Automotive SoC Design

Day - 1 Introduction to SoC Design

A **System on Chip (SoC)** is an integrated circuit that combines all essential components of a computing system—including processors, memory, peripherals, and communication interfaces—onto a single chip. SoCs are designed to be power-efficient, compact, and optimized for specific application requirements, balancing performance, cost, and power consumption.

Design Specifications for a SoC

Designing a System on Chip (SoC) involves balancing multiple competing requirements known as **design drivers**. These drivers determine key architectural and implementation choices.

1. Performance

- The SoC must meet the required **functional throughput** and **timing constraints**.
- Performance depends on CPU speed, memory bandwidth, bus architecture, and peripheral access times.
- Applications such as real-time systems, high-resolution video processing, or gaming demand high performance.

2. Power Consumption

- Especially critical in **mobile, IoT, and battery-operated devices**.
- Designers optimize both **dynamic power** (switching activity) and **static power** (leakage).
- Techniques like **clock gating**, **power gating**, and **DVFS (Dynamic Voltage and Frequency Scaling)** are commonly used.

3. Cost

- Lower **production cost** is essential for consumer products.
- Influenced by:
  - Silicon area (smaller dies = cheaper to fabricate).
  - Number of metal layers.
  - IP licensing fees.
  - Test and packaging overhead.
- Reusing **pre-verified IP blocks** can significantly reduce development cost.

4. Size Constraints

- Particularly important in **mobile, wearable, and embedded systems** where board space is limited.
- A smaller SoC can:
  - Reduce system size.
  - Lower material usage.
  - Improve thermal characteristics.

5. Customization

- SoCs may be:
  - **Fully custom-designed** for specific applications (e.g., Apple Silicon for iPhones).
  - **Modular** using reusable IP blocks to accelerate development (common in FPGAs and commercial SoCs).
- Customization enables product differentiation and performance tuning for target use cases.

> Each design decision in SoC development is guided by these drivers, and successful designs typically strike the right balance among them based on the end application.

Components of a SoC

1. Processing Units

- **CPUs (Central Processing Units):** Handle general-purpose processing tasks, such as running operating systems and application logic.
- **GPUs (Graphics Processing Units):** Specialized for massively parallel processing, ideal for graphics rendering, video processing, and AI workloads.
- **DSPs (Digital Signal Processors):** Optimized for real-time numerical processing, common in audio, sensor data, and communication systems.

> **Why not always use a powerful GPU or CPU?**
> SoC design must balance **power consumption, area, cost, and complexity**. For simple tasks, a small CPU suffices and conserves power and silicon area.

2. Memory

- **RAM (Random Access Memory):** Volatile memory for temporary data storage during program execution.
- **ROM (Read-Only Memory):** Non-volatile memory storing firmware, bootloaders, or other permanent instructions/data essential for SoC operation.

3. Peripherals

Interfaces for communication and control outside the SoC:

- Timers
- UART (Universal Asynchronous Receiver/Transmitter)
- SPI (Serial Peripheral Interface)
- I2C (Inter-Integrated Circuit)
- CAN (Controller Area Network)
- GPIO (General Purpose Input/Output)
- USB
- SDIO (Secure Digital Input Output)

These peripherals facilitate interaction with sensors, user inputs, communication buses, and storage devices.

4. Interconnects (Bus Fabric)

Efficient communication between CPUs, memory, and peripherals requires structured bus architectures rather than direct wiring to meet timing, bandwidth, and modularity constraints. Common industry-standard interconnects include the **AMBA** (Advanced Microcontroller Bus Architecture) family by ARM:

- **AXI (Advanced eXtensible Interface):**
  High-performance bus supporting burst transfers, out-of-order transactions, and pipelining. Used for CPU-memory and high-speed peripheral communications.
- **AHB (Advanced High-performance Bus):**
  Medium-performance bus for communication between CPUs and high-speed peripherals or DMA controllers.
- **APB (Advanced Peripheral Bus):**
  Low-bandwidth, low-power bus used for simple peripherals such as GPIO, UART, and timers.

5. System Components

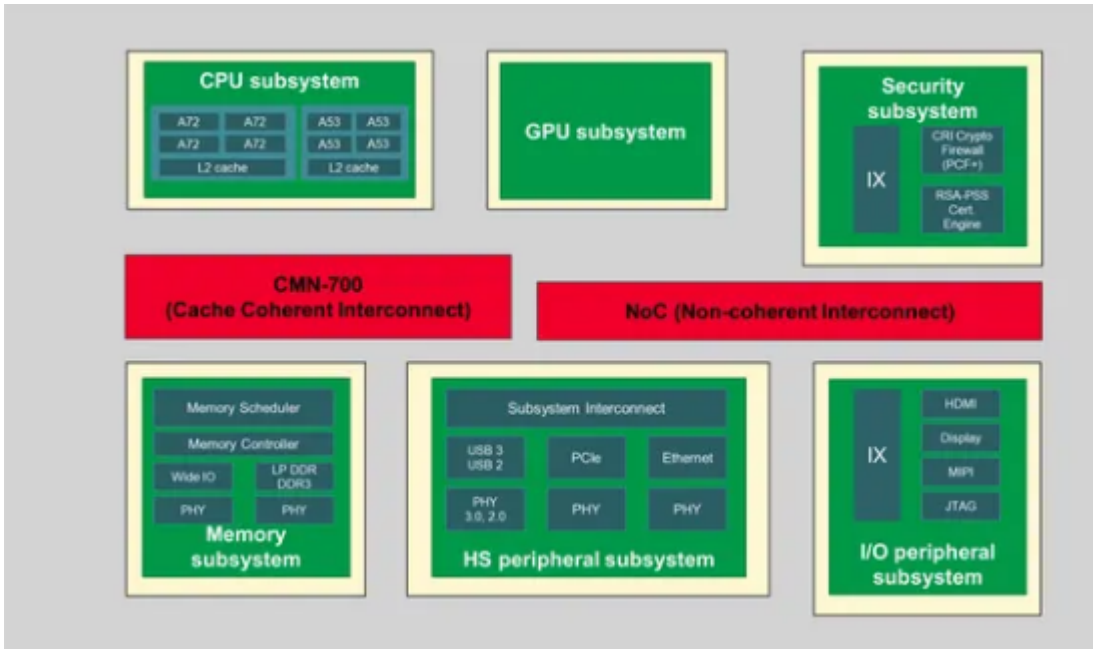
Essential for stable SoC operation:

- **Clock Generators:** Provide timing signals to synchronize operations.
- **Reset Controllers:** Manage system reset signals to initialize hardware into known states.
- **Power Management Units / Regulators:** Control voltage and power modes to optimize energy efficiency.

6. Security Blocks

Modern SoCs often embed hardware security features to protect against software and physical attacks:

- **MPUs (Memory Protection Units):** Enforce access permissions on memory regions.
- **TrustZone Controllers:** Provide hardware-enforced secure and non-secure execution environments (ARM-specific).
- **Cryptographic Engines:** Accelerate encryption, decryption, hashing, and secure key storage.



Types of SoC Architectures

Monolithic Architecture

- All system functions are integrated into a single chip, typically as an ASIC (Application-Specific Integrated Circuit).
- Optimized for a specific task or product.
- Pros: High performance, low power, compact.
- Cons: Hard to modify, not reusable, expensive to redesign.

Modular IP-based Architecture

- SoC is built using modular and reusable IP (Intellectual Property) blocks.
- These IP blocks include processor cores, memory controllers, peripheral interfaces, etc.
- Advantages:
  - Faster design and verification using pre-designed components.
  - Easier customization and upgrade paths.
  - Widely adopted in industry (e.g., ARM Cortex cores as IP blocks).

Processor Families

Cortex-M Series

- Designed for low-power, real-time microcontroller-class applications.
- Suitable for embedded systems, wearables, and IoT nodes.
- Examples: Cortex-M0, M3, M4, M7

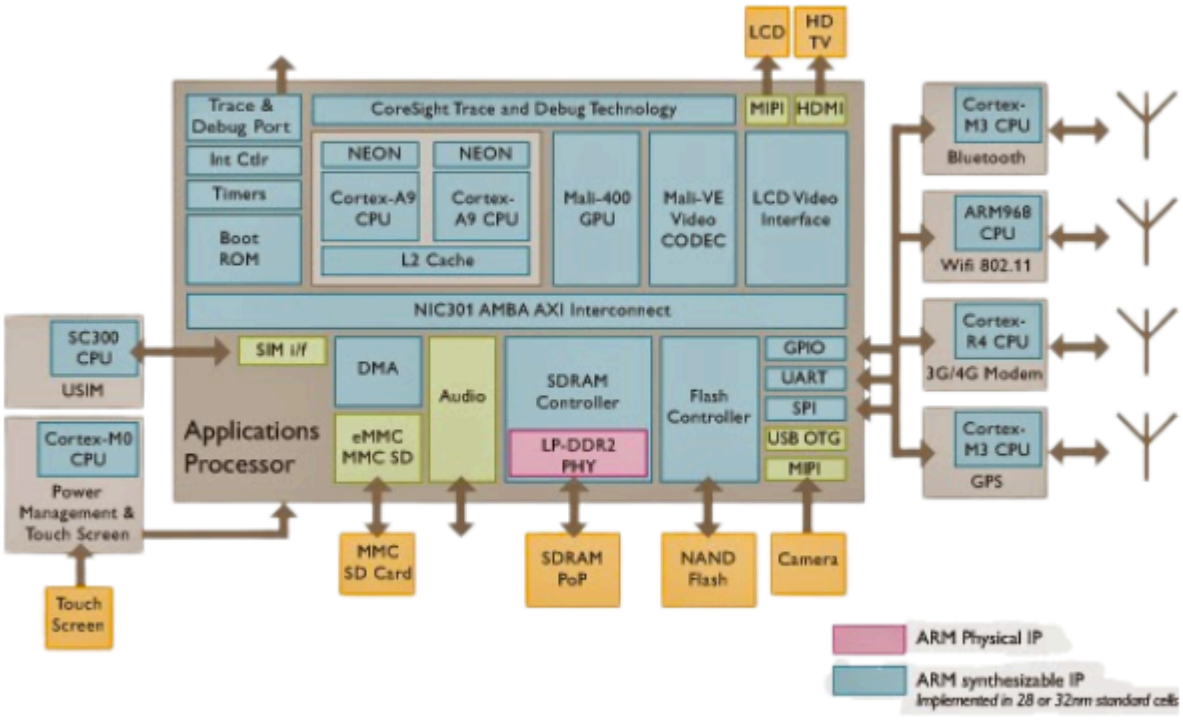
Cortex-A Series

- High-performance application-class cores.
- Designed to run full operating systems like Linux or Android.
- Found in smartphones, tablets, digital TVs, and other high-end consumer devices.
- Examples: Cortex-A7, A9, A53, A76

MCU vs SoC

Feature	MCU (Microcontroller Unit)	SoC (System on Chip)
Complexity	Simpler	More complex
Integration	CPU, small RAM/ROM, basic peripherals	CPU, GPU, memory, wireless, etc.
Target Use	Control tasks, real-time systems	Rich applications, multimedia, AI
OS Support	Often bare-metal or RTOS	Full OS like Linux, Android

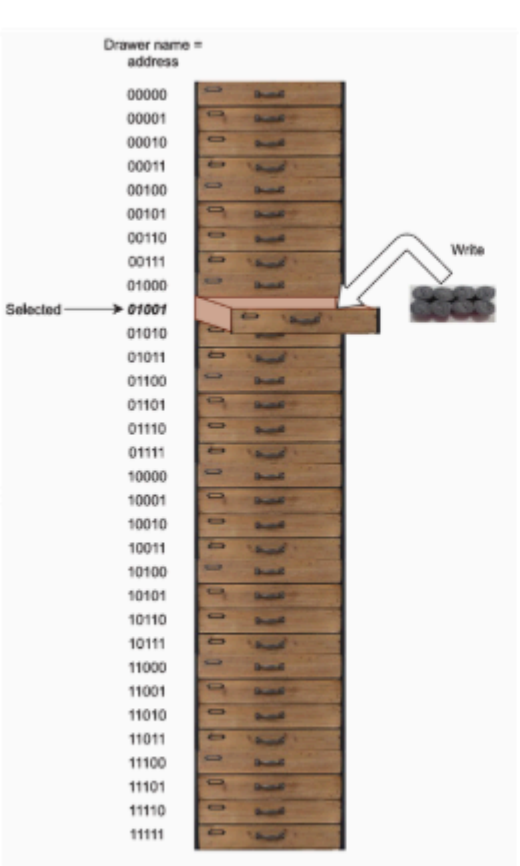
- **MCUs** are suitable for basic control-oriented tasks with strict power and timing constraints.
- **SoCs** provide higher integration and are suited for performance-intensive applications requiring multitasking and connectivity.

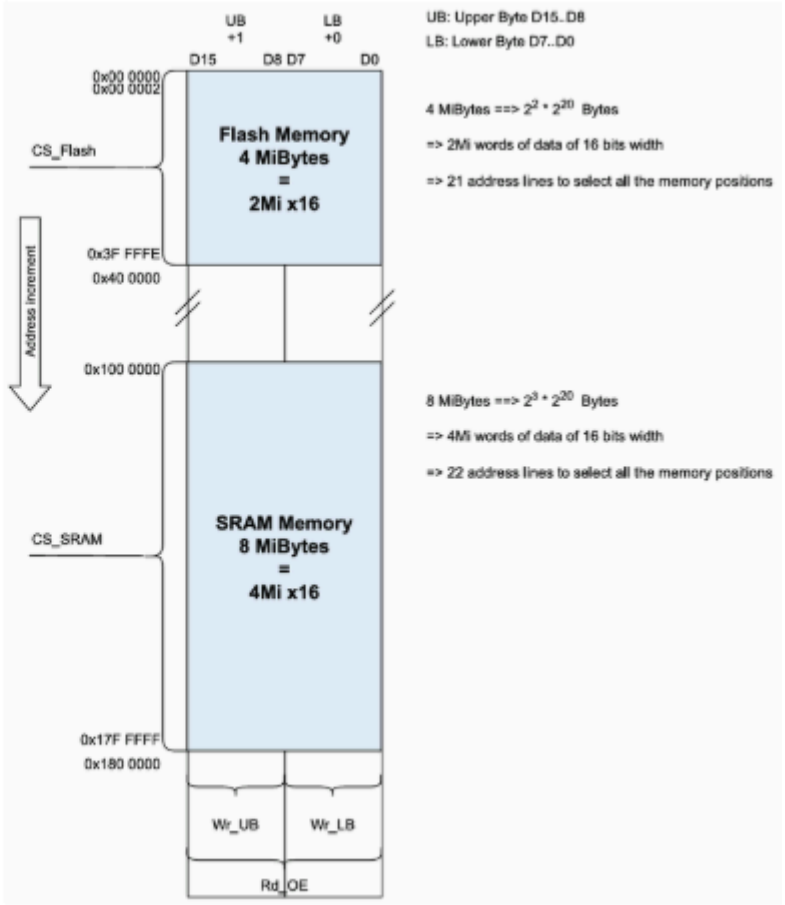


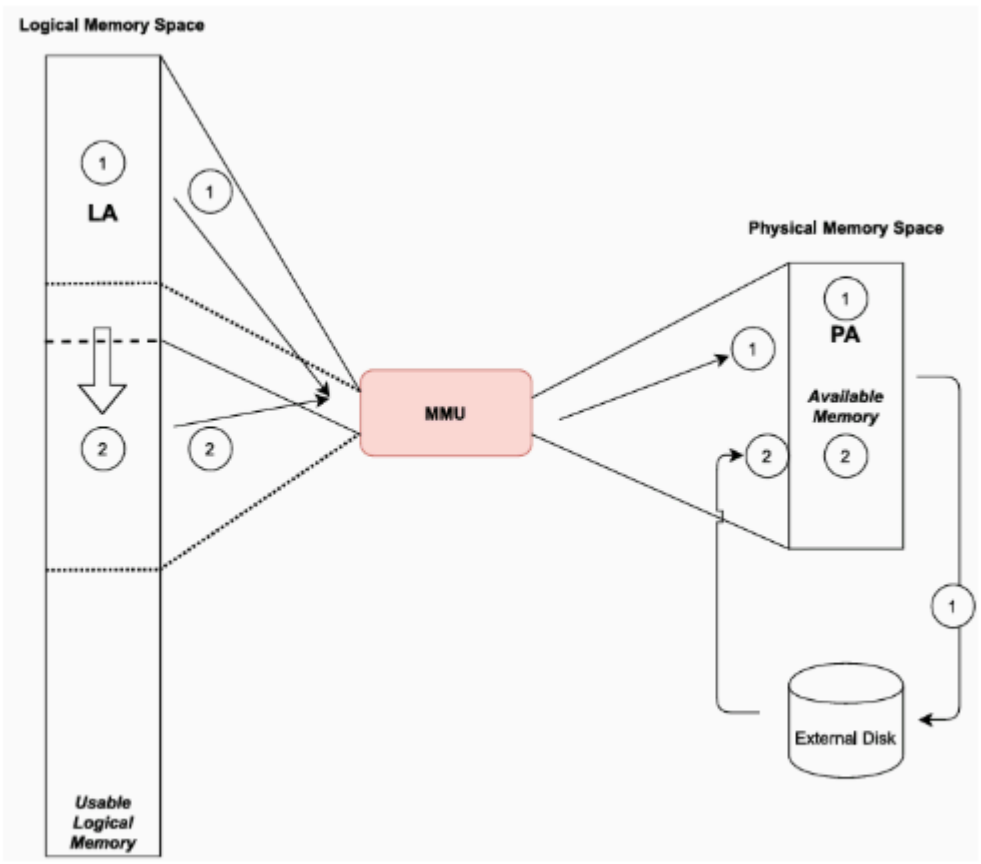
Memory Management

Memory Types & Hierarchy

- **On-Chip vs Off-Chip Memory:**
  On-chip memory (like SRAM and caches) is faster but limited in size. Off-chip memory (e.g., DRAM, Flash) offers larger capacity but slower access.
- **Memory Hierarchy:**
  Registers → L1/L2 Cache → RAM → Flash
- **Cache:**
  Small, fast memory layers (L1, L2) that store frequently accessed data to reduce latency.
- **MMU (Memory Management Unit) / MPU (Memory Protection Unit):**
  Translate virtual addresses to physical addresses and enforce access controls to improve security and multitasking.
- **Boot and Update:**
  Boot code resides in non-volatile memory (Flash/ROM). Firmware updates can be performed via interfaces like JTAG, UART, or over-the-air.







MMU with virtual memory

DMA (Direct Memory Access)

DMA enables peripherals to transfer data directly to/from memory without CPU intervention, significantly improving system efficiency.

Without DMA:

- Peripheral requests data → CPU requests memory → CPU passes data to peripheral.
- CPU becomes a bottleneck, wasting cycles on simple data moves.

With DMA:

- CPU configures DMA controller with source/destination and transfer size.
- DMA engine moves data independently of CPU.
- CPU freed to perform other tasks, improving throughput and responsiveness.

Summary:
Peripheral ↔ DMA ↔ Memory (bypasses CPU after setup)

GPIO (General-Purpose Input/Output)

- Configurable digital pins for input or output.
- Used for reading buttons, switches, sensors or controlling LEDs, relays, and other devices.
- Programmable direction and state.

UART (Universal Asynchronous Receiver/Transmitter)

- Serial communication protocol transmitting data bit-by-bit asynchronously.
- Uses two lines: TX (Transmit) and RX (Receive).
- Common for debugging, console I/O, and communication with low-speed peripherals.
- Communication requires both devices to agree on **baud rate** (e.g., 9600, 115200).

SPI (Serial Peripheral Interface)

- Synchronous serial communication protocol with separate clock line.
- Signals: MOSI, MISO, SCLK, and Slave Select (SS).
- Faster and more flexible than UART but limited to short distances.
- Used with sensors, SD cards, displays, ADCs, etc.

System Timers

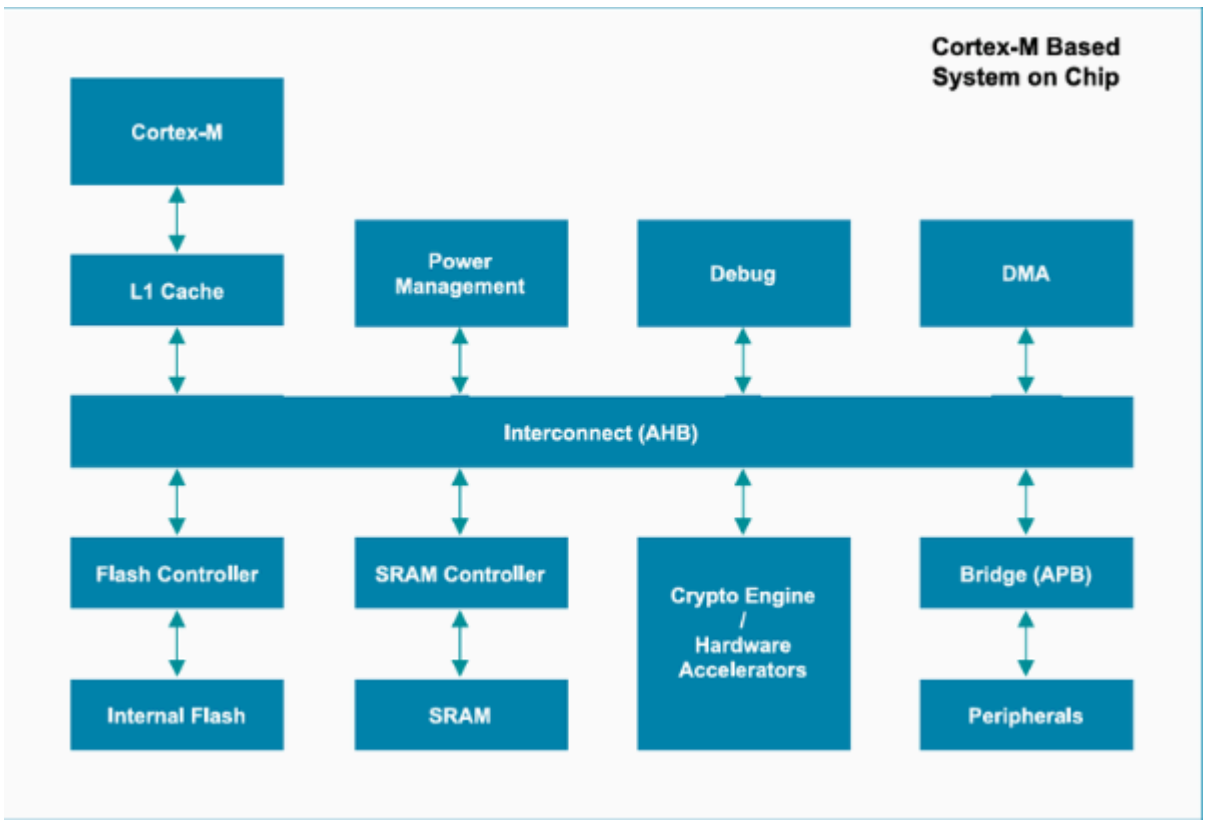
- Hardware timers tracking system uptime or generating periodic interrupts.
- Used for task scheduling, timeouts, and event timestamping.
- Often integrated with OS tick in RTOS environments.

