Communication Protocol

All of these protocols are digital communication protocols

Latency

- **Definition**: Time between an event and the corresponding action.
- **Input Latency**: Delay between new input data ready and software reading it.
- **Output Latency**: Delay between device idle and software sending output data.
- **Real-Time System**: Guarantees small, bounded latency to meet performance specs.

Bandwidth

- **Definition**: Maximum data flow rate (bytes/second).
- **Types**:
    - **Average Bandwidth**: Long-term data transfer rate.
    - **Peak Bandwidth**: Short-term maximum rate.

Priority

- **Definition**: Determines the order of task processing.
- **High vs. Low**: High-priority tasks can preempt lower-priority ones.
- **Equal Priority**: Prevents monopolization of resources.
- **Soft Real-Time**: Uses priority but without strict timing guarantees.

Synchronization

- **Purpose**: Aligns microcontroller and I/O device transitions.
- **Methods**: Five mechanisms for managing data transfers (detailed separately).

1. Blind Cycle Synchronization

Definition

- A method where the software waits a fixed amount of time after initiating an I/O operation, assuming it will complete within that delay.

Input Device

- **Process**:
    1. Trigger the input hardware.
    2. Wait for a fixed time.
    3. Read data from the device.

- **Example**: Reading from a sensor with predictable timing.

Output Device

- **Process**:
    1. Write data to the device.
    2. Trigger the device.
    3. Wait for a fixed time.
- **Example**: Displaying characters on an LCD screen with a known delay (e.g., 37 µs).

Characteristics

- **"Blind" Nature**: No feedback or status check from the I/O device.
- **Appropriate Use**: When I/O speed is predictable and consistent.

Example: Stepper Motor Control

- Repeated 8-step sequence:
    1. Output 0x05, wait 1 ms.
    2. Output 0x06, wait 1 ms.
    3. Output 0x0A, wait 1 ms.
    4. Output 0x09, wait 1 ms.
- Result: Constant speed motor spinning. in other words delay

2. Busy-Wait Synchronization unbuffered

Definition

- A method where the software continuously checks the status of an I/O device until it reaches the **done** state.

Input Device

- **Process**:
    1. Wait until the device signals new data is ready.
    2. Read data from the device.

Output Device

- **Process (two variations)**:
    1. Write data, trigger the device, then wait until it finishes.
    2. Wait until the device finishes previous output, then write new data and trigger it.

Characteristics

- **Busy Loop**: The CPU remains occupied checking device status instead of doing other tasks.
- **Use Cases**: Simple systems where real-time constraints are not critical.
- **Example**: UART communication in basic applications.

3. Interrupt Synchronization buffered

Definition

- Uses hardware to trigger special software execution when an event occurs, eliminating the need for continuous polling.

Input Device

- **Process**:
    1. Hardware triggers an interrupt when new data is ready.
    2. Software interrupt service reads data and stores it in global RAM.

Output Device

- **Process**:
    1. Hardware triggers an interrupt when the device is idle.
    2. Software interrupt service fetches data from a global structure and writes it to the device.

Periodic Interrupts

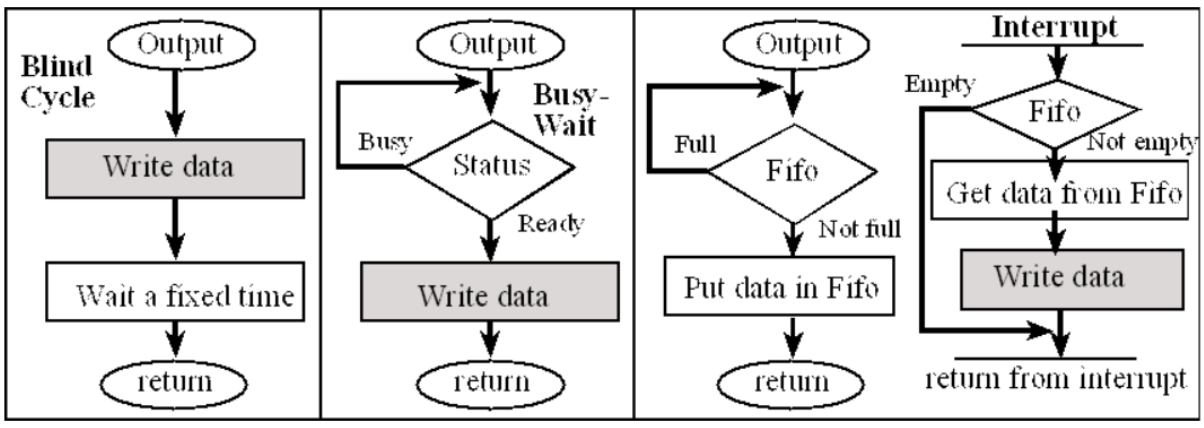
- **Use**: Hardware timers generate periodic interrupts for regular tasks, such as reading from an ADC.

