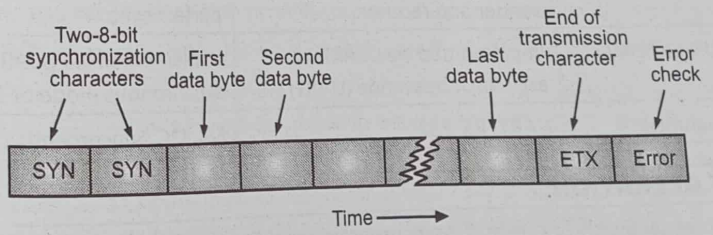

Fig. 3.4: Synchronous data transmission

- As shown in the above Fig. 3.4 each block of data contains a data of large characters.
- At the beginning of each block unique series of bits identifying beginning of the block is added, whereas at the end of the block another special code ETX signaling end of transmission is attached.
- The two 8-bit synchronization (SYN) characters' signal indicates start of transmission which is also indication for receiver to receive the continuous data after it.
- As soon as ETX code is received, receiver recognizes the end of transmission.
- In between this SYN and ETX signal, some error detection and correction codes are also added.
- As the extra 8-bit synchronization characters, SYN and ETX are added to data signal, the speed of synchronous transmission is extremely high.
- This is because the data is in the form of multiword block and also the SYN and ETC signals used are much less than start and stop signal in case of asynchronous data transmission.
- Thus, synchronous data transmission is much faster than asynchronous data transmission.

Table 3.4: Difference between Asynchronous and Synchronous Serial Communication

Sr. No.	Feature	Asynchronous Communication	Synchronous Communication
1.	Clock	No shared clock between sender and receiver (separate clock used for transmitter and receiver).	Uses a shared clock to synchronize data transmission (same clock used for transmitter and receiver).
2.	Data Synchronization	Data is synchronized using start and stop bits.	Data is synchronized with the clock signal or sync pulses.
3.	Speed	Slower due to overhead of start/stop bits.	Faster due to continuous stream of data.
4.	Hardware Complexity	Simpler (fewer lines: Tx, Rx, GND).	More complex (requires an additional clock line).
5.	Timing	Independent timing between sender and receiver.	Sender and receiver must be in perfect sync.
6.	Usage in AVR	Implemented via USART in asynchronous mode (UART).	Implemented via USART in synchronous mode or SPI.
7.	Examples	UART, RS-232.	SPI, I2C, Synchronous USART.

3.7 USART OPERATION

USART (Universal Synchronous and Asynchronous Serial Receiver and Transmitter) consists of Central Processing Unit (CPU) accessible I/O Registers and I/O pins as shown in below Fig. 3.5.

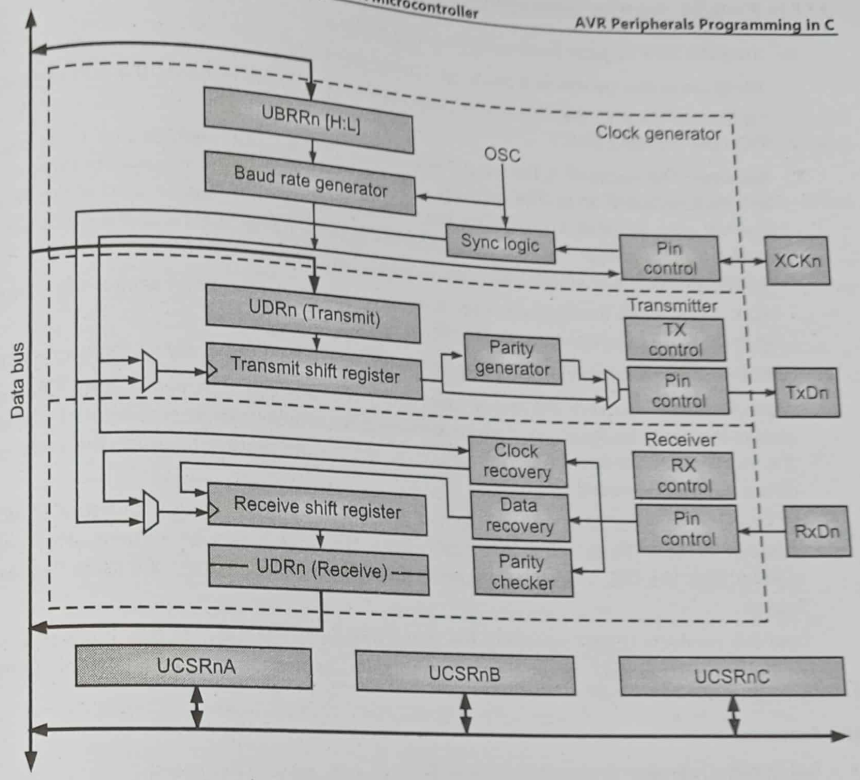

Fig. 3.5: USART block diagram

Source: From data sheet of Atmel.

There are three main blocks of USART separated by dashed lines as shown in above Fig. 3.5.

1. **Clock Generator:** It consists of synchronization logic for the external clock input used by synchronous slave operation and the baud rate generator. UBRRn[H:L] (USART Baud Rate Register High & Low) sets the baud rate (communication speed). Baud rate generator uses values and oscillator input to generate proper timing for transmission/reception.

   The XCKn (Transfer Clock) pin is only used by Synchronous Transfer mode.

   SYNC Logic used in synchronous mode to synchronize with external clock input (XCKn) pin which is used in synchronous mode to transmit/receive clock externally.

2. **Transmitter:** The Transmitter consists of a single write buffer, a serial Shift Register, Parity Generator, and Control logic for handling different serial frame formats. The write buffer allows a continuous transfer of data without any delay between frames.

   UDRn (Transmit) is USART Data Register for writing data to be transmitted.

Transmit Shift Register shifts out data bit by bit serially.

Parity Generator generates a parity bit if parity is enabled.

TX Control controls the transmission process and Pin Control manages the output to the TxDn pin (transmit pin).

3. **Receiver:** The Receiver is the most complex part of the USART module due to its clock and data recovery units. The recovery units are used for asynchronous data reception. Receiver also includes a Parity Checker, Control logic, a Shift Register and a two level receive buffer (UDRn). The Receiver supports the same frame formats as the Transmitter and can detect Frame Errors, Data OverRun, and Parity Errors.

RxDn (Receive Pin) through which serial data is received. Pin Control manages input from the RxDn pin. Rx Control controls the reception operation.

Clock Recovery extracts clock signal from the received data (used in asynchronous mode).

Data Recovery recovers the actual data from the received bitstream. Parity Checker checks the parity bit for errors if parity is enabled. Receive Shift Register temporarily holds the incoming serial data bits. UDRn (Receive) is the Data register to which the received data is eventually loaded and read by the CPU.

UCSRnA, UCSRnB, UCSRnC (USART Control and Status Registers A/B/C) configure USART behavior (enable Tx/Rx, set frame size, parity, stop bits, interrupt settings, etc.) and monitor flags like TXC (Transmit Complete), RXC (Receive Complete), UDRE (Data Register Empty)

Data Bus connects USART system to the microcontroller's internal data bus. It allows CPU to interact with USART by writing/reading UDRn, setting baud rate and configuring registers.

USART Features:

- Full Duplex Operation (Independent Serial Receive and Transmit Registers).
- Asynchronous or Synchronous Operation.
- Master or Slave Clocked Synchronous Operation.
- High Resolution Baud Rate Generator.
- Supports Serial Frames with 5, 6, 7, 8, or 9 data bits and 1 or 2 stop bits.
- Odd or Even Parity Generation and Parity Check Supported by Hardware.
- Data OverRun Detection.
- Framing Error Detection.
- Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter.
- Three Separate Interrupts on TX Complete, TX Data Register Empty, and RX Complete.
- Multi-processor Communication mode.
- Double Speed Asynchronous Communication mode.
- Start Frame Detection.

3.7.1 Synchronous Clock Operation

Synchronous communication is used in systems with a single master and one or more slaves. Unlike asynchronous communication, no start or stop bits are typically used, making it faster and more efficient.

Key signals like Bidirectional Data Line (TxDn/RxDn – shared line) and Clock Line (XCKn driven by master) are used.

Master-Slave configuration uses Master which generates the baud rate clock and drives the clock signal on XCKn. Slave uses external clock from master via XCKn. There is no need for internal baud rate generator.

In Data Synchronization, Data Output (on TxDn) changes at one clock edge, Data Input (on RxDn) is sampled on the opposite clock edge. This ensures reliable and synchronized data transfer.

USART Mode Selection:

- Set UMSELn = 1 in UCSRnC register to enable synchronous mode.
- Set UCPOLn to configure clock edge polarity:
  - **0:** Sample on rising, change on falling edge.
  - **1:** Sample on falling, change on rising edge.

3.7.2 Asynchronous Data Reception

In Asynchronous communication, two pins are used; Transmit (TxDn) and Receive (RxDn).

The TxDn of one device connects to the RxDn of the other device.