Watchdog Timer

- Safety timer that resets the system if software becomes unresponsive.
- Requires periodic "kicks" or refreshes by running software.
- Prevents system hangs or crashes from locking the device indefinitely.

System Boot Process

1. **Power-Up and Reset:**
   Power-On Reset (POR) initializes system to a known state. Other reset sources include external reset pins, watchdog timers, software-triggered resets, and brown-out detection.
2. **Vector Table Fetch:**
   Processor fetches initial Stack Pointer (SP) and Reset Handler address from vector table at a fixed memory location (usually 0x0000_0000).
   - ARM Cortex-M variants support relocating vector table via VTOR.
   - TrustZone-enabled processors have separate secure and non-secure vector tables.
3. **Reset Handler Execution:**
   Initializes stack pointer limits, calls `SystemInit()` to configure clocks, PLLs, and peripherals, then transfers control to C runtime start-up.
4. **System Initialization (`SystemInit`):**
   Configures system clocks, flash wait states, bus matrix arbitration, and peripheral clocks.
5. **C Runtime Initialization:**
   - Zero-initializes `.bss` section (uninitialized variables).
   - Copies initialized data from flash to RAM (`.data` section).
   - Sets up heap and stack memory.
   - Calls `main()` to start application execution.



Day 2 - Memory Protection Unit (MPU) – ARM Cortex-M

The **Memory Protection Unit (MPU)** is a hardware feature in ARM Cortex-M processors that improves system **stability and security** by dividing memory into regions and enforcing **access permissions**.

Why is it needed?

In embedded systems, multiple tasks or components may share memory. Without protection, a bug in one task could corrupt another task's memory. The MPU prevents this by restricting who can access what.

Key Features

- Region-based protection (e.g., code, stack, heap, peripherals)
- Controls for read/write, execute, and access privileges
- Differentiates **privileged** vs **unprivileged** code
- Does **not require an MMU**, so it’s suitable for microcontrollers

Example Use Case

Assume:

- Task A has stack at memory address `0x20001000–0x20001FFF`
- Task B has stack at memory address `0x20002000–0x20002FFF`

The MPU can restrict Task A from writing into Task B’s stack. If it does, a **Memory Management Fault** is raised.

Real-World Analogy

Think of MPU like assigning rooms in a building:

- Each task has its own room (memory region)
- Only authorized people (privileged code) can enter locked rooms
- Unauthorized access raises an alarm (fault)

Benefits

- Catches memory access bugs
- Prevents task interference
- Enables safer multitasking
- Adds basic security for embedded systems

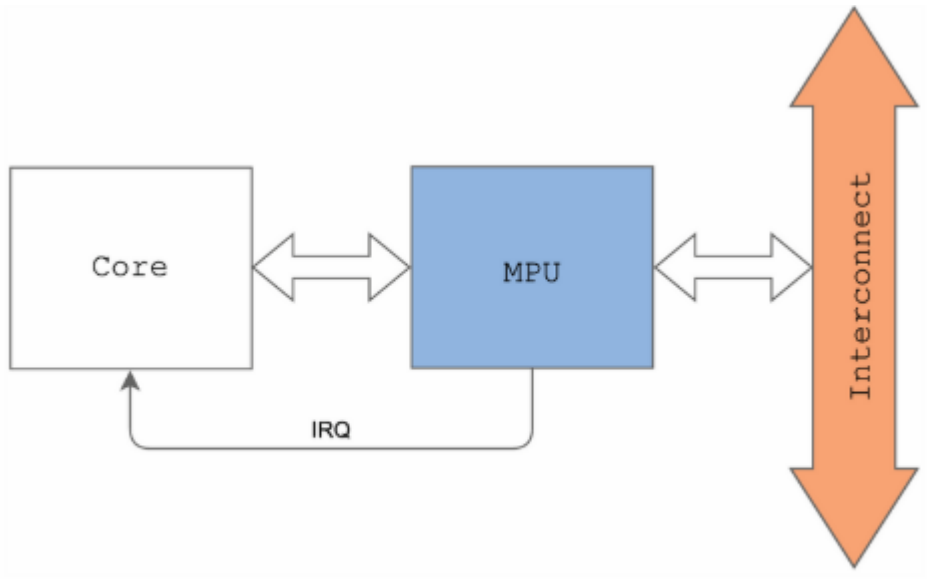


Figure 3.24 Location of the MPU in a system

ARM MPU Architecture (v7-M)

The ARM Cortex-M MPU architecture (ARMv7-M) supports up to **8 configurable regions**, each with flexible control over access permissions and memory attributes.

Region Configuration

Each MPU region is defined using:

- **Base Address**
- **Region Size** (power-of-two aligned)
- **Control Bits** (e.g., access type, execute permission)

Regions can be enabled or disabled **dynamically**.

Control & Status Registers

- `MPU_TYPE` : Indicates number of supported regions.
- `MPU_CTRL` : Enables/disables MPU and background region.
- `MPU_RNR` : Selects current region number.
- `MPU_RBAR` : Region Base Address Register.
- `MPU_RASR` : Region Attribute and Size Register.

Register Name	Access	Address	Function
MPU_TYPE	RO	0xE000_ED90	Gives the numbers of regions that can be configured
MPU_CTRL	RW	0xE000_ED94	Enable or disable the MPU
MPU_RNR	RW	0xE000_ED98	Select the region to be configured
MPU_RBAR	RW	0xE000_ED9C	Set the base address of the region selected
MPU_RASR	RW	0xE000_EDA0	Set the size and attributes of a region (NOT in Arm-v8-M)

Region Attributes

- **XN (Execute Never)**: Marks region as non-executable.
- **AP (Access Permissions)**: Defines read/write rights for privileged/unprivileged code.
- **Memory Types**: Specifies cacheable, bufferable, strongly ordered, device memory, etc.

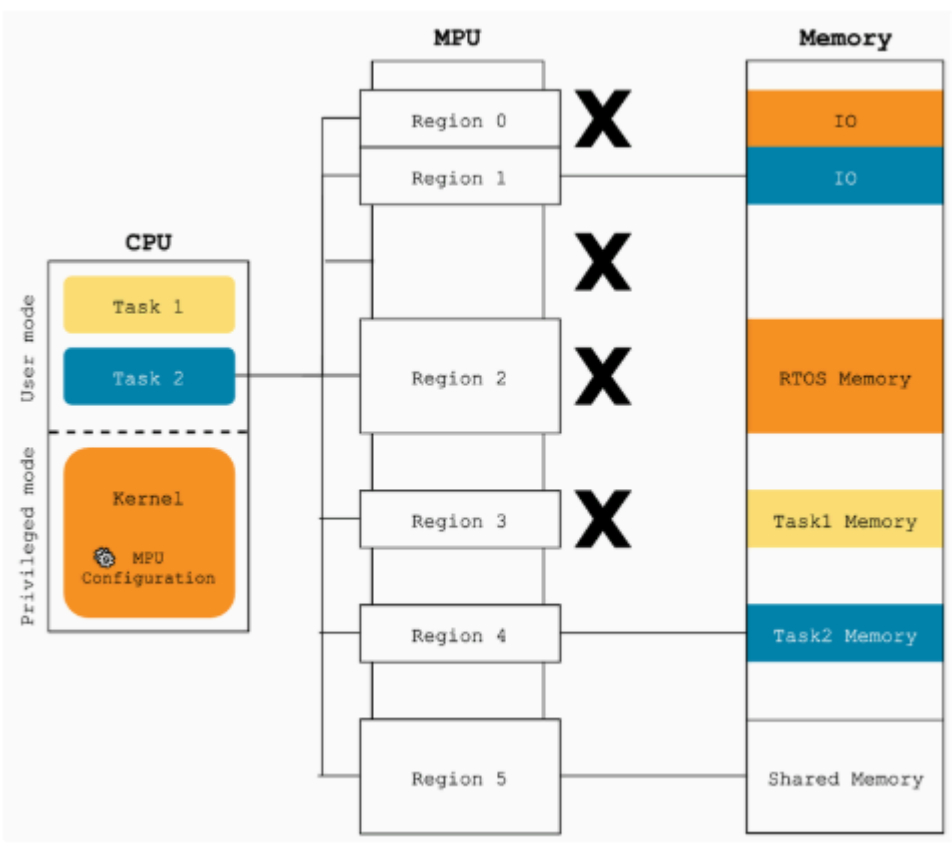
Region Overlap Priority

If regions **overlap**, the region with the **higher region number** takes precedence. This allows finer-grained overrides.

Example:

To mark a region `0x20000000–0x20000FFF` as read-only for unprivileged code:

- Set base = `0x20000000`
- Size = 4KB
- Access = Privileged RW, Unprivileged RO
- XN = 0 (executable allowed if needed)



MPU Memory Region Types

Memory regions in the ARM Memory Protection Unit (MPU) are categorized into **types** that determine how memory accesses behave in terms of **caching, buffering, and access ordering**. These types are critical for balancing **performance, correctness, and predictability** in embedded systems.

1. Normal Memory

- **Usage**: Used for regions like Flash, RAM, and ROM—typically where code and data are stored.
- **Features**:
  - Supports **speculative access**, **prefetching**, and **out-of-order execution**.
  - Can be marked **cacheable** and/or **bufferable** for improved performance.
- **Typical Use Case**: Executing code, reading/writing to application data stored in RAM.

2. Device Memory

- **Usage**: For memory-mapped **peripheral regions**.
- **Features**:
  - **No speculative access**—the processor never prefetches or reorders reads/writes.
  - Writes can be **buffered** but must **appear in program order**.
  - Ensures proper ordering for I/O operations like UART, SPI, etc.
- **Typical Use Case**: Reading from or writing to I/O devices or hardware registers.

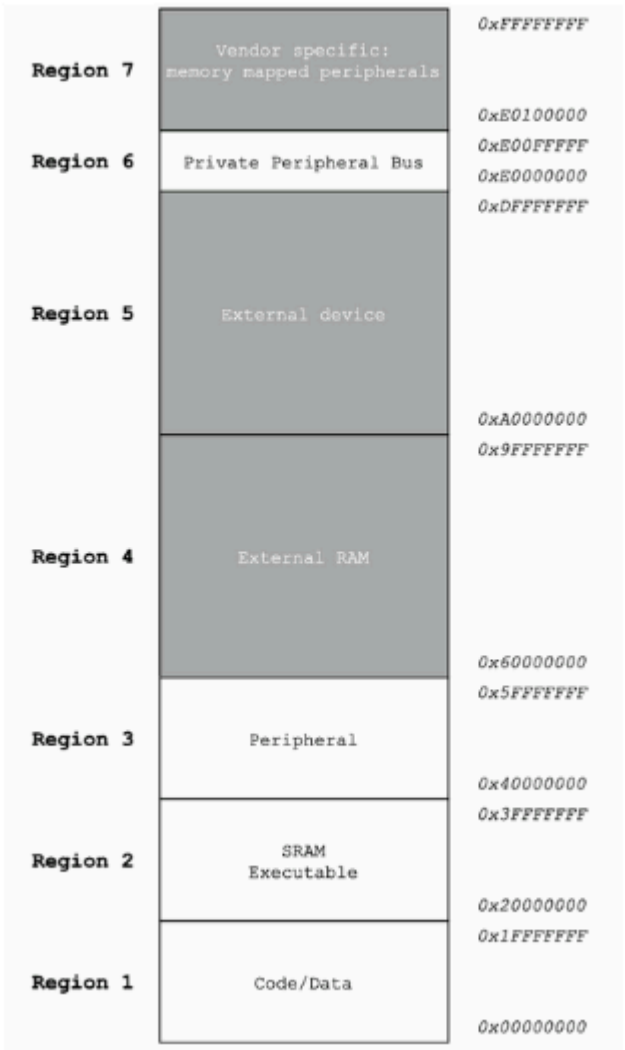
> Example: Writing to a GPIO pin must occur exactly in the order your code specifies. Any reordering may result in undefined behavior.

3. Strongly Ordered Memory

- **Usage**: For memory where **strict access sequencing** is required.
- **Features**:
  - No buffering, no caching.
  - **Access completes only when it reaches memory**—ensuring absolute ordering and visibility.
  - Every memory operation completes before the next begins.
- **Typical Use Case**: Synchronization areas between processors or DMA-accessed memory where strict consistency is required.

Why This Matters

- Using **normal memory** for peripherals can cause bugs due to speculative reads/writes or reordering.
- Using **strongly ordered** memory unnecessarily can hurt performance.
- Correct region typing ensures **safe interactions** between the processor and memory or peripherals while allowing for performance optimizations where possible.



Summary Table

Type	Cacheable	Bufferable	Speculative	Use Case
Normal	Yes	Yes	Yes	Code & RAM regions
Device	No	Yes	No	I/O peripherals
Strongly Ordered	No	No	No	Critical sync regions

Region Overlapping in ARM MPU

In ARM Cortex-M processors, the Memory Protection Unit (MPU) allows memory regions to **overlap**. When this happens, **priority determines which region's settings take effect**.

- Lower-numbered regions have lower priority.
- Higher-numbered regions override the access permissions of overlapping lower-numbered ones.

This mechanism allows flexible and hierarchical memory protection. For example, you can define a large region as read-write and then define a smaller overlapping region with stricter permissions (e.g., read-only or execute-never).

Why Use Overlapping?

- Protect critical sections inside a general memory region.
- Apply stricter access control without changing the base layout.
- Dynamically change access based on region priority.

Example Setup

Assume the following three regions:

- **Region 0:** `0x20000000` – `0x20004000` (16 KB)
- **Region 1:** `0x20001000` – `0x20005000` (16 KB)
- **Region 2:** `0x20002000` – `0x20004000` (8 KB)

Priority Order:

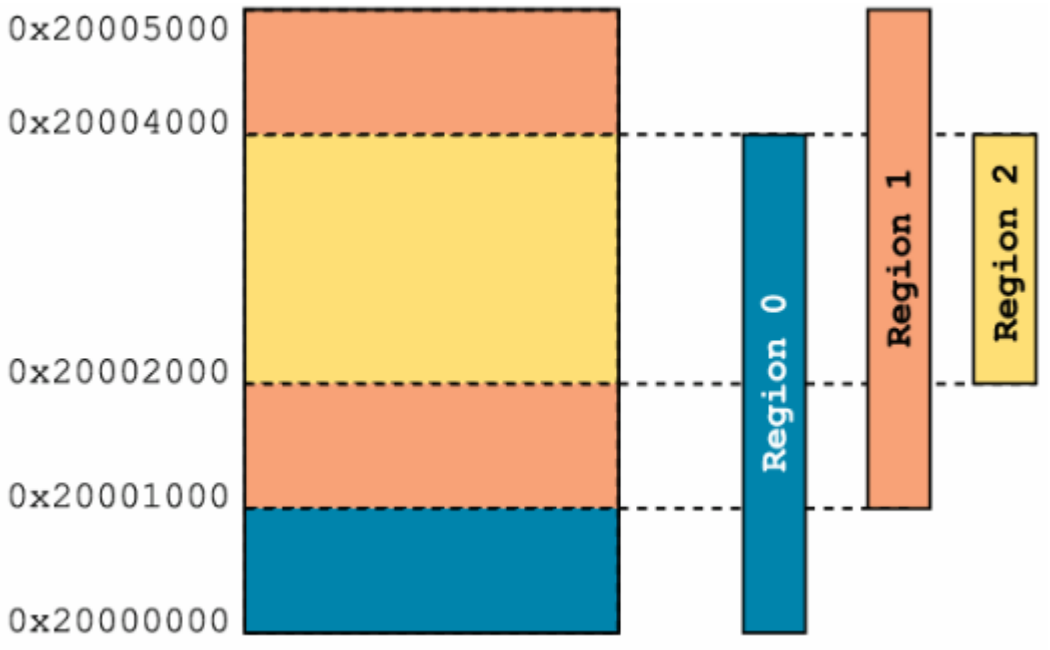
- Region 0: Lowest priority
- Region 1: Medium priority
- Region 2: Highest priority

Behavior:

- From `0x20000000` to `0x20001000`: only Region 0 applies.
- From `0x20001000` to `0x20002000`: Region 1 overrides Region 0.
- From `0x20002000` to `0x20004000`: Region 2 overrides both Region 0 and Region 1.
- From `0x20004000` to `0x20005000`: only Region 1 applies.

This setup could be used to:

- Allow general read-write access in Region 0,
- Make Region 1 read-only,
- Mark Region 2 as execute-never to block code execution in that sensitive sub-region.



SoC Memory Attributes

When configuring memory regions in a System on Chip (SoC), several attributes define how the processor and peripherals interact with that memory. Here's a brief overview of common terms:

- **Sharable Memory**
  Memory marked as sharable means it can be safely accessed by multiple bus masters or processors without causing data corruption. This is important in multi-core SoCs or systems with multiple DMA engines to maintain cache coherency and proper synchronization.
- **Execute Never (XN)**
  This attribute prevents code execution from the specified memory region. Marking memory as Execute Never is a security feature to avoid running code from data or peripheral regions, helping to mitigate certain attacks like code injection.
- **Bufferable**
  Bufferable memory allows write operations to be temporarily stored in a buffer before being written to memory. This can improve performance by reducing wait times, especially for burst or sequential writes. However, buffering can affect memory ordering and visibility, so it must be used carefully.
- **Readable**
  Indicates the memory region can be read from by the processor or bus master. Read access permissions ensure that the data stored can be fetched as needed.
- **Writeable**
  Indicates the memory region can be written to. Write permissions allow the processor or bus masters to modify the content of the memory region.

In-Order vs Out-of-Order Execution

Modern CPUs use pipelining and parallelism to speed up instruction execution. The way they **issue**, **execute**, and **retire** instructions gives rise to two major models:

In-Order Execution

- Instructions are **fetched, decoded, executed, and completed in the exact order** they appear in the program.

- Simple to implement, predictable behavior.
- But can be **inefficient** if a stall happens (e.g., waiting for data), because all subsequent instructions must wait.

Example:

```
1. LOAD R1, [100]
2. ADD R2, R1, R3
3. STORE [200], R2
```

If `LOAD` takes time (waiting for memory), the CPU must **pause** even if `ADD` or `STORE` could be done earlier.

Out-of-Order Execution (OoO)

- In Out-of-Order Execution, the CPU executes instructions **as soon as their operands are ready**, **regardless of their original program order**.
- However, results are always **committed in program order** to maintain correctness.
- This model is used to **maximize instruction-level parallelism (ILP)** and improve CPU throughput.
- Requires complex hardware mechanisms like:
  - **Register renaming** (to avoid false dependencies)
  - **Reservation stations** (buffer instructions waiting for operands)
  - **Reorder buffer** (to retire instructions in order)

Example:

```
1. LOAD R1, [100]      ; slow - waiting for memory
2. ADD R2, R4, R5      ; ready to execute
3. MUL R3, R6, R7      ; ready to execute
```

Here, even though `LOAD` is stalled, the CPU can execute `ADD` and `MUL` out of order, improving efficiency.

Feature	In-Order Execution	Out-of-Order Execution
Execution order	Strict program order	Executes instructions as soon as ready
Hardware complexity	Simple	Complex (register renaming, reservation stations, reorder buffer)
Performance	Can stall and waste cycles	Maximizes instruction-level parallelism (ILP)
Predictability	Easier to predict	More complex to verify and debug

---

MPU Region Configuration in Armv6-M and Armv7-M Architectures

In Armv6-M and Armv7-M architectures, the **Memory Protection Unit (MPU)** allows flexible configuration of memory regions to enforce access permissions and memory attributes for enhanced security and system stability. The configuration is mainly controlled via the **MPU_RASR** (Region Attribute and Size Register), which works alongside **MPU_RNR** (Region Number Register) to select and configure individual regions.

> **Note:** The MPU_RASR register is *not* present in Armv8-M; that architecture uses a different MPU configuration scheme.

Key Fields of the MPU_RASR Register

- **XN (Execute Never)**
  A 1-bit field that, when set, disables instruction fetches from the region, preventing code execution from that memory area.
- **AP (Access Permissions), TEX (Type Extension), C (Cacheable), B (Bufferable)**
  These fields collectively specify the memory access permissions and cache behavior:
  - **AP** defines read/write permissions for privileged and unprivileged accesses.
  - **TEX, C, B** configure the memory type and cache policy (e.g., strongly ordered, device, normal memory, cacheable, bufferable).
- **S (Shareable)**
  Indicates if the region is shareable, i.e., whether multiple bus masters or processors can safely share the region with cache coherency considerations.
- **SRD (Sub-Region Disable)**
  An 8-bit mask that allows disabling of up to eight equal-sized sub-regions within the configured region. Each bit corresponds to

one sub-region:

- The least significant bit disables the first sub-region.
- The most significant bit disables the eighth sub-region.
  This fine-grained control allows flexible exclusion of specific parts of a region.

- **SIZE**
  Specifies the size of the region as a power-of-two number of bytes (e.g., 32 bytes, 64 bytes, 128 bytes, etc.).
- **ENABLE**
  Enables or disables the region. When disabled, the region's attributes and permissions are ignored.

