**UART**: Universal Asynchronous receiver and transmitter

It is a hardware peripheral used for serial communication. It transmits and receives the one byte at a time, asynchronously.

Eg: Communication between microcontrollers, sensors, GPU modules, PCs etc.

UART hardware generates an interrupt (eg: UART\_RX\_ISR) every time when there is a arrival of new byte. If our component don’t want to process the data immediately we can store the data in ring buffer temporarily.

**ISR**: Interrupt service routine

Special function in embedded system and microcontrollers that is automatically executed in response to an interrupt.

**Serial Interface**:

In this, data is sent one bit at a time over a single wire/channel. In parallel communication, multiple bits at once using multiple lines are sent

**Asynchronous**:

No clock signal is shared between sender & receiver. Both devices agree common baud rate (like 9600bps, speed). Each byte is wrapped with start/stop bits so, receiver knows when data starts and ends.

Basics on cores:

In computing and embedded systems, a **core** refers to the **individual processing unit** inside a **CPU (Central Processing Unit)** or **microcontroller**. Each **core** can **independently execute instructions**, meaning it acts like a **mini processor** on its own.

**Think of It Like This:**

Imagine a CPU is a **kitchen**, and each **core** is a **chef**.

* 1 chef = 1 task at a time
* 4 chefs = 4 tasks can be done in parallel

So, a **multi-core** CPU can handle **multiple instructions or tasks** at the same time.

A modern **automotive microcontroller** (like the Infineon AURIX or NXP S32K) may have:

* 2 or more **CPU cores**
* Each core might:
  + Run its own **AUTOSAR OS instance**
  + Handle a different function (e.g., safety-critical logic vs. diagnostics)
  + Use **inter-core communication** to share data

When an **ECU (Electronic Control Unit)** receives **sensor data**, the system must go through a series of **key tasks** to **safely, efficiently, and accurately** process that data. These tasks are distributed across **cores** depending on the **AUTOSAR architecture**, safety requirements, and system performance needs.

**🧠 Overview: Main Tasks After Receiving Sensor Data**

Let’s break it down step-by-step:

**🔹 1. Signal Acquisition (Sensor Interface Layer)**

**Where:** MCAL (Microcontroller Abstraction Layer)  
**Core Role:** Core 0 or hardware-triggered ISR

* Read data from ADC (Analog), SPI, I2C, UART, or digital GPIO.
* Perform **basic validation** (e.g., signal within range).
* May use **interrupts** to handle incoming data fast.

➡️ **Result:** Raw sensor value in memory.

**🔹 2. Preprocessing / Filtering**

**Where:** Complex Driver or Sensor Abstraction Layer  
**Core Role:** Core 0 or Core 1

* Apply **digital filters** (low-pass, Kalman, etc.).
* Perform **scaling**, **unit conversion**, **debouncing**.
* Handle **sensor fusion** if multiple sensors measure the same thing.

➡️ **Result:** Cleaned-up, usable signal data.

**🔹 3. Diagnostics & Monitoring**

**Where:** BSW (DEM, DCM)  
**Core Role:** Core 2 (diagnostics/core health)

* Check for:
  + Sensor short/open circuit
  + Plausibility errors
  + Missing data
* Report faults via **DTCs (Diagnostic Trouble Codes)**

➡️ **Result:** Error flags and diagnostic info stored/reported.

**🔹 4. Data Communication**

**Where:** AUTOSAR COM Stack (PDU Router, COM module, CAN/Ethernet MCAL)  
**Core Role:** Core 2 or Core 3 (communication core)

* Pack and transmit sensor data via:
  + **CAN**
  + **LIN**
  + **Ethernet**
  + **FlexRay**
* Handle message timing and scheduling (based on **ComTxMode**)

➡️ **Result:** Sensor data shared with other ECUs.