Each character transmission consists of; 1 Start bit (always a space logic level), 5 to 9 data bits and 1 or more Stop bits (always marks logic level).

Each bit duration $= \frac{1}{\text{Baud rate seconds}}$. The baud rate controls the speed of operation.

An on-chip Baud Rate Generator derives standard baud rate frequencies from the system oscillator.

3.7.3 Clock Generation

The Clock Generation logic generates the base clock for the Transmitter and Receiver. The USART supports four modes of clock operation:

- Normal asynchronous
- Double Speed asynchronous
- Master synchronous
- Slave synchronous

Number of signals are used for clock generation. The signal txclk is transmitter clock (internal signal), rxclk is receiver base clock (internal signal), xcki is input from XCK pin (internal signal) which is used for synchronous slave operation. xcko is clock output to XCK pin (internal signal) which is used for synchronous master operation and OSC is System clock frequency.

3.7.4 Baud Rate Generator

Internal clock generation is used for the Asynchronous and Synchronous Master modes of operation. The USART Baud Rate Register (UBRRn) and the down-counter connected to it function as a programmable prescaler or baud rate generator. The down counter, running at system clock (fosc), is loaded with the UBRRn value, each time the counter has counted down to zero or when the UBRRnL Register is written. A clock is generated each time the counter reaches zero. This clock is the baud rate generator clock output (= fosc/(UBRRn + 1)).

The table below contains equations for calculating the baud rate (in bits per second) and for calculating the UBRRn value for each mode of operation using an internally generated clock source.

Operating Mode	Equation for Calculating Baud Rate (1)	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2X = 0)	$BAUD = \frac{f_{osc}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{16BAUD} + - 1$
Asynchronous Double Speed mode (U2X = 1)	$BAUD = \frac{f_{osc}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{8BAUD} + - 1$
Synchronous Master mode	$BAUD = \frac{f_{osc}}{2(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{2BAUD} + - 1$

3.8 SFR (SPECIAL FUNCTION REGISTER)

The USART uses several Special Function Registers (SFRs) to manage serial communication. These registers help in configuration, data transmission and status monitoring. SFRs are listed below:

1. UDR – USART I/O Data Register:

It is used for both transmission and reception of data. It loads data into the transmit buffer and reads data from the receive buffer.

Bit structure of UDR is given below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UDR7	UDR6	UDR5	UDR4	UDR3	UDR2	UDR1	UDR0

These bits represent the data bits D7-D0 of the character to be transmitted or that has been received. UDR7 to UDR0 hold the actual data byte.

2. UCSRA – USART Control and Status Register A:

The UCSRA (USART Control and Status Register A) is an 8-bit register that provides important status flags and one control bit for USART operation.

The bit structure of UCSRA is as follows:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM

Bit explanation is given in below table.

Bit	Name	Full Form	Description
7	RXC	Receive Complete	Set when a character is received and ready to be read from UDR. Cleared by reading UDR.
6	TXC	Transmit Complete	Set when the entire frame in the transmit shift register has been shifted out. Cleared by writing a 1 to it.
5	UDRE	USART Data Register Empty	Set when UDR is ready to receive new data for transmission.
4	FE	Frame Error	Set if the next character in the receive buffer had a framing error (missing stop bit).
3	DOR	Data OverRun	Set if a data overrun has occurred (new data received before previous was read).
2	PE	Parity Error	Set if a parity error occurred in received data (only if parity is enabled).
1	U2X	Double the USART Transmission Speed	When set, doubles the baud rate in asynchronous mode.
0	MPCM	Multi-processor Communication Mode	Used to enable multi-processor communication in asynchronous mode.

3. UCSRB – USART Control and Status Register B:

The bit structure of UCSRB – USART Control and Status Register B is given below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

Explanation of each bit is given in below table.

Bit	Name	Full Form	Description
7	RXCIE	RX Complete Interrupt Enable	Enables interrupt when RXC flag is set (i.e., data received).
6	TXCIE	TX Complete Interrupt Enable	Enables interrupt when TXC flag is set (i.e., transmission done).
5	UDRIE	USART Data Register Empty Interrupt Enable	Enables interrupt when UDRE flag is set (i.e., ready to send next byte).
4	RXEN	Receiver Enable	Enables the USART receiver circuitry.
3	TXEN	Transmitter Enable	Enables the USART transmitter circuitry.
2	UCSZ2	Character Size Bit 2	Used with UCSZ1 and UCSZ0 in UCSRC to select character size (5 to 9 bits).
1	RXB8	Receive Data Bit 8 (for 9-bit data)	Used to read the 9^{th} data bit if 9-bit communication is enabled.
0	TXB8	Transmit Data Bit 8 (for 9-bit data)	Used to write the 9^{th} data bit in 9-bit data transmission.

4. UCSRC – USART Control and Status Register C:

The bit structure of UCSRC – USART Control and Status Register C is given below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

Explanation of each bit given below.

Bit	Name	Full Form	Description
7	URSEL	Register Select	Must be 1 to write to UCSRC (used to differentiate it from UBRRH).
6	UMSEL	USART Mode Select	0 = Asynchronous 1 = Synchronous
5	UPM1	Parity Mode bit 1	Together with UPM1M0 00 = None, 10 = Even, 11 = Odd parity
4	UPM0	Parity Mode bit 0	Combined with UPM1 to select parity
3	USBS	Stop Bit Select	0 - 1 stop bit 1 - 2 stop bits
2	UCSZ1	Character Size bit 1	With UCSZ0 (UCSRC) and UCSZ2 (UCSRB): selects character size
1	UCSZ0	Character Size bit 0	See above.
0	UCPOL	Clock Polarity (for synchronous)	0 = Rising edge 1 = Falling edge (only affects synchronous mode).

UPM bit settings are:

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

USBS Bit Settings.

USBS	Stop Bit(s)
0	1-bit
1	2-bit

UCSZ Bits Settings.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

5. UBRRH and UBRRL – USART Baud Rate Registers:

UBRR is a 16-bit register containing UBRRL and UBRRH.

The UBRRH contains the four MSB (most significant bits) and the UBRRL contains the eight LSB (least significant bits) of the USART baud rate.

Baud Rate Formula for Asynchronous Normal Mode:

$$\text{Baud Rate} = \frac{F_{osc}}{(16 \times \text{UBRR} + 1)}$$

Where, F_{osc} = System oscillator frequency

$$\text{UBRR} = \frac{F_{CPU}}{(16 \times \text{Baud Rate})} - 1$$

If clock frequency (F_CPU) = 16 MHz and Normal (asynchronous) mode (U2X = 0), then baud rates and corresponding decimal / hex values are given in below table.

Baud Rate	UBRR (Decimal)	UBRR (Hex)
2400	416	0x01A0
4800	207	0x00CF
9600	103	0x0067
14400	68	0x0044
19200	51	0x0033
28800	34	0x0022
38400	25	0x0019
57600	16	0x0010
76800	12	0x000C
115200	8	0x0008
230400	3	0x0003
250000	3	0x0003
500000	1	0x0001

UBRR: 12-bit value formed by combining UBRRH and UBRRL.

Bit Structure of UBRRH:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
URSEL	-	-	-	UBRR11	UBRR10	UBRR9	UBRR8

UBRR11–UBRR8: Bits 11 to 8 of the 12-bit baud rate value.

URSEL: Must be 0 when accessing UBRRH (if shared with UCSRC in older AVRs like ATmega16/32)

Bits 6-4 are unused/reserved.

Bit Structure of UBRRL:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UBRR7	UBRR6	UBRR5	UBRR4	UBRR3	UBRR2	UBRR1	UBRR0

These bits represent UBRR[7 - 0], the lower 8 bits of the 12-bit UBRR value.

3.9 C PROGRAMS FOR DATA TRANSMISSION AND RECEPTION

Steps to transfer data serially are as follows:

1. Set the baud rate using the UBRR (USART Baud Rate Register).

   Use formula, $UBRR = \left(\frac{F_{CPU}}{16 \times \text{Baud Rate}}\right) - 1$

2. Configure the USART Control Registers. UCSRB will enable Transmitter.
   UCSRC – Set Frame Format, for 8-bit data, no parity, 1 stop bit.
3. Load data into UDR for transmission. Wait for the transmit buffer to be empty, then send data.
4. Wait until transmission is complete. Clear the flag if necessary.

Following are the C programs for USART data transmission and reception using ATmega16/32. These programs use polling (no interrupts) and assume asynchronous mode, 8-bit data, no parity, 1 stop bit, and baud rate = 9600 bps with F_CPU = 16 MHz.