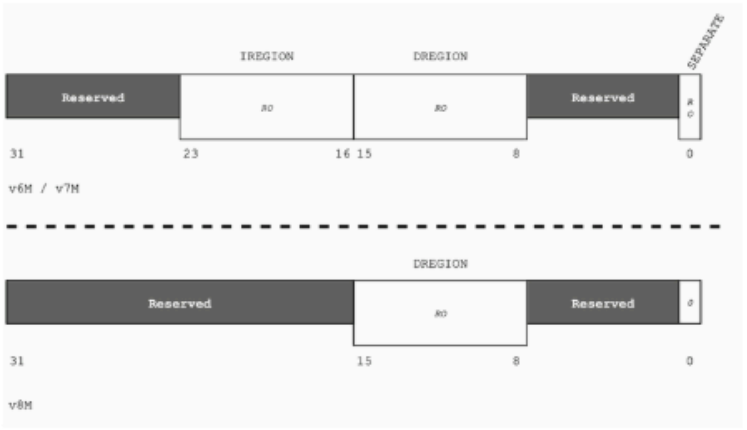


Figure 3.28 Fields of the MPU_TYPE register (above: Arm-v6-M & Arm-v7-M; below: Arm-v8-M)

Figure 3.29 MPU_CTRL register fields

Figure 3.30 Region number register (MPU_RNR)

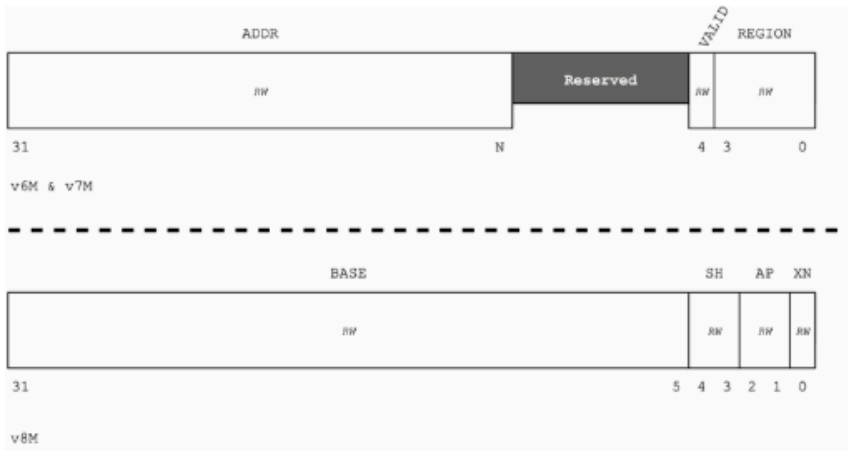


Figure 3.31 Fields of the region base address register (MPU_RBAR) (above: Arm-v6-M & Arm-v7-M; below: Arm-v8-M)

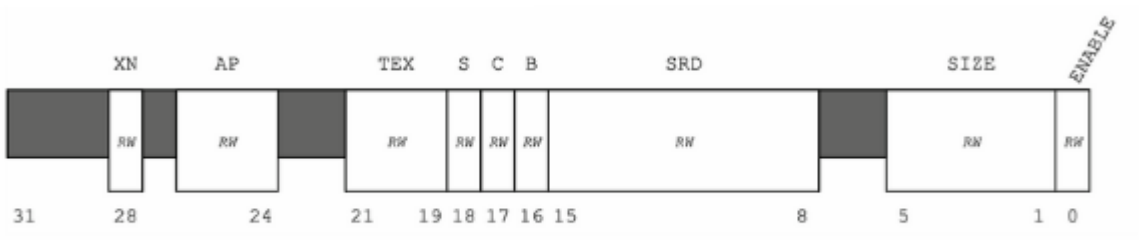


Figure 3.32 MPU region attributes and size register (MPU_RASR) fields (not in Arm-v8-M)

MPU Configuration Process

Configuring the Memory Protection Unit (MPU) involves several clear steps to define and enable memory regions with specific access permissions and attributes.

1. **Disable the MPU**
   Before making any changes, the MPU must be disabled to safely update region settings without causing conflicts.
2. **Configure Each Memory Region**
   For every region you want to protect or configure, specify:
   - **Base Address:** The starting memory address of the region.
   - **Size:** How large the region is (must be a power-of-two size).
   - **Attributes:** Access permissions (read/write/execute), cache settings, shareability, etc.
3. **Enable Each Region**
   Use the **MPU_RASR** register (or equivalent CMSIS functions) to enable the configured region with its attributes.
4. **Repeat for All Regions**
   Repeat steps 2 and 3 until all desired regions are configured.
5. **Enable the MPU**
   Once all regions are set up, enable the MPU globally using the **MPU_CTRL** register (or CMSIS API). This activates protection based on your configurations.

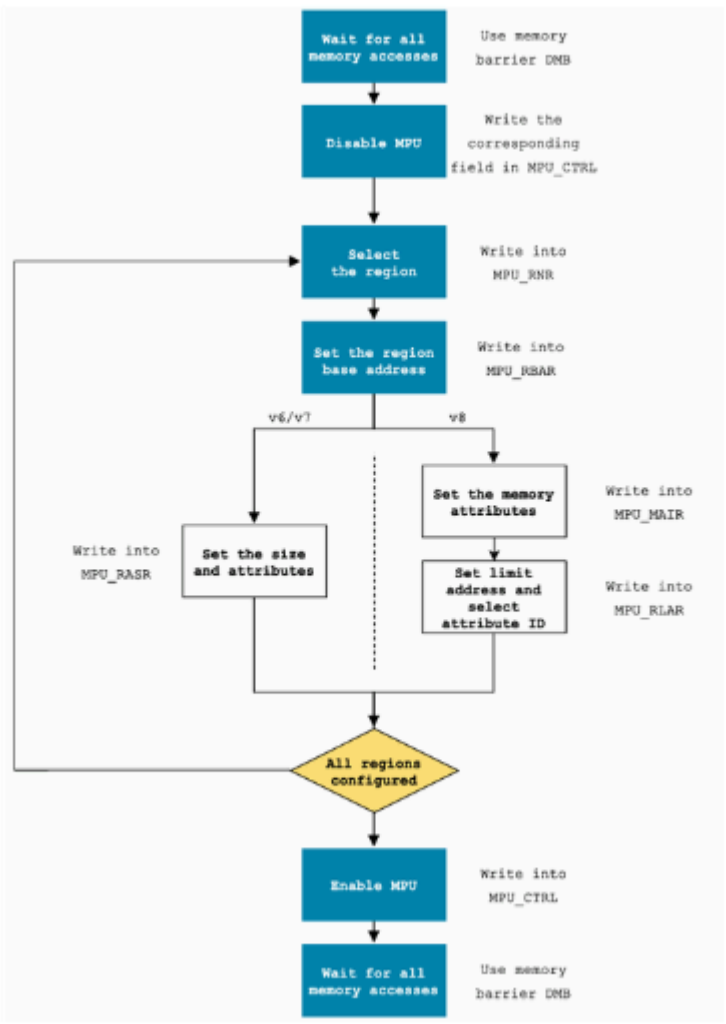
Using CMSIS APIs

The ARM CMSIS (Cortex Microcontroller Software Interface Standard) provides helper functions to simplify MPU configuration, such as:

- `ARM_MPU_SetRegion()` : Configure a specific MPU region easily.
- `ARM_MPU_Enable()` : Enable the MPU with proper settings.

These functions handle register accesses and make the process less error-prone.

> **Important Note:**
>
> - **Plan Your Regions Carefully:**
>   Avoid overlapping or leaving gaps between regions, as these can cause unexpected access behavior or security holes.
> - **Static vs Dynamic Configuration:**
>   Most embedded systems use static MPU configurations set once during startup. Some advanced systems may update MPU settings dynamically at runtime for changing protection needs.



MPU Diagnostics and Fault Handling

When the MPU detects an illegal memory access, it triggers a **fault exception** that the processor handles to protect the system and help debug the issue.

How Faults Are Reported

- **MMFSR (Memory Management Fault Status Register):**
  Indicates the type of fault that occurred (e.g., instruction access violation, data access violation).
- **MMFAR (Memory Management Fault Address Register):**
  Holds the memory address that caused the fault, helping to pinpoint the exact problematic access.

Using Debuggers and IDEs

Debug tools like **GDB** or IDEs such as **Code Composer Studio (CCS)** can read these registers, allowing developers to trace and diagnose faults effectively.

Common Causes of MPU Faults

1. **Instruction Access Violation**
   Occurs when the CPU tries to fetch and execute an instruction from a memory region marked as non-executable (e.g., execute-never region).
   Example: Attempting to run code from a data-only memory area.

2. **Data Access Violation**
   Happens when a program tries to read or write to a memory region without proper permissions.
   Example: Writing to read-only flash memory or accessing restricted RAM.
3. **Fault on Unstacking (Return from Exception)**
   Triggered if the processor tries to restore context from an invalid or protected memory area during exception return.
4. **Fault on Stacking (Exception Entry)**
   Occurs if storing context onto the stack during exception entry targets a forbidden memory region.
5. **Fault During Floating-Point Lazy State Preservation**
   On Cortex-M processors with an FPU (e.g., M4, M7, M23, M33), the MPU can fault if saving the floating-point state lazily attempts to access illegal memory.

Why Fault Handling Matters

- **Detect Bugs:** Faults highlight invalid memory accesses or protection violations early, preventing undefined behavior.
- **Debugging Aid:** Fault registers give precise info to help fix problems quickly.
- **System Stability:** Proper exception handlers can log faults, recover gracefully, or reset the system to avoid crashes.

Example Scenario

Suppose a program tries to execute code from a peripheral memory region accidentally marked as **execute-never**. The MPU will trigger an **instruction access violation fault**, causing the processor to enter the memory management fault handler. The MMFSR register indicates the fault type, and the MMFAR register shows the exact address. Using a debugger, a developer can identify this mistake and correct the memory attributes or code placement.

Dual Core Lockstep (DCLS) Overview

The **TEALLOCKSTEP DCLS** processor subsystem is a fault-tolerant architecture designed to improve system reliability by running two identical processor cores in lockstep. This means both cores execute the exact same instructions simultaneously, and their outputs are continuously compared to detect faults or errors.

Components of the DCLS Subsystem

- **TEAL Cortex-M33 Processors:**
  Two identical Cortex-M33 cores run in parallel without any modification, executing the same code and sharing the same clock.
- **DCLS Controller (`tealdcctl`):**
  Manages synchronization between the two cores and controls resynchronization flip-flops that help align timing between outputs.
- **Comparator Logic (`tealdccm`):**
  Compares the outputs from both cores after they have been resynchronized to detect any discrepancies indicating faults.
- **Redundant Core Input Delay Flip-Flops (`tealdcreg`):**
  These delay elements ensure input signals are properly timed for both cores to maintain synchronization.

How It Works

- Both cores receive the **same external clock (CLKIN)** and execute instructions in perfect sync.
- Critical output signals from both cores pass through **multi-stage resynchronization flip-flops** before reaching the comparator. This ensures that timing differences do not cause false error detection.
- The **comparator logic continuously checks** if outputs from both cores match.
- If a mismatch is detected, it signals a fault, indicating one core might be malfunctioning.
- External reset signals (like **nPORESET** and **nSYSRESET**) must be synchronized to the clock to ensure both cores reset simultaneously.
- Both cores must have the **RAR parameter set to 1**, enabling lockstep operation mode.

An easy example

Imagine two runners running side-by-side in a race, expected to step in perfect sync. If one runner stumbles or moves out of sync, a judge immediately notices the difference. Similarly, in dual core lockstep, if one processor core produces a different output than the other at any point, the system detects this mismatch as a fault.

Benefits

- **Improved Fault Detection:** Errors like hardware failures or soft errors are quickly caught by comparing both cores' outputs.

- **High Reliability:** Ideal for safety-critical applications (e.g., automotive, aerospace) where system failure is not an option.
- **Minimal Modification:** The Cortex-M33 cores run unmodified, simplifying integration.

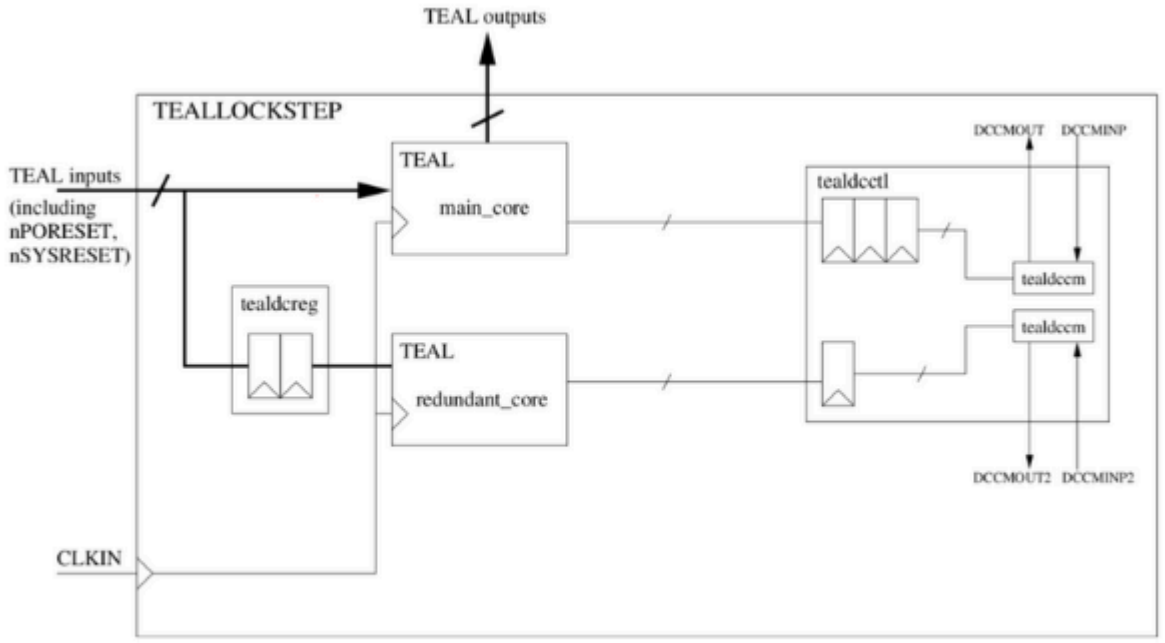


Figure 6-1 Cortex-M33-based DCLS processor

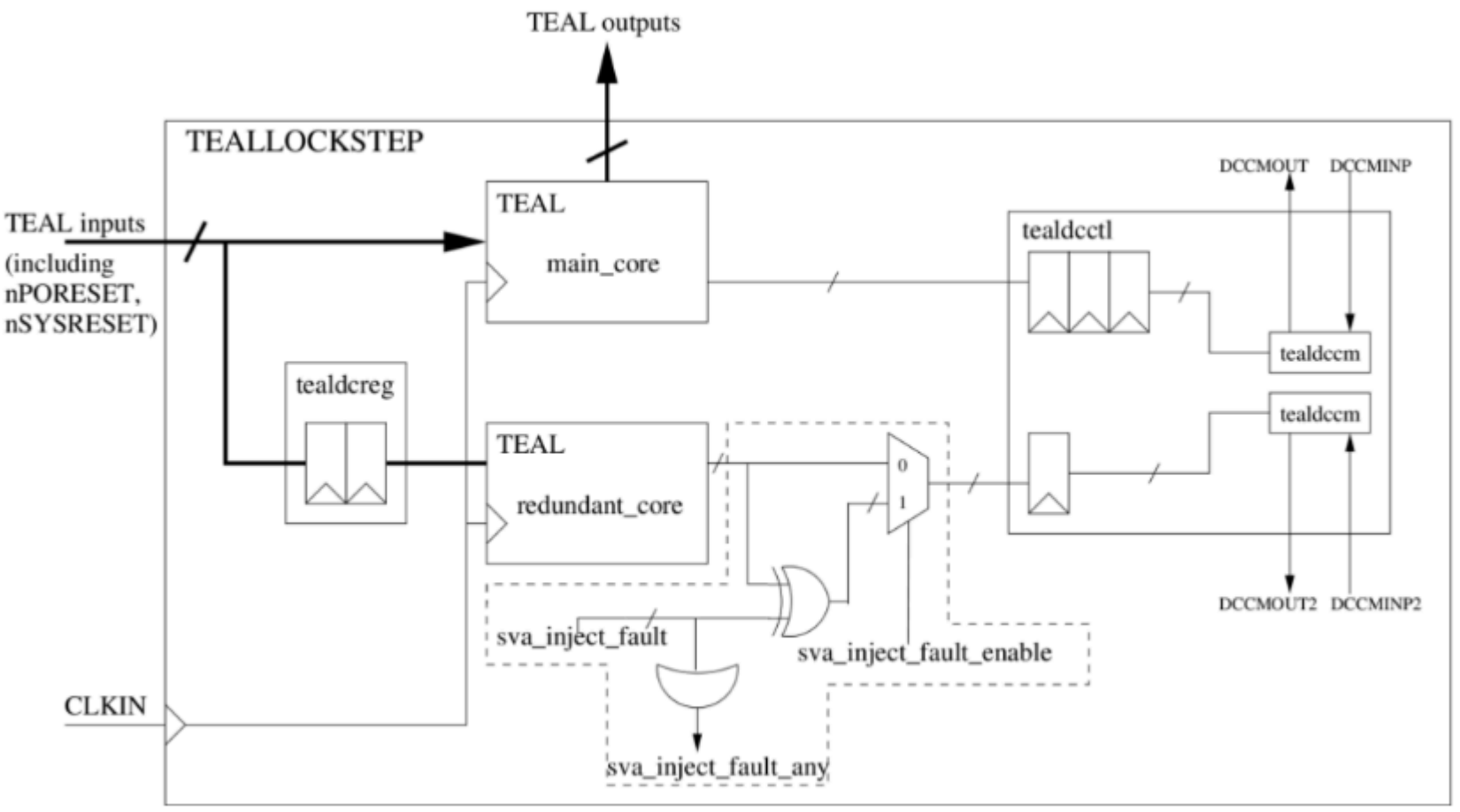
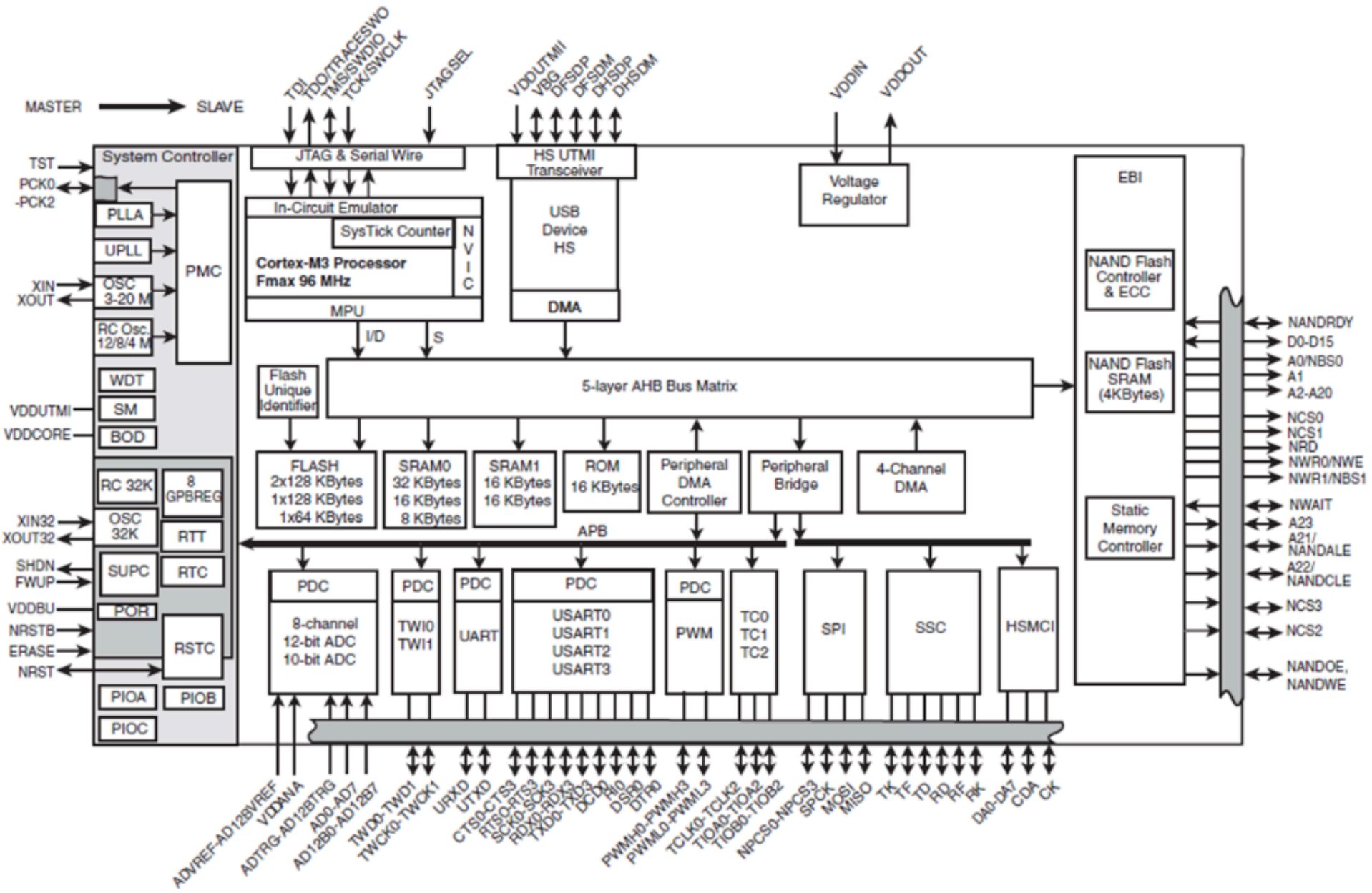


Figure 6-2 DCLS processor with fault injection logic

Day 3(Part 1) - CAN

> Part 1 contains every detailed information about CAN.



Introduction to CAN (Controller Area Network)

- Developed by Robert Bosch GmbH in 1986 for in-vehicle communication
- Standardized as ISO 11898 in 1993
- Multi-master, message-based protocol used in automotive and industrial systems

> There is no concept of **ADDRESSES** in CAN. It is **identifier-based**, not address-based.
> A **CAN frame** is a structured packet of data that includes not just the payload (data), but also addressing, control, and error-detection fields. It’s how devices on a CAN network communicate reliably and in real time.

This means:

- There’s no "source" or "destination" address in the message
- The **Frame ID** conveys the meaning or type of data (e.g., engine temperature, vehicle speed), not the identity of sender/receiver
- Nodes **don’t know** which node sent the message — only what kind of message it is.

```
Frame ID = 0x100 → Engine temperature
Frame ID = 0x200 → Vehicle speed
Frame ID = 0x300 → Airbag status

All nodes receive these frames, but only nodes that care about, say, speed (0x200) will process it.
```

Evolution of CAN

Classic CAN

- Original protocol supporting up to 8-byte payloads
- Widely used in Electronic Control Units (ECUs)

CAN FD (Flexible Data-rate)

- Introduced in 2012 to extend capabilities of Classic CAN

- Supports two bit rates: arbitration phase and data phase
- Increases payload size from 8 bytes to 64 bytes
- Defined in ISO 11898-1:2015

CAN XL

- Development began in 2018
- Initiated by Volkswagen (VW) for larger frame sizes and higher bandwidth
- Supports data rates up to 10 Mbit/s
- Payload capacity up to 2048 bytes
- Allows tunneling of Ethernet frames within CAN XL

Communication Model

- CAN is broadcast-based; all nodes see all frames
- Messages identified by a Frame ID, not by destination address
- Frame ID also acts as a priority: lower ID = higher priority

Key Features

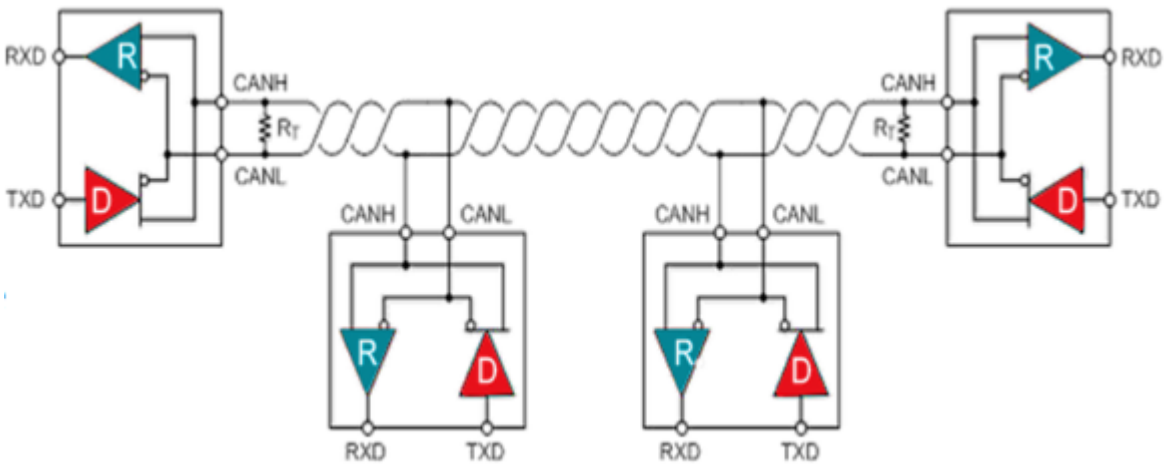
- Robust error handling: CRC checks, ACK bits, error frames
- Arbitration ensures deterministic communication
- Lightweight wiring and protocol simplicity
- Widely adopted in automotive, industrial, and medical fields

Small Example

```
Node A sends:
  Frame ID = 0x100
  Data = Engine temperature = 90°C

Node B sends:
  Frame ID = 0x300
  Data = Window position = OPEN

Since 0x100 < 0x300, Node A's message gets higher priority on the bus.
```



CAN Transceiver & Bit Logic

Role of the Transceiver

- The **CAN transceiver** sits between the CAN controller (inside a microcontroller) and the physical CAN bus.
- It converts **TTL/CMOS logic levels** (e.g., 0V and 5V) from the microcontroller to the **differential signals** used on the CAN bus.
- It also receives differential signals from the bus and translates them back to logic levels.

