Advanced Assembly & Microcontroller Interfacing Cheat Sheet

Chapter 7: Advanced Assembly

Assembler Directives

🔑 **Assembler Directives**: Instructions to the assembler, guiding the assembly process.

- **LDI (Load Immediate)**: Loads an 8-bit constant directly into a register.
  - Example: `LDI R20, 8`
- **HIGH and LOW**: Extracts high and low bytes of a 16-bit value.
  - Example: `LDI R20, LOW(0x1234)` (R20 = 0x34)
  - Example: `LDI R21, HIGH(0x1234)` (R21 = 0x12)

Addressing Modes

🧠 **Addressing Modes**: Define how machine language instructions identify operands, specifying how to calculate the effective memory address.

1. **Single-Register (Immediate)**: Operations on a single register or loading an immediate value.
   - `INC Rd`: Increment register Rd. (e.g., `INC R19`)
   - `DEC Rd`: Decrement register Rd. (e.g., `DEC R23`)
   - `LDI Rd, K`: Load immediate value K into register Rd. (e.g., `LDI R19, 25`)
   - *Relates to*: A form of Immediate Addressing.
2. **Two-Register Addressing Mode**: Operations between two registers.
   - `ADD Rd, Rr`: Add Rr to Rd.
   - `SUB Rd, Rr`: Subtract Rr from Rd.
3. **Direct Addressing Mode**: Instruction directly specifies the memory address.
   - `LDS Rd, address`: Load Rd with content of specified memory address.
   - `STS address, Rs`: Store content of Rs into specified memory address.
   - *Contrast with*: Register Indirect.
4. **I/O Direct Addressing Mode**: Used for accessing I/O memory locations.
   - `OUT address, Rr`: Send Rr content to specified I/O address.
   - `IN Rd, address`: Load Rd with content of specified I/O address.
5. **Register Indirect Addressing Mode**: A register (X, Y, or Z) contains the memory address.

    - `LD Rd, X`: Load Rd with content pointed to by X register.
    - `ST X, Rd`: Store Rd content into location pointed to by X register.
    - **Registers**:
        - `X`: R27:R26
        - `Y`: R29:R28
        - `Z`: R31:R30
6. **Auto-increment and Auto-decrement**: Special cases of register indirect addressing.
    - **Post-increment**: Address register incremented *after* memory access.
        - `LD Rd, X+`: Load Rd from X, then increment X.
        - `ST X+, Rs`: Store Rs to X, then increment X.
    - **Pre-decrement**: Address register decremented *before* memory access.
        - `LD Rd, -X`: Decrement X, then load Rd from X.
        - `ST -X, R31`: Decrement X, then store R31 to X.
7. **Register Indirect with Displacement**: Accesses memory with an offset (q) from a register's base address.
    - `STD Z+q, Rr`: Store Rr into location Z+q. (q is 0-63)
    - `LDD Rd, Z+q`: Load from Z+q into Rd.

🔑 **Storing Fixed Data in Flash Memory**:

- `.DB` data directive: Allocates ROM (program code) memory in byte-sized chunks.
- `.DW` defines values in two bytes.
- **LPM (Load Program Memory)**: Used to read data from Flash memory.
    - `LPM Rd, Z`: Loads data from address pointed by Z into Rd.
    - `LPM Rd, Z+`: Loads data from address pointed by Z into Rd, then increments Z.

Macros

🔑 **Macro**: A symbolic name representing a sequence of instructions or statements.

- Can take up to 10 parameters (`@0` to `@9`).
- Invoked by its name.
- `MACRO ... .ENDMACRO`

EEPROM

🔑 **EEPROM (Electrically Erasable Programmable Read-Only Memory)**: Non-volatile memory for persistent data.