Common Initialisation Code:

```
#define F_CPU 16000000UL
#define BAUD 9600
#define UBRR_VALUE ((F_CPU / (16UL * BAUD)) - 1)
#include <avr/io.h>
// USART Initialization
void USART_Init(unsigned int ubrr)
{
   // Set baud rate
   UBRRH = (unsigned char)(ubrr>> 8);   // High byte
   UBRRL = (unsigned char)ubrr;         // Low byte
   UCSRB = (1 << RXEN) | (1 << TXEN);   // Enable receiver and transmitter
   UCSRC = (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0);
                                        // 8-bit data, 1 stop bit, no parity
}
```


USART Data Transmission (Sending a Character 'A')

```
// Transmit a single character
void USART_Transmit(unsigned char data)
{
   while (!(UCSRA & (1 << UDRE))); // Wait until UDR is empty
   UDR = data;                     // Load data into transmit register
}
```

Example: Send 'A' continuously

```
int main(void)
{
   USART_Init(UBRR_VALUE);
   while (1)
   {
      USART_Transmit('A');
      for (long i = 0; i< 20000; i++);   // Simple delay
   }
}
```

USART Data Reception (Receiving a Character)

```
// Receive a single character
unsigned char USART_Receive(void)
{
   while (!(UCSRA & (1 << RXC)));    // Wait until data is received
   return UDR;                       // Get and return received data
}
```

Example: Receive a character and retransmit it (Echo)

```
int main(void)
{
   USART_Init(UBRR_VALUE);
   unsigned char receivedChar;
   while (1)
   {
      receivedChar = USART_Receive();   // Receive character
      USART_Transmit(receivedChar);     // Echo it back
   }
}
```

3.10 I2C AND SPI

3.10.1 Introduction of I2C

- I2C (Inter-Integrated Circuit) is a serial communication protocol used to connect low-speed devices like sensors, EEPROMs, RTCs, and displays with microcontrollers using only two wires.
- It was originally started by Philips.

Key Features:

- It is two-wire interface; SDA (Serial Data Line) carries the data and SCL (Serial Clock Line) carries the clock. Both lines are open-drain and require pull-up resistors.
- It supports multiple masters and multiple slaves.
- In synchronous communication – data is transferred with clock.
- Its speed : Standard Mode - 100 kbps, Fast Mode - 400 kbps and High-Speed Mode - up to 3.4 Mbps.
- Master in I2C initiates communication, generates clock. Slave in I2C responds to master's request.

Working:

1. Master sends START condition.
2. Master sends slave address with R/W bit.
3. Slave sends ACK if address matches.
4. Master sends/receives data.
5. Communication ends with STOP condition.

Advantages:

1. Simple 2-wire connection.
2. Supports many devices (up to 127 slaves).
3. Low pin usage.

Limitations:

1. Slower than SPI.
2. Short-distance communication (typically < 1 meter).
3. Shared bus requires careful addressing and timing.

3.10.2 Introduction of SPI

- SPI (Serial Peripheral Interface) is a synchronous, full-duplex serial communication protocol used for fast communication between a master and one or more slave devices.
- It was originally started by Motorola Corporation and then adapted by many semiconductor chip companies.
- It uses only 2 pins for data transfer called SDI (Din) and SDO (Dout).
- The SPI bus has the SCLK (shift clock) and CE (chip enable).
- Thus, SDI, SDO, SCLK and CE makes the SPI 4 -wire interface.

Key Features:

1. It uses 4 signal lines.
2. It can be used for full-duplex as simultaneous transmission and reception.
3. Synchronous data transfer is synchronized with a clock signal.
4. It is faster than I2C.
5. It is typically used for short-distance, high-speed communication.
6. Master controls clock, initiates communication and Slave responds to master.

Working:

1. Master selects slave by pulling SS low.
2. Master generates clock (SCK).
3. Data is shifted out on MOSI and read on MISO.
4. Data is shifted in/out bit-by-bit on each clock edge.
5. SS is pulled high after communication ends.

Advantages:

1. It is faster than I2C.
2. It is simple protocol.
3. It supports full-duplex communication.
4. It is efficient for short-distance, high-speed communication.

Limitations:

1. It requires more pins than I2C.
2. It do not have No acknowledgment system (unlike I2C).
3. It is not suitable for communication over longer distances.
4. In SPI, Master must control multiple SS lines for multiple slaves.

Table 3.5: Difference between I2C and SPI

Feature	I2C (Inter-Integrated Circuit)	SPI (Serial Peripheral Interface)
Lines Used	2 lines: SDA (data), SCL (clock)	4 lines: MOSI, MISO, SCK, SS
Number of Wires	Fewer (2 wires)	More (at least 4 wires)
Speed	Up to 400 kbps (Fast mode), 3.4 Mbps (HS)	Typically up to 10 Mbps or more
Data Direction	Half-duplex (one direction at a time)	Full-duplex (send and receive simultaneously)
Master/Slave Support	Multi-master, multi-slave	Single master, multi-slave
Complexity	More complex protocol	Simpler protocol
Distance	Better for medium distances (up to 1m)	Better for short distances (<1m)
Common Uses	Sensors, EEPROMs, RTCs	SD cards, displays, high-speed sensors
Bus Sharing	Shared bus, all devices connected to SDA/SCL	Each slave needs individual SS from master
Protocol Overhead	Higher (due to addressing and ACK)	Lower (fast and simple)

3.10.3 Specifications

Specifications of I2C and SPI protocols, particularly relevant for embedded systems like AVR microcontrollers (e.g., ATmega16/32) is given below:

I2C (Inter-Integrated Circuit):

Specification	Details
Bus Type	Multi-master, multi-slave
Wires/Lines	2 (SDA - Data, SCL - Clock)
Speed Modes	Standard: 100 kbps, Fast: 400 kbps, Fast Plus: 1 Mbps, High Speed: 3.4 Mbps
Data Direction	Half-duplex
Addressing	7-bit (up to 127 devices), or 10-bit extended addressing
Clock Generation	Only by master
Start/Stop Condition	Required to initiate and terminate communication
Acknowledgment	Yes (ACK/NACK) after each byte
Bus Arbitration	Yes (multi-master supported)
Distance	Up to 1 meter recommended
Error Checking	Basic (ACK/NACK)
Pull-up Resistors	Required on SDA and SCL lines
Power Consumption	Lower (fewer lines switching)
Complexity	Higher due to addressing and protocol control

SPI (Serial Peripheral Interface):

Specification	Details
Bus Type	Single master, multiple slaves (multi-master possible but not common)
Wires/Lines	4 (MOSI, MISO, SCK, SS); one SS line per slave
Speed	Typically up to 10 Mbps or higher
Data Direction	Full-duplex
Addressing	No addressing; slaves selected using separate SS (Slave Select) lines
Clock Generation	Master generates clock
Start/Stop Condition	Not required
Acknowledgment	No acknowledgment system
Bus Arbitration	Not supported
Distance	Short (a few feet at high speed)
Error Checking	Not built-in
Pull-up Resistors	Not required
Power Consumption	Slightly higher (due to more switching lines)
Complexity	Lower (simple and fast protocol)

3.10.4 Bus Signals

I2C Bus signals:

Signal	Name	Direction	Function
SCL	Serial Clock Line	From Master	Carries clock pulses to synchronize data transfer between master and slave
SDA	Serial Data Line	Bi-directional	Transfers data between master and slave (used for both sending and receiving)

SPI Bus Signals:

Signal	Name	Direction (from Master)	Function
MOSI	Master Out Slave In	Output	Master sends data to slave
MISO	Master In Slave Out	Input	Slave sends data to master
SCK	Serial Clock	Output	Clock signal generated by master
SS	Slave Select	Output (per slave)	Selects which slave to communicate with

3.10.5 SFR used in AVR

SFR used for I2C:

In many applications including AVR data sheet, I2C is referred as Two Wire Interface (TWI). TWI in AVR is composed of 4 submodules; bit rate generation module, bus interface unit, address match unit and control unit as shown in below Fig. 3.6.

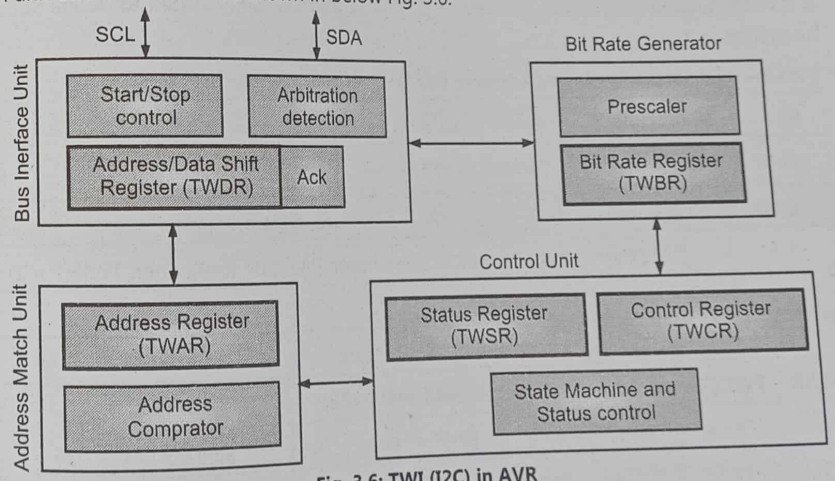

Fig. 3.6: TWI (I2C) in AVR

The bit rate generation unit controls the frequency of the system clock (SCL) when operating in master mode. The bus interface unit detects and generates START, REPEATED START and STOP conditions. It also detects arbitration. The address match unit compares the received address byte with the 7-bit address in TWI address register and informs the control unit upon address match. The control unit controls the TWI module and generates the responses according to the settings in the TWI control register. It also sets the content of status register according to current state.