Characteristics

- **Efficiency**: Reduces CPU load compared to busy-wait.
- **Complex Systems**: Handles multiple I/O devices effectively.
- **Real-Time Systems**: Ensures timely responses to events.

Example

- **Data Acquisition System**: Reads ADC values at fixed intervals using timer interrupts.



Periodic Polling

- **Definition**: Uses a clock interrupt to check the I/O status periodically.
- **Input Device**:
    1. A **ready flag** is set when the device has new data.
    2. At the next interrupt, the software reads the data and stores it in global RAM.
- **Output Device**:
    1. A **ready flag** is set when the device is idle.
    2. At the next interrupt, the software writes new data to the device.
- **Use Case**: When I/O devices do not directly support interrupts, but periodic checks are still necessary.

DMA (Direct Memory Access)

- **Definition**: Transfers data directly between memory and I/O devices without CPU intervention.
- **Input Device**:
    1. Hardware requests DMA when new data is available.
    2. DMA controller automatically reads data and saves it in memory.
- **Output Device**:
    1. DMA requests a transfer when the output device is idle.
    2. DMA controller reads data from memory and writes it to the device.
- **Use Case**: High-speed data acquisition and systems requiring **high bandwidth** and **low latency**.
- **Not Covered**: Specific implementation details for DMA on LM4F120/TM4C123 are advanced topics.

Hardware States

- **Idle**: Device is inactive; no I/O operations are occurring.
- **Busy**: Device is performing operations (flag = 0).
- **Ready**: Device is ready for data transfer (flag = 1).
- **Flag Usage**:
    - Set by hardware when the task is complete.
    - Software reads the flag to determine whether the device is busy or ready.
    - Software clears the flag after processing, signaling task completion.

Universal Asynchronous Receiver Transmitter (UART)

UART is a hardware communication protocol used for serial communication between a microcontroller and other devices (like sensors, computers, or printers). It works by sending data one bit at a time over a communication line.

The **baud rate** is the number of bits transmitted per second. It determines how fast data is sent or received. The reciprocal of the baud rate gives the **bit time**, the time it takes to send one bit.

Serial Transmission Frame
A data frame is the basic structure used in serial communication. It typically includes:

- **Start Bit (0)**: Marks the beginning of a frame.
- **Data Bits (5-8)**: The actual information being sent, often 8 bits.
- **Parity Bit (optional)**: Used for error detection (even/odd/no parity).
- **Stop Bit (1)**: Marks the end of the frame

Components Involved in UART Transmission

Data Output Pin: Outputs the digital signal representing the data being transmitted.

FIFO (First-In, First-Out) Buffers:

- UART transmission uses a **16-element FIFO buffer**.
- The FIFO buffer stores data temporarily before transmission.

- **Shift Register**:
  - The **10-bit shift register** handles actual data transmission, including the start bit, data bits, and stop bit.
  - This register is **not directly accessible** by the programmer.

2. Transmission Process

1. **Checking TXFF (Transmit FIFO Full Flag)**:
   - If **TXFF = 1**, the FIFO is full, and the software must wait.
   - If **TXFF = 0**, there is space in the FIFO, and data can be written.
2. **Writing Data to UART Data Register**:
   - The data is written to the **UART data register (e.g., UART0_DR_R)**.
   - Data bits are transmitted in the following order:
     - **Start bit (0)** → **b0 (LSB)** → **b1** → … → **b7 (MSB)** → **Stop bit (1)**.