- ATmega328 has 1024 bytes.
- **Registers**:
    1. `EEARH:EEARL`: EEPROM Address Register

    2. `EEDR`: EEPROM Data Register
    3. `EECR`: EEPROM Control Register
- **Reading from EEPROM**:
    1. Wait for `EEWE` to be zero.
    2. Write address to `EEAR`.
    3. Set `EERE` to one.
    4. Read data from `EEDR`.
- **Writing to EEPROM**:
    1. Wait for `EEWE` to be zero.
    2. (Optional) Write address to `EEAR`.
    3. (Optional) Write data to `EEDR`.
    4. Set `EEMWE` to one.
    5. Within 4 clock cycles, set `EEWE` to one.

Checksum

🧠 **Checksum**: Used to detect data corruption.

- **Calculating Checksum Byte**:
    1. Add bytes together, drop carries.
    2. Take 2's complement of the sum.
- **Testing Checksum**:
    1. Add bytes together, drop carries.
    2. Add the checksum byte to the sum.
    3. If result is not zero, data is corrupted.

Memory Layout

🧠 **Memory Layout (AVR Microcontrollers)**:

- **PROGMEM**: Program code and fixed data. Non-volatile.
- **SRAM**: Global variables, virtual methods, heap space, stack. Volatile (data lost on power off). Fast access.
- **EEPROM**: Data area to persist values. Non-volatile.

Chapter 8: AVR Programming in C

Languages

🔑 **Language Levels**:

- **High Level (e.g., VB)**: Easy to develop, portable, easy to update.

- **C Language**: Acceptable performance, portable.
- **Low Level (Assembly)**: High performance, not portable, direct hardware control.

Accessing I/O Registers

- Direct manipulation of DDR (Data Direction Register) and PORT (Output Port Register) using hexadecimal values.
  - Example: `DDRD = 0xFF;` (Set PORTD as output)
  - Example: `PORTD = 0xAA;` (Send 0xAA to PORTD)
- Reading from PIN (Input Pin Register).
  - Example: `PORTD = PINB + PINC;`

Data Types

- Use `unsigned` whenever possible.
- Use `unsigned char` instead of `unsigned int` when appropriate to conserve space.

Time Delays in C

- **For Loops**: Use a `for` loop (e.g., `for(i=0; i<42150; i++) {}`).
  - ⚠️ Affected by clock frequency and compiler optimization.
- **Predefined Functions (Atmel Studio)**:
  - Include `#define F_CPU 8000000UL` and `#include <util/delay.h>`.
  - Use `_delay_us(value)` and `_delay_ms(value)`.

Bit-wise Logical Operators

📌 **Bit Manipulation in C**:

- **Setting a Bit to 1**: Use `|` (OR) operator with `(1 << bit_number)`.
  - Example: `PORTB |= (1<<4);` (Sets bit 4 of PORTB to 1)
- **Clearing a Bit to 0**: Use `&` (AND) and `~` (NOT) operators.
  - Example: `PORTB &= ~(1<<4);` (Clears bit 4 of PORTB to 0)
- **Checking a Bit**: Use `&` (AND) operator.
  - Example: `if (((PINC & (1<<5)) != 0))` (Checks if bit 5 of PINC is high)
- **Shift Operations**: `>>` (right shift), `<<` (left shift).

Memory Types in AVR (C Access)

🔑 **Flash Memory**:

- Non-volatile. Large size. Stores program code, lookup tables, fixed data.

- **Accessing**:
    1. Include `<avr/pgmspace.h>`.
    2. Declare data with `const unsigned char PROGMEM name[] = {...};`.
    3. Read using `pgm_read_byte(&name[i]);`.

🔑 **EEPROM**:

- Non-volatile. Smaller size. Stores small, modifiable data (e.g., configuration settings).
- **Accessing**:
    1. Include `<avr/io.h>` and `<avr/eeprom.h>`.
    2. Reserve location with `unsigned char EEMEM myVar;`.
    3. Read/write using `eeprom_read_byte(&myVar)` and `eeprom_write_byte(&myVar, value)`.

🔑 **RAM**:

- Volatile (data lost on power off). Fast access. Stores data manipulated during execution.