In AVR, 6 main registers are associated with TWI as given in below table along with their function.

Register	Full Name	Function
TWBR	TWI Bit Rate Register	Sets TWI clock frequency
TWSR	TWI Status Register	Indicates status (e.g., START sent, ACK received)
TWCR	TWI Control Register	Enables TWI, START/STOP control, interrupt flags
TWDR	TWI Data Register	Holds data for transmission or reception
TWAR	TWI (Slave) Address Register	Sets slave address and general call bit
TWAMR	TWI Address Mask Register	Used for address masking in multi-slave configurations

Bit structure and description of each bit of all registers is given below:

1. **TWBR – TWI Bit Rate Register (Address: 0×20):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0

Bits	Symbol	Description
7-0	TWBR[7:0]	Sets the bit rate for TWI

It is used in combination with prescaler bits (TWPS1 and TWPS0 in TWSR) to define SCL clock frequency.

2. **TWSR – TWI Status Register (Address: 0×21):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0

Bits	Symbol	Description
7-3	TWS[7:3]	Status bits: Show current TWI state (only when TWINT = 1)
2	–	Reserved
1-0	TWPS1, TWPS0	Prescaler bits for bit rate

3. **TWAR – TWI (Slave) Address Register (Address: 0×22):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE

Bits	Symbol	Description
7-1	TWS [6:0]	7-bit slave address
0	TWGCE	General Call Recognition Enable (1 = Enable)

4. **TWDR – TWI Data Register (Address: 0×23):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0

Bits	Symbol	Description
7-0	TWD [7:0]	Holds data to be sent/received

5. **TWCR – TWI Control Register (Address: 0×56):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE

Bit	Symbol	Description
7	TWINT	TWI Interrupt Flag (1 = operation complete)
6	TWEA	Enable Acknowledge (1 = ACK returned after receive)
5	TWSTA	TWI START condition bit
4	TWSTO	TWI STOP condition bit
3	TWWC	Write Collision Flag
2	TWEN	TWI Enable (1 = enable TWI operations)
1	–	Reserved
0	TWIE	TWI Interrupt Enable

6. **TWAMR – TWI Address Mask Register (optional, used in slave mode):**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	–

Bits	Symbol	Description
7-1	TWAM [6:0]	Used to mask bits in slave address
0	–	Reserved

Basic Steps in TWI Master Transmission:

1. Initialize TWI (set TWBR and prescaler in TWSR)
2. Send START condition (TWCR)
3. Send slave address with R/W bit
4. Send data byte (write to TWDR)
5. Wait for ACK (check TWSR)
6. Send STOP condition (TWCR)

SFR used for SPI:

There are 3 main registers used for SPI in AVR.

7. SPCR – SPI Control Register (Address: 0×2D):

Bit structure of SPCR is given below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR9

Description of each bit is given below:

Bits	Symbol	Description
7	SPIE	SPI Interrupt Enable (1 = Enable interrupt)
6	SPE	SPI Enable (1 = Enable SPI)
5	DORD	Data Order (1 = LSB first, 0 = MSB first)
4	MSTR	Master/Slave Select (1 = Master, 0 = Slave)
3	CPOL	Clock Polarity (1 = SCK high when idle)
2	CPHA	Clock Phase (1 = Data sampled on trailing edge of SCK)
1	SPR1	SPI Clock Rate Select 1
0	SPR0	SPI Clock Rate Select 0

8. SPSR – SPI Status Register (Address: 0×2E):

Bit structure of SPSR is given below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SPIF	WCOL	–	–	–	–	–	SPI2X

Description of each bit is given below:

Bits	Symbol	Description
7	SPIF	SPI Interrupt Flag (Set when transmission complete)
6	WCOL	Write Collision Flag (Set if SPDR written during transfer)
5-1	–	Reserved (Read as 0)
0	SPI2X	Double SPI Speed Bit (1 = Double speed)

9. SPDR – SPI Data Register (Address: 0×2F):

Bit structure of SPDR is given below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Description of each bit is given below:

Bits	Symbol	Description
7-0	D7-D0	SPI Data Register (Read/Write)

Write: Writing to SPDR starts transmission.

Read: After transmission, the received data is read from here.

3.10.6 Error Handling and Addressing

3.10.6.1 Error Handling and Addressing in SPI

Addressing in SPI:

- SPI is a master-slave protocol without built-in addressing like I2C.
- Selection of slave is handled externally using the /SS (Slave Select) line.
- Multiple slaves are supported using individual /SS lines from the master to each slave.
- Only the slave whose /SS line is low (active) will respond to SPI communication.

Error Handling in SPI:

SPI has limited error handling, as it is a simple protocol.

Main concerns are:

1. **Write Collision Error (WCOL):**
   - Occurs when writing to SPDR while a transmission is ongoing.
   - Detected using the WCOL bit (bit 6) in SPSR.
   - WCOL has to be checked before writing (wait for SPIF (transfer complete) before writing again).
2. **SPI Interrupt Flag (SPIF):**
   - Bit 7 in SPSR.
   - Set when a transmission is complete.
   - SPI must be checked to know when it is safe to write the next byte.

3.10.6.2 Error Handling and Addressing in I2C

Addressing in I2C:

- Supports 7-bit or 10-bit addressing.
- Each slave is assigned a unique address.
- In AVR, the slave address is set in the TWAR (TWI Address Register):
  - Bits 7:1 = Slave address.
  - Bit 0 = General Call Recognition enable.

Error Handling in I2C:

AVR uses the TWSR (TWI Status Register) to identify status and errors:

Error/Condition	TWSR Value (Hex)	Meaning
0×38	Arbitration lost.	Two masters tried to access the bus.
0×20	SLA+W transmitted, NACK received.	Slave did not acknowledge write request.
0×30	Data byte transmitted, NACK received.	Slave did not acknowledge data.
0×48	SLA+R transmitted, NACK received.	Slave did not acknowledge read request.

Always check TWSR after each operation (START, SLA+W, DATA). Repeat START or STOP depending on the error. Handle arbitration loss by restarting communication.

The summarised table is given below:

Feature	SPI	I2C (TWI)
Addressing	External (via SS pin)	Built-in (via TWAR)
Error Handling	WCOL, SPIF	TWSR status codes
Slave Selection	Manual with SS	Automatic via address match
Collision Detection	WCOL	Arbitration Lost (TWSR = 0 × 38)

3.10.7 C Programs to Transfer and Receive Information

1. **SPI:**

For SPI Master Transmission Steps are as follows:

(i) Set MOSI, SCK and SS as output in DDRB.

(ii) Enable SPI, set device as Master, and configure clock rate via SPCR.

(iii) Load data into SPDR (SPI Data Register).

(iv) Wait for transmission complete by polling the SPIF bit in SPSR.

(v) (Optional) Repeat for continuous transmission.

Example in codes:

```
DDRB |= (1<<PB5)|(1<<PB7)|(1<<PB4);      // Step 1
SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);     // Step 2
SPDR = data;                             // Step 3
while(!(SPSR & (1<<SPIF)));              // Step 4
```

For SPI Slave, Reception Steps are as follows:

(i) Set MISO as output, other SPI pins as inputs.

(ii) Enable SPI via SPCR.

(iii) Wait for SPIF flag in SPSR to become high.

(iv) Read the received byte from SPDR.

Example in codes:

```
DDRB |= (1<<PB6);          // Step 1
SPCR = (1<<SPE);           // Step 2
while(!(SPSR & (1<<SPIF)));// Step 3
char received = SPDR;      // Step 4
```

2. **I2C:**

For I2C Master Transmission Steps are as follows:

(i) Set SCL frequency by setting TWBR and TWSR.

(ii) Enable TWI by setting TWEN in TWCR.

(iii) Send START condition using TWCR.

(iv) Wait for TWINT flag to confirm START has been sent.

(v) Send slave address with write bit (SLA+W) via TWDR.

(vi) Wait for TWINT, check TWSR for ACK.

(vii) Send data byte via TWDR.

(viii) Wait for TWINT, check for ACK.

(ix) Send STOP condition.

Example in codes:

```
TWSR = 0x00; TWBR = 0x48;
TWCR = (1<<TWEN);                                  // Step 1
TWCR = (1<<TWSTA)|(1<<TWEN)|(1<<TWINT);            // Step 2
while (!(TWCR & (1<<TWINT)));                      // Step 3
TWDR = 0xA0; TWCR = (1<<TWEN)|(1<<TWINT);          // Step 4
TWDR = 0x55; TWCR = (1<<TWEN)|(1<<TWINT);          // Step 5-6
TWCR = (1<<TWSTO)|(1<<TWEN)|(1<<TWINT);            // Step 7-8
                                                   // Step 9
```

For I2C Master Reception Steps are as follows:

(i) Initialize TWI as in transmission (steps 1-2).

