Peripherals

1.1 Embedded Systems Communication

Embedded systems rely on hierarchical communication architectures that scale from tightly integrated on-die circuitry to wide-area networked interactions. These communications are conventionally categorized into four interdependent domains: On-Chip, On-board, In-System, and External, each defined by its physical scope, latency requirements, and protocol characteristics. Together, they enable seamless coordination between computation, hardware, subsystem, and end users.

- *On-Chip*: This innermost layer operates entirely within a single integrated circuit, where speed is paramount and physical area, while constrained, remains secondary to latency and bandwidth. Communication occurs between the CPU, memory, and integrated peripherals (e.g., timers, ADCs, DMA controllers) via high-speed parallel buses, commonly transferring data in 8-, 16-, or 32-bit chunks per clock cycle. Parallelism maximizes throughput with minimal protocol overhead, enabling cycle-accurate access to critical resources. Because signals remain confined within the silicon die, noise and skew are tightly controlled—allowing clock frequencies to reach hundreds of MHz or even GHz. This layer forms the real-time foundation: every instruction fetch, interrupt response, or peripheral register access depends on its efficiency.
- *On-board*: When communication extends beyond the chip-across discrete components on a single PCB-design priorities shift. Each additional signal requires a dedicated pin on the IC package and a trace on the board, increasing cost, size, and weight due to larger packages and more complex routing. As a result, serial buses dominate: they transmit data one bit at a time, minimizing pin count and simplifying PCB layout. Common protocols include:
  - SPI for high throughput, full-duplex, point-to-point links (e.g., to an SD card or display driver),
  - I^2C for multi-master, moderate speed, lower-bandwidth buses (e.g., environmental sensors),
  - UART for asynchronous device-to-device links (e.g., GPS modules or debug consoles),
  - I^2S for digital audio streaming.

  If bandwidth becomes limiting, designers may widen the bus (e.g., dual- or quad-SPI), increase clock rates, or adopt embedded SerDes (serializer/deserializer) links—though the latter introduces greater complexity. Signal integrity (e.g., impedance matching, crosstalk mitigation) becomes critical, as trace lengths and board stackup now influence timing margins.
- *In-System*: This tier coordinates multiple embedded nodes—such as Electronic Control Units (ECUs) in an automobile, subsystems in industrial machinery, or distributed sensors in a smart building. Here, determinism, fault tolerance, and electromagnetic compatibility (EMC) outweigh raw speed. Protocols like CAN, LIN, and FlexRay are purpose-built for harsh, electrically noisy environments. CAN, for instance, uses differential signaling

and built-in error detection with acknowledgment and retransmission to ensure message integrity — critical for safety-critical functions like braking or steering. Bus arbitration, time-triggered scheduling (e.g., FlexRay), and redundancy further enhance robustness. Cabling and connectors add cost, weight, and failure points, so minimizing conductor count remains important — even as systems grow more interconnected. Modern platforms increasingly supplement classical buses with Ethernet (e.g., 100BASE-T1, TSN) to support high-bandwidth domains (e.g., cameras, LiDAR), while preserving legacy interfaces for low-complexity nodes.

- *External*: The outermost domain connects the embedded system to external entities—users, cloud services, or other physical systems (e.g., phone — thermostat, vehicle — traffic infrastructure). Communication now traverses uncontrolled environments: longer cables, connectors, or wireless channels introduce significant noise vulnerability, latency, and security risks. As a result, robust error control is mandatory: end-to-end checksums, acknowledgments (ACK/NACK), retransmission (e.g., TCP), and forward error correction (FEC) are common. Solutions diverge based on application needs:

  - Wired, high-performance: USB 2.0/3.0, Ethernet (10/100/1000BASE-T), or CAN FD—where bandwidth justifies cables and shielding. Wider buses or higher clock speeds (e.g., SuperSpeed USB at 5 Gbps) compensate for serial bit-by-bit transmission.
  - Wireless, portable: Wi-Fi (802.11 a/b/g/n/ac/ax), LTE/5G, or IEEE 802.15.4 (Zigbee/Thread)—enabling mobility and flexible deployment, albeit with trade-offs in power, range, and interference resilience.

  While this layer often operates at human timescales (hundreds of milliseconds to seconds), it depends entirely on the underlying tiers: commands traverse the stack downward, while status and telemetry flow upward, making end-to-end latency and data consistency system-wide concerns.

These four domain form a cohesive, nested hierarchy: each builds on the services and abstractions of the one below it. Communication is inherently bidirectional: control and configuration descend, while sensing and status ascend, and system performance hinges on optimizing transitions between layers. Modern embedded design thus requires co-engineering of hardware, firmware, and protocols across all tiers to achieve responsiveness, reliability, and scalability in real-world applications.

Bus. In embedded systems, a bus is a shared communication infrastructure that enables data, address, and control signals to be exchanged among multiple system components (e.g., processor, memory, peripherals). Unlike point-to-point links, buses allow multiple devices to connect to a common pathway, reducing wiring complexity and cost, but introducing the need for arbitration and protocol coordination. A bus implementation involves two interdependent layers:

- *Hardware Infrastructure* defines the physical medium: conductive traces on a PCB, cables (e.g., ribbon or shielded twisted pair), connectors, and transceivers. For instance, while desktop PCI uses edge connectors and backplanes, embedded systems favor on-board traces (e.g., SPI lines routed between MCU and sensors) or compact connectors (e.g., CAN on a DB9 or OBD-II port).
- *Software Infrastructure* governs how communication occurs: addressing, data framing, error handling, clocking, and arbitration. The PCI bus protocol, for example, specifies command

phases, burst transfers, and retry mechanisms — similar concepts appear in embedded protocols like CAN (with identifier-based arbitration) or I^2C (with START/STOP conditions and ACK/NACK).

Embedded bus architecture commonly involve:

- *Master*: The device (typically the processor or DMA engine) that initiates and controls transactions by issuing addresses and control signals.
- *Slave*: A target device (e.g., memory chip, sensor, or peripheral register) that responds to master requests. Multiple slaves coexist on the same bus.
- *Address Decoder*: Logic (often built into the slave or a dedicated chip) that monitors the address bus and activates the correct slave when its address range is selected.
- *Multiplexer*: Used in some designs to route data or address lines between multiple potential sources or destinations — e.g., to share a data bus among several peripherals under master control.

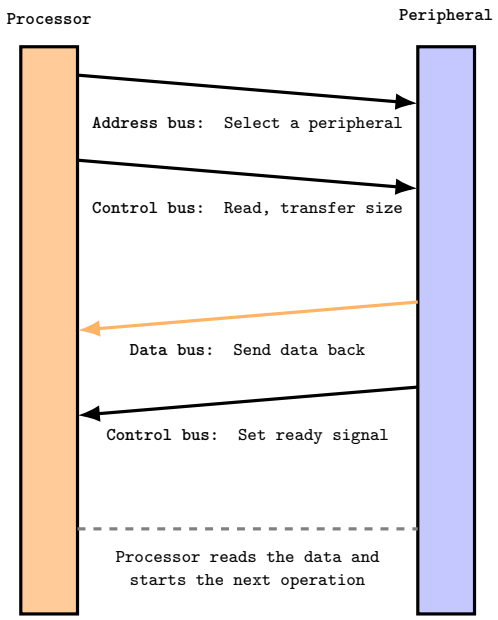
A typical synchronous parallel bus — though increasingly rare in modern low-pin-count embedded designs — comprises three fundamental signal groups:

(1) *Data Bus*: A bidirectional set of lines (e.g., 8, 16, or 32-bit wide) that carries payload information between master and slave.

(2) *Address Bus*: A unidirectional set of lines specifying which slave (or which register within a slave) is being accessed. The width determines the addressable space (e.g., a 16-bit address bus can select up to 64 KB of memory-mapped I/O).

(3) *Control Bus*: Timing and command signals that orchestrate the transaction, such as Read/Write to indicate data direction, Chip Select or Slave Select, Clock, and Ready or Wait for flow control.

An example. The communication cycle begins when the processor places the address of a target peripheral — or a specific register within it — onto the address bus. At the same time, it drives the control bus with signals indicating the operation type (e.g., read), data width, and timing parameters. The master then enters a wait state, allowing the peripheral time to prepare the requested data. Once ready, the slave places the data onto the data bus and asserts a ready (or acknowledge) signal on the control bus to confirm completion. Upon detecting this handshake, the master latches the data and concludes the transaction, immediately advancing to the next stage — whether initiating another transfer, processing the received value, or continuing program execution.



ARM AMBA System Bus. The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard on-chip interconnect specification developed by Arm to define communication protocols between functional blocks (IP cores) in system-on-chip (SoC) designs. By providing a consistent, modular interface framework, AMBA significantly enhances IP reuse, accelerates integration, and supports right-first-time development — especially critical for complex multiprocessor systems incorporating numerous processors, memory controllers, and peripherals. As a cornerstone of modern SoC design, AMBA is extensively deployed in power and performance-sensitive applications, including smartphones, tablets, and embedded devices.

Among the AMBA protocol family, the Advanced High-performance Bus (AHB) targets high-speed, synthesizable system designs. AHB supports multiple bus masters, pipelined transfers, and high-bandwidth operation, making it suitable for connecting high-throughput components such as CPUs, DMA controllers, and high-seed memories. A streamlined variant, AHB-Lite, simplifies the full AHB specification — most notably by assuming a single bus master — thereby reducing logic complexity and verification effort. In an AHB-Lite transaction, the master initiates read or write operations by driving address, control, and data (for writes) signals onto the bus. Slaves respond with data (for reads), along with handshake signals: `HREADY` (indicating transfer completion) and `HRESP` (signaling success or error conditions). Slave selection is managed by the address decoder, which asserts the corresponding `HSELx` signal for the targeted slave during each access, ensuring only one slave responds per cycle.

1.2 Addressing

Memory Mapping. The ARMv6-M architecture (used in Cortex-M0 and M0+ processors) defines a flat, linear 32-bit address space, yielding a total of 4 GiB of addressable locations. This space is logically partitioned into predefined regions — not all of which correspond to physical memory — and the layout is standardized across all ARMv6-M implementations to ensure software portability. The canonical ARMv6-M memory map is divided as follows:

Address Range	Region Name	Description
`0x0000_0000` - `0x1FFF_FFFF`	Code (Alias)	Typically maps to Flash (via aliasing mechanisms); supports execution-in-place (XIP).
`0x2000_0000` - `0x3FFF_FFFF`	SRAM	On-chip RAM (often tightly coupled memory, TCP-like behavior).
`0x4000_0000` - `0x5FFF_FFFF`	Peripheral	Memory-mapped I/O space for system and external peripherals (e.g., UART, GPIO, timers).
`0x6000_0000` - `0x9FFF_FFFF`	External RAM	Optional external memory (e.g., SDRAM, SRAM via FSMC/EBI).
`0xA000_0000` - `0xDFFF_FFFF`	External Devices	Memory-mapped external peripherals (e.g., LCD controllers, Ethernet MACs).
`0xE000_0000` - `0xFFFF_FFFF`	Private Peripheral Bus (PPB)	System control space: NVIC, SysTick, MPU, debug components (e.g., DWT, ITM, ROM table).

Table 1 / Memory map of a typical ARM Cortex-M-based microcontroller.

Crucially, all I/O in ARMv6-M is memory-mapped — there is no dedicated I/O address space or I/O instructions. Peripherals — both general-purpose (e.g., GPIO at `0x4002_0000`) and system-critical (e.g., NVIC at `0xE000_E100`) — occupy fixed or configurable regions within the unified 32-bit address map.

Though all peripherals use the same address space, they are not treated like normal memory. The architecture assigns memory type attributes per region, enforced by the optional Memory Protection Unit (MPU) or hardwired in the bus matrix. These attributes ensure that accesses to peripheral registers have deterministic timing and side effects, preventing compiler or hardware optimizations (e.g., reordering, caching, write combining) that would break device driver logic. Additionally, while ARMv6-M is byte-addressable, not all peripheral support arbitrary byte or half-word accesses — even if the address is aligned. Access width must match the register definition; mismatched accesses may cause bus faults or undefined behavior.

Isolated (Port-Mapped) I/O. In contrast, isolated I/O, also known as port-mapped I/O (PMIO), uses a separate address space dedicated exclusively to I/O devices — distinct from the main memory address space. Key features:

- A dedicated I/O address space, typically much smaller (e.g., 64 KiB).
- Specialized instructions for I/O access, e.g., `IN` and `OUT` on x86, or `IN` / `OUT` mnemonics on 8051 or Z80.
- The CPU's memory-access instructions (e.g., `LDR`, `STR`) cannot access I/O ports; only I/O instructions can.
- Physical separation: often implemented via a distinct set of control signals (e.g., `/IOR`, `/IOW` on ISA bus vs. `/MEMR`, `/MEMW`).

Advantages for isolated I/O include a larger effective memory space (since I/O doesn't consume main memory addresses, the full physical memory address space remains available for RAM/ROM), clear separation of concerns (I/O operations are syntactically and semantically distinct, reducing accidental accesses), and simpler hardware decoding (I/O port decode logic can be simpler and faster than full 32-bit memory decoding). However, it also brings drawbacks, since it requires additional instructions and CPU microarchitecture support, cannot use general-purpose addressing modes for I/O without extra overhead, inline assembly is often needed, and is harder to virtualize or remap.

1.3 Peripherals.

In embedded systems, peripheral controllers like GPIO (General Purpose Input/Output), UART (Universal Asynchronous Receiver-Transmitter), and SPI (Serial Peripheral Interface) are typically accessed via memory-mapped interfaces. To interact with these peripherals, software reads from and writes to specific registers located at fixed address offsets. For example, the Xilinx UARTLite has a Receive FIFO at offset `0h`, a Transmit FIFO at `04h`, a Status Register at `08h`, and a Control Register at `0Ch`.

C-Level Memory Access. We can interact with these hardware registers using C pointers. We can define `peek` (read) and `poke` (write) functions to demonstrate this mechanism. The pointer is dereferenced to access the specific memory location assigned to the device.

C

```c
// Reading from a register
int peek(char *location) {
    return *location;  // dereference location pointer to read
}

// Writing to a register
void poke(char *location, char newval) {
    (*location) = newval;  // Write newval to address 'location'
}

// Example usage
#define DEV1 0x1000
dev_status = peek(DEV1);  // Read status
poke(DEV1, 8);            // Write 8 to the device
```

When writing this type of low-level code, specific keywords ensure the compiler handles variables correctly:

- `const` : Makes a variable or pointer parameter unmodifiable.
- `static` : Preserves a variable's value after its scope ends and prevents it from being placed on the stack.
- `volatile` : This is critical for embedded I/O. It tells the compiler that the variable (or register) can change in the background (e.g., by hardware) without the software's explicit action. It prevents the compiler from optimizing away reads/writes to that address.

Data Transmission Protocols. Data transmission between devices relies on knowing when data is valid. This is handled in two main ways:

(1) *Synchronous Communication*: Uses a separate clock signal. The receiver samples data on a specific clock edge (e.g., rising edge). This is common in parallel communication or protocols like SPI.

(2) *Asynchronous Communication*: Does not use a clock signal. Instead, the receiver infers bit times based on a fixed baud rate and specific framing delays relative to a reference event (like a start bit).

UART (Universal Asynchronous Receiver-Transmitter) is the standard implementation for asynchronous serial communication. Since there is no clock, it relies on strict framing to signal the start and end of data. The line is held high during the Idle state. The transmission frame consists of:

- *Start Bit*: The transmitter pulls the line low to signal the beginning of a message.
- *Data Bits*: Usually 8 bits are sent, starting with the LSB.
- *Parity Bits*: Used for error detection. In Even Parity, the bit is set to make the total number of 1s even. In Odd Parity, it makes the total count odd. This can detect an odd number of bit errors but fails if an even number of bits are flipped.
- *Stop Bit*: The line is pulled high to end the frame and return to the Idle state.

There are two primary models for managing peripheral data flow:

(1) *Polling (Busy Wait)*: In polling, the processor continually interrogates the peripheral's status register to check if it is ready to send or receive data. While simple to implement, this wastes CPU cycles because the processor is fully occupied "waiting" for the hardware. The following code demonstrates a blocking send routine for the UARTLite. It sits in a `while` loop checking the "Transmit FIFO Full" bit in the status register before writing a byte.

C

```c
void XUartLite\_SendByte(int *BaseAddress, char Data) {
    // Busy wait: keep looping while the Transmit FIFO is full
    while (XUartLite\_IsTransmitFull(BaseAddress));

    // Write data to the TX FIFO register
    XUartLite\_WriteReg(BaseAddress, XUL\_TX\_FIFO\_OFFSET, Data);
}
```

(2) *Interrupts*: Interrupts are asynchronous events that trigger the execution of a specific procedure called an interrupt handler. These allow the CPU to only be active regarding the peripheral when a status change actually occurs, allowing it to do other work otherwise, but they introduce complexity in debugging and predictability. The flow is as follows:

(a) The peripheral asserts an interrupt signal.

(b) The processor saves its current context (PC, registers) and jumps to the interrupt handler.

(c) The processor acknowledges the interrupt (often by toggling an acknowledgment bit in the interrupt controller or the peripheral).

(d) After handling the event, the processor restores the context and resumes the main program.

In Xilinx designs, interrupts are managed by an AXI Interrupt Controller. To enable interrupts on a peripheral like the AXI GPIO, you must set the Global Interrupt Enable (GIE) bit and the Interrupt Enable Register (IER) for the specific channel.