AVR Timer Programming

🔑 **Counter Register**: 8-bit or 16-bit register incrementing with each clock cycle.

- `Load`: Load new value.
- `Up`: Clock input.
- `Cout`: Carry-out flag.

🔑 **Generic Timer/Counter Uses**: Delay generation, counting, waveform generation, capturing.

🔑 **Key Timer Registers in AVR**:

- `TCNTn` (Timer/Counter register)
- `TOVn` (Timer Overflow flag)
- `TCCRn` (Timer Counter control register)
- `OCRn` (Output Compare Register)
- `OCFn` (Output Compare Match Flag)
- All are byte-addressable.

Timer Modes

1. **Normal Mode**:
    - Timer/counter increments from loaded value to `0xFF` (8-bit) or `0xFFFF` (16-bit).
    - Rolls over to `0x00` and sets `TOVn` (Timer Overflow) flag.
    - **Delay Calculation**: (Max Value - Start Value) * Clock Period.

        - For 8-bit: `(256 - TCNTn) * T_clock`.
        - For 16-bit: `(65536 - TCNTn) * T_clock`.
    - To get largest delay, set `TCNTn` to `0`.
2. **CTC (Clear Timer on Compare Match) Mode**:
    - Timer counts up until `TCNTn` equals `OCRn` (Output Compare Register).
    - Timer is then cleared to `0` and `OCFn` (Output Compare Flag) is set.
    - **Delay Calculation**: `(OCRn + 1) * T_clock`.
    - `OCRn` is loaded with `(Desired Clocks - 1)`.

🧠 **Generating Large Delays**:

- Nested loops.
- **Prescalers**: Divides system clock frequency for slower timer clock, allowing longer delays.
    - `T_timer_clock = T_system_clock * Prescaler_value`.
    - Example Prescalers: 1, 8, 64, 256, 1024.
- **Bigger Counters**: Use 16-bit timers (e.g., Timer1) for larger delays.

Timer 0 (8-bit) vs. Timer 2 (8-bit) vs. Timer 1 (16-bit)

- **Timer0/Timer2**: 8-bit timers.
- **Timer1**: 16-bit timer (`TCNT1H`, `TCNT1L`). Has two control registers (`TCCR1A`, `TCCR1B`).
- All timers have different `TOVn`, `OCFn` flags and `TCCRn` registers, and often different prescaler options.

Chapter 12: Interrupts

🔑 **Interrupts**: Allows a device to signal the microcontroller when it needs attention, pausing current program flow.

- **Polling**: Continuously checking device status (inefficient).
- **Interrupt vs. Polling**: Interrupts are event-driven, polling is constant checking.

🧠 **Steps in Executing an Interrupt**:

1. Microcontroller finishes current instruction.
2. Program Counter (PC) saved on stack.
3. Jumps to Interrupt Vector Table (fixed memory location).
4. Executes Interrupt Service Routine (ISR) until `RETI`.
5. Saved PC retrieved from stack, program resumes.

🔑 **Interrupt Control Unit (ICU)**: Manages interrupt requests.

- **Registers**: `SREG` (Status Register, contains Global Interrupt Enable 'I' flag), `PCICR`, `TIMSK0/1/2`, `EIMSK`.

🔑 **Interrupts: Masking and Enabling**:

- Upon reset, all interrupts disabled (masked).
- Must be enabled by software.
- `CLI`: Clears 'I' flag to 0 (disables interrupts).
- `SEI`: Sets 'I' flag to 1 (enables interrupts).
- `sei()` in C: Enables global interrupts.