(ii) Send START condition.

(iii) Send slave address with read bit (SLA+R).

(iv) Wait for TWINT, check for ACK.

(v) Enable ACK and wait for data (to read multiple bytes).

(vi) Read data from TWDR.

(vii) Send STOP condition after reception.

Example in codes:

```
TWCR = (1<<TWSTA)|(1<<TWEN)|(1<<TWINT);            // Step 2
TWDR = 0xA1; TWCR = (1<<TWEN)|(1<<TWINT);          // Step 3
TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWEA);             // Step 5
while (!(TWCR & (1<<TWINT)));
uint8_t data = TWDR;                               // Step 6
TWCR = (1<<TWSTO)|(1<<TWEN)|(1<<TWINT);            // Step 7
```

3.10.7.1 C Programs to Transmit and Receive Information for I2C and SPI

1. SPI:

Transmit Data (SPI Master Mode)

```
#include <avr/io.h>
void SPI_MasterInit(void)
{
   // Set MOSI, SCK and SS as output
   DDRB |= (1<<PB5)|(1<<PB7)|(1<<PB4);
   // Enable SPI, Master, set clock rate fck/16
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
void SPI_Transmit(char data)
{
   SPDR = data;                    // Load data into SPI data register
   while(!(SPSR & (1<<SPIF)));     // Wait until transmission complete
}
int main(void)
{
   SPI_MasterInit();               // Initialize SPI master
   while (1)
   {
      SPI_Transmit('A');           // Transmit character 'A'
   }
}
```

Receive Data (SPI Slave Mode):

```
#include <avr/io.h>
void SPI_MasterInit(void)
{
   // Set MOSI, SCK and SS as output
   DDRB |= (1<<PB5)|(1<<PB7)|(1<<PB4);
   // Enable SPI, Master, set clock rate fck/16
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
void SPI_Transmit(char data)
{
   SPDR = data;                    // Load data into SPI data register
   while(!(SPSR & (1<<SPIF)));     // Wait until transmission complete
}
```



```
int main(void)
{
   SPI_MasterInit();
   while (1)                       // Initialize SPI master
   {
      SPI_Transmit('A');
   }                               // Transmit character 'A'
}
```

2. I2C (TWI) Communication:

Master Transmit Function:

```
#include <avr/io.h>
void TWI_Init(void)
{
   TWSR = 0x00;          // Prescaler = 1
   TWBR = 0x48;          // SCL frequency (set for 100kHz with 16MHz clock)
   TWCR = (1<<TWEN);     // Enable TWI
}
void TWI_Start(void)
{
   TWCR = (1<<TWSTA)|(1<<TWEN)|(1<<TWINT);    // Send START
   while (!(TWCR & (1<<TWINT)));              // Wait for TWINT flag
}
void TWI_Stop(void)
{
   TWCR = (1<<TWSTO)|(1<<TWEN)|(1<<TWINT);    // Send STOP
}
void TWI_Write(uint8_t data)
{
   TWDR = data;                    // Load data
   TWCR = (1<<TWEN)|(1<<TWINT);    // Start transmission
   while (!(TWCR & (1<<TWINT)));   // Wait for completion
}
int main(void)
{
   TWI_Init();             // Initialize TWI (I2C)
   while (1)
   {
      TWI_Start();         // Send START condition
      TWI_Write(0xA0);     // SLA+W (slave address + write)
      TWI_Write(0x55);     // Data byte
      TWI_Stop();          // Send STOP condition
   }
}
```

Master Receive Function:

```
#include <avr/io.h>
void TWI_Init(void)
{
    TWSR = 0x00;
    TWBR = 0x48;
    TWCR = (1<<TWEN);
}
void TWI_Start(void)
{
    TWCR = (1<<TWSTA)|(1<<TWEN)|(1<<TWINT);
    while (!(TWCR & (1<<TWINT)));
}
void TWI_Stop(void)
{
    TWCR = (1<<TWSTO)|(1<<TWEN)|(1<<TWINT);
}
void TWI_Write(uint8_t data)
{
    TWDR = data;
    TWCR = (1<<TWEN)|(1<<TWINT);
    while (!(TWCR & (1<<TWINT)));
}
uint8_t TWI_Read_ACK(void)
{
    TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWEA);   // Read with ACK
    while (!(TWCR & (1<<TWINT)));
    return TWDR;
}
int main(void)
{
    uint8_t data;
    TWI_Init();
    while (1)
    {
        TWI_Start();
        TWI_Write(0xA1);          // SLA+R
        data = TWI_Read_ACK();  // Read 1 byte
        TWI_Stop();
    }
}
```


3.11 ON-CHIP ADC

The on-chip ADC (Analog-to-Digital Converter) of AVR ATmega16/32 microcontrollers is an essential peripheral that allows the microcontroller to interface with analog world devices (like sensors).

The on-chip ADC is 10-bit ADC, having 8 Single-ended Channels, 7 Differential Channels in TQFP Package. Only 2 Differential Channels with Programmable Gain at 1x, 10x, or 200x.

Features:

- ADC resolution is 10-bit (0 to 1023).
- 8 multiplexed input channels (ADC0 to ADC7).
- Results are stored in output registers ADCH and ADCL.
- ADC gives 10-bit output in ADCH (low byte) and ADCL (high byte), thus only 10-bits are useful out of 16-bits.
- ADC clock is derived from system clock using presclaer.
- Conversion time depends on ADC clock (must be 50-200 kHz).
- ADC has various modes such as Single conversion mode, Free running mode, Auto Trigger mode.
- Selectable 2.56V ADC Reference Voltage.
- Voltage reference sources are AREF pin, AVCC and Internal 2.56V reference.
- Start conversion method started by setting a bit or using auto-triggering.
- 7 Differential Input Channels.
- 2 Differential Input Channels with Optional Gain of 10x and 200x.
- 0 – VCC ADC Input Voltage Range.
- Interrupt on ADC conversion complete.
- It has Sleep Mode Noise Canceller.

Important Characteristics of ADC:

1. **Resolution:** The ADC has n-bit resolution, n can be 8, 10, 12, 16 or 24 bits. It is defined as, "the smallest change in analog input voltage that can be detected by the ADC".

   $$\text{Resolution} = \frac{V_{ref}}{2^n}$$

2. **Conversion time:** Conversion time is defined as, "the time taken by ADC to convert analog input into digital output".
3. **Accuracy:** Accuracy of an ADC refers to how close the digital output of the ADC is to the actual analog input voltage. Accuracy is the measure of the difference between the actual analog input voltage and the digitally converted value.

3.11.1 Block Diagram of ADC

The ATmega32 has a 10-bit successive approximation ADC. The ADC is connected to an 8-channel Analog Multiplexer which allows 8 single-ended voltage inputs constructed from the pins of Port A. The single-ended voltage inputs refer to 0 V (GND).

Block diagram of ADC is as shown in below Fig. 3.7.

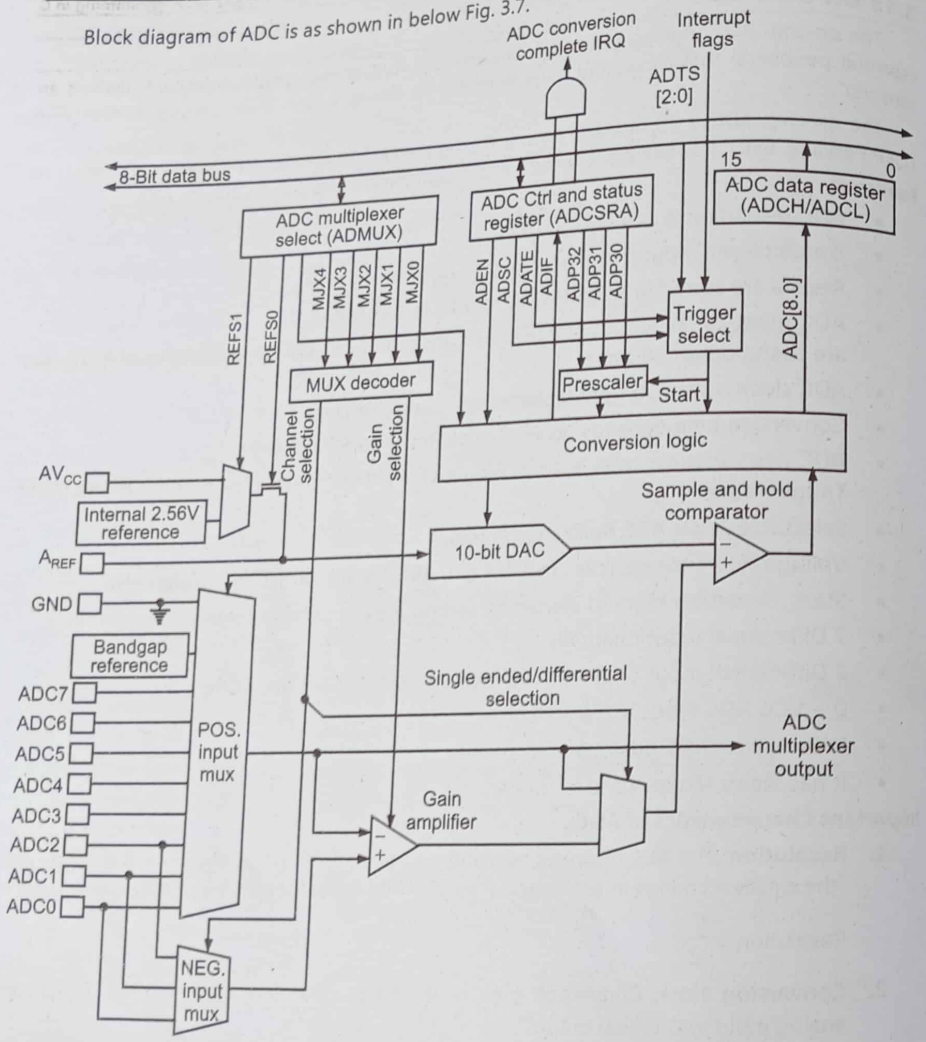

Fig. 3.7: Block diagram of ADC

Source: From datasheet of Atmel.