CAN Bus Physical Layer

- The CAN bus uses **two wires**: `CANH` (CAN High) and `CANL` (CAN Low)
- These lines carry a **differential signal**, which helps cancel out external **electromagnetic noise**.

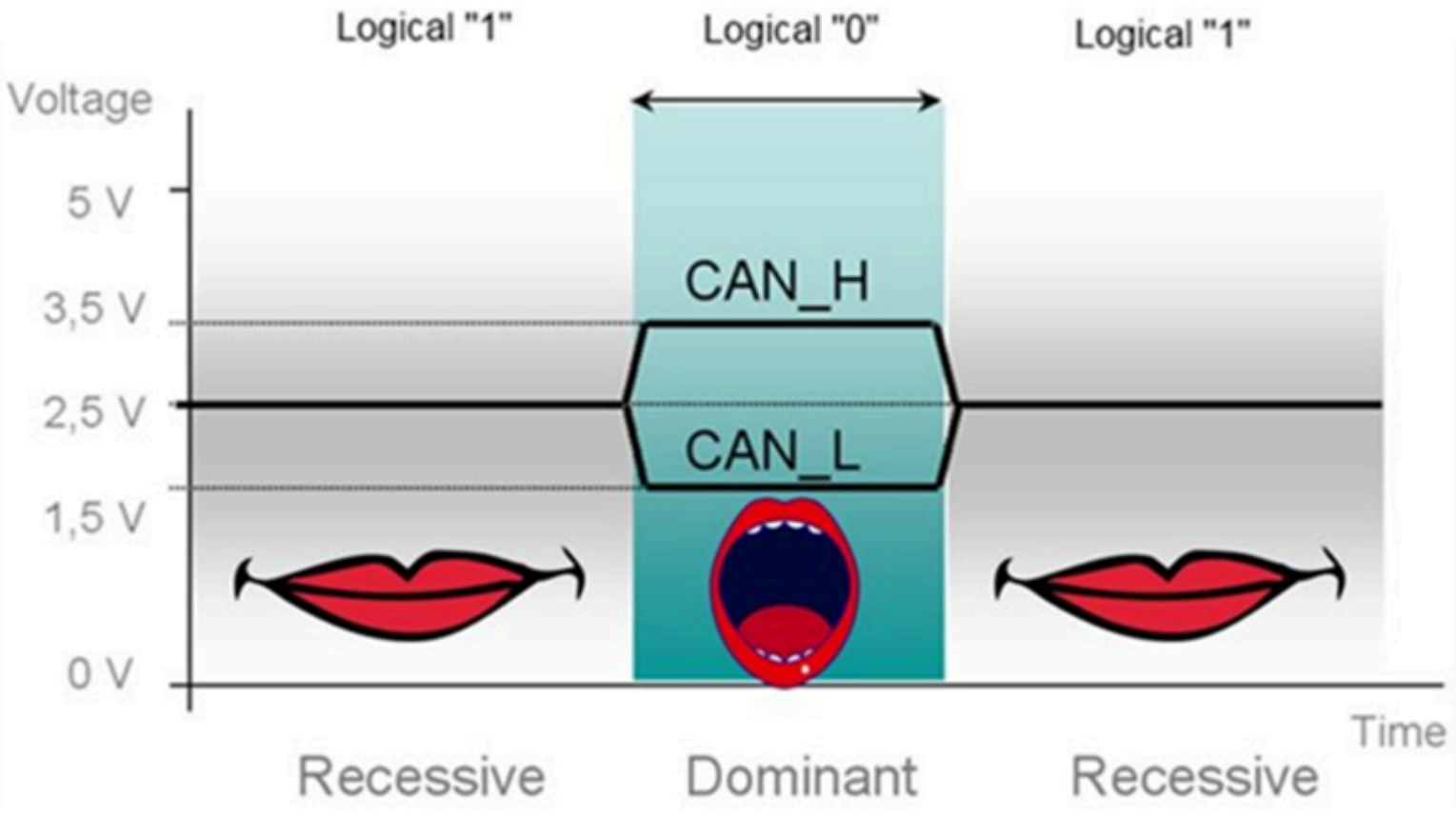
Differential Signaling Logic

Logical Bit	CANH Voltage	CANL Voltage	Bus State
Dominant (0)	≈ 3.5V	≈ 1.5V	Logic 0
Recessive (1)	≈ 2.5V	≈ 2.5V	Logic 1

- **Dominant bit (0):** CANH > CANL → drives the bus actively
- **Recessive bit (1):** CANH ≈ CANL → no differential voltage

Noise Immunity

- Since the CAN bus uses differential signaling:
  - If the same noise affects both CANH and CANL, the difference between them remains unchanged
  - This enables **reliable communication in electrically noisy environments**



CAN - Arbitration

What is Arbitration?

- **Arbitration** is the process by which the CAN protocol resolves conflicts when **multiple nodes attempt to transmit at the same time**.
- It ensures that the **highest priority message** (based on **Frame ID**) gets transmitted without delay, while others back off automatically.

How It Works

- Arbitration happens during the **Identifier field** of the CAN frame.
- CAN uses **bitwise arbitration**:
  - All nodes monitor the bus while transmitting.
  - The bus uses a **wired-AND logic**:
    - **Dominant bit (0)** wins over **recessive bit (1)**
- As soon as a node sees a bit on the bus that doesn't match what it's sending (i.e., it tries to send recessive but sees dominant), it **stops transmitting**.

Priority Rule

- **Lower the Identifier (Frame ID), higher the priority**
- This enables time-critical messages (e.g., braking or engine speed) to take precedence over less critical ones (e.g., air conditioning status)

Small Example

Each node attempts to transmit a message with the following 7-bit identifier:

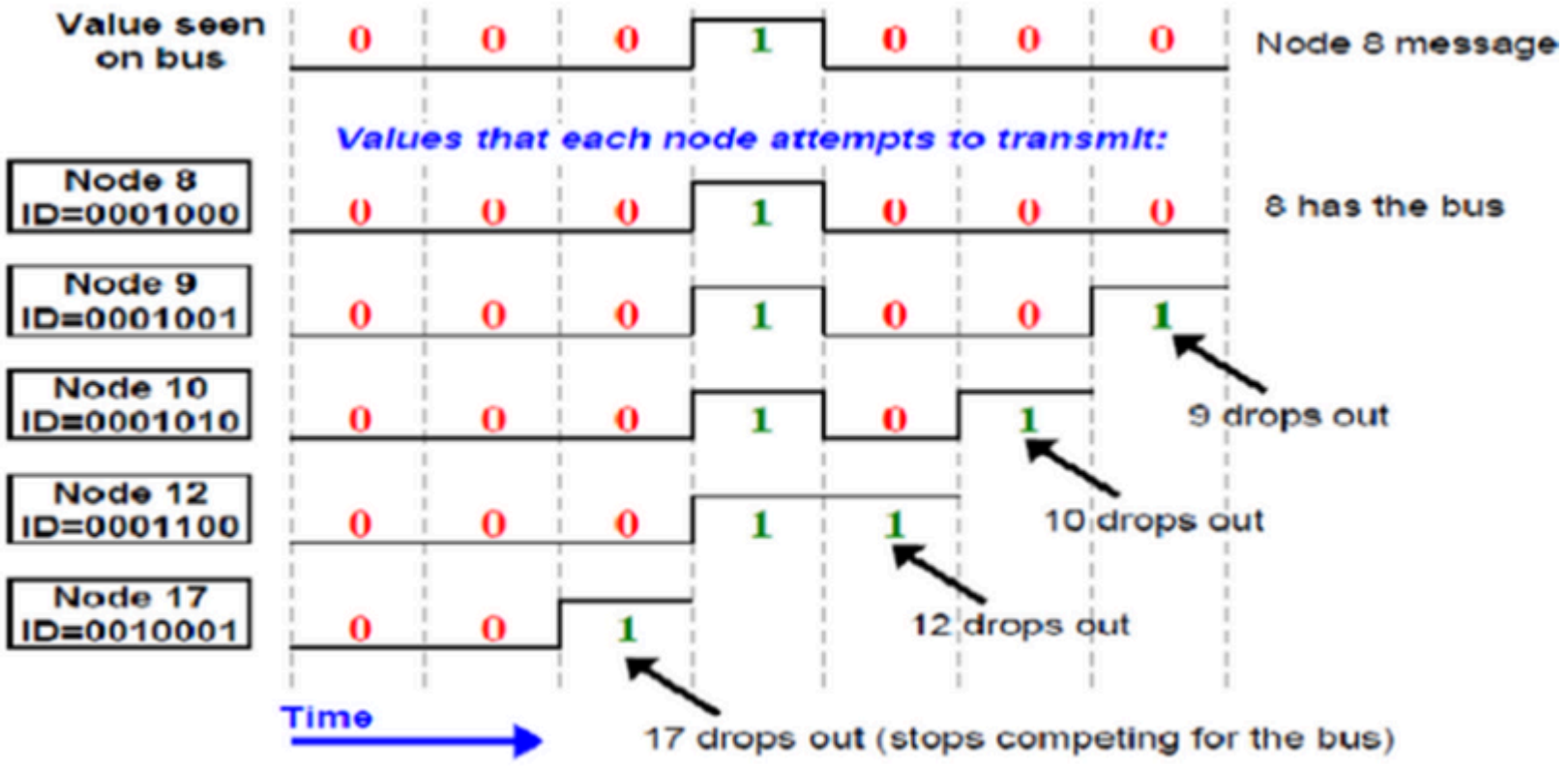
- Node 8 → `0001000`
- Node 9 → `0001001`
- Node 10 → `0001010`
- Node 12 → `0001100`
- Node 17 → `0010001`

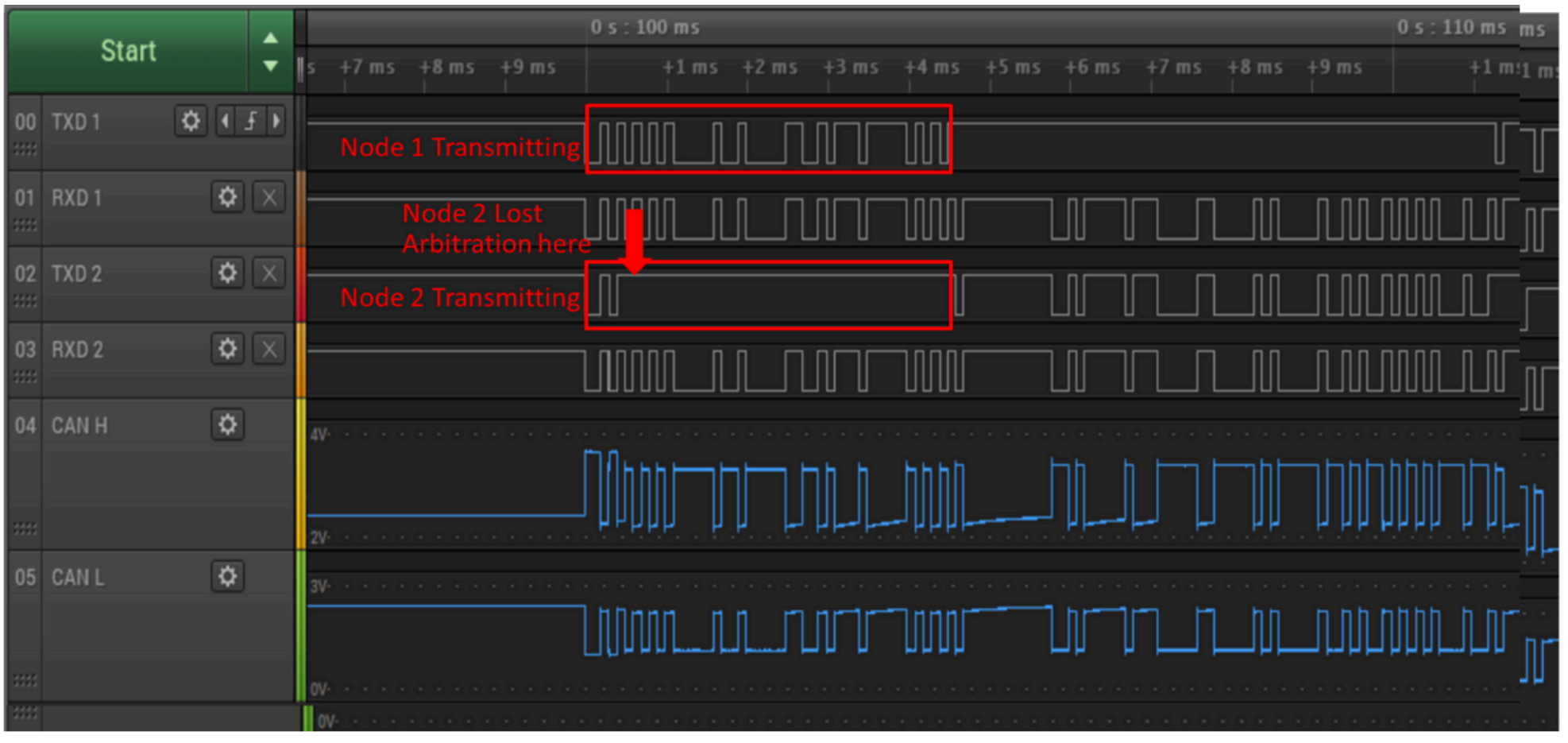
All nodes start transmitting at the same time.

Arbitration Process (Bit-by-Bit)

At each bit position, nodes compare the bit they're sending with the value seen on the bus:

- **Bit 1 to 4**: All nodes send `0`, bus shows `0`. All nodes stay in arbitration.
- **Bit 5**: Node 17 sends `1`, others send `0`. Bus shows `0`.
  → Node 17 loses arbitration and stops transmitting.
- **Bit 6**: Node 12 sends `1`, others send `0`. Bus shows `0`.
  → Node 12 drops out.
- **Bit 7**: Node 10 sends `1`, others send `0`. Bus shows `0`.
  → Node 10 drops out.
- **Bit 8**: Node 9 sends `1`, Node 8 sends `0`. Bus shows `0`.
  → Node 9 drops out.
- Node 8 is the only remaining transmitter.
- It wins arbitration and gains control of the bus.





CAN 2.0 Frame Types

The CAN 2.0 protocol defines **four types of frames** that can be transmitted over the bus:

1. Data Frame

- **Purpose:** Carries actual data from a sender to receivers
- **Structure:** Contains the Identifier, Control field, Data field (up to 8 bytes), CRC, ACK, and EOF
- **Most commonly used frame**

2. Remote Frame (Remote Transmission Request - RTR)

- **Purpose:** Requests data from another node
- **Difference from Data Frame:**
  - Has **no data field**
  - The **RTR bit is set to 1**
- The node with the corresponding data responds with a **Data Frame**

3. Error Frame

- **Purpose:** Transmitted by any node that detects an error (e.g., CRC mismatch, bit error)
- **Structure:**

- Consists of an **Error Flag** (6 or more dominant bits) and an **Error Delimiter**
- Alerts all nodes to discard the current frame and prepare for retransmission

4. Overload Frame

- **Purpose:** Injects a delay between frames when a node is not ready to process the next message
- **Structure:**
  - Similar to an Error Frame: Overload Flag + Overload Delimiter
- Sent by a node that needs more time before receiving the next frame

Frame Type	Purpose	Contains Data?	Special Bits
Data Frame	Transmit data	Yes	RTR = 0
Remote Frame	Request data from another node	No	RTR = 1
Error Frame	Signal an error to all nodes	No	Error Flag
Overload Frame	Request delay before next message	No	Overload Flag

---

CAN 2.0A – Data Frame Structure (Standard Frame Format)

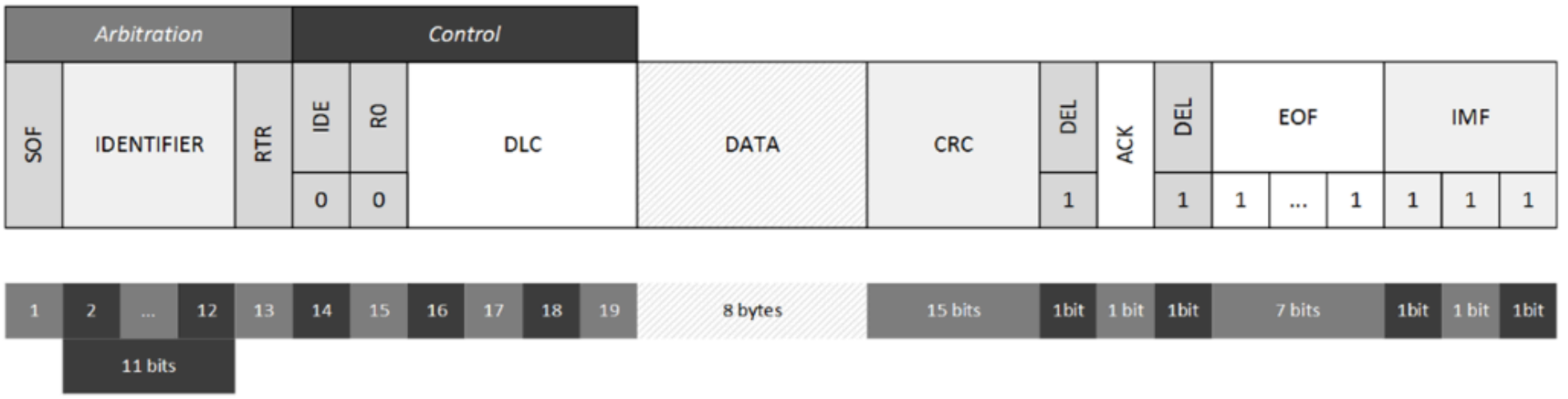
The **CAN 2.0A data frame** is the most common message format used in Controller Area Networks. It follows a structured layout for reliable communication between ECUs (Electronic Control Units) in real-time systems like automotive networks.

CAN 2.0A Data Frame Format (Field-by-Field)

Field	Bits	Description
SOF	1	**Start of Frame** – Always dominant (`0`). Indicates the start of a frame.
Identifier	11 (2–12)	Unique message ID used for **prioritization and arbitration**. Lower value = higher priority.
RTR	1 (13)	**Remote Transmission Request** – `0` for data frame, `1` for remote frame.
IDE	1 (14)	**Identifier Extension** – `0` for standard (11-bit) frame.
r0	1 (15)	Reserved bit – must be dominant (`0`).
DLC	4 (16–19)	**Data Length Code** – Number of data bytes (0 to 8).
Data	0–64	Payload – Carries up to 8 bytes (0–64 bits) of application data.
CRC	15	**Cyclic Redundancy Check** – Used for error detection.
CRC Delimiter	1	Always recessive (`1`). Separates CRC field from ACK.
ACK Slot	1	Receiver sets dominant (`0`) if frame was received without error.
ACK Delimiter	1	Always recessive (`1`). Follows the ACK slot.
EOF	7	**End of Frame** – All recessive (`1`). Marks the end of the frame.
IFS	3	**Inter-Frame Space** – Recessive idle bits before next frame begins.

Additional Notes on the Data Frame

- **Arbitration** happens using the **identifier + RTR bit**. Lower identifier values have higher priority.
- **Control field** includes IDE, r0, and DLC – all critical to define the frame type and data length.
- **Data field** is optional and may contain 0–8 bytes depending on DLC.
- **CRC** ensures data integrity. The sender computes this and receivers validate it.
- **ACK field** allows receivers to acknowledge correct reception – if no dominant bit is seen here, the sender knows there was an error.
- **EOF and IFS** ensure clear separation between successive frames and give nodes time to process.



CAN 2.0B – Extended Data Frame Structure

The **CAN 2.0B extended data frame** supports a **29-bit identifier**, allowing more unique message types than the standard 11-bit format in CAN 2.0A. It includes additional fields for extended identification and maintains backward compatibility with standard CAN nodes.

Key Differences in Extended Frame Format

In the extended format:

- The **RTR bit** is replaced by **SRR (Substitute Remote Request)**, which is always recessive (`1`).
- The **IDE bit** is always recessive (`1`) to indicate the presence of an extended identifier.
- Arbitration uses both the **base 11-bit identifier** and the **18-bit identifier extension**.
- Standard frames (with the same first 11 bits) have higher priority due to dominance arbitration.
- Some additional reserved bits are included to maintain compatibility.