3. UART Registers for Transmission

- **Transmit Data Register (UART0_DR_R)**:

    - It is **write-only**, meaning data can be written to it, but the contents cannot be read back.
    - Writing to this register starts the transmission if space is available in the FIFO.
- **Transmit and Receive Data Registers**:
    - Though they share the same address, they are separate hardware registers.

4. Flags for Transmission and Reception

- **TXFF (Transmit FIFO Full)**:
    - **0**: FIFO has space (ready for new data).
    - **1**: FIFO is full (wait before writing more data).
- **RXFE (Receive FIFO Empty)**:
    - **0**: Data is available in the receive FIFO.
    - **1**: Receive FIFO is empty (no data to read).

1. UART Transmission

- **Data Writing**: Writing a byte to UART0_DR_R places it in a **16-element FIFO**.
- **Data Transmission**: Data moves from FIFO to a **10-bit shift register** (1 start bit, 8 data bits, 1 stop bit).
- **Timing**: Transmission uses a clock running at **16x the baud rate (Baud16)**.
    - Example: At **9600 bps**, each frame takes **1.04 ms**, up to **16.7 ms** delay if FIFO is full.

2. UART Reception

- **Start Bit Detection**: A **1-to-0 edge** triggers data reception.
- **Data Bits Sampling**: Bits are read with timing spaced by 16 clock cycles.
- **FIFO Buffering**: A **12-bit FIFO** holds **8 data bits** and **4 error flags**.

4. Errors

- **Overrun Error (OE)**: FIFO full; data lost.
- **Framing Error (FE)**: Wrong stop bit, often due to **baud rate mismatch** (>5%).
- **Break Error (BE)**: Line held low too long.
- **Parity Error (PE)**: Caused by incorrect parity (not used here).

Initialization Steps

- **Enable Clocks**:
  - Activate the UART clock in RCGCUART.
  - Activate the GPIO port clock in RCGCGPIO.
- **Configure Pins**:
  - Enable transmit/receive pins as **digital** signals.
  - Set **AFSEL** for alternate function.
  - Configure **PCTL** to select UART functionality.
- **Enable UART**:
  - Set **TXE (Transmit Enable)**, **RXE (Receive Enable)**, and **UARTEN** to **1** in UART0_CTL_R.
  - During initialization, clear UARTEN before setting other configurations.

Baud Rate Calculation

$$\text{Baud rate} = \frac{\text{Bus clock frequency}}{16 \times \text{divider}}$$

IBRD = divider , FBRD = fraction from divider * 64 (6 bit fraction)

There is 8 uart channels

UART Initialization (UART_Init)

1. **Activate UART1 module**: Enable the clock for UART1 using the system control register.
2. **Enable Port C clock**: Activate the clock for Port C to allow UART1 pins to function.
3. **Disable UART1**: Temporarily disable UART1 to configure its settings safely.
4. **Set integer and fractional baud rate divisors**: Calculate and configure the baud rate divisor using bus clock frequency and desired baud rate.
5. **Configure data format**: Set line control for 8-bit data, no parity, one stop bit, and enable FIFO buffers.
6. **Enable UART1**: Turn on UART1 by setting the appropriate control register bit.

7. **Enable alternative functions on pins**: Configure PC5 and PC4 for UART1 transmit and receive functions.
8. **Enable digital functionality**: Activate digital mode on PC5 and PC4 for UART communication.
9. **Set pin control for UART**: Configure port control to assign UART functionality to PC5 and PC4.
10. **Disable analog mode**: Ensure PC5 and PC4 operate in digital mode by disabling analog functions.

Receiving Data (UART_InChar)

1. **Wait until data is available**: Continuously check if the receive buffer is not empty.
2. **Read received data**: Retrieve and return the data byte from the data register.

Transmitting Data (UART_OutChar)

1. **Wait until buffer is ready**: Check if the transmit buffer is not full.
2. **Write data**: Send data by writing it to the data register.