Types of Interrupts

1. **External Interrupts**: Triggered by external signals on specific pins (e.g., `INT0` on PD2).
   - `EIMSK`: External Interrupt Mask Register.
   - `EICRA`: External Interrupt Control Register A (sets edge/level trigger).
   - **Edge Trigger**: On rising or falling edge.
   - **Level Trigger**: When signal is at a specific level (high/low).
2. **Timer Interrupts**: Generated by internal timers.
   - `TIMSKn`: Timer Interrupt Mask Registers.
   - `TIFRn`: Timer Interrupt Flag Registers.
   - Can be generated on Timer Overflow (`TOVn`) or Compare Match (`OCFn`).
3. **Pin Change Interrupts**: Trigger when logic level of *any* pin in a group changes.
   - `PCMSKn` (Pin Change Mask Registers for PORTB/C/D).
   - `PCICR` (Pin Change Interrupt Control Register).
   - `PCIE0/1/2` (Pin Change Interrupt Enable bits).

🔑 **Interrupt Priority**: Processed based on priority; lower addresses typically have higher priority.

⚠️ **Interrupt Inside an Interrupt**:

- When ISR executes, 'I' flag is cleared (disabling further interrupts).
- 'I' flag is set again on `RETI`.
- Problem: If a higher priority interrupt occurs during an ISR, it will not be serviced until the current ISR finishes, unless explicitly re-enabled within the ISR (generally not recommended without careful design).

🧠 **Task Switching and Resource Conflict**:

- **Problem**: Interrupts can cause resource conflicts when multiple tasks use the same registers.
- **Solution 1: Different Registers**: Use dedicated registers for different tasks.

- **Solution 2: Context Saving (Software Stack)**:
  - `PUSH Rxx` at ISR start to save registers.
  - `POP Rxx` before `RETI` to restore registers.
  - **Saving SREG**: Also save `SREG` if ISR modifies flags.

Chapter 13: LCD and Keyboard

LCD (Liquid Crystal Display)

🔑 **LCD Internal Components**:

- **DDRAM (Data Display RAM)**: 128x8 RAM, holds data to be displayed.
- **CGRAM (Character Generator RAM)**: 64x8 RAM, stores fonts of first 8 characters (0-7). Can be modified to define custom characters.
- **Cursor (Address Counter)**: Points to a location in DDRAM or CGRAM.
- **Data Register**: 8-bit, data written here goes to where cursor points.
- **Command Register**: Commands LCD (e.g., clear screen, set cursor).

🔑 **LCD Commands (Examples)**:

- `01H`: Clear display screen.
- `02H`: Return home (cursor to 0).
- `10H`: Shift cursor left.
- `14H`: Shift cursor right.
- `06H`: Shift cursor right after displaying char (default).
- `04H`: Shift cursor left after displaying char.
- `38H`: Initialize to 2 lines & 5x7 font.
- `0CH`/`0EH`/`0FH`: Display ON/OFF, Cursor ON/OFF/Blinking.
- `80H-FFH`: Set DDRAM address (cursor position).
- `40H-7FH`: Set CGRAM address.

🔑 **LCD Pins**:

- **VSS, VCC**: Power supply (+5V).
- **VEE**: Contrast control.
- **D0-D7**: Data pins.
- **R/W (Read/Write)**: Low for Write, High for Read.
- **E (Enable)**: High-to-low pulse activates internal latch for data/command transfer.
- **RS (Register Select)**: `0` for Command Register, `1` for Data Register.

🧠 **LCD Programming Flow**:

1. **Initialization**: Send commands like `0x38`, `0x0E`, `0x01` for 8-bit mode. For 4-bit mode: `0x33`, `0x32`, `0x28`.
2. **Sending Commands**:
   - Set `RS=0`, `R/W=0`.
   - Put command on `D0-D7`.
   - Send high-to-low pulse on `E`.
3. **Sending Data**:
   - Set `RS=1`, `R/W=0`.
   - Put data on `D0-D7`.
   - Send high-to-low pulse on `E`.
4. **4-Bit Mode**:
   - Send high nibble to `D4-D7` first.
   - Then, swap and send low nibble to `D4-D7`.
5. **Changing Fonts (CGRAM)**:
   - Set cursor to CGRAM address (e.g., `40H` for char 0, row 0).
   - Write font data (row by row) to data register.
   - 📌 After changing CGRAM, set cursor back to DDRAM!