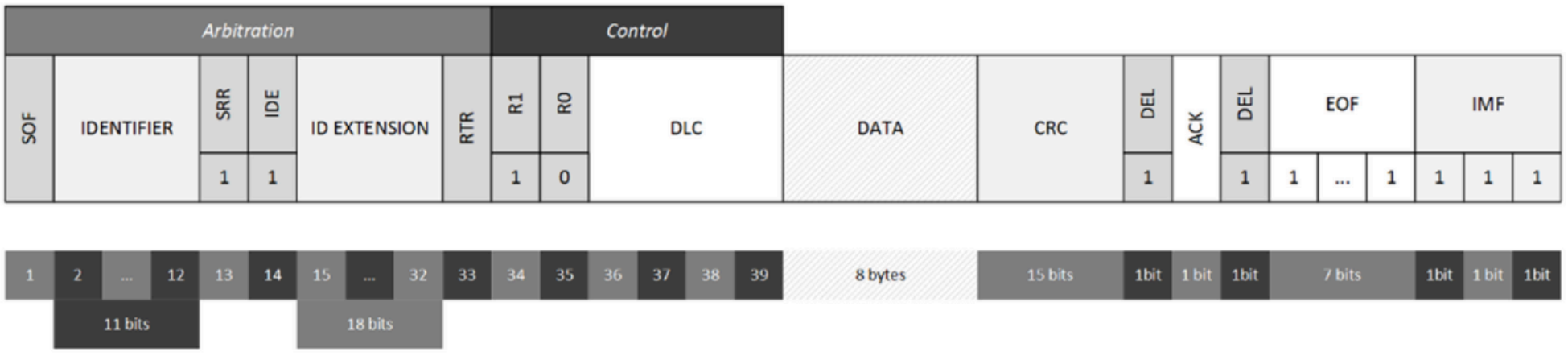
Additional Fields in Extended Frame (Not in Standard Frame)

Field	Bits	Description
SRR	1 (Bit 13)	**Substitute Remote Request** – Always recessive. Used instead of RTR in extended frames.
IDE	1 (Bit 14)	**Identifier Extension** – Always recessive (`1`) to signal an extended frame.
ID Extension	18 (15–32)	Additional 18 bits that, with the 11-bit base ID, form the 29-bit extended identifier. Used for further arbitration among extended frames.
RTR	1 (Bit 33)	Remote Transmission Request – Recessive in data frames. Present after full 29-bit ID.
r1	1 (Bit 34)	Reserved bit – Must be dominant (`0`). Reserved for future protocol extensions.
r0	1 (Bit 35)	Reserved bit – Must be dominant (`0`). Also used in CAN 2.0A.

TLDR

- Arbitration still works bit-by-bit using **dominant (0)** vs **recessive (1)**, but now across **29 bits**.
- A **standard frame** with the same first 11 bits as an extended frame **will win arbitration**, since it transmits fewer bits (shorter frames dominate).
- CAN 2.0B controllers can operate in:
  - **Passive mode**: Ignore extended frames.
  - **Active mode**: Send and receive extended frames.

This format offers higher message ID capacity while maintaining compatibility with existing CAN 2.0A devices.



CAN 2.0A – Remote Frame (RTR Frame)

The **Remote Frame** in CAN 2.0A is used by a node to request data from another node without sending a payload itself. It shares the same structure as a **Data Frame**, with one key difference: the **RTR bit is recessive (1)** to indicate that this is a **request**, not a data transmission.

Key Characteristics of a Remote Frame

- **RTR Bit**: Always **recessive (`1`)** to signal a remote frame.
- **Identifier**: Matches the identifier of the corresponding data frame it's requesting.
- **Data Length Code (DLC)**: Indicates how many bytes are expected in the response, but **no data is included** in the remote frame itself.
- **Data Field**: Always **empty (0 bytes)**.
- **Priority Rule**: A data frame with the same identifier will **win arbitration** over a remote frame. This ensures that if both are sent simultaneously, the actual data is transmitted.

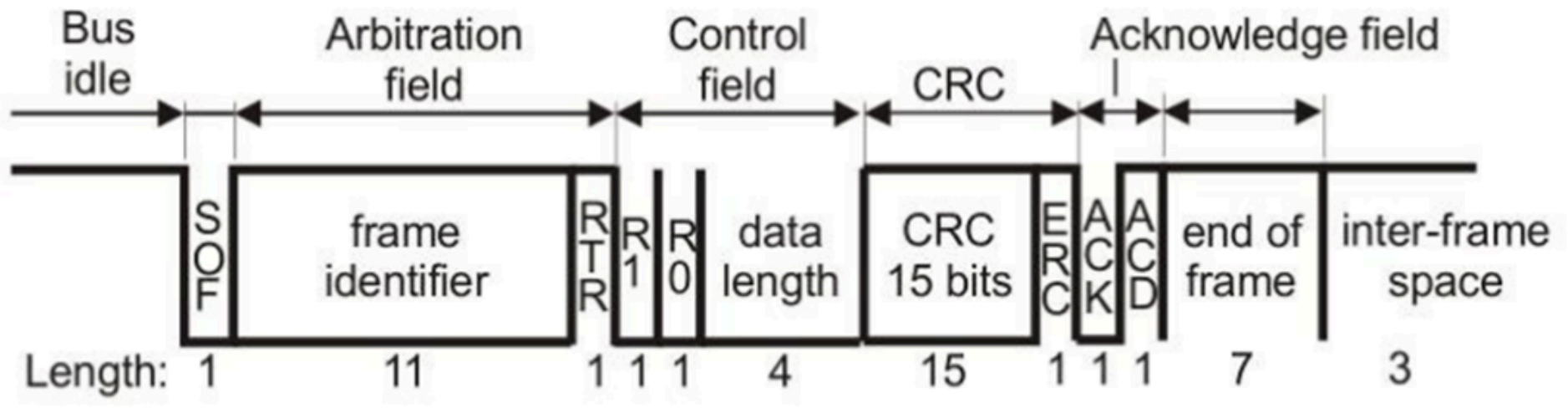
Purpose and Use

Remote frames are useful in **polled communication setups**, where a central node queries other nodes for their status or data. The responder will reply with a **data frame** that shares the same identifier.

TLDR

Remote frames allow nodes to **request specific data** without broadcasting payloads. They maintain the same format and identifier structure as data frames but serve purely to initiate a response.

- **No data payload**
- **Recessive RTR bit**
- **Same ID as the requested data**
- **Lower priority than the matching data frame**



CAN – Overload Frame

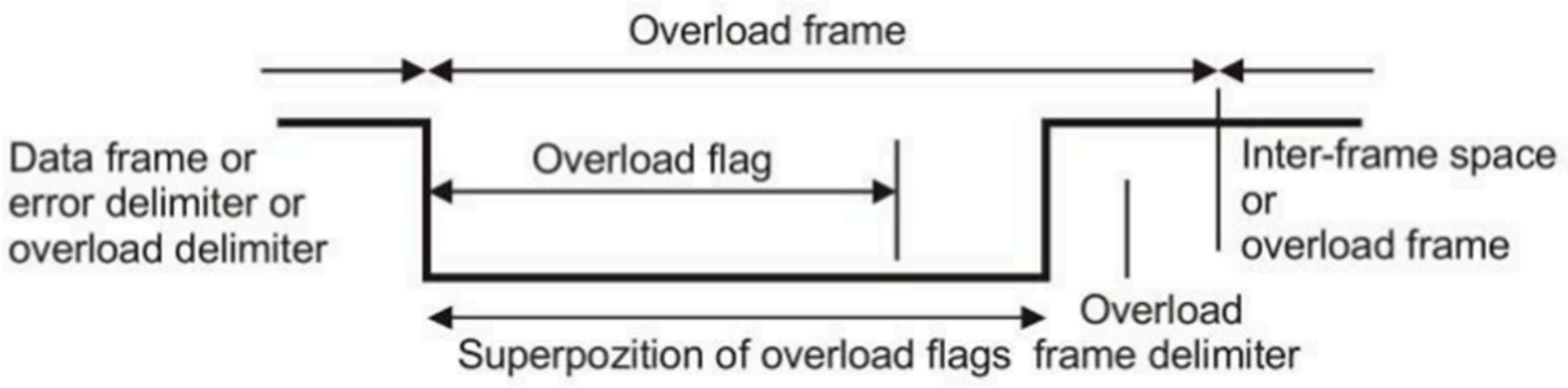
The **Overload Frame** in CAN is a special type of frame transmitted by a receiver to **introduce a delay** before the next data or remote frame is sent. It is not used for error signaling, but rather as a flow control mechanism.

Key Characteristics

- **Purpose**:
  - To **postpone** the transmission of the next frame when the receiver is **not ready** to handle more data (e.g., due to processing delays).
- **Trigger Conditions**:
  - Detection of a **dominant bit** in the **last bit of the End Of Frame (EOF)** field.
  - Detection of a dominant bit in the **first two bits of the Inter-Frame Space (IFS)**.
- **Structure**:
  - Consists of **two dominant bits**, followed by **six recessive bits** (similar to an error frame, but with different meaning and usage).
- **Does NOT indicate an error**:
  - Overload frames are **not error frames**.
  - They **do not cause retransmission** of the previous frame.

TLDR

- Sent by a **receiver**, not the transmitter.
- Used to **delay** the next frame for **timing or processing reasons**.
- Indicates **temporary unavailability**, not a communication fault.
- Helps maintain **system stability** in high-traffic or slower-processing nodes.



CAN – Error Frame

The **Error Frame** in CAN is used to signal the detection of a fault in a transmitted or received message. It allows nodes to quickly notify others of an issue so that corrective action (e.g., retransmission) can occur.

Structure of an Error Frame

An error frame consists of two main parts:

1. Error Flag (6 bits)

Indicates that an error has been detected. There are two types:

- **Active Error Flag**:
  - Bit Pattern: `000000` (all dominant)
  - Sent by a node in **error-active** state.
  - This non-stuffed dominant sequence **intentionally violates** CAN rules to **force detection** by all other nodes.
  - Triggers immediate reaction on the bus.
- **Passive Error Flag**:
  - Bit Pattern: `111111` (all recessive)
  - Sent by a node in **error-passive** state.
  - It does **not disrupt** the bus because it blends into idle time.
  - Used when a node has exceeded the error threshold but still needs to report issues without disturbing others.

2. Error Delimiter (8 bits)

- Always `11111111` (all recessive).

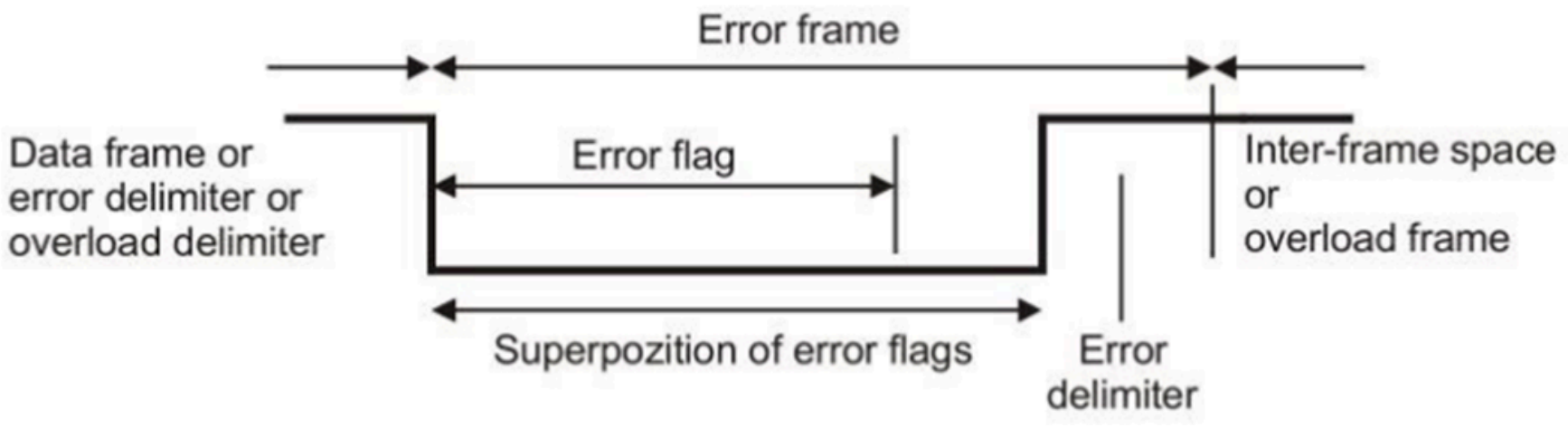
- Signals the end of the error frame.
- Allows the bus to return to idle and prepare for recovery.

Key Behavior

- **Transmission is immediately stopped** when an error is flagged.
- **The message will be retransmitted** after bus recovery.
- Error frames are part of CAN's **robust fault handling** to ensure data integrity across unreliable channels.

TLDR

- **Active Error Flag**: Dominant, disruptive, triggers bus-wide error response.
- **Passive Error Flag**: Recessive, silent, does not disrupt others.
- **Error Delimiter**: Recessive, always 8 bits, ends the error frame.
- Designed to **detect, broadcast, and recover from faults** automatically.



CAN Error Detection Mechanisms

The CAN protocol uses five mechanisms simultaneously to detect errors:

1. Monitoring

- The transmitter monitors the bus while transmitting.
- If it detects a different bit value on the bus than what it is sending (except during ACK), it identifies an error and may send an error frame.

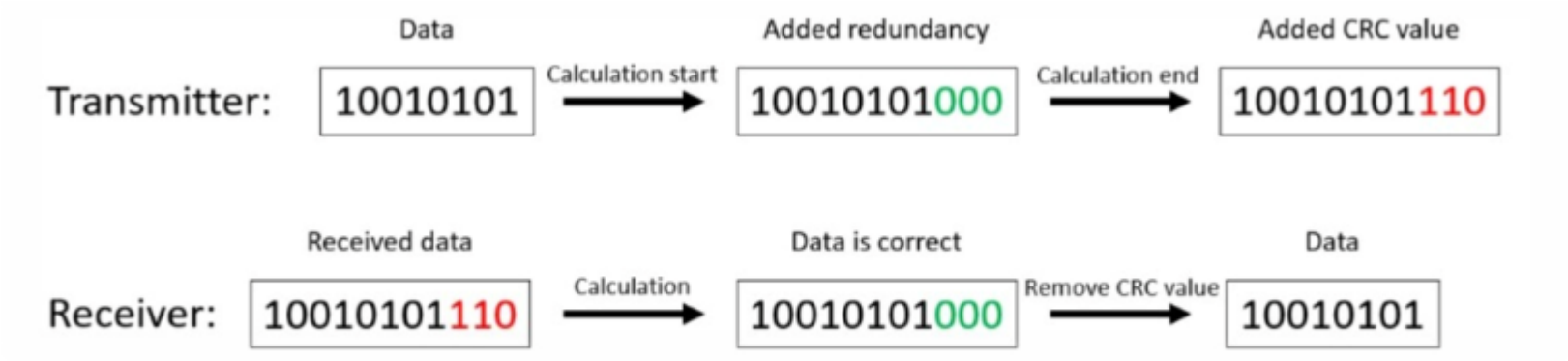
Example:

```
Transmitter sends: 1 (recessive)
Bus shows: 0 (dominant)
→ Arbitration lost or error detected
```

2. CRC (Cyclic Redundancy Check)

- A CRC is computed on the data and transmitted with the frame.
- The receiver recalculates the CRC.
- If there's a mismatch, the receiver sends an error frame.

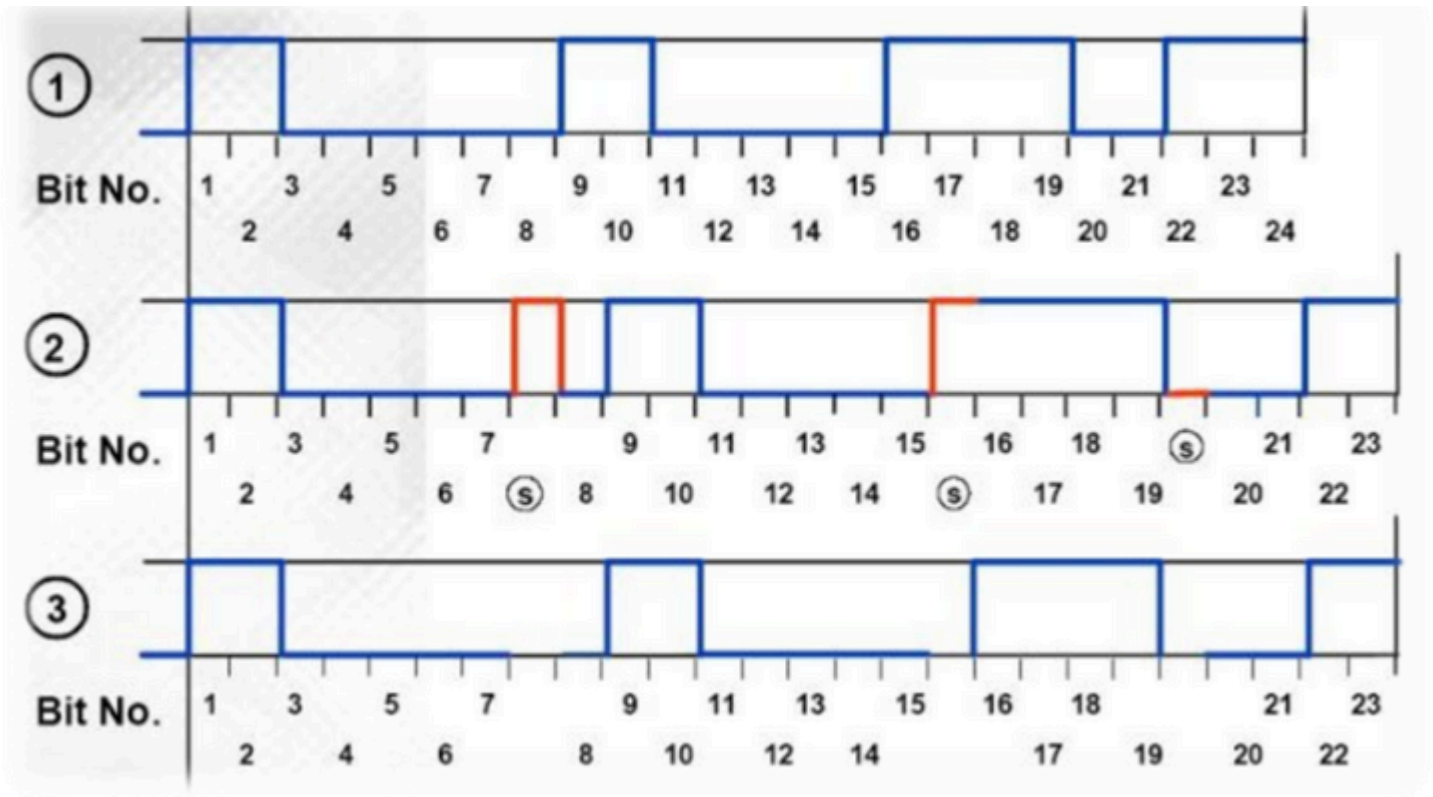
```
Transmitted CRC: 0x3F
Calculated CRC by receiver: 0x7A
→ CRC mismatch → error frame
```



3. Bit Stuffing

- After 5 consecutive bits of the same polarity, the transmitter inserts a bit of opposite polarity.
- The receiver expects this pattern.
- If violated, it triggers an error frame.

```
Data: 11111 → Transmitter sends: 111110
Receiver expects: 11111 + 0
If next bit is 1 instead of 0 → error frame
```



4. Frame Format Check

- Certain bits must have specific values:
  - CRC and ACK delimiters must be recessive (1)
  - End of frame field must be all recessive
- Invalid values result in an error frame.

```
CRC delimiter = 0 (dominant) → should be 1 (recessive)
→ Error frame sent

Data Length Code (DLC) = 9 → allowed, but ignored
→ No error frame
```

5. Acknowledgment Check

- After a frame is transmitted, the sender expects a dominant (0) ACK bit from at least one receiver.
- If not received, the transmitter sends an error frame and retries.

```
Transmitter sends data frame → ACK bit remains recessive (1)
→ No receiver acknowledged → error frame sent, retransmit
```

Mechanism	Checks For	Error Frame Sent?
Monitoring	Bit mismatch during transmission	Yes

Mechanism	Checks For	Error Frame Sent?
CRC Check	CRC mismatch between sender/receiver	Yes
Bit Stuffing	Invalid stuffed bit sequence	Yes
Format Check	Illegal bit levels in specific fields	Yes
ACK Check	No acknowledgment from any receiver	Yes

CAN – Fault Confinement

To ensure reliability, each CAN node implements **fault confinement** using two error counters:

- **Transmit Error Counter (TEC)** → $CT \in \mathbb{N}$
- **Receive Error Counter (REC)** → $CR \in \mathbb{N}$

Both counters start at 0 and are **incremented or decremented** based on the node's transmission/reception performance.

The higher these values, the **less reliable** the node is considered to be. Based on these counters, a node can be in one of **three error states**:

Error States

State	Condition	Capabilities	Error Flag Sent
Error-Active	$CR \leq 127$ and $CT \leq 127$	Normal communication	Active Error Flag
Error-Passive	$CR > 127$ or $CT > 127$	Communicates with 8-bit delay	Passive Error Flag
Bus-Off	$CT \geq 256$	Cannot transmit, can still receive	No error flag (isolated)

Behavior

- **Error-Active:**
  - Node can transmit/receive normally.
  - Sends **Active Error Flags** (dominant bits).
- **Error-Passive:**
  - Node still participates, but with reduced influence.
  - Sends **Passive Error Flags** (recessive bits).
  - Adds delay before retransmission.
- **Bus-Off:**
  - Node is **disconnected** from the bus (cannot transmit).
  - Can still **receive** messages.
  - Requires manual reset or timeout to rejoin the bus.

> This mechanism helps **contain faults** by demoting misbehaving nodes and protecting the integrity of the bus.

CAN Error Flags and Bus-Off Behavior

Active Error Flag

- Sent by a node in the **Error-Active** state when it detects an error.
- Consists of **6 dominant bits**, which **violates bit-stuffing rules**.
- This violation is **intentional** to trigger an immediate reaction from all nodes.
- Nodes that detect this will:
  - Also send an error flag (if not in Bus-Off)
  - Or remain silent (if in Bus-Off)
- The result is a **chain reaction**, alerting all nodes on the bus.

Passive Error Flag

- Sent by a node in the **Error-Passive** state when it detects an error.
- Consists of **6 recessive bits**.

- Since it does **not violate bit-stuffing**, other nodes **do not detect it**.
- No chain reaction occurs.
- Only the node itself knows an error occurred.