Description of the various blocks of ADC is as follows:

1. **AVCC/AREF/Internal 2.56V Reference:** AVCC is an analog power supply. It should be connected to VCC through a low-pass filter if ADC is used. AREF is an external voltage reference input. Internal 2.56V can be selected via ADMUX for ADC reference voltage.

2. **ADC Multiplexer (MUX):** POS INPUT MUX & NEG INPUT MUX select the analog input channels (ADC0-ADC7). It can select either single-ended or differential inputs. MUX Decoder decodes the input from ADMUX register to select which channel or input pair is connected to the ADC.
3. **Gain Amplifier:** It is used in differential mode. It amplifies the difference between two selected input voltages. The Gain options are configured via ADMUX register.
4. **Sample and Hold Comparator:** It samples the selected analog voltage and holds it for conversion and compares the held analog value with the DAC output during conversion.
5. **10-BIT DAC (Digital-to-Analog Converter):** It converts the digital value back to analog during the successive approximation process. It is used internally by the comparator to perform ADC.
6. **Conversion Logic:** It is the core of the ADC which implements the Successive Approximation Register (SAR) algorithm to convert the held analog voltage into a 10-bit digital value.
7. **Prescaler:** It divides the system clock to provide an appropriate clock to the ADC. The ADC works best at a clock between 50 kHz and 200 kHz. Prescaler is configured in ADCSRA register.
8. **Trigger Select and Start:** It determines the source that starts ADC conversion. It can be manual (software by setting ADSC) or automatic (timer, external interrupt etc.). It is controlled using ADATE (Auto Trigger Enable) in ADCSRA and ADTS bits in SFIOR register.
9. **ADC Control and Status Register (ADCSRA):** Main control registers are:

   ADEN: Enable ADC

   ADSC: Start Conversion

   ADATE: Auto Trigger Enable

   ADIF: ADC Interrupt Flag

   ADIE: ADC Interrupt Enable

   ADPS2:0: Prescaler selection
10. **ADC Multiplexer Selection Register (ADMUX):** It selects reference voltage (REFS1, REFS0), left/right result adjustment (ADLAR) and input channel (MUX4:0)
11. **ADC Data Registers (ADCL + ADCH):** It stores the 10-bit result of conversion and read order depends on ADLAR (Left Adjust Result) bit in ADMUX. Right Adjusted (default) is lower bits in ADCL, upper bits in ADCH.
12. **ADC Conversion Complete IRQ:** When conversion is done, ADIF is set. If ADIE is enabled, an interrupt is generated.

3.11.2 Operation of ADC

The ADC converts an analog voltage (0 to V_{ref}) into a 10-bit digital value (0 to 1023), using a process called successive approximation. The operation involves multiple steps from configuration to data conversion and retrieval as follows:

1. **Select Reference Voltage (V_{ref}):** Use the ADMUX register (REFS1 and REFS0 bits) to select AREF pin (external voltage reference), AVCC (supply voltage with external capacitor at AREF), Internal 2.56V reference.

   This voltage acts as the maximum analog value (mapped to digital value 1023).

2. **Select Input Channel:** Use MUX4:0 bits in ADMUX to choose one of them. Single-ended inputs: ADC0 to ADC7 measures voltage with respect to GND. Differential inputs: Pairs like (ADC1 - ADC0), optionally amplified.

3. **Configure Result Adjustment (Optional):** Set ADLAR (bit 5 in ADMUX): 0 → Right-adjusted (default), ADCL holds low bits, ADCH high bits and 1 → Left-adjusted, makes it easier to read only high 8 bits (useful for 8-bit precision).

4. **Enable ADC and Set Prescaler:** In ADCSRA register: Set ADEN = 1 to enable ADC and set ADPS2:0 to divide system clock down to 50-200 kHz for accurate conversions.

5. **Start Conversion:** Start the conversion process is:

   (i) Set ADSC = 1 in ADCSRA.

   (ii) ADC starts the Successive Approximation process: It compares the input analog voltage to a series of binary-weighted voltages from the internal 10-bit DAC.

   (iii) The Sample-and-Hold Circuit captures the input during conversion.

6. **Wait for Conversion to Complete:** Wait until ADSC becomes 0, or Poll ADIF (ADC Interrupt Flag) in ADCSRA, or Use ADC interrupt if ADIE = 1.

7. **Read the Result:** Read from ADCL and then ADCH (if right-adjusted). Final value ranges from 0 for 0 V (1023 for V_{ref}).

   Formula for digital output is:

$$\text{Digital Output} = \frac{V_{in}}{V_{ref}} \times 1023$$

8. **Use Auto Trigger Mode:** If ADATE = 1 in ADCSRA then ADC conversion can be triggered automatically by Timer overflow or External interrupts or Free-running mode. The trigger source is selected via ADTS2:0 bits in SFIOR.

3.11.3 SFR used in ADC

Special Function Registers of ADC are listed below:

1. **ADMUX - ADC Multiplexer Selection Register:**

Bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Fig. 3.8

Description of each bit:

Bit	Name	Description
7	REFS1	Reference Selection Bit 1
6	REFS0	Reference Selection Bit 0
5	ADLAR	ADC Left Adjust Result
4-0	MUX4:0	Analog Channel Selection Bits (ADC0–ADC7, differential inputs)

Voltage reference selection for ADC is though REFS1 and REFS0 as:

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56 V voltage reference with external capacitor at AREF pin

For example, REFS1=0, REFS0=1 → AVCC as reference, ADLAR=0 → Right adjusted (10-bit result in ADCH+ADCL), MUX4:0=00000 → Select ADC0

2. **ADCSRA – ADC Control and Status Register A:**

Bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Fig. 3.9

Description of each bit:

Bit	Name	Description
7	ADEN	ADC Enable (1 = Enable). Writing this bit to '1' enables the ADC. By writing it to '0', the ADC is turned off.
6	ADSC	ADC Start Conversion. In Single Conversion mode, write this bit to one to start each conversion.
5	ADATE	ADC Auto Trigger Enable. When this bit is written to one, Auto Triggering of the ADC is enabled.
4	ADIF	ADC Interrupt Flag. This bit is set when an ADC conversion completes and the Data Registers are updated.
3	ADIE	ADC Interrupt Enable. When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.
2-0	ADPS2:0	ADC Prescaler Select Bits. These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

ADC Prescaler selection:

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

For example, ADEN=1 → ADC enabled, ADSC=1 → Start conversion,
ADPS2:0 = 110 → Prescaler = 64

3. ADCL & ADCH – ADC Data Registers:

These two registers store the 10-bit result, ADCL holds lower 8 bits and ADCH holds upper 2 bits (if right adjusted), full 8 MSBs (if left adjusted) as shown below. It always read ADCL first, then ADCH (important for correct data capture).

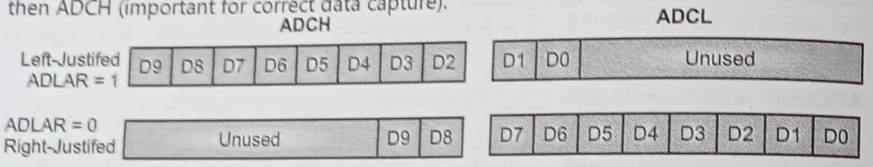

Fig. 3.10

ADCH:

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
Read/Write	7 R	6 R	5 R	4 R	3 R	2 R	1 R	0 R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Fig. 3.11

ADCL:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0							ADCL
Read/Write	7 R	6 R	5 R	4 R	3 R	2 R	1 R	0 R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Fig. 3.12

When an ADC conversion is complete, the result is found in these two registers. If differential channels are used, the result is presented in two's complement form.

4. SFIOR – Special Function IO Register:

Bit	7	6	5	4	3	2	1	0
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Fig. 3.13

Description of bits:

Bit(s)	Name	Description
7-5	ADRS2-0	ADC Auto Trigger Source

It is used only if ADATE = 1 in ADCSRA.