Keyboard

🔑 **Debouncing**: Correct way to read keys, preventing a single physical key press from being registered as multiple clicks due to mechanical bounce.

🧠 **Matrix Keyboard**:

- Used to save MCU pins. Keys are arranged in rows and columns.
- Requires:
  - **Key press detection**: Detecting if *any* key is pressed.
  - **Key identification (scanning)**: Identifying *which* key is pressed by scanning rows/columns.

Serial Port UART Protocol

🔑 **Serial Communication Characteristics**:

- **Serial**: Data sent one bit at a time.
- **Asynchronous**: No shared clock; uses start/stop bits.
- **Full-duplex**: Simultaneous two-way communication.

🧠 **Communication Directions**:

- **Simplex**: One-way (e.g., radio broadcast).
- **Half Duplex**: Two-way, but not simultaneous (e.g., walkie-talkie).

- **Full Duplex**: Simultaneous two-way (e.g., telephone).

🔑 **Line Coding**: Presenting data using signals (NRZ-L, NRZ-I for digital; ASK, FSK, PSK for analog).

🔑 **UART (Universal Asynchronous Receiver/Transmitter)**:

- A hardware peripheral for serial, asynchronous, full-duplex communication.
- USART (Universal Synchronous Asynchronous Receiver/Transmitter) supports both sync and async.

🔑 **Parity Bit**: Simple error-detecting code.

- **Even Parity**: Parity bit makes total count of 1s even.
- **Odd Parity**: Parity bit makes total count of 1s odd.

📌 **Serial Communication Data Frame**:

- Start bit (low)
- Data bits (e.g., 8 bits)
- (Optional) Parity bit
- Stop bit(s) (high)

🔑 **UART in AVR**:

- **Control Registers**: `UBRR` (Baud Rate Register), `UCSRA`, `UCSRB`, `UCSRC`.
- **Send/Receive Register**: `UDR` (UART Data Register).
- **Status Register**: `UCSRA`.
- **Baudrate Calculation**: `Baudrate = System_clock / (16 * (UBRR + 1))`

🔑 **Key UART Registers (`UCSROB`, `UCSRC`, `UCSRA`)**:

- **UCSROB**:
    - `RXCIE0`, `TXCIE0`, `UDRIE0`: Interrupt enables.
    - `RXEN0`, `TXEN0`: Receiver/Transmitter enable.
    - `UCSZ02`: Character Size bit (combined with UCSRC's `UCSZ01:00`).
- **UCSRC**:
    - `UMSEL01:00`: USART Mode Select (Async, Sync, Master SPI).
    - `UPM01:00`: Parity Mode (Disabled, Even, Odd).
    - `UCSZ01:00`: Character Size bits.
    - `UCPOL0`: Clock Polarity (for synchronous mode).
- **UCSRA**:
    - `RXC0`: Receive Complete Flag.
    - `TXC0`: Transmit Complete Flag.

    - `UDRE0`: Data Register Empty Flag.
    - `FE0`: Frame Error Flag.
    - `DOR0`: Data OverRun Flag.
    - `PE0`: Parity Error Flag.

🧠 **Programming AVR for Serial Transfer (TX)**:

1. Enable transmitter (`UCSR0B = (1<<TXEN0)`).
2. Set frame format (8-bit data, no parity, 1 stop bit) (`UCSR0C = (1<<UCSZ01)|(1<<UCSZ00)`).
3. Set baud rate (`UBRR0L`).
4. Write character to `UDR0`.
5. Monitor `UDRE0` flag in `UCSRA` to ensure `UDR` is ready for next byte.

🧠 **Programming AVR for Serial Receive (RX)**:

1. Enable receiver (`UCSR0B = (1<<RXEN0)`).
2. Set frame format (`UCSR0C`).
3. Set baud rate (`UBRR0L`).
4. Monitor `RXC0` flag in `UCSRA` for new data.
5. Read received byte from `UDR0`.