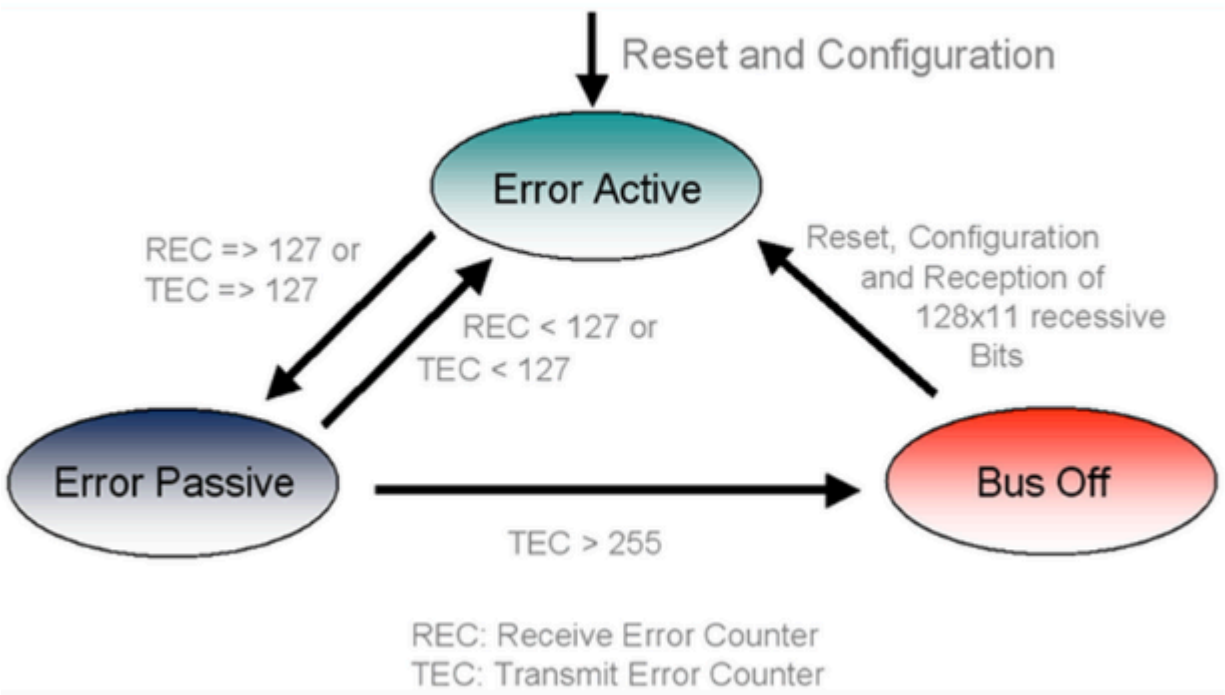
Bus-Off State

- A node enters **Bus-Off** when its **Transmit Error Counter (CT) ≥ 256**.
- In this state:
  - The node **cannot transmit** any frames or error flags.
  - It may **still receive** data from the bus.
  - It can **only increment** internal counters when further errors are detected.

Recovery from Bus-Off

- **Automatic recovery** may occur if the counters **decrease below the threshold** (CT < 256).
- If the node **fails to recover**, it may require:
  - **Manual reset**
  - Or **complete exclusion** from the network for safety

> Bus-Off is a self-protection and network-protection mechanism designed to isolate faulty nodes.



CAN – Bit Timing

In CAN communication, each bit time is divided into several segments to allow synchronization and timing adjustments across all nodes on the bus.

Bit Time Segments

1. SYNC SEG (Synchronization Segment)

- Used to synchronize all nodes.
- Expected that a signal edge (transition) lies within this segment.
- It is always **1 time quantum (Tq)** long.

2. PROP SEG (Propagation Segment)

- Compensates for physical delays in the network.
- Accounts for:
  - Signal propagation delay on the bus
  - Input comparator delay
  - Output driver delay
- Typical value: **2 × total delay**

3. PHASE SEG1

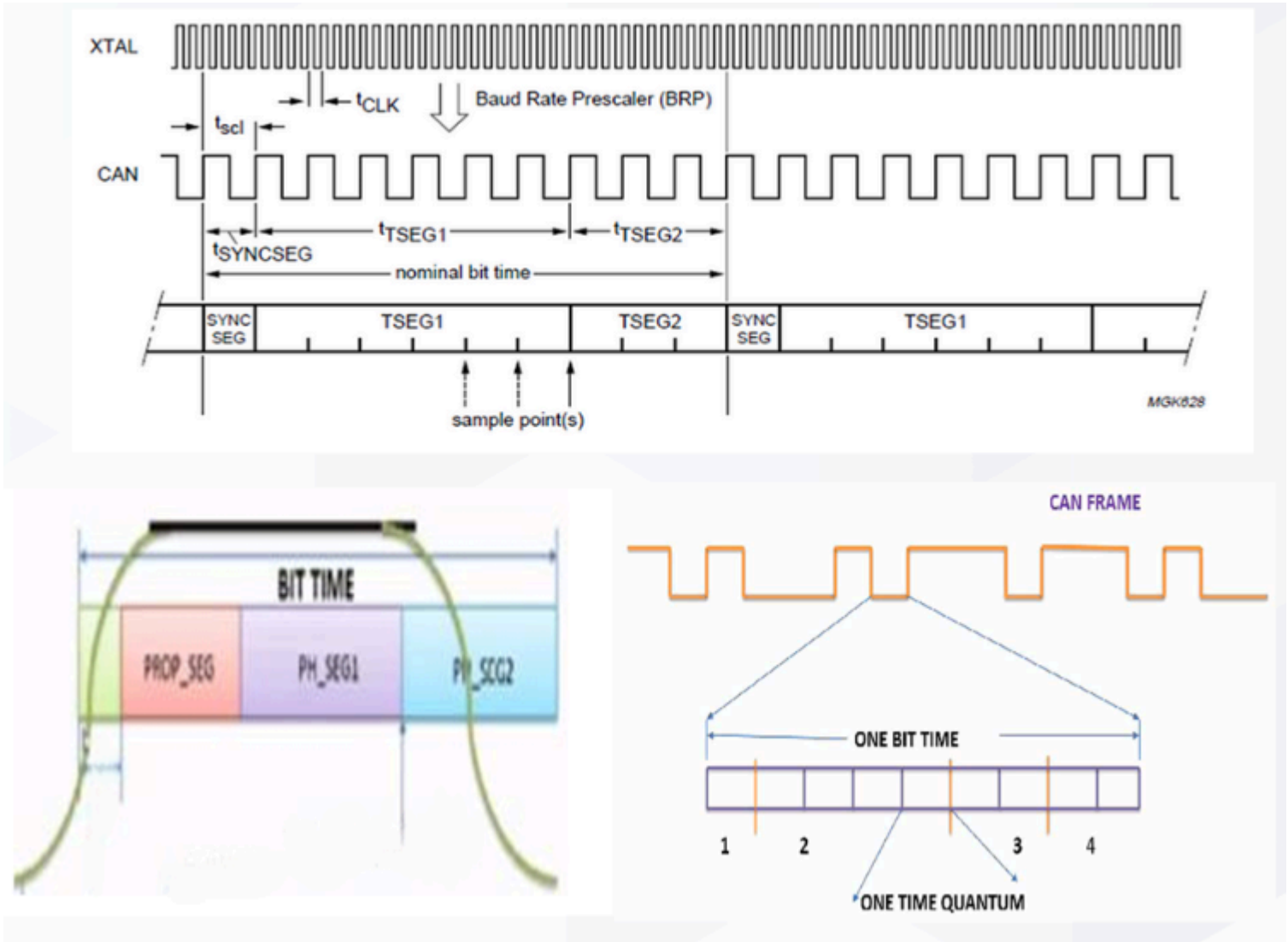
- A phase buffer segment used to compensate for edge phase errors.
- Can be **lengthened** during resynchronization.
- Also includes part of the propagation delay.

4. PHASE SEG2

- Another phase buffer segment.
- Can be **shortened** during resynchronization.
- Must be **greater than or equal to the information processing time** of the node.

```
|<--- SYNC SEG --->|<--- PROP SEG --->|<--- PHASE SEG1 --->|<--- PHASE SEG2 --->|
```

> Bit timing ensures all nodes sample the bit at the correct time and stay synchronized, even with physical signal delays.



CAN FD (Flexible Data-Rate)

CAN FD is an enhanced version of the Classical CAN protocol, designed to support higher data throughput and improved efficiency — while maintaining compatibility with the existing CAN architecture.

To promote **design transparency** and **implementation flexibility**, CAN FD is organized in layers based on the **ISO/OSI reference model**.

Key Features of CAN FD

- **Data Efficiency**
  - Supports up to **64 bytes** of data per frame.
  - Classical CAN supports only **8 bytes**.
  - This reduces overhead per byte and boosts bandwidth.
- **Higher Bitrate and Lower Latency**
  - CAN FD can achieve bitrates up to **8 Mbps** (or more with some extensions).
  - Enables **5–8× faster data transfer**, reducing latency significantly.
- **Extended Frame Structure**
  - Similar to Classical CAN but adds fields like:
    - **EDL** (Extended Data Length)

    - **BRS** (Bit Rate Switch)
- Allows switching to higher bitrate for the data field only.

Why CAN FD?

Classical CAN works well for small control signals (like engine status), but falls short when larger or faster data transmission is needed — such as:

- Firmware updates
- Sensor fusion (e.g., cameras, LiDAR)
- Complex diagnostics

CAN FD solves this by **allowing more data per frame** and **transmitting that data faster**.

Example: CAN vs CAN FD

Imagine you want to transmit a 48-byte message.

Classical CAN:

- Max payload = 8 bytes/frame
- So, **6 separate frames** are needed
- Each frame has overhead (identifier, CRC, etc.)

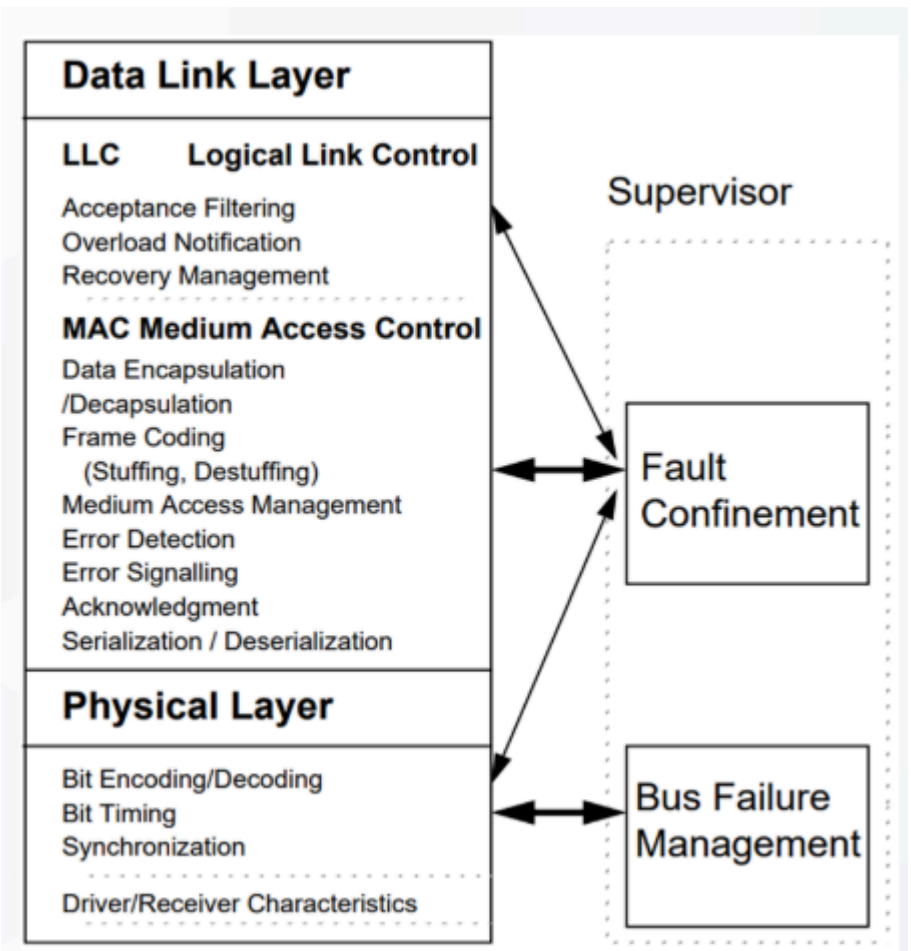
CAN FD:

- Max payload = 64 bytes/frame
- So, **entire message fits in 1 frame**
- Less overhead, much faster delivery

```
Classical CAN:
  Frame 1 → 8 bytes
  Frame 2 → 8 bytes
  ...
  Frame 6 → 8 bytes
  → Total = 6× overhead + 48 bytes

CAN FD:
  Frame 1 → 48 bytes
  → Total = 1× overhead + 48 bytes
```

Result: **Fewer frames, lower bus load, and higher speed**!



CAN FD – Data Length Code Format

DLC in Classical CAN Protocol

- **Size:** 4 bits
- **Data Length Range:** 0 to 8 bytes per frame
- Simple and efficient for small control messages.

DLC in CAN FD Protocol

- **Size:** 4 bits (same base size as Classical CAN)
- **Data Length Range:** Extended to support **0 to 64 bytes** per frame
- Allows larger payloads, improving data throughput and efficiency.

Key Differences and Benefits

Feature	Classical CAN	CAN FD
DLC Size	4 bits	4 bits (base), extended
Max Data Payload	8 bytes	Up to 64 bytes
Use Case	Small control data	Larger data, faster comm

> CAN FD's extended DLC enables sending bigger data chunks efficiently, making it suitable for modern applications needing high-speed and large payloads.

	Number of Data Bytes	Data Length Code			
		DLC3	DLC2	DLC1	DLC0
Codes in CAN and CAN FD Format	0	0	0	0	0
	1	0	0	0	1
	2	0	0	1	0
	3	0	0	1	1
	4	0	1	0	0
	5	0	1	0	1
	6	0	1	1	0
	7	0	1	1	1
CAN Format	8	1	0/1	0/1	0/1
Codes in CAN FD Format	8	1	0	0	0
	12	1	0	0	1
	16	1	0	1	0
	20	1	0	1	1
	24	1	1	0	0
	32	1	1	0	1
	48	1	1	1	0
	64	1	1	1	1

CAN FD – Frame Format

CAN FD frames extend the classical CAN frame format by adding new control bits to enable flexible data length and higher bit rates.

Key Fields

Field	Value (Bit Level)	Role / Description
IDE (Identifier Extension)	0 (dominant)	Indicates frame uses the **BASE FORMAT** (standard CAN FD frame).
EDL (Extended Data Length)	1 (recessive)	Differentiates CAN FD frame from classical CAN (where this bit is replaced by `r0`). Marks the frame as CAN FD format.

Field	Value (Bit Level)	Role / Description
BRS (Bit Rate Switch)	0 or 1 (dominant or recessive)	Indicates if bit rate switches from arbitration phase to data phase (0 = no switch, 1 = switch to higher bit rate during data).
ESI (Error State Indicator)	0 or 1 (dominant or recessive)	Shows node's error state (0 = error-active, 1 = error-passive).
r1 (Reserved Bit 1)	1 (recessive)	Used in **CAN EXTENDED FORMAT** to indicate extended identifiers.

Additional Information

- **IDE**: Always dominant (0) in CAN FD base format frames.
- **EDL**: Must be recessive (1) to mark the frame as CAN FD.
- **BRS**: Enables faster data transfer in data phase by switching bit rate.
- **ESI**: Allows nodes to indicate their health state during communication.
- **r1**: Reserved, usually set to recessive, relevant in extended frame format.

Example

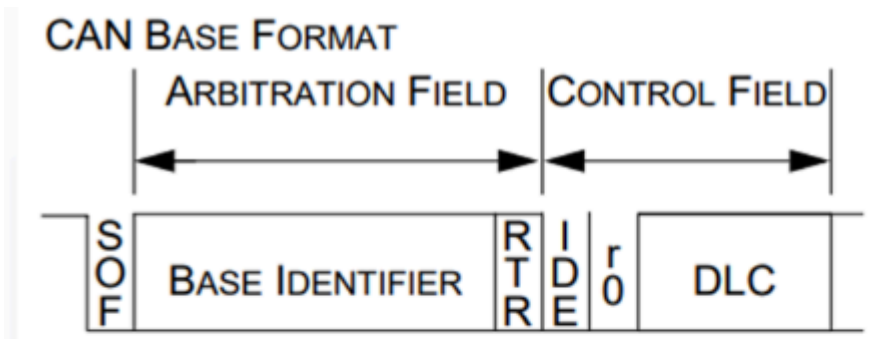
Suppose a CAN FD frame is sent with the following:

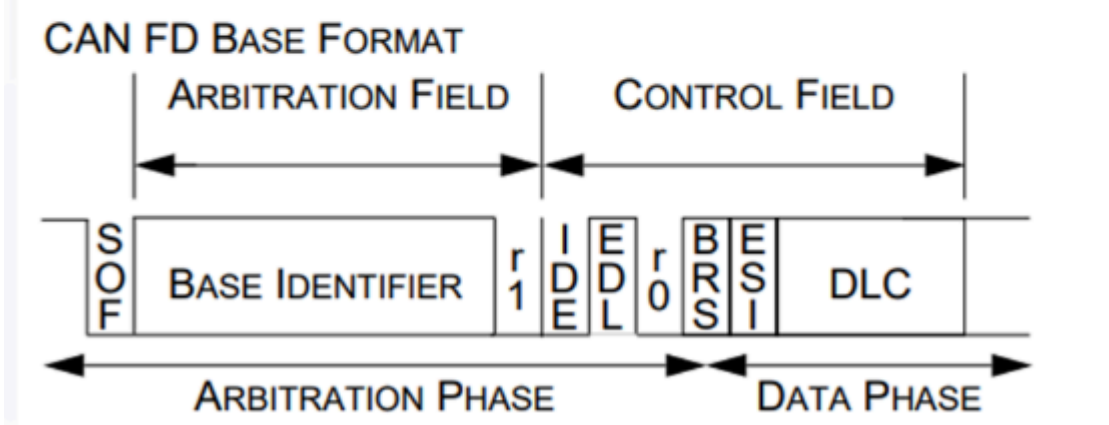
- IDE = 0 (base frame)
- EDL = 1 (CAN FD frame)
- BRS = 1 (bit rate switches to faster during data phase)
- ESI = 0 (node is error-active)
- r1 = 1 (standard for extended format)

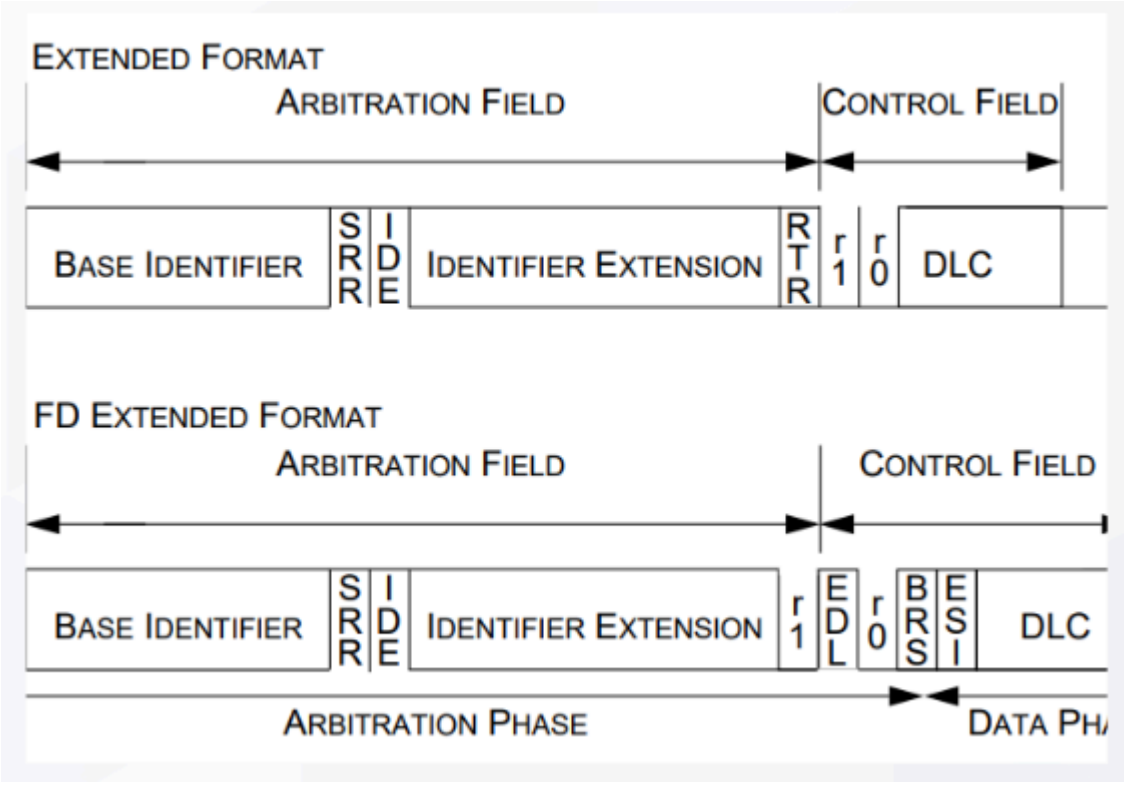
This means the frame:

- Uses CAN FD format,
- Will transmit arbitration at normal bitrate,
- Switches to a higher bitrate during data transmission,
- Is sent by a healthy (error-active) node.

> This extended frame format enables CAN FD to support faster data rates and longer data payloads while maintaining backward compatibility.







CAN – Overload Conditions

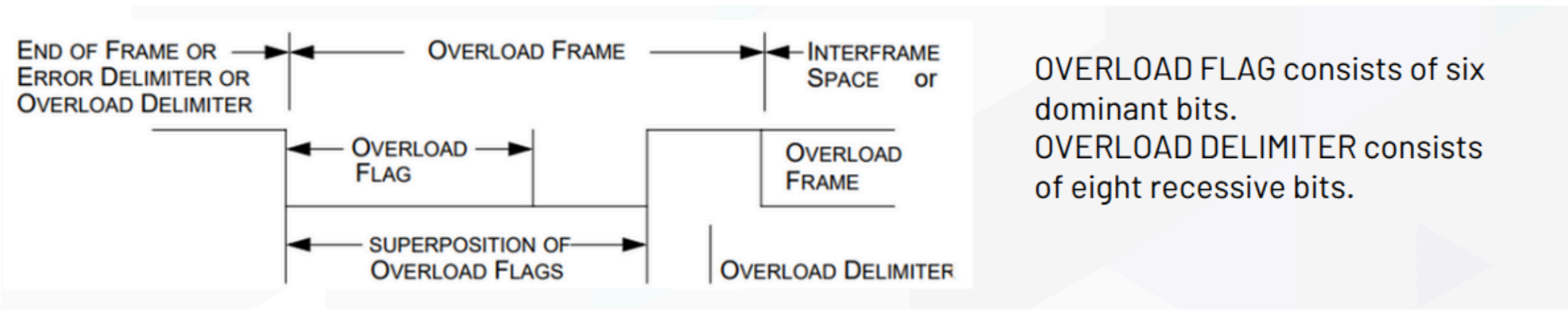
There are **three types of overload conditions** that cause the transmission of an **OVERLOAD FLAG** in the CAN network:

1. **Receiver Internal Condition**
   When a receiver needs to delay the next DATA FRAME or REMOTE FRAME, it signals an overload condition internally and triggers an overload flag.
2. **Dominant Bit During Intermission**
   If a dominant bit is detected by any node at the **first or second bit of the INTERMISSION** period, it causes an overload condition and the transmission of an overload flag.
3. **Dominant Bit at Specific Delimiters (CAN FD specific)**

   - If a CAN FD node samples a dominant bit at the **eighth bit (last bit) of an ERROR DELIMITER or OVERLOAD DELIMITER**, or
   - If a CAN FD receiver samples a dominant bit at the **last bit of END OF FRAME**,
     then it will start transmitting an **OVERLOAD FRAME** (distinct from an ERROR FRAME).