**🔹 5. Application Logic / Control Algorithms**

**Where:** Application Layer (AUTOSAR SWCs)  
**Core Role:** Core 0 or Core 1 (safety/functional logic)

* Control decisions based on sensor values:
  + E.g., adjust fuel injection, brake force, steering angle
* Logic is often **ASIL-rated** (e.g., ASIL D for safety-critical)

➡️ **Result:** Commands to actuators or next-layer systems.

**🔹 6. Actuator Control (Output Processing)**

**Where:** MCAL + Complex Drivers  
**Core Role:** Core 0 or Core 1

* Convert processed data into actuator control signals:
  + PWM, DAC, GPIO output
* May involve closed-loop control (PID)

➡️ **Result:** Physical response (motor turns, light changes, brake applies).

**🔹 7. Logging, Reporting, OTA, Debug (Optional)**

**Where:** Diagnostic/Telematics modules  
**Core Role:** Core 3 or Background Tasks

* Store sensor data to memory or send to cloud.
* Provide runtime info via UDS or XCP/DAQ.

**🧩 Core Distribution Example**

| **Core** | **Tasks** |
| --- | --- |
| Core 0 | ISR for signal capture, safety logic, actuator control |
| Core 1 | Sensor preprocessing, filter, fusion |
| Core 2 | Diagnostics (DTCs), communication stack (CAN, Ethernet) |
| Core 3 | Logging, OTA updates, debugging |

**🔁 Data Flow Summary Diagram**

mathematica

CopyEdit

[SENSOR]

↓

MCAL (ADC/SPI/UART)

↓

Sensor Abstraction → Filtering → Diagnostic Check

↓

Application SWC (Control Logic)

↓

PDU Router / COM Stack → CAN / Ethernet

↓

[Other ECUs or Actuators]

**Two Data Paths in an ECU**

**🔹 1. Data IN (Reception)**

**Used to perform fusion or control logic.**

**Flow (Reception via CAN):**

vbnet

CopyEdit

CAN Bus

↓

CAN Interface (MCAL)

↓

PDU Router

↓

AUTOSAR COM Module

↓

Signal to Application SWC (RTE)

↓

Filtering → Fusion → Control Logic

**Purpose:**

* You **receive sensor data** from another ECU via CAN.
* It is **processed**: validated, filtered, possibly fused with other data.
* Then passed to logic that makes decisions (e.g., brake, steer).

**🔹 2. Data OUT (Transmission)**

**Used to send processed or derived data to other ECUs.**

**Flow (Your ECU wants to send data out):**

objectivec

CopyEdit

Your Application SWC (Control Result)

↓

Filtering / Output Limiting

↓

RTE → COM Signal

↓

PDU Router

↓

CAN Interface

↓

CAN Bus → Other ECU

**Purpose:**

* You **generate new data** or **results** (e.g., fused object position).
* You want to **send that data** to another ECU over CAN.
* Before sending, the data may be **clamped**, **rate-limited**, or **filtered**.

**🧠 Key Understanding**

🔸 **"Filtering" happens both before and after communication, but for different purposes.**

| **Direction** | **When Filtered?** | **Purpose of Filtering** |
| --- | --- | --- |
| **Receiving (Rx)** | After reception from CAN | Clean noisy input, validate, fuse |
| **Transmitting (Tx)** | Before sending to CAN | Limit jitter, protect output |

**✅ Full Round-Trip Example**

Let’s say you're working on a **Parking Assist ECU**.

**It receives:**

* Distance from ultrasonic sensors (via CAN)
* Camera data (via Ethernet)

**It processes:**

* Sensor fusion to determine object location

**It transmits:**

* Final "Object Detected" signal to the HMI ECU (via CAN)

**📌 Flow Summary:**

**➤ Reception Path**

kotlin

CopyEdit

CAN Receive → PDU Router → COM → RTE → SWC

SWC filters / fuses data

**➤ Processing**

pgsql

CopyEdit

SWC runs logic: Is object within danger zone?

**➤ Transmission Path**

lua

CopyEdit

SWC output → COM Signal → PDU Router → CAN Transmit