Non-blocking Data Reception (UART_InCharNonBlocking)

1. **Check if data is available**: Read the status register to determine if data is ready.
2. **Return data or 0**: Retrieve data if available, or return 0 if no data is ready.

```
// UART Initialization - Configures UART1 for 115200 baud rate
void UART_Init(void) {  // Should be called only once
  SYSCTL_RCGCUART_R |= 0x00000002;  // Activate UART1
  SYSCTL_RCGCGPIO_R |= 0x00000004;  // Activate port C

  UART1_CTL_R &= ~0x00000001;  // Disable UART1
```

```
  // Set baud rate: 115200 = 80 MHz / (16 * 43.40278) -> IBRD = 43, FBRD = 26
  UART1_IBRD_R = 43;  // IBRD = int(80,000,000/(16*115,200)) = int(43.40278)
  UART1_FBRD_R = 26;  // FBRD = round(0.40278 * 64) = 26

  UART1_LCRH_R = 0x00000070;  // 8-bit data, no parity, one stop bit, enable FIFOs

  UART1_CTL_R |= 0x00000001;  // Enable UART1

  // Configure PC4 and PC5 as UART1 RX and TX
  GPIO_PORTC_AFSEL_R |= 0x30;  // Enable alternate function on PC4 and PC5
  GPIO_PORTC_DEN_R |= 0x30;    // Enable digital I/O on PC4 and PC5
  GPIO_PORTC_PCTL_R = (GPIO_PORTC_PCTL_R & 0xFF00FFFF) + 0x00220000;  // Set PC4, PC5 as UART1 RX and TX
  GPIO_PORTC_AMSEL_R &= ~0x30; // Disable analog functions on PC4 and PC5
}

// Wait for new input, then return ASCII code
char UART_InChar(void) {
  while ((UART1_FR_R & 0x0010) != 0);  // Wait until RXFE (Receive FIFO Empty) is 0
  return ((char)(UART1_DR_R & 0xFF));   // Read and return the received data
}

// Wait for buffer to be not full, then output data
void UART_OutChar(char data) {
  while ((UART1_FR_R & 0x0020) != 0);  // Wait until TXFF (Transmit FIFO Full) is 0
  UART1_DR_R = data;  // Write data to the transmit register
}

// Immediately return input or 0 if no input is available
char UART_InCharNonBlocking(void) {
```

```
  if ((UART1_FR_R & UART_FR_RXFE) == 0) {  // If RXFE is 0 (data is available)
    return ((char)(UART1_DR_R & 0xFF));  // Read and return the received data
  } else {
    return 0;  // Return 0 if no data is available
  }
}
```

Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time Tbit*(N+1.5)

SUMMARY OF REGISTERS :

UART Registers:

1. **CGCUART Register (UART Clock Gating Control)**

   Enables/disables the clock to UART peripherals.

1. **UARTn_CTL_R (Control Register):**
   - **Description:** Controls the operation of the UART module.
   - **Key Bits:**
     - UARTEN: Enables the UART (set to 1 to enable, clear to disable).
     - TXE: Enables the transmitter (set to 1 to enable, clear to disable).
     - RXE: Enables the receiver (set to 1 to enable, clear to disable).
2. **UARTn_IBRD_R (Integer Baud Rate Divisor Register):**
   - **Description:** Holds the integer part of the baud rate divisor.
   - **Calculation:** IBRD = int(Bus Clock / (16 * Baud Rate)).
3. **UARTn_FBRD_R (Fractional Baud Rate Divisor Register):**
   - **Description:** Holds the fractional part of the baud rate divisor.
   - **Calculation:** FBRD = round((Fractional part of divisor) * 64).
4. **UARTn_LCRH_R (Line Control Register):**
   - **Description:** Configures the word length, parity, stop bits, and FIFO settings.
   - **Key Bits:**
     - WLEN: Word length (typically set to 8 bits).
     - FEN: Enables FIFOs (set to 1 to enable, clear to disable).

- STP2: Configures stop bits (set to 1 for 2 stop bits, clear for 1 stop bit).