I2C (Inter-Integrated Circuit) Protocol (TWI)

🔑 **I2C Overview**:

- Developed by Philips.
- Connects many devices (up to ~128) to MCU using two wires.
- **Two Wires**:
    - **SDA (Serial Data)**: Bidirectional, open-drain data line.
    - **SCL (Serial Clock)**: Bidirectional, open-drain clock line.
- Lines idle in HIGH state (passively pulled high).
- Standard data rate: 100 kbits/s. Fast Mode: 400 kbits/s.

🧠 **I2C Protocol**:

- **Sending Bits**: SDA values change when SCL is low. Receiver reads SDA on falling edge of SCL.
- **Start Condition**: Master pulls SDA low while SCL is high.
- **Stop Condition**: Master pulls SDA high while SCL is high.
- **Repeated Start**: New Start condition before a Stop condition.
- **Packet Format**: 9 bits long (8 data + 1 ACK).
- **Acknowledge (ACK)**: Receiver pulls SDA low (ACK) or leaves high (NACK) for 9th bit.

🔑 **Master vs. Slave**: | Feature | Master | Slave | | :--------- | :-------------------------------------- | :------------------------------------- | | **Role** | Begins communication, chooses slave | Responds to master | | **Clock** | Makes clock | - | | **Address**| - | Each slave has a unique address | | **Data** | Sends or receives data | - |

- Multiple masters possible. Each device can be both master and slave.

🧠 **Steps of an I2C Communication**:

1. Start
2. Address (slave address + Read/Write bit)
3. Send/Receive (Write or Read)
4. Acknowledge
5. Send/Receive a byte of data
6. Acknowledge
7. Stop

🔑 **Multibyte Burst Write**: Transmit slave address (write), address of first location, then consecutive data bytes. 🔑 **Multibyte Burst Read**: Transmit slave address (write), address of first location, then Repeated Start, slave address (read), then read consecutive data bytes.

🔑 **I2C (TWI) Unit in AVR Components**:

- Bit Rate Generation Unit
- Bus Interface Unit
- Address Match Unit
- Control Unit

🔑 **I2C (TWI) Registers**:

- **TWSR (TWI Status Register)**: `TWS` (Status bits), `TWPS` (Prescaler bits).
    - `SCL_freq = XTAL_freq / (16 + (2 * TWBR) * (4^TWPS))`
- **TWBR (TWI Bit Rate Register)**: Sets the clock rate.
- **TWCR (TWI Control Register)**:
    - `TWINT`: TWI Interrupt Flag.
    - `TWEA`: TWI Enable Acknowledge.
    - `TWSTA`: TWI Start Condition bit.
    - `TWSTO`: TWI Stop Condition bit.
    - `TWWC`: TWI Write Collision Flag.
    - `TWEN`: TWI Enable.
    - `TWIE`: TWI Interrupt Enable.
- **TWDR (TWI Data Register)**: Data byte.
- **TWAR (TWI Address Register)**: Slave address (7 bits + TWGCE for general calls).

🧠 **TWI Master Mode Programming**:

- **Initializing**: Set `TWBR`, `TWPS`, enable `TWEN`.
- **Transmit START**: Set `TWEN`, `TWSTA`, `TWINT`.
- **Send Data**: Copy data to `TWDR`, set `TWEN`, `TWINT`, poll `TWINT`.
- **Receive Data**: Set `TWEN`, `TWINT`, poll `TWINT`, read from `TWDR`. (Use `TWEA` for ACK/NACK).
- **Transmit STOP**: Set `TWEN`, `TWSTO`, `TWINT`.

🧠 **TWI Slave Mode Programming**:

- **Initializing**: Set `TWAR` (slave address), enable `TWEN`, `TWINT`, `TWEA`.
- **Listening**: Poll `TWINT` (or use interrupt) to detect master addressing.
- **Send Data**: Copy data to `TWDR`, set `TWEN`, `TWEA`, `TWINT`, poll `TWINT`.
- **Receive Data**: Set `TWEN`, `TWINT`, poll `TWINT`, read from `TWDR`.

RTC (Real-Time Clock)

🔑 **RTC (Real-Time Clock)**: Keeps track of time and date, typically uses a 32.768 kHz quartz crystal.

- **DS1307**:
    - 64 bytes of RAM.
    - Uses BCD format.
    - Bit 7 of address 0 (CH) must be zero to enable oscillator.
    - Address 07 is control register.
    - **Register Pointer**: Specifies byte for next R/W, automatically increments.
- **Write to DS1307**: Start, DS1307 address (write), location address, data bytes, Stop.
- **Read from DS1307**: Start, DS1307 address (read), receive data bytes, Stop. (Register pointer must be set first via a write operation).
- **DS3231**: Similar to DS1307.

SPI Protocol (Serial Peripheral Interface)

🔑 **SPI Overview**:

- Synchronous, Full-duplex, Serial, Fast communication, Short distances.
- **Pins**:
    - `MOSI` (Master Out Slave In) / `SDO` (Serial Data Out)
    - `MISO` (Master In Slave Out) / `SDI` (Serial Data In)
    - `SCK` (Shift Clock)
    - `SS` (Slave Select) / `CE` (Chip Enable)

🔑 **Master vs. Slave**:

- **Master**: Begins communication (pulls `SS/CE` low), provides clock.
- **Slave**: Responds to master.
- **Internal Circuit**: Shift register in master and slave; bits shifted simultaneously with each clock cycle.

🔑 **AVR Registers for SPI**:

- **SPCR (SPI Control Register)**:
  - `SPIE`: Interrupt Enable.
  - `SPE`: SPI Enable.
  - `DORD`: Data Order (LSB/MSB first).
  - `MSTR`: Master Mode Select (1 for Master, 0 for Slave).
  - `CPOL`: Clock Polarity.
  - `CPHA`: Clock Phase.
  - `SPR1, SPR0`: Clock Rate Select bits.
- **SPSR (SPI Status Register)**:
  - `SPIF`: SPI Interrupt Flag (transfer complete).
  - `WCOL`: Write Collision.
  - `SPI2X`: Double SPI Speed.
- **SPDR (SPI Data Register)**: Data to be sent/received.

🧠 **SPI Clock Rate**: Determined by `SPI2X`, `SPR1`, `SPR0` bits to divide `Fosc`.

SPI Modes (CPOL vs CPHA)

- **Mode 0**: `CPOL=0`, `CPHA=0` (Read on rising edge, changed on falling edge)
- **Mode 1**: `CPOL=0`, `CPHA=1` (Read on falling edge, changed on rising edge)
- **Mode 2**: `CPOL=1`, `CPHA=0` (Read on falling edge, changed on rising edge)
- **Mode 3**: `CPOL=1`, `CPHA=1` (Read on rising edge, changed on falling edge)

🧠 **Programming SPI as Master**:

1. Set `DDRB` for `MOSI`, `SCK`, `SS` as output.
2. Set `SPCR` with `SPE` (enable SPI), `MSTR` (master mode), `SPR0/SPR1/SPI2X` (clock rate), `CPOL/CPHA` (mode).
3. Pull `SS` low to enable slave.
4. Write data to `SPDR`.
5. Wait for transfer finish (`SPSR & (1<<SPIF)`).
6. Read received data from `SPDR`.
7. Pull `SS` high to disable slave.

🧠 **Programming SPI as Slave**:

1. Set `DDRB` for `MISO` as output.
2. Set `SPCR` with `SPE` (enable SPI) only (`MSTR` defaults to 0 for slave).
3. Write data to `SPDR` to prepare for master's read.
4. Wait for transfer finish (`SPSR & (1<<SPIF)`).
5. Read received data from `SPDR`.