

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),
 - I²C 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),
 - I²S 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 domains 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 I²C (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.