   Note: In this case, the node's error counters are **not incremented**.

> Overload frames are used to provide extra delay or recovery time in the communication without penalizing nodes with error states.



CAN FD – Operating Modes

A CAN FD unit operates in one of four main states:

Mode	Description
Integrating	The unit waits to detect eleven consecutive recessive bits after startup or bus-off recovery. Once detected, it switches to Idle.
Idle	The unit is ready to detect a START OF FRAME and can switch to either Receiver or Transmitter mode.
Receiver	The unit listens to activity on the CAN bus and is not transmitting.
Transmitter	The unit originates a message and remains Transmitter until the bus goes idle or it loses arbitration.

Special Modes

Bus Monitoring Mode

- **Function:**
  Receives DATA and REMOTE FRAMES but only transmits recessive bits (i.e., it does not actively transmit data).
- **Handling Dominant Bits:**
  When dominant bits must be sent (e.g., ACK slot, OVERLOAD flag), these are internally rerouted to monitoring logic. The bus remains recessive externally, allowing passive observation.

Restricted Operation Mode

- **Function:**
  Can transmit and receive DATA and REMOTE FRAMES and acknowledge valid frames.
- **Error Handling:**
  Does **not** send ACTIVE ERROR or OVERLOAD frames. Instead, on errors, it waits for the bus to go idle to resynchronize without incrementing error counters.

> These modes allow CAN FD nodes to adjust behavior for diagnostics, error management, or passive monitoring while maintaining bus integrity.

CAN – Controller Block Architecture

A CAN controller typically consists of the following key components that handle protocol logic, buffering, and message filtering:

Protocol Controller

Responsible for handling core CAN protocol operations:

- **Data Encapsulation:**
  Frames raw data into the standard CAN frame format.
- **Error Detection:**
  Implements mechanisms like **CRC**, **bit stuffing**, and **frame checks** to detect transmission errors.
- **Acknowledgment Handling:**
  Manages ACK bits and responds to successfully received messages.
- **Bit Encoding:**
  Converts logical bits into the NRZ (Non-Return-to-Zero) format used on the CAN bus.

Hardware Acceptance Filter

- Filters incoming frames based on **identifier** matching.
- Prevents the CPU from processing irrelevant messages.
- Enhances efficiency by allowing only relevant frames through to the receive buffer.

Message Buffers

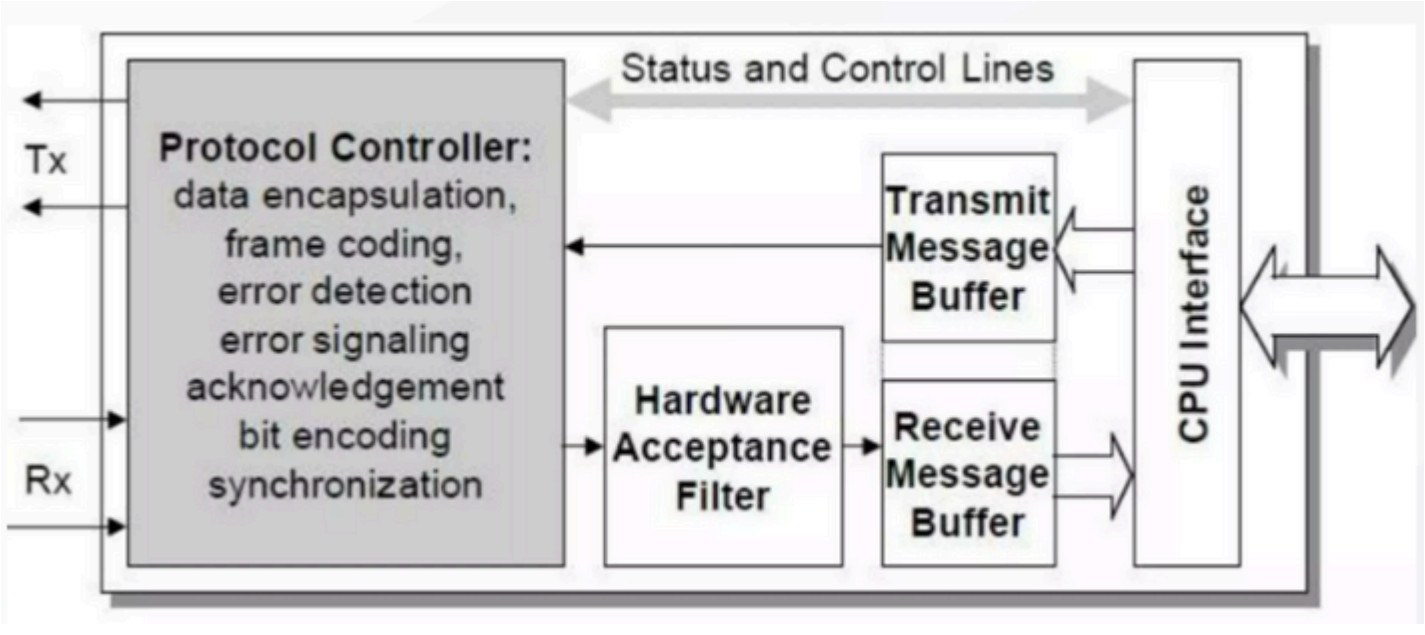
Transmit Message Buffer:

- Holds messages waiting to be transmitted.
- Ensures proper ordering and reliable transmission.

Receive Message Buffer:

- Stores incoming messages temporarily.
- Helps manage the flow of data before it's passed to the microcontroller.

> These components work together to ensure reliable, efficient, and real-time communication over the CAN bus.



Day 3(Part 2) - SPI

SPI(Serial Peripheral Interface) is a widely used synchronous, full duplex interface for communication between a microcontroller (main) and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and others.

Key Features

- **Communication Type:** Synchronous, full duplex
- **Data Synchronization:** Data from main and subnode synchronized on rising or falling clock edge
- **Data Transmission:** Both main and subnode can transmit simultaneously
- **Wiring:** Available as 3-wire or 4-wire interface; focus here is on 4-wire SPI

4-Wire SPI Signals

- **Clock (SCLK):** Generated by the main device; synchronizes data transfer
- **Chip Select (CS):** Active low signal used to select a subnode device
- **MOSI (Main Out, Subnode In):** Data from main to subnode
- **MISO (Main In, Subnode Out):** Data from subnode to main

Interface

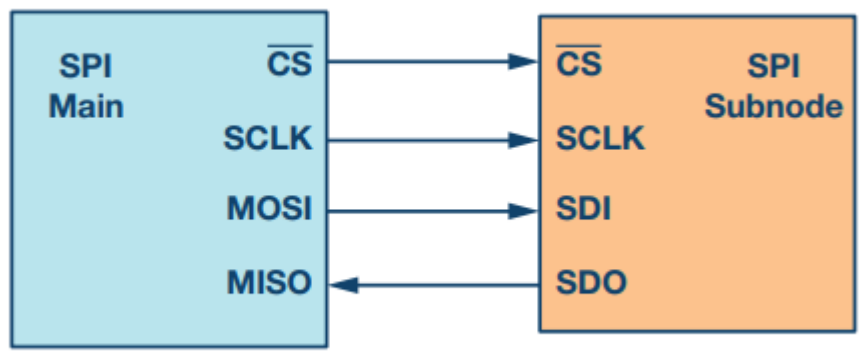


Figure 1. SPI configuration with main and a subnode.

Additional Details

- SPI supports higher clock frequencies than I^2C (refer to product datasheet for max frequency)
- Only one main device per SPI bus, but multiple subnodes are possible
- Each subnode requires a dedicated chip select line from the main
- Chip select is active low and pulled high when subnode is disconnected from the bus

Use Case: Analog Devices SPI Switches and Muxes

- SPI-enabled switches and multiplexers from Analog Devices help reduce the number of GPIO pins required in system board designs by managing multiple SPI subnodes efficiently.

SPI – Data Transmission

To initiate SPI communication, the **master** must generate the clock signal and activate the **chip select (CS)** line to choose the slave device. The CS signal is typically **active low**, meaning the master must drive it **low (logic 0)** to select the slave.

SPI is a **full-duplex interface**, allowing both the master and slave to transmit data simultaneously:

- The master transmits data on the **MOSI (Master Out, Slave In)** line.
- The slave sends data back on the **MISO (Master In, Slave Out)** line.

During communication:

- Data is shifted out serially on the **MOSI** line.
- At the same time, data is sampled from the **MISO** line.
- The **serial clock (SCLK)** synchronizes both shifting and sampling operations.

SPI offers flexibility in timing:

- Users can configure the interface to sample and/or shift data on either the **rising** or **falling** edge of the clock.
- These settings are often programmable and should match between master and slave devices.

> The number of data bits per SPI transfer is device-dependent. Always refer to the device's data sheet for specifics.

SPI – Clock Polarity and Clock Phase

In SPI communication, the master can configure two timing parameters to control how data is sampled and shifted:

Clock Polarity (CPOL)

- **CPOL** sets the **idle state** of the clock line (SCLK).
- The idle state is defined as the period:
  - **Before** transmission begins (when CS goes from high to low).
  - **After** transmission ends (when CS returns from low to high).
- If **CPOL = 0**, the clock is **low** when idle.
- If **CPOL = 1**, the clock is **high** when idle.

Clock Phase (CPHA)

- **CPHA** determines **when data is sampled and shifted** relative to the clock edge.
- If **CPHA = 0**, data is sampled on the **first clock edge**.
- If **CPHA = 1**, data is sampled on the **second clock edge**.

SPI Modes

There are four possible SPI modes, defined by combinations of CPOL and CPHA:

Mode	CPOL	CPHA	Clock Idle State	Data Capture Edge	Data Shift Edge
0	0	0	Low	Rising edge	Falling edge
1	0	1	Low	Falling edge	Rising edge
2	1	0	High	Falling edge	Rising edge
3	1	1	High	Rising edge	Falling edge

> The master must select the SPI mode that matches the slave's requirement. Always consult the slave device's data sheet to determine the correct CPOL and CPHA settings.

Table 1. SPI Modes with CPOL and CPHA

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic high	Data sampled on the falling edge and shifted out on the rising edge
3	1	1	Logic high	Data sampled on the rising edge and shifted out on the falling edge

SPI Communication Timing Modes

Figures 2 through 5 illustrate example timing diagrams of SPI communication in four different modes. In these examples:

- Data is shown on the **MOSI** and **MISO** lines.
- The **start and end of transmission** are indicated by the **dotted green line**.
- The **sampling edge** (where data is read) is indicated by the **orange dotted line**.
- The **shifting edge** (where data is changed) is indicated by the **blue dotted line**.

> **Note:** These figures are for illustration purposes only. For successful SPI communication, always refer to the product datasheet and ensure the timing specifications of the specific device are met.

SPI Mode 0 (CPOL = 0, CPHA = 0)

- **Clock Polarity (CPOL)** = 0
  The idle state of the clock signal is **low**.
- **Clock Phase (CPHA)** = 0
  Data is **sampled on the rising edge** of the clock (orange dotted line).
  Data is **shifted on the falling edge** of the clock (blue dotted line).

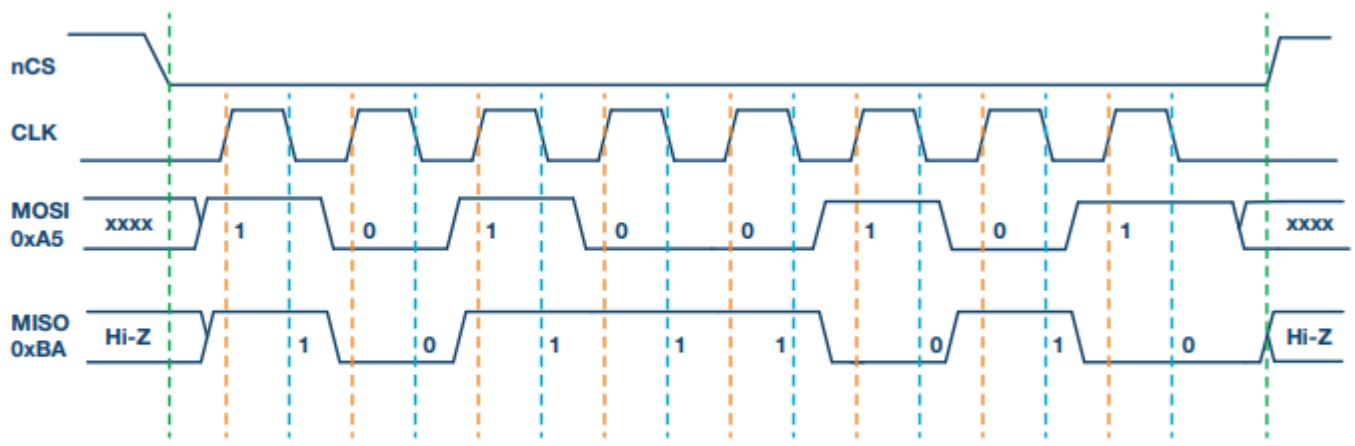


Figure 2. SPI Mode 0, CPOL = 0, CPHA = 0: CLK idle state = low, data sampled on rising edge and shifted on falling edge.

SPI Mode 1 (CPOL = 0, CPHA = 1)

- **Clock Polarity (CPOL)** = 0
  The idle state of the clock signal is **low**.
- **Clock Phase (CPHA)** = 1
  Data is **sampled on the falling edge** of the clock (orange dotted line).
  Data is **shifted on the rising edge** of the clock (blue dotted line).

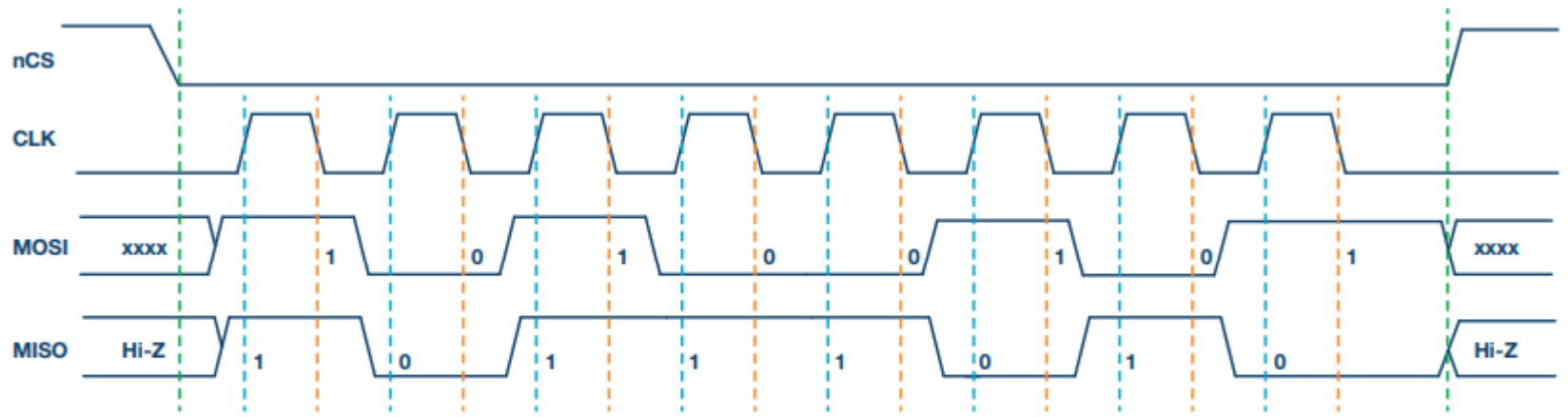


Figure 3. SPI Mode 1, CPOL = 0, CPHA = 1: CLK idle state = low, data sampled on the falling edge and shifted on the rising edge.

SPI Mode 2 (CPOL = 1, CPHA = 0)

- **Clock Polarity (CPOL)** = 1
  The idle state of the clock signal is **high**.
- **Clock Phase (CPHA)** = 0
  Data is **sampled on the falling edge** of the clock (orange dotted line).
  Data is **shifted on the rising edge** of the clock (blue dotted line).

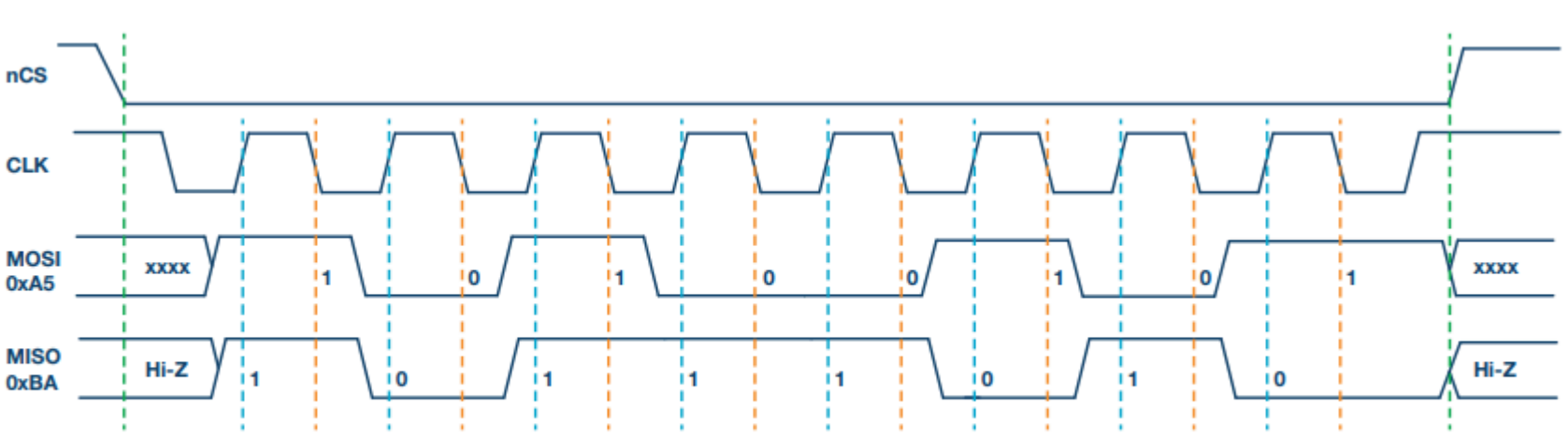


Figure 4. SPI Mode 2, CPOL = 1, CPHA = 0: CLK idle state = high, data sampled on the falling edge and shifted on the rising edge.

SPI Mode 3 (CPOL = 1, CPHA = 1)

- **Clock Polarity (CPOL)** = 1
  The idle state of the clock signal is **high**.
- **Clock Phase (CPHA)** = 1
  Data is **sampled on the rising edge** of the clock (orange dotted line).
  Data is **shifted on the falling edge** of the clock (blue dotted line).

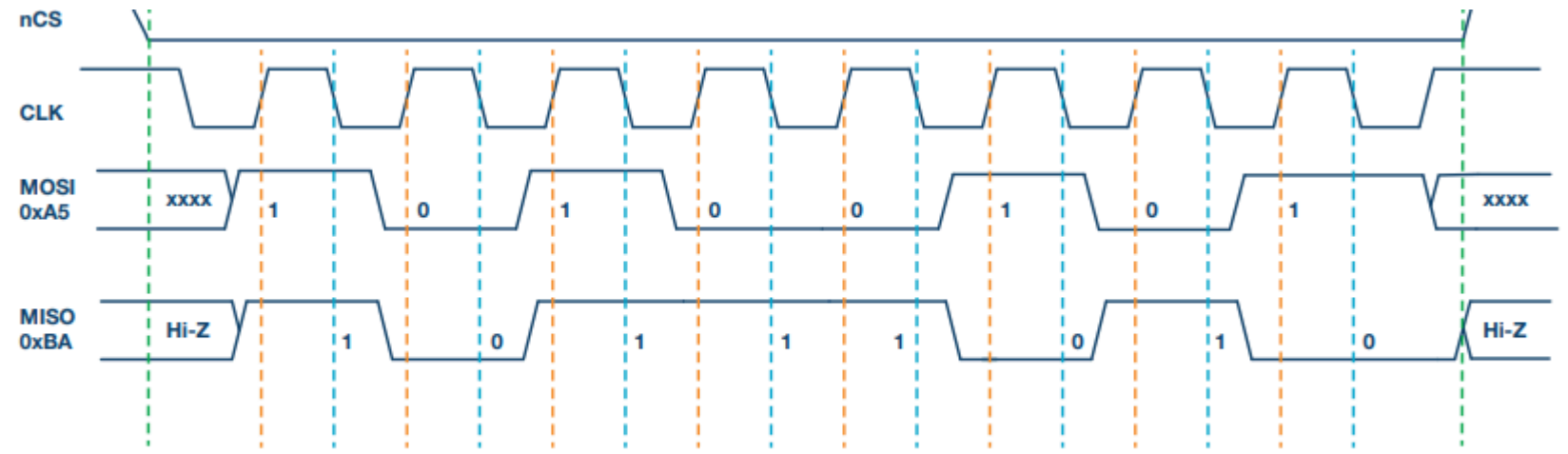


Figure 5. SPI Mode 3, CPOL = 1, CPHA = 1: CLK idle state = high, data sampled on the rising edge and shifted on the falling edge.

Multi-Subnode Configuration

Multiple subnodes (slaves) can be used with a single SPI main (master). The subnodes can be connected in **regular mode** or **daisy-chain mode**.

Regular SPI Mode

- In regular mode, each subnode requires an individual **chip select (CS)** line from the SPI main.

- When the chip select signal is enabled (pulled low) by the main, the clock and data on the **MOSI** and **MISO** lines are available for the selected subnode.
- If multiple chip select signals are enabled simultaneously, the data on the **MISO** line becomes corrupted because the main cannot determine which subnode is transmitting.
- As shown in **Figure 6**, increasing the number of subnodes increases the number of chip select lines from the main. This can quickly increase the number of inputs and outputs required from the main and limit how many subnodes can be used.
- Techniques such as using a **multiplexer (mux)** to generate chip select signals can help increase the number of subnodes supported in regular mode.