5. **UARTn_FR_R (Flag Register):**
   - **Description:** Contains flags indicating the status of the UART.
   - **Key Flags:**
     - RXFE: Receive FIFO empty (set to 1 if no data is available).
     - TXFF: Transmit FIFO full (set to 1 if FIFO is full).
     - BUSY: UART is busy transmitting data.
6. **UARTn_DR_R (Data Register):**
   - **Description:** Holds the data to be transmitted or the received data.
   - **For Transmit:** Write data to this register to send.
   - **For Receive:** Read data from this register after it has been received.
7. **UARTn_RSR_R (Receive Status Register):**
   - **Description:** Contains error flags for received data.
   - **Key Flags:**
     - OE: Overrun error (set if new data arrives before previous data was read).
     - BE: Break error (set if a break signal is received).
     - PE: Parity error (set if parity check fails).
     - FE: Framing error (set if the stop bit is incorrect).
8. **UARTn_IFLS_R (Interrupt FIFO Level Select Register):**
   - **Description:** Configures the interrupt trigger levels for the FIFOs.
   - **Key Bits:**
     - Defines thresholds for interrupt generation when the FIFO reaches certain fill levels.

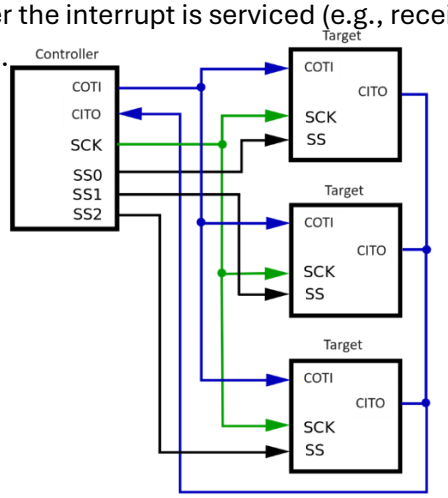
Value	Description
0x0	TX FIFO ≤ ⅞ empty
0x1	TX FIFO ≤ ¾ empty
0x2	TX FIFO ≤ ½ empty (default)
0x3	TX FIFO ≤ ¼ empty
0x4	TX FIFO ≤ ⅛ empty
0x5-0x7	Reserved

Value	Description
0x0	RX FIFO ≥ ⅛ full
0x1	RX FIFO ≥ ¼ full
0x2	RX FIFO ≥ ½ full (default)
0x3	RX FIFO ≥ ¾ full
0x4	RX FIFO ≥ ⅞ full
0x5-0x7	Reserved

9. **UARTn_IM_R (Interrupt Mask Register):**

- **Description:** Controls which UART interrupts are enabled.
- **Key Flags:**
  - Enables or disables interrupts for events like receive FIFO not empty, transmit FIFO empty, etc.

10. **UARTn_MIS_R (Masked Interrupt Status Register):**
    - **Description:** Provides the masked interrupt status for the UART.
    - **Key Flags:**
      - Indicates which interrupts are pending based on enabled interrupts.
11. **UARTn_ICR_R (Interrupt Clear Register):**
    - **Description:** Clears the interrupt flags.
    - **Key Bits:**
      - Clears specific interrupt flags after the interrupt is serviced (e.g., receive FIFO empty, transmit FIFO empty).



SPI (synco):

All chips share bus signals

SPI Example: Secure Digital Card Access

uses **4 wires**: Clock (SCK), Master Out Slave In (MOSI),

Master In Slave Out (MISO), and Chip Select (CS).

I2C (Serial):

- Uses **2 wires**: Serial Data (SDA) and Serial Clock (SCL).
- It can support multiple devices on the same bus but is slower than SPI.
- Each device has a unique 7-bit address.

Protocol	Device Addressing	Signals Req. for Bidirectional Communication with N devices	Speed	Topology
UART (Point to Point)	None	2*N (TxD, RxD)	Fast – Tens of Mbit/s	Point-to-point full duplex
UART (Multi-drop)	Added by user in software	2 (TxD, RxD)	Fast – Tens of Mbit/s	Multi-drop
SPI	Hardware chip select signal per device	3+N for SCLK, MOSI, MISO, and one SS per device	Fast – Tens of Mbit/s	Multi-point full-duplex, multi-drop half-duplex buses
I²C	In packet	2 (SCL, SDA)	Moderate – 100kbit/s, 400 kbit/s, 1Mbit/s, 3.4Mbit/s. Packet overhead.	Multi-point half-duplex bus