Center of Excellence: Supercomputing for AI & Big-Data

Architectural Insights and Programming Techniques for Embedded Systems

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center

Contents

Overview of Embedded Systems and their Application

Introduction to Micro-controller and Microprocessors

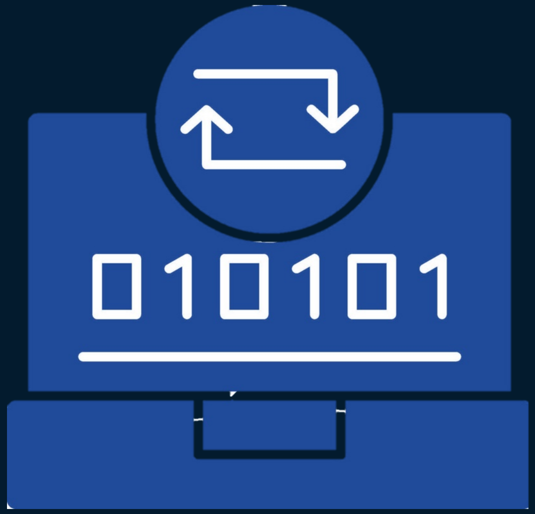
Embedded System Architectures

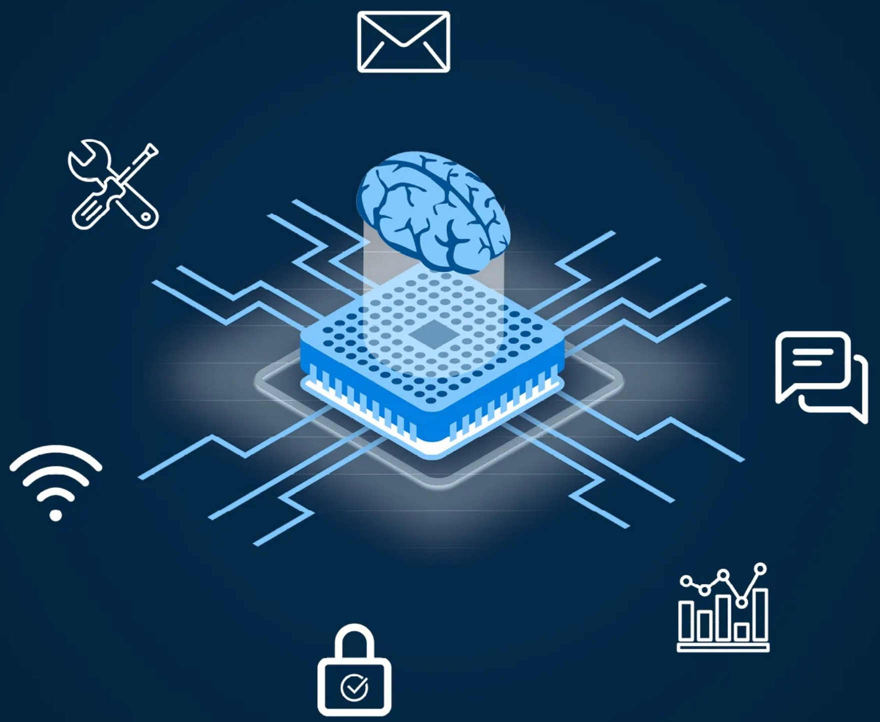
Memory Mapping and Bus Architecture

Embedded System Clock Tree

Embedded processor Instruction Set Architecture

Embedded Systems



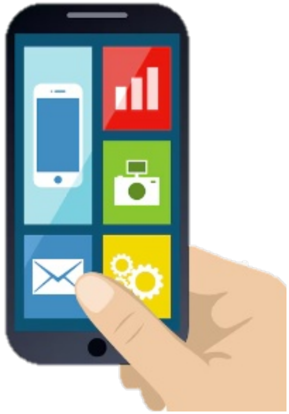


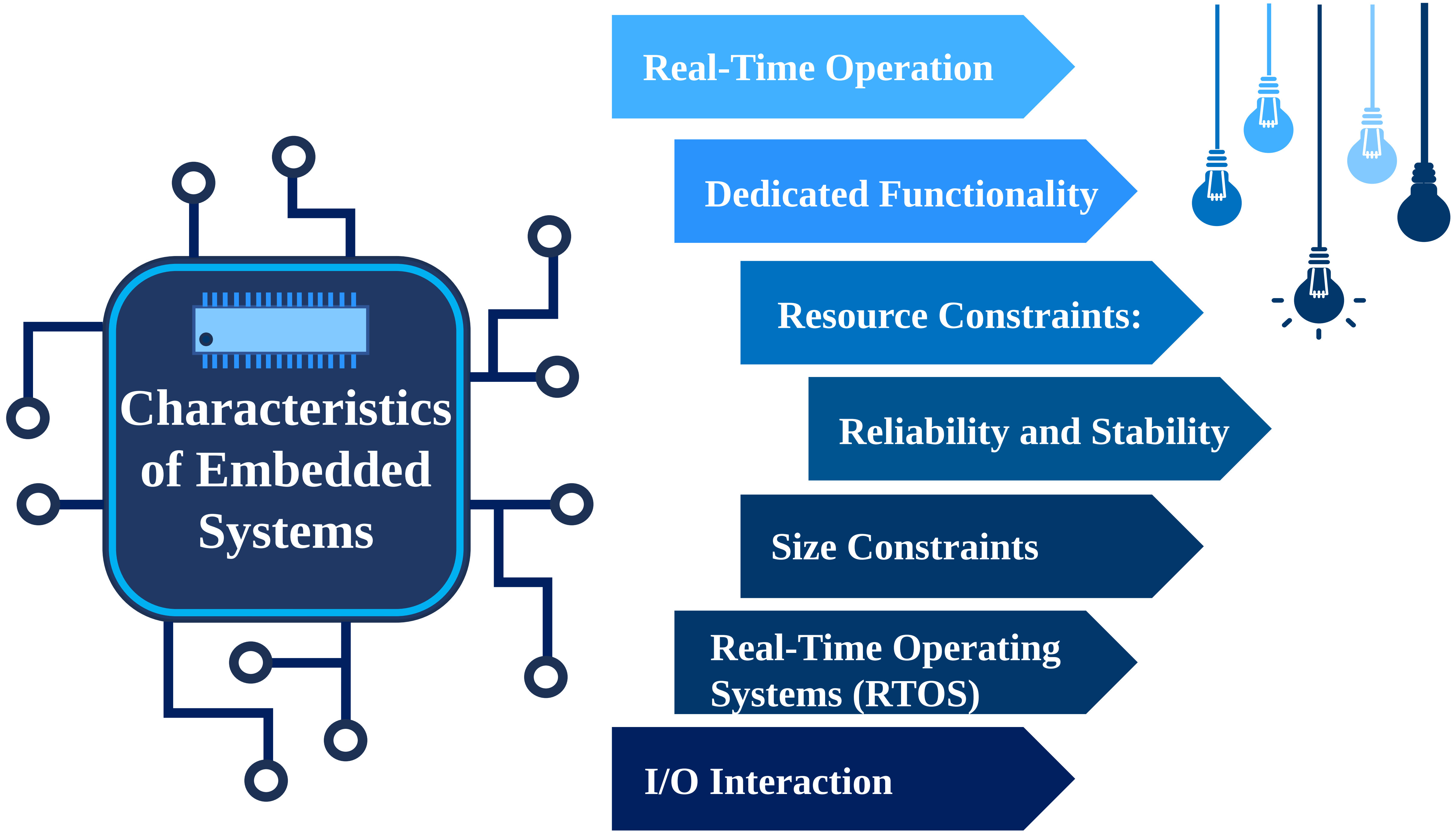
Getting Started with Embedded Systems

What is an Embedded system ?

Embedded systems are computing devices that are designed for specific tasks or functions within a larger system. They are embedded as part of a complete device, often with real-time computing constraints and limited resources.

Examples of Embedded Systems in everyday life ?





Characteristics of Embedded Systems
Real-Time Operation
Dedicated Functionality
Resource Constraints:
Reliability and Stability
Size Constraints
Real-Time Operating Systems (RTOS)
I/O Interaction

Applications of Embedded Systems

01

Manufacturing Equipment

02

Domestic Appliances

03

Audio/Video Equipment

04

Gamming

05

Telecommunication

06

Medical Devices

07

Cars And Vehicles

08

Sensor Integration

Types of Embedded Systems

1. Based on Application

- Real Time
- Stand alone
- Networked
- Mobile

2. Based on Performance of Microcontroller

- Small Scale
- Large Scale
- Sophisticated

3. Based on Complexity

- Hard-Real Time
- Soft Real Time

4. Based on Functional Requirements

- Control Systems
- Monitoring Systems
- Data Acquisition Systems

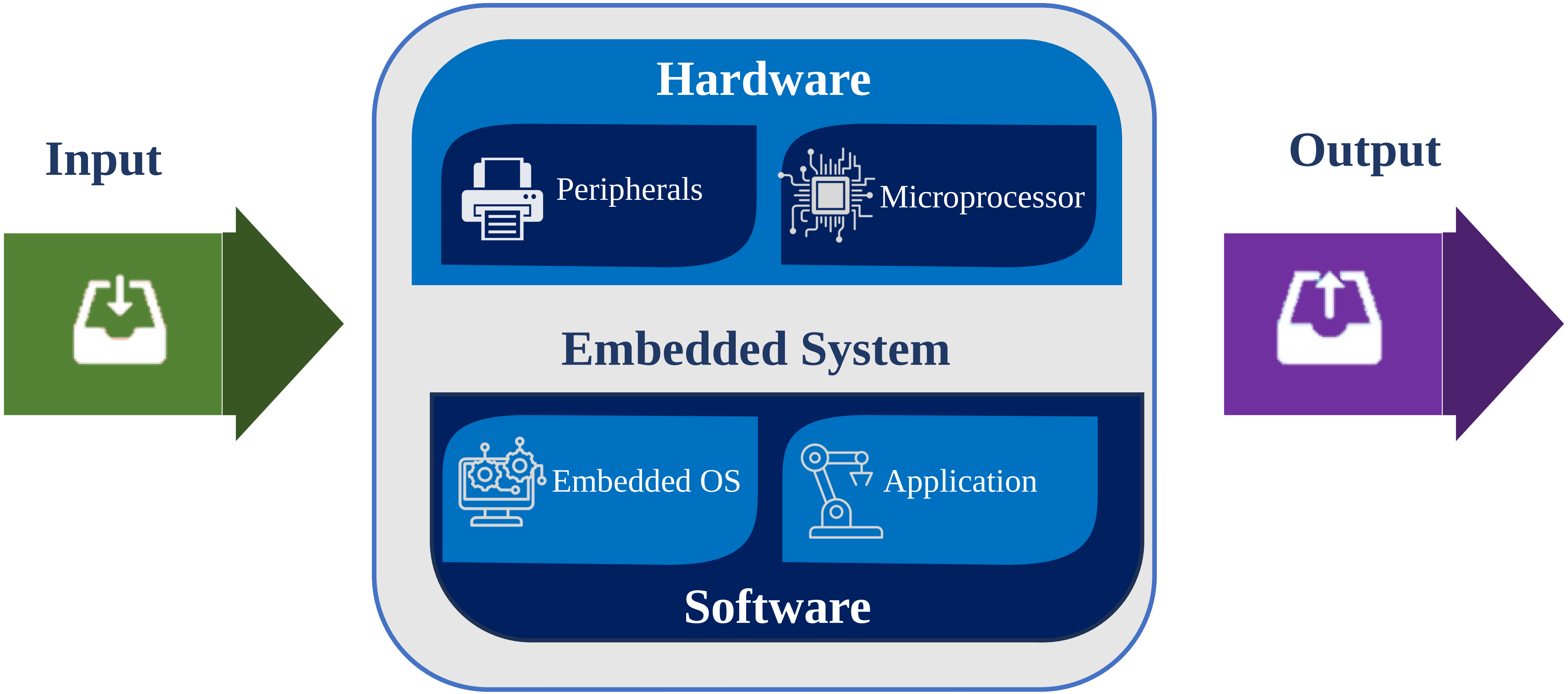
EMBEDDED WORLD IS RUNNING

Now **30 billion** embedded systems are in operation **worldwide**

70% of all **medical devices** use embedded systems

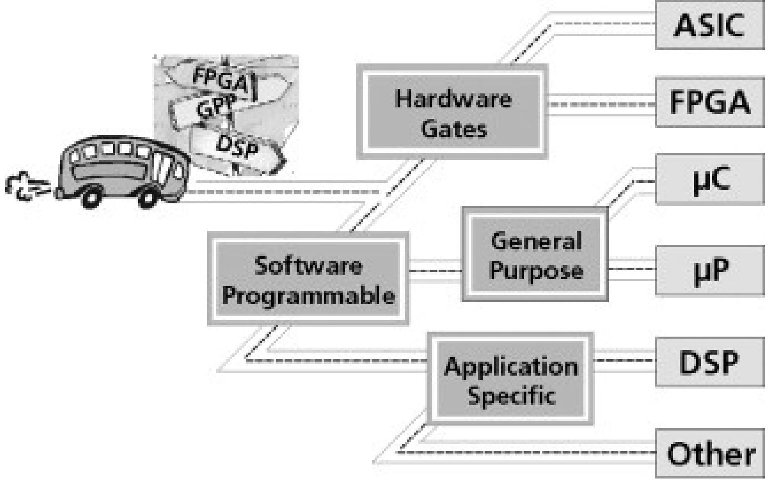
Most IoT devices contain embedded systems

Basic Embedded Architecture



Contents

Microprocessors and Microcontrollers



Microprocessor:

A central processing unit (CPU) on a single integrated circuit (IC) designed to perform general-purpose computation.

Key Characteristics: High processing power, requires external components for I/O, memory, and storage.

Examples: Intel Core, AMD Ryzen

Microcontrollers:

An integrated circuit designed to perform specific control functions, containing a CPU, memory, and I/O peripherals on a single chip.

Key Characteristics: Compact, low power, designed for specific tasks.

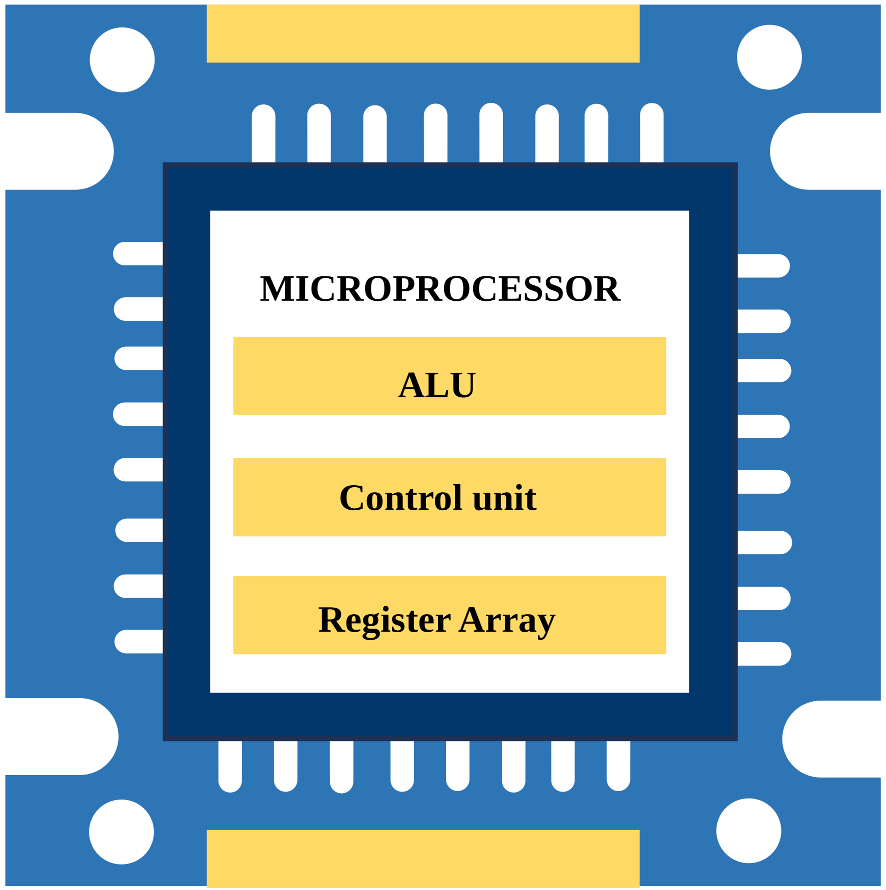
Examples: Arduino (ATmega328), PIC (Microchip PIC16F877A)

Key Differences

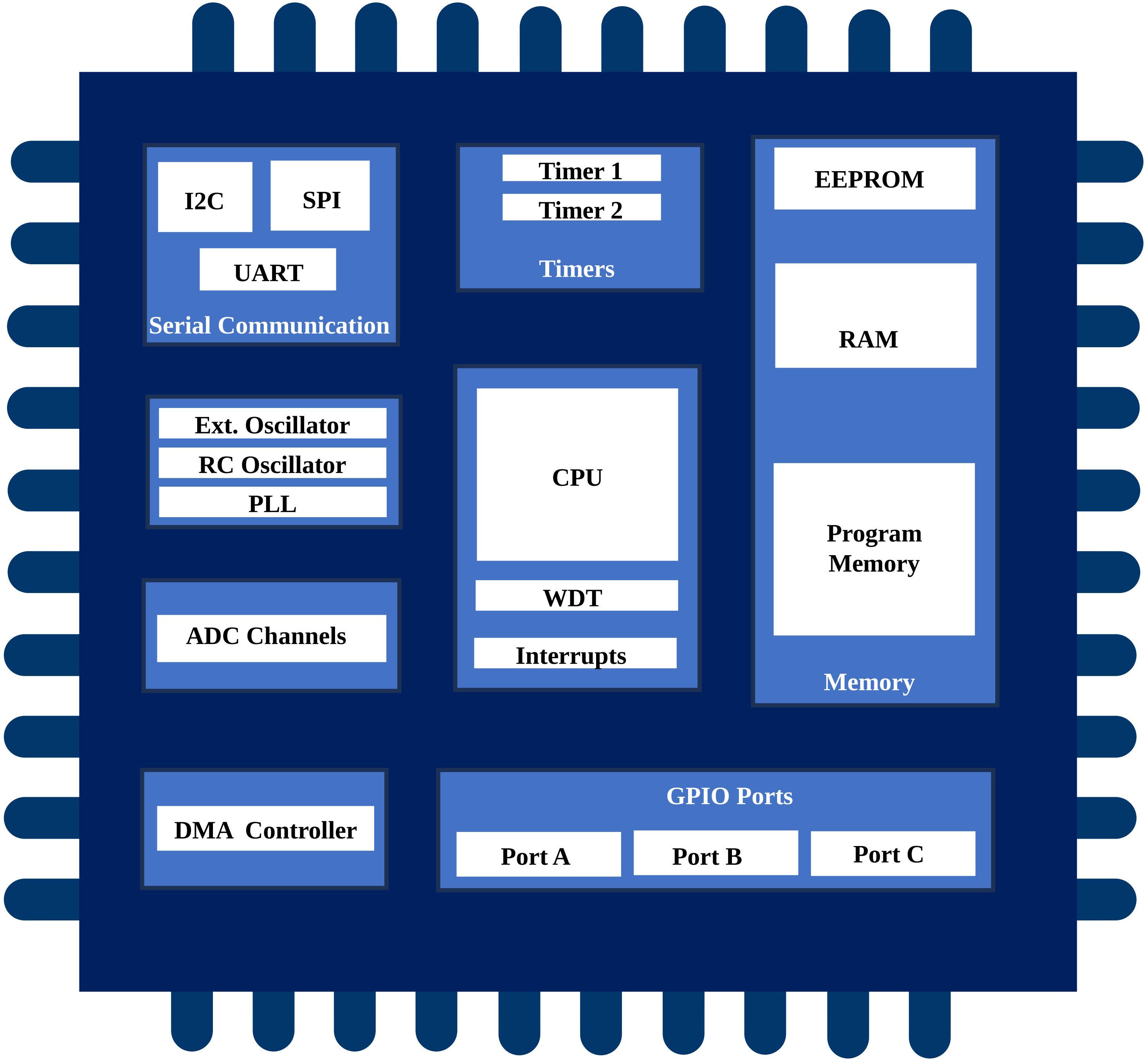
- **Architecture**:
  - Microprocessors: CPU only, needs external components.
  - Microcontrollers: CPU, memory, I/O peripherals integrated.
- **Usage**:
  - Microprocessors: General-purpose computing, PCs, servers.
  - Microcontrollers: Embedded systems, appliances, automotive.
- **Power Consumption**:
  - Microprocessors: Higher power consumption.
  - Microcontrollers: Lower power consumption.
- **Complexity and Cost**:
  - Microprocessors: More complex, higher cost.
  - Microcontrollers: Simpler, cost-effective for specific tasks.

Major Units in Computer Architecture

- **Memory Management Unit (MMU)**
  - Purpose: Translates virtual addresses to physical addresses, handles memory protection, and manages virtual memory.
  - Integration: Essential for supporting sophisticated memory management required by modern operating systems.
- **Bus System**
  - Purpose: Facilitates communication between the CPU, memory, and peripherals.
  - Components:
    - Address Bus: Carries memory addresses.
    - Data Bus: Transfers data.
    - Control Bus: Sends control signals.
  - Integration: Crucial for ensuring all parts of the computer system can communicate effectively.
- **Input/Output (I/O) System**
  - Purpose: Manages data flow between the CPU and external devices.
  - Components:
    - I/O Controllers: Interfaces that manage the interaction between the system and peripheral devices.
    - I/O Ports: Connection points for external devices.
  - Integration: Often considered part of the bus system, as it involves communication pathways.
- **System Software and Firmware**
  - BIOS/UEFI: Initializes hardware and provides a runtime environment for the operating system.
  - Operating System: Manages hardware resources and provides services for application software.
  - Firmware: Low-level software embedded in hardware to control device-specific functions.



MICROPROCESSOR
ALU
Control unit
Register Array



I2C
SPI
UART
Serial Communication
Timer 1
Timer 2
Timers
EEPROM
RAM
Ext. Oscillator
RC Oscillator
PLL
CPU
Program Memory
WDT
ADC Channels
Interrupts
Memory
DMA Controller
GPIO Ports
Port A
Port B
Port C

Microprocessors and Microcontrollers

Applications of Microprocessors

- Personal Computers
- Servers and Workstations
- Gaming Consoles
- Smartphones and Tablets
- High-performance computing systems

Applications of Microcontrollers

- Home Appliances (Microwaves, Washing Machines)
- Automotive Systems (Engine Control Units, Airbags)
- Consumer Electronics (Remote Controls, Toys)
- Industrial Automation (Robotic Controls, Sensors)
- IoT Devices (Smart Home Devices, Wearables)

Architecture Comparison

Key Components: ALU, Control Unit, Registers, Memory (RAM/ROM), I/O Ports, Timers

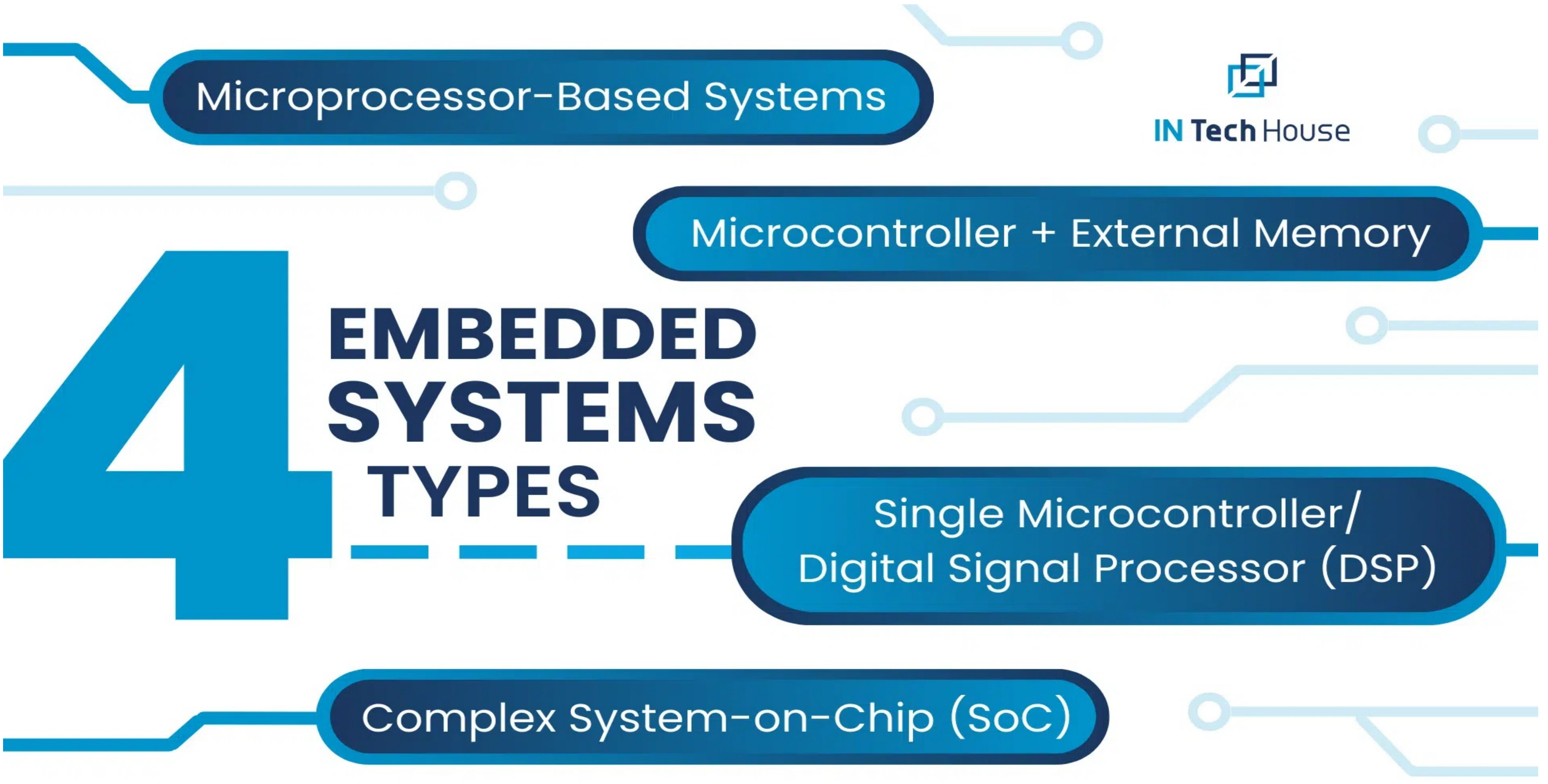
Development Tools

- Microprocessors: Compilers, Debuggers, IDEs (e.g., GCC, Visual Studio)
- Microcontrollers: Integrated Development Environments (IDEs), Simulators, Debuggers (e.g., MPLAB, Keil uVision, Arduino IDE)

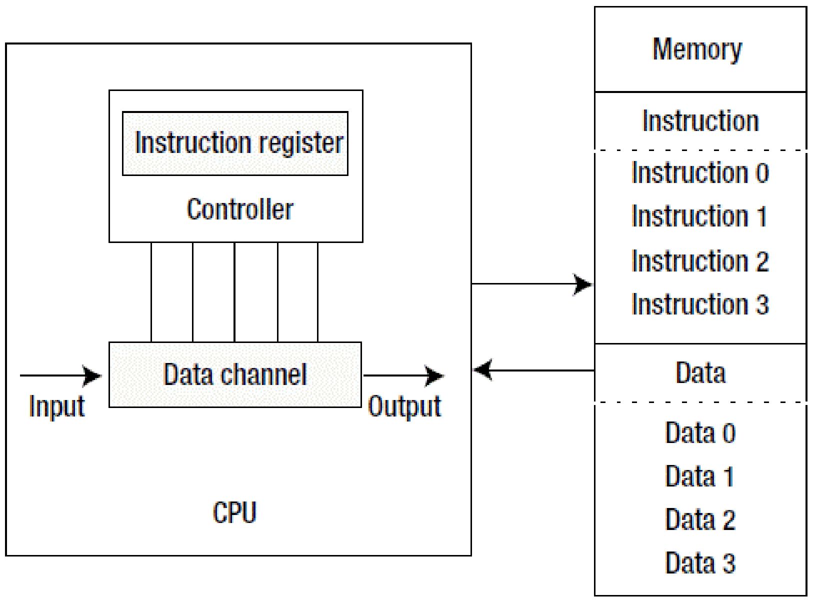
Trends and Future Directions

Increasing Integration and Miniaturization AI and Machine Learning Integration IoT and Edge Computing Low Power and Energy-Efficient Designs

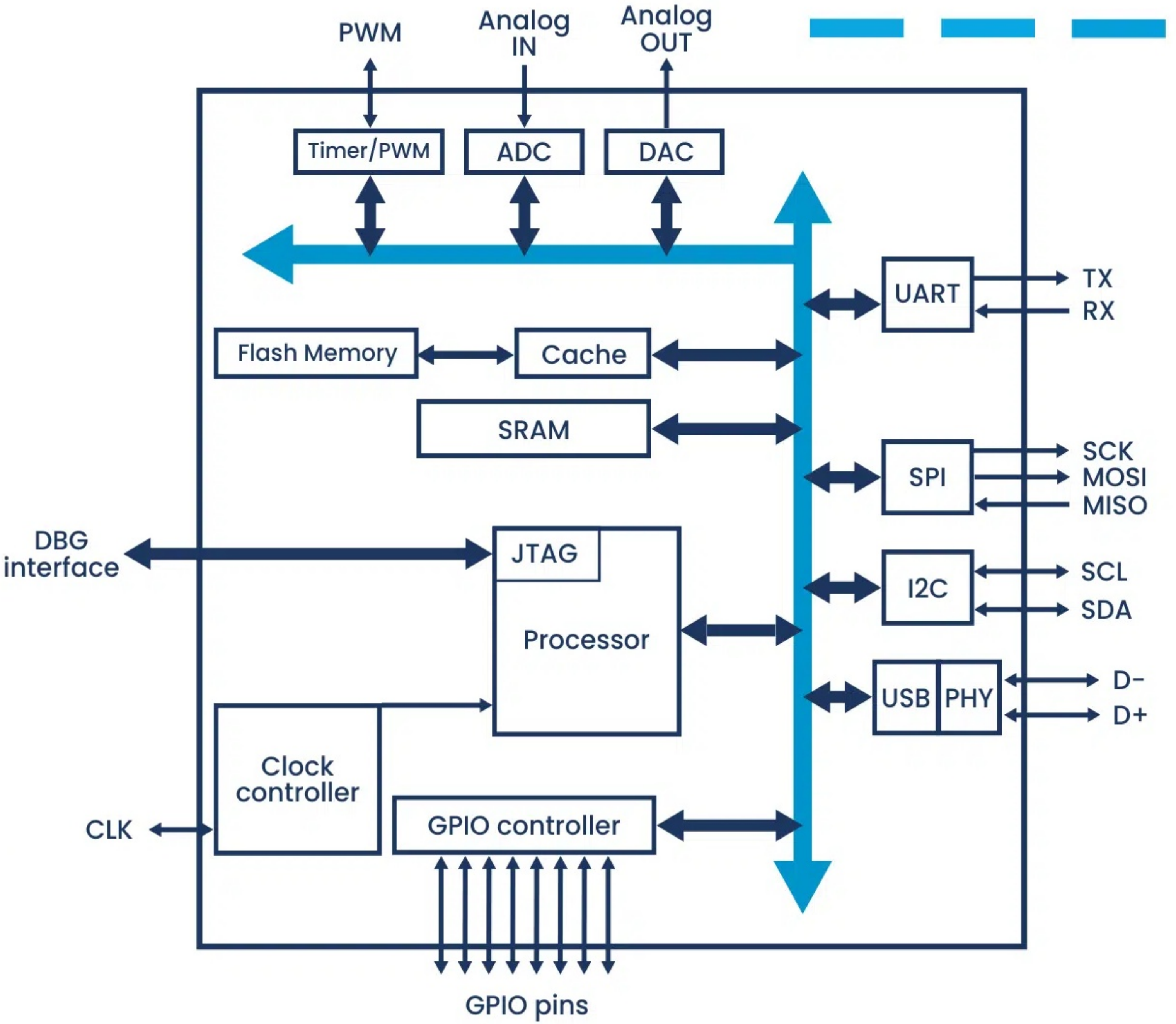
Contents



https://intechhouse.com/blog/embedded-systems-architecture/

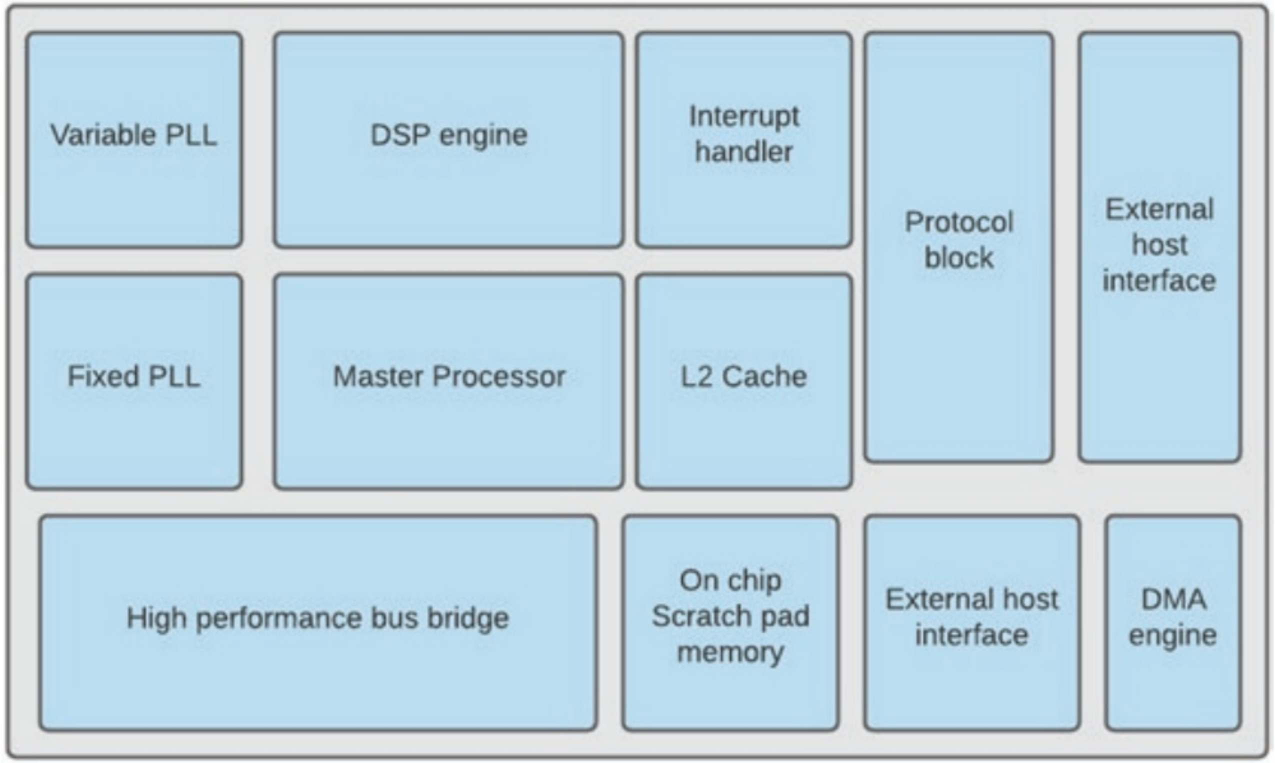


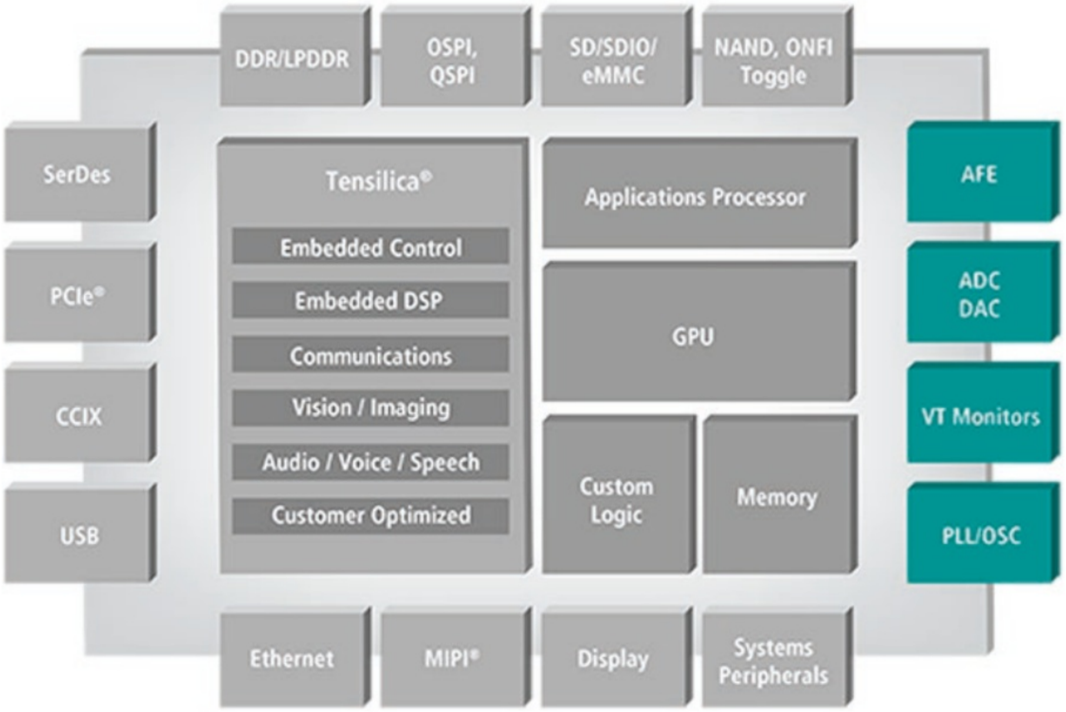
Instruction register
Controller
Data channel
Input
Output
CPU
Memory
Instruction
Instruction 0
Instruction 1
Instruction 2
Instruction 3
Data
Data 0
Data 1
Data 2
Data 3



PWM
Analog IN
Analog OUT
Timer/PWM
ADC
DAC
UART
TX
RX
Flash Memory
Cache
SRAM
SPI
SCK
MOSI
MISO
DBG interface
JTAG
Processor
I2C
SCL
SDA
USB
PHY
D-
D+
Clock controller
CLK
GPIO controller
GPIO pins