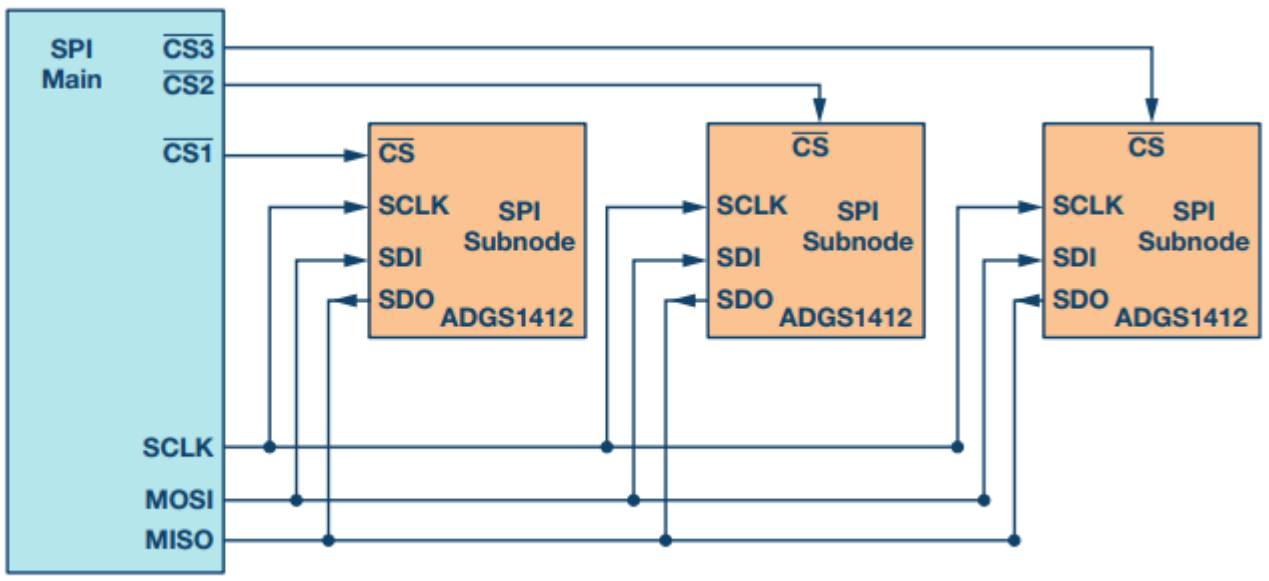


Figure 6. Multi-subnode SPI configuration.

Daisy-Chain Mode

- In daisy-chain mode, all subnodes share the same **chip select** signal, which is tied together.
- Data propagates sequentially from one subnode to the next.
- All subnodes receive the same SPI clock simultaneously.
- The main's data output connects directly to the first subnode, which then passes data to the next subnode, and so on.
- Because data passes through each subnode in sequence, the number of clock cycles needed to transmit data to a specific subnode is proportional to its position in the chain.
  - For example, in **Figure 7**, for an 8-bit system, the third subnode requires 24 clock pulses to have valid data, compared to only 8 clock pulses in regular SPI mode.
- **Figure 8** illustrates the clock cycles and data propagation through the daisy chain.
- Note: Daisy-chain mode is **not supported by all SPI devices**. Always refer to the product datasheet to verify support for this mode.

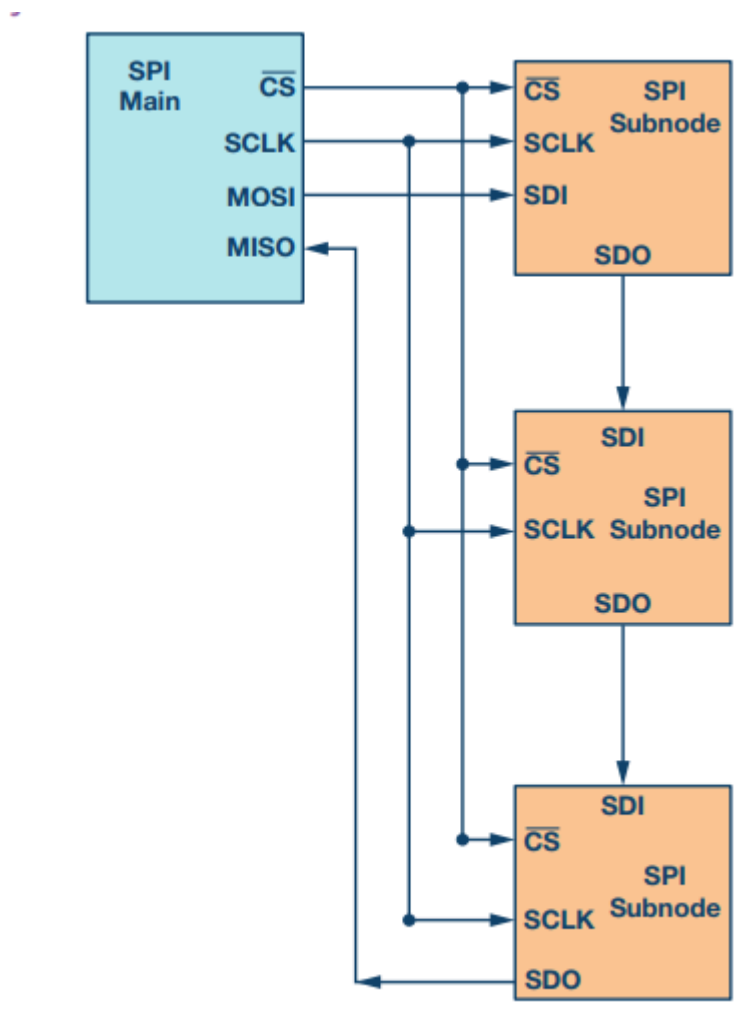


Figure 7. Multi-subnode SPI daisy-chain configuration.

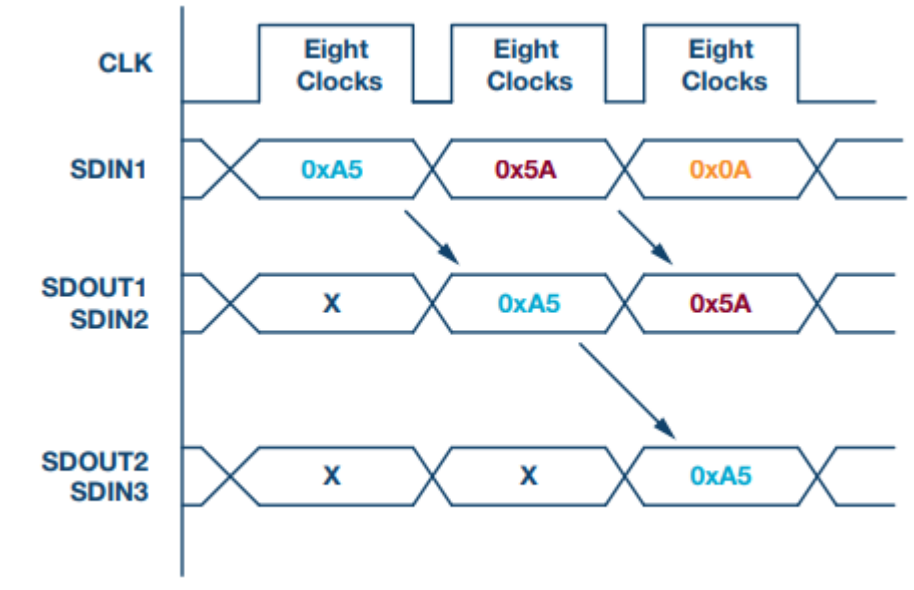


Figure 8. Daisy-chain configuration: data propagation.

Analog Devices SPI Enabled Switches and Muxes

The newest generation of Analog Devices (ADI) SPI-enabled switches offer significant space savings without compromising precision switch performance. This section discusses a case study on how SPI-enabled switches or multiplexers (muxes) can simplify system-level design and reduce the number of required GPIOs.

ADG1412 Switch Example

- The **ADG1412** is a quad, single-pole, single-throw (SPST) switch, requiring **four GPIOs** connected to the control inputs (one GPIO per switch).
- Figure below shows the connection between a microcontroller and one ADG1412.

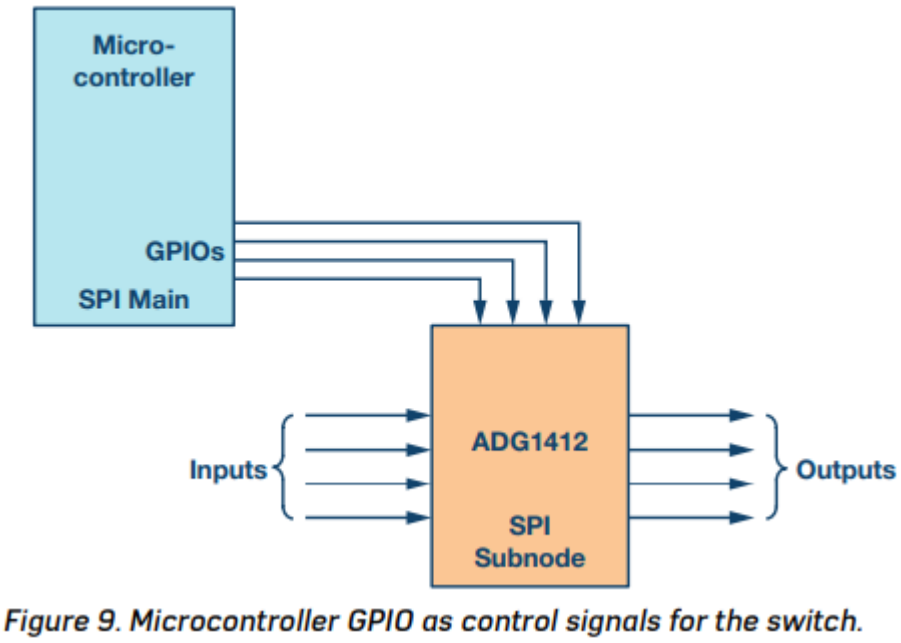


Figure 9. Microcontroller GPIO as control signals for the switch.

Scaling with Multiple Switches

- As the number of switches on the board increases, the number of GPIOs needed also increases significantly.
- For example, in a **4 × 4 cross-point matrix** configuration, four ADG1412s are used, requiring **16 GPIOs** from the microcontroller (one per control input).
- Figure below illustrates this setup using 16 GPIOs.

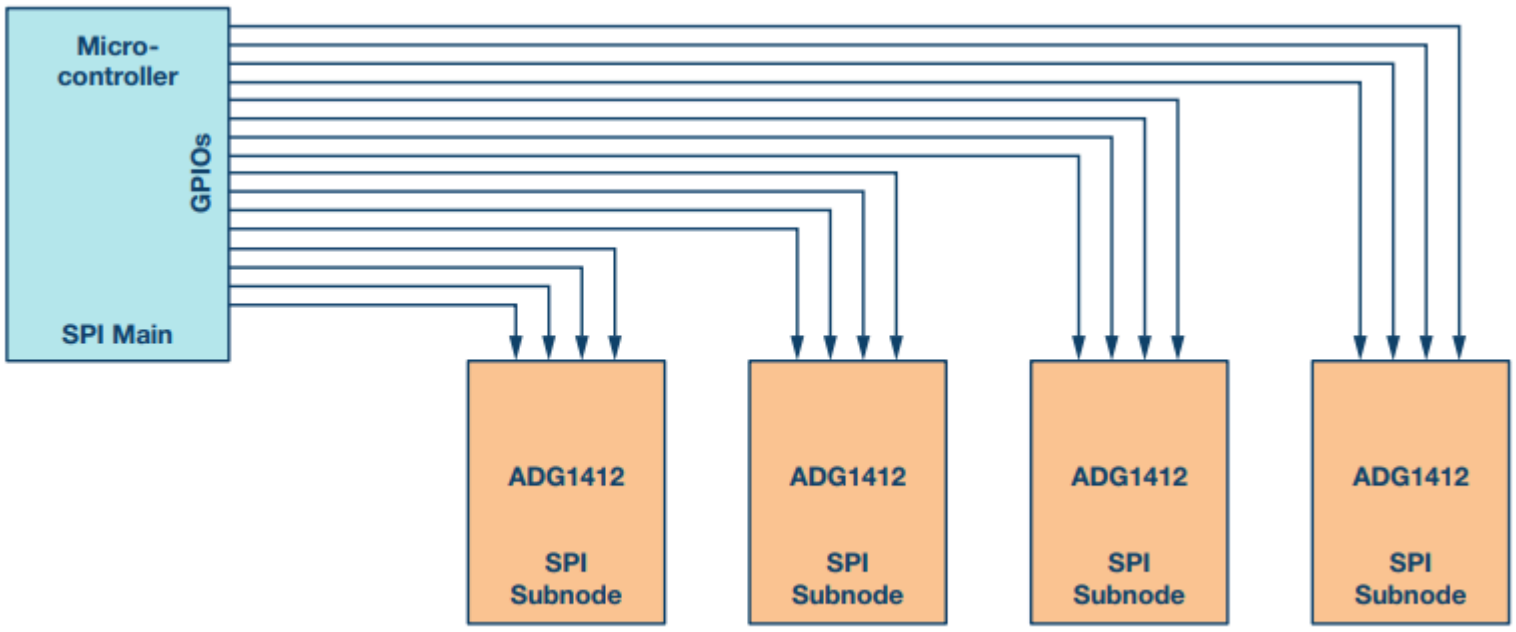


Figure 10. In a multi-subnode configuration, the number of GPIOs needed increases tremendously.

Reducing GPIO Usage

Serial-to-Parallel Converter Approach

- One method to reduce GPIO usage is to use a **serial-to-parallel converter** (see Figure below).
- This device outputs parallel signals connected to switch control inputs and is configured via an SPI serial interface.
- **Drawback:** This adds complexity and cost by introducing an additional component.

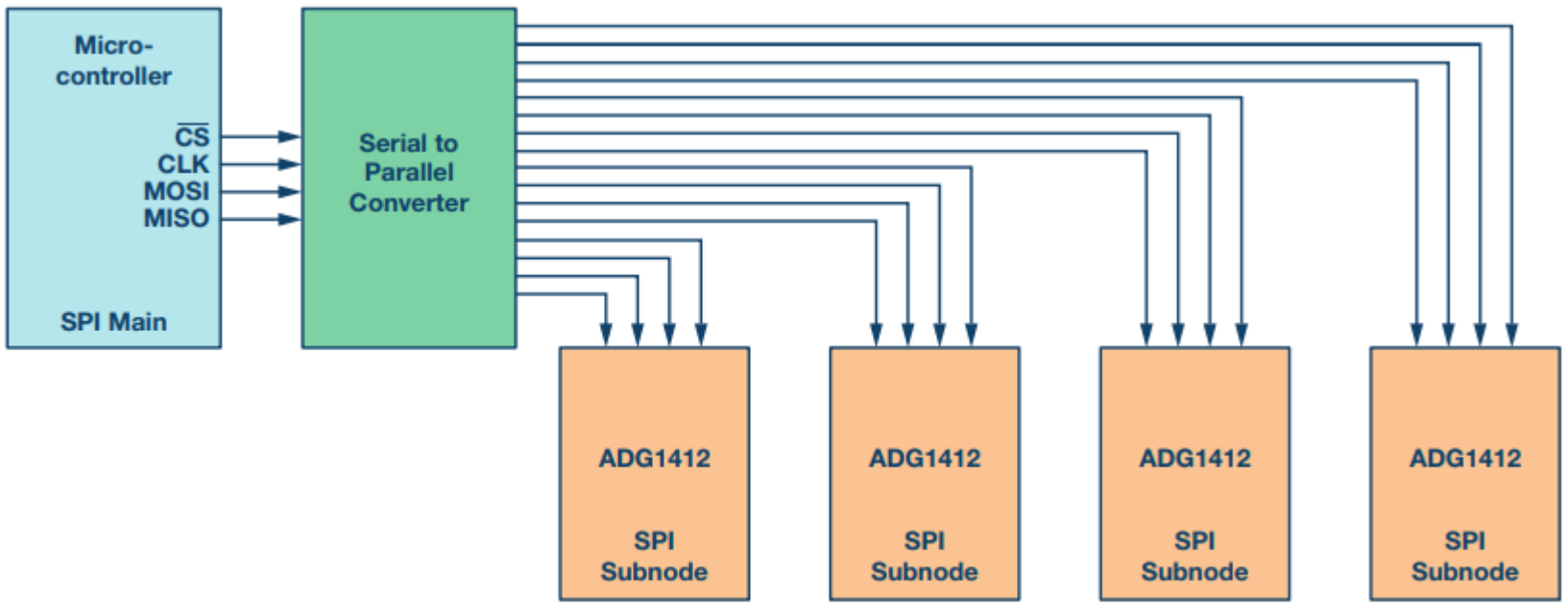


Figure 11. Multi-subnode switches using a serial-to-parallel converter.

SPI Controlled Switches Approach

- An alternative is to use **SPI controlled switches**, reducing GPIO requirements and eliminating the need for extra converters.
- As shown in figure below, only **7 microcontroller GPIOs** are needed to provide SPI signals to four ADGS1412 switches.

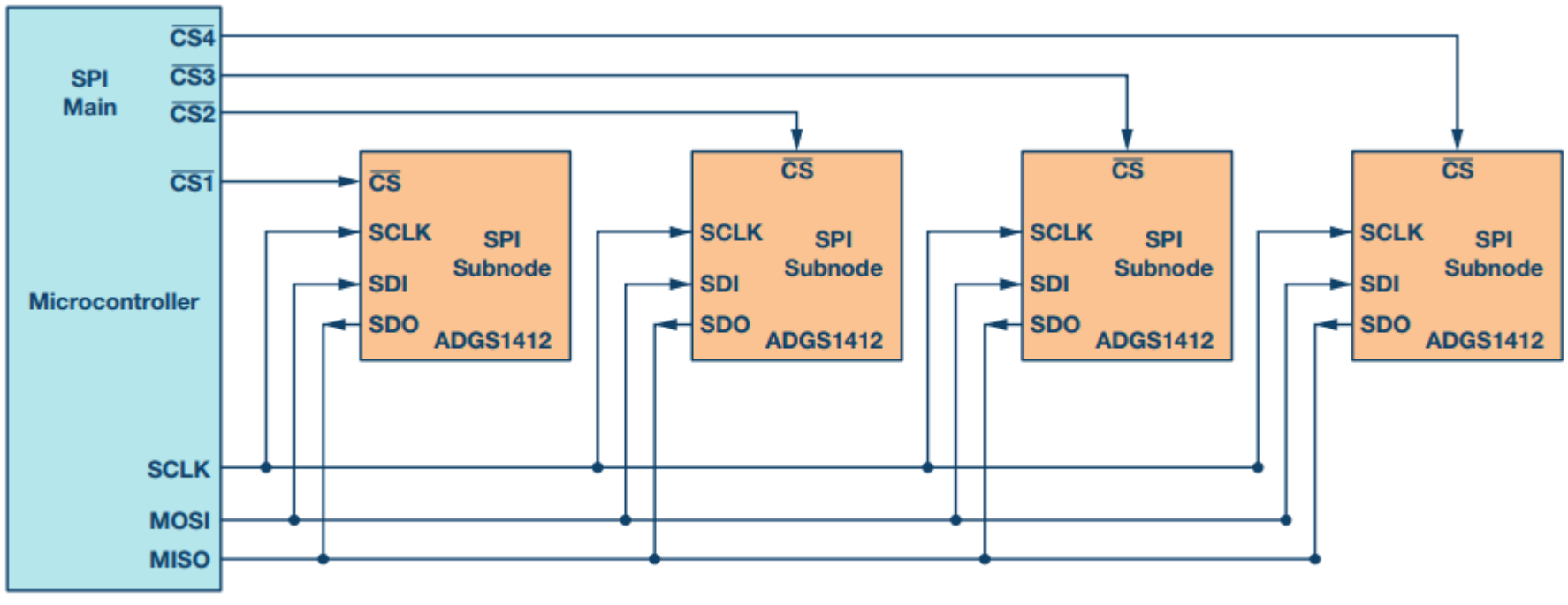


Figure 12. SPI enabled switches save up microcontroller GPIOs.

- These switches can be configured in **daisy-chain mode**, further optimizing GPIO usage.
- In daisy-chain configuration, **only 4 GPIOs** are needed from the microcontroller, regardless of the number of switches.

- **Figure below illustrates this concept (for simplicity, using four switches).

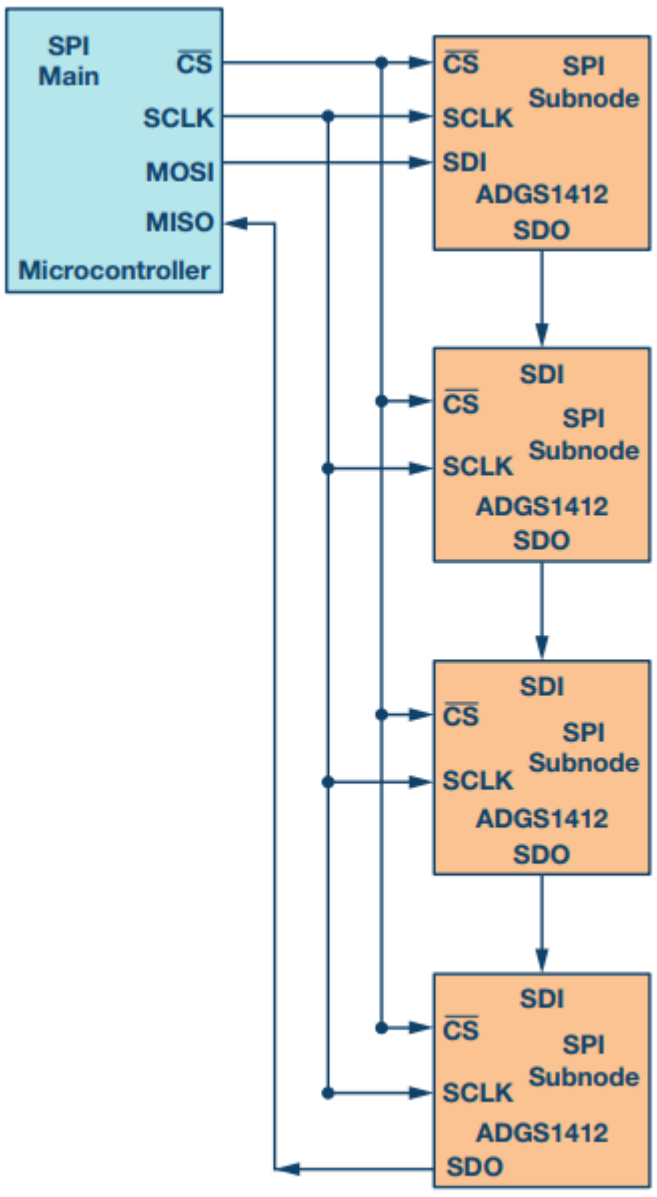


Figure 13. SPI enabled switches configured in a daisy chain to further optimize the GPIOs.

Additional Notes

- The ADGS1412 datasheet recommends a **pull-up resistor** on the SDO pin.
  Please refer to the ADGS1412 datasheet for more details on daisy-chain mode.
- As the number of switches increases, benefits in **board simplicity** and **space savings** become significant.
- In a **4 × 8 crosspoint configuration** (eight quad SPST switches on a 6-layer board), ADI SPI-enabled switches offer an estimated **20% overall board space reduction**.
- For detailed insights, see the article Precision SPI Switch Configuration Increases Channel Density.
- Analog Devices offers several SPI-enabled switches and multiplexers. For more information, visit the Analog Switches & Multiplexers product category.