For example, trigger sources, Free running, External interrupt and Timer overflow.

ADC Auto Trigger Source Selections:

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

Bit 4 is reserved bit for future use in the ATmega32.

3.11.4 C Programs to Convert the Analog Signal to Digital

The ADC operates in various modes based on how conversions are triggered and handled as explained below:

(i) Single conversion mode: It starts one conversion when the ADSC bit is set in ADCSRA. It is triggered manually (software-controlled). It is used for simple and controlled analog-to-digital conversion.

(ii) Free Running Mode: It is continuous ADC conversion without manual start each time. It is triggered automatically (after one conversion ends, the next begins). It is used for continuous monitoring of analog signal (e.g., sensors).

(iii) Auto Trigger Mode (with External/Timer Events): ADC conversion starts automatically upon hardware events like timer overflow, external interrupts, comparator output, etc. It is triggered by selecting from 8 hardware sources via SFIOR register. It is used for precise periodic or event-based conversions.

(iv) Interrupt Mode: ADC notifies completion via interrupt. It reduces polling, allows multitasking.

(v) Polling: Polling is a technique where the microcontroller repeatedly checks a flag or status bit in a register to determine whether a hardware event like ADC conversion is complete.

Polling is fine for simple/small programs. Polling should be avoided in real-time or multi-tasking systems.

Examples:

1. **Write a AVR C program to convert analog input to decimal output using polling method.**

Steps for Programming AVR using Polling:

(i) Make the pin for selected ADC channel as input pin.
(ii) Turn on ADC module.
(iii) Select the conversion speed by using ADPS2-0.
(iv) Select voltage reference and ADC input channel using REFS0REFS1 in ADMUX register.
(v) Activate the start conversion bit by writing a '1' to the ADSC bit of ADCSRA.
(vi) Wait for the conversion to be completed by polling the ADIF bit in the ADCSRA register.
(vii) Once the ADIF bit becomes high, read ADCH and ADCL register to get the digital output.
(viii)Go back to step 5.

ADC connection:

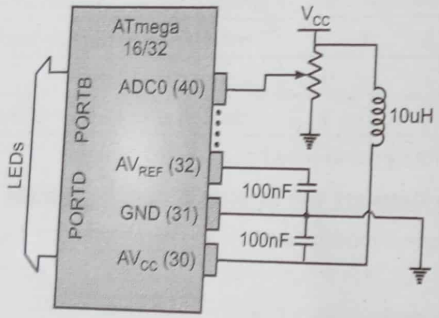

Fig. 3.14: ADC connection with AVR

AVR C Program:

```
#int main (void)
include <avr/io.h>
{
   DDRB = 0xFF;      // make PORT B an output
   DDRD = 0xFF;      // make output PORT D an output
   DDRA = 0;         // make Port A an input for ADC input
   ADCSRA = 0x87;    // make ADC enable and select ck/128
   ADMUX = 0xC0      // data will be right justified
```



```
   while (1)
   {
      ADCSRA |=(1<<ADSC);                    // start conversion
      while ((ADCSRA&(1<<ADIF)==0);          // wait for conversion to complete
      PORTD = ADCL;                          // transfer low byte to PORTD
      PORTB = ADCH;                          // transfer high byte to PORTB
   }
   return 0;
}
```

2. **Write a AVR C program to convert ADC Value to Voltage (Assuming V_{ref} = 5V)**

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#define F_CPU 8000000UL
#define VREF 5.0
void ADC_init()
{
   ADMUX = (1 << REFS0);          // AVCC with external capacitor
   ADCSRA = (1 << ADEN)|          // Enable ADC
  (1 << ADPS2) | (1 << ADPS1); // Prescaler = 64
}
uint16_t ADC_read(uint8_t channel)
{
   channel &= 0x07;
   ADMUX = (ADMUX & 0xF8) | channel;
   ADCSRA |= (1 << ADSC);
   while (ADCSRA & (1 << ADSC));
   return ADC;
}
float ADC_to_voltage(uint16_t adc_value)
{
   return (adc_value * VREF) / 1023.0;
}
int main()
{
   uint16_t adc_val;
   float voltage;
   ADC_init();
```

```
    while (1)
    {
        adc_val = ADC_read(0);
        voltage = ADC_to_voltage(adc_val);
        // Here you can send the voltage to LCD, UART, etc.
        _delay_ms(500);
    }
}
```

3. **Write AVR C program to convert analog input to digital output using for Auto Triggered Free-Running Mode.**

```
void ADC_init_freerun()
{
    ADMUX = (1 << REFS0);                    // AVCC as Vref, ADC0
    ADCSRA = (1 << ADEN) |                   // Enable ADC
             (1 << ADATE) |                  // Enable Auto Trigger
             (1 <<ADSC)  |                   // Start conversion
             (1 << ADPS2) | (1 << ADPS1);    // Prescaler = 64
    SFIOR &= ~(1 << ADTS2);                  // Free Running Mode
}
int main()
{
    ADC_init_freerun();
    DDRC = 0xFF;
    while (1)
    {
        uint16_t adc_val = ADC;
        PORTC = adc_val>> 2;
        _delay_ms(300);
    }
}
```

Exercise

[I] Multiple Choice Questions:

1. Which of the following is not a mode of Timer/Counter in AVR?
   (a) Normal mode (b) CTC mode
   (c) PWM mode (d) Oscillator mode
2. What is the maximum count of an 8-bit timer?
   (a) 127 (b) 255
   (c) 511 (d) 1023

3. In timer mode, the counter increments on _____.
   (a) External events (b) Clock pulses
   (c) Software trigger (d) ADC conversion
4. Which register in AVR is used to set prescaler bits for Timer 0?
   (a) TCCR0 (b) TCNT0
   (c) TIMSK (d) OCR0
5. To generate a 1-second delay using Timer0 in Normal mode (with 8-bit timer), which of the following is needed?
   (a) Counter overflow and software delay (b) CTC mode
   (c) External clock (d) ADC trigger
6. Which register holds the data to be transmitted via USART?
   (a) UBRR (b) UCSRA
   (c) UCSRB (d) UDR
7. What is the function of the RXC bit in UCSRA?
   (a) Indicates transmitter is ready (b) Indicates receiver has received data
   (c) Enables receiver (d) Starts baud rate generator
8. USART in AVR supports _____.
   (a) Only synchronous communication (b) Only asynchronous communication
   (c) Both synchronous and asynchronous (d) Only full duplex
9. Baud rate is determined by which register?
   (a) UCSRA (b) UBRR
   (c) UCSRC (d) UDR
10. In serial communication, simplex mode means _____.
    (a) One-way communication (b) Two-way communication
    (c) Simultaneous send and receive (d) Error correction mode
11. How many lines does I2C use?
    (a) 1 (b) 2
    (c) 3 (d) 4
12. SPI communication is _____.
    (a) Single-wire (b) Two-wire
    (c) Three-wire (d) Four-wire
13. Which AVR register is used to write SPI data?
    (a) SPDR (b) SPCR
    (c) SPSR (d) TWDR
14. I2C uses which kind of addressing?
    (a) MAC address (b) Port number
    (c) Slave ID address (d) IP address

15. In SPI, master and slave exchange data _____.
(a) Asynchronously (b) Using polling
(c) Simultaneously (d) With interrupts only

16. What is the resolution of ADC in ATmega16/32?
(a) 8-bit (b) 10-bit
(c) 12-bit (d) 16-bit

17. ADCSRA is used for _____.
(a) Setting UART speed (b) Controlling SPI
(c) ADC control and status (d) Counter reset

18. Which bit starts ADC conversion?
(a) ADSC (b) ADEN
(c) ADIF (d) AREF

19. Which register stores ADC conversion result?
(a) ADMUX (b) ADCH
(c) ADC (d) ADCSR

20. Step size of a 10-bit ADC with 5V reference is _____.
(a) 1.0 mV (b) 2.5 mV
(c) 4.88 mV (d) 5.0 mV

Answers

1. (d)	2. (b)	3. (b)	4. (a)	5. (a)	6. (d)	7. (b)	8. (c)	9. (b)	10. (a)
11. (b)	12. (d)	13. (a)	14. (c)	15. (c)	16. (b)	17. (c)	18. (a)	19. (c)	20. (c)

[II] State True or False:

1. AVR timers can be used only for delay generation, not for event counting.
2. Timer 0 and Timer 2 are 8-bit timers, while Timer 1 is a 16-bit timer.
3. TCCR0 is used to configure Timer 1.
4. TCNTn registers hold the current count value of the timer.
5. In counter mode, the timer counts external pulses instead of internal clock cycles.
6. Delay in C can be generated by using timer overflow and polling.
7. Serial communication sends multiple bits at the same time.
8. In full duplex mode, data can be sent and received simultaneously.
9. USART stands for Universal Synchronous and Asynchronous Receiver Transmitter.

10. UDR register is used for both sending and receiving data in USART.
11. UCSRB is used to set the baud rate of USART.
12. Asynchronous communication does not require a shared clock signal.
13. SPI uses two lines for communication: SDA and SCL.
14. I2C allows multiple masters on the same bus.
15. In I2C, each slave device has a unique 7-bit or 10-bit address.
16. The SPCR register in AVR is used to configure I2C.
17. In SPI, both master and slave can send and receive data simultaneously.
18. I2C is slower than SPI but requires fewer pins.
19. The ADC in ATmega16/32 is a 10-bit converter.
20. ADMUX register is used to select the ADC channel and reference voltage.
21. ADC conversion is started by setting the ADEN bit in ADCSRA.
22. The ADC result is available in the ADC register (combination of ADCL and ADCH).
23. The reference voltage (V_{ref}) affects the resolution and range of the ADC.
24. Step size of ADC is defined as $\frac{V_{ref}}{2^n}$, where n is ADC resolution.

Answers

1. False	2. True	3. False	4. True	5. True
6. True	7. False	8. True	9. True	10. True
11. False	12. True	13. False	14. True	15. True
16. False	17. True	18. True	19. True	20. True
21. False	22. True	23. True	24. True	

[III] Fill in the Blanks:

1. AVR timers can operate in two modes: _____ and _____.
2. A Timer counts based on the _____ clock, while a Counter counts external event signals.
3. In counter mode, the external clock input is received on the _____ pin for Timer/Counter0.
4. Serial communication transmits data _____ bit at a time, while parallel communication transmits _____ bits simultaneously.
5. In simplex communication, data flows in _____ direction; in duplex, data flows in _____ directions.
6. In USART transmission, the start bit is followed by data bits, optional parity bit, and one or more _____ bits.

7. I2C is a _____ line communication protocol, while SPI uses _____ lines.
8. The two main lines in I2C are _____ and _____.
9. SPI uses the lines: _____, _____, _____, and SS (Slave Select).
10. I2C addressing is based on _____-bit or _____-bit addresses.
11. The ADC has a resolution of _____ bits in ATmega16/32.
12. The _____ register stores the result of ADC conversion.
13. The _____ register is used to select the input channel and reference voltage.
14. ADC starts conversion when the _____ bit is set in the ADCSRA register.
15. In asynchronous mode, there is no separate _____ line; the clock is not shared.

Answers

1. Timer mode and Counter mode
2. Internal
3. T0
4. One, multiple
5. One, both
6. Multiple
7. Two, four
8. SCL (clock), SDA (data)
9. SCK (clock), MOSI (Master Out Slave In), MISO (Master In Slave Out)
10. 7, 10
11. 10
12. ADCL/ADCH (ADC Data Registers)
13. ADMUX (ADC Multiplexer Selection Register)
14. ADSC (ADC Start Conversion)
15. clock

[IV] Short Answer Questions:

1. What is the primary function of timers in AVR microcontrollers?
2. What is the difference between Timer and Counter modes in AVR?
3. Name the SFRs used for configuring Timer0, Timer1 and Timer2.
4. What is the use of the TCCR and TCNT registers in AVR timers?
5. How is delay generated using Timer0 in C programming?
6. How do you configure Timer1 in counter mode using C?
7. What is the role of the UDR register in USART communication?
8. List the SFRs used for USART programming in AVR.
9. What is I2C and how many lines does it use?
10. Mention the specifications of I2C protocol.
11. What are the four bus signals used in SPI communication?
12. Name the SFRs used for SPI and I2C (TWI) in AVR.
13. What is the resolution of the on-chip ADC in ATmega16/32?
14. What is the function of the ADMUX register in ADC operation?
15. Which registers hold the result of the ADC conversion?
16. How is the ADC conversion started in C programming?

[V] Long Answer Questions:

1. What is the difference between serial and parallel communication?
2. Explain simplex and duplex communication with examples.
3. What is the key difference between asynchronous and synchronous communication?
4. Write a C program to transmit a character using USART.
5. Write a C program to receive a character using USART.
6. Compare I2C and SPI based on speed, number of lines, and complexity.
7. Explain master-slave configuration in SPI.
8. How is addressing handled in I2C protocol?
9. Write a simple C code to send data using SPI.
10. Write a simple C code to receive data using I2C.
11. Draw or describe the block diagram of AVR's on-chip ADC.
12. Write a C program to read analog voltage using ADC and display the result.
13. Explain the concept of Timer and Counter in AVR micro-controllers. Discuss the differences between Timer and Counter operations with suitable examples.
14. Describe the various Special Function Registers (SFRs) used in Timer0, Timer1, and Timer2 of AVR. Mention their roles in configuring timer operations.
15. Explain the steps and write a C program to generate a precise time delay using Timer0 in Normal mode.
16. Discuss how AVR counters can be used for event counting. Write a C program to implement a simple counter using external pulses.
17. What is serial communication? Compare serial and parallel communication in terms of speed, hardware complexity and application. 7
18. Differentiate between simplex, half-duplex, and full-duplex communication with suitable real-world examples.
19. Compare asynchronous and synchronous communication in USART. Describe the working of USART in asynchronous mode in AVR.
20. Explain the operation of the USART module in AVR. Describe the role of UDR, UCSRA, UCSRB, and UBRR registers. Write C programs for data transmission and reception.
21. Define I2C protocol. Discuss its specifications, bus signals, and advantages in embedded communication.
22. Explain SPI communication protocol. Describe the function of MISO, MOSI, SCK and SS lines. Include master-slave configuration.
23. Compare I2C and SPI in terms of speed, number of wires, communication complexity and addressing.
24. Explain how master-slave configuration is implemented in I2C. How are devices identified and addressed in a multi-slave I2C setup?

25. Discuss the SFRs used in AVR for SPI and TWI (I2C). Explain error handling and data transfer mechanisms with suitable C code.
26. Explain the features of the on-chip ADC module in ATmega16/32. Include the resolution, number of channels, voltage reference options, and conversion speed.
27. Describe the step-by-step procedure for performing analog-to-digital conversion using AVR ADC. Include the configuration of ADMUX and ADCSRA registers.
28. Write a C program to read an analog input using the ADC and display the digital output value. Explain each part of the code.
29. Write a C program to generate delay of 100 msec using Timer1, assume XTAL 16 MHZ.
30. Write a C program to toggle LED connected to PD4 every 50 msec using Timer 0 with 1024 prescalar in normal mode.

■■■

Syllabus ...

1. **Introduction to Embedded System and Microcontroller [06 Lectures]**
   - **Introduction to Embedded Systems:** Embedded Systems: Introduction, Characteristics, Elements and Applications. Design Metrics: NRE Cost, Unit Cost, Time to Market, Safety, Maintenance, Size, Cost and Power Dissipation. Software Development Tools: Editor, Assembler, Linker, Compiler, IDE, ICE, Programmer and Simulator.
   - **Microcontroller and Architectures:** History, Introduction, Classification, Applications. Differences between Microcontroller and Microprocessor, Criteria for Choosing a Microcontroller. Architectures - Harvard and Von-Neumann Architecture, RISC vs CISC. Concept of Pipelining.
2. **Fundamentals of AVR and Its Programming in C [06 Lectures]**
   - **AVR Architecture:** Overview of AVR, Classification of AVR Family, AVR (ATmega16/32) Architecture, AVR Processor Memory Map, CPU Registers, ALU, I/O Ports, Peripherals in AVR.
   - **Programming of AVR in C:** Basic Structure, Data Types, Operators, Library Files, Delay Functions and Bitwise Operation Syntax. Simple C Programs: Data Transfer Operation, Arithmetic Operation, Decision-making and Code Conversion.
3. **AVR Peripherals Programming in C [10 Lectures]**
   - **AVR Timer Programming:** Introduction, Difference between Timer and Counter operation, Basic SFR Registers used – Timer 0, 1 & 2, C Programs for Delay Generation, Counter Programming.
   - **AVR Serial Port Programming:** Basics of Serial Communication (Serial Vs Parallel, Simplex Vs Duplex), Difference between Asynchronous and Synchronous Communication, USART Operation, SFR used, C Programs for Data Transmission and Reception.
   - **I2C and SPI:** Introduction, Specifications, Bus Signals, Master-slave Configuration, Error Handling and Addressing. SFR used in AVR, C Programs to Transfer and Receive Information.
   - **On-chip ADC:** Features, Block Diagram, Operation, SFR used, C Programs to Convert the Analog Signal to Digital.
4. **Real World Interfacing with AVR and Case Studies [08 Lectures]**
   - **I/O Device Interfacing:** LED, Push Button, Buzzer, Seven Segment Display, Thumbwheel Switch, DC and Stepper Motor, Relay Interfacing, 16*2 LCD Interfacing, DAC Interfacing (Waveform Generation using DAC).
   - **Case Studies:** Traffic Light Controller using AVR, Single Digit Event Counter using Opto-interrupter and SSD, Real Time Clock using IC DS1307 Chip, Temperature Monitoring System using LM35 Sensor, Smart Phone Controlled Devices using Bluetooth Module HC05.

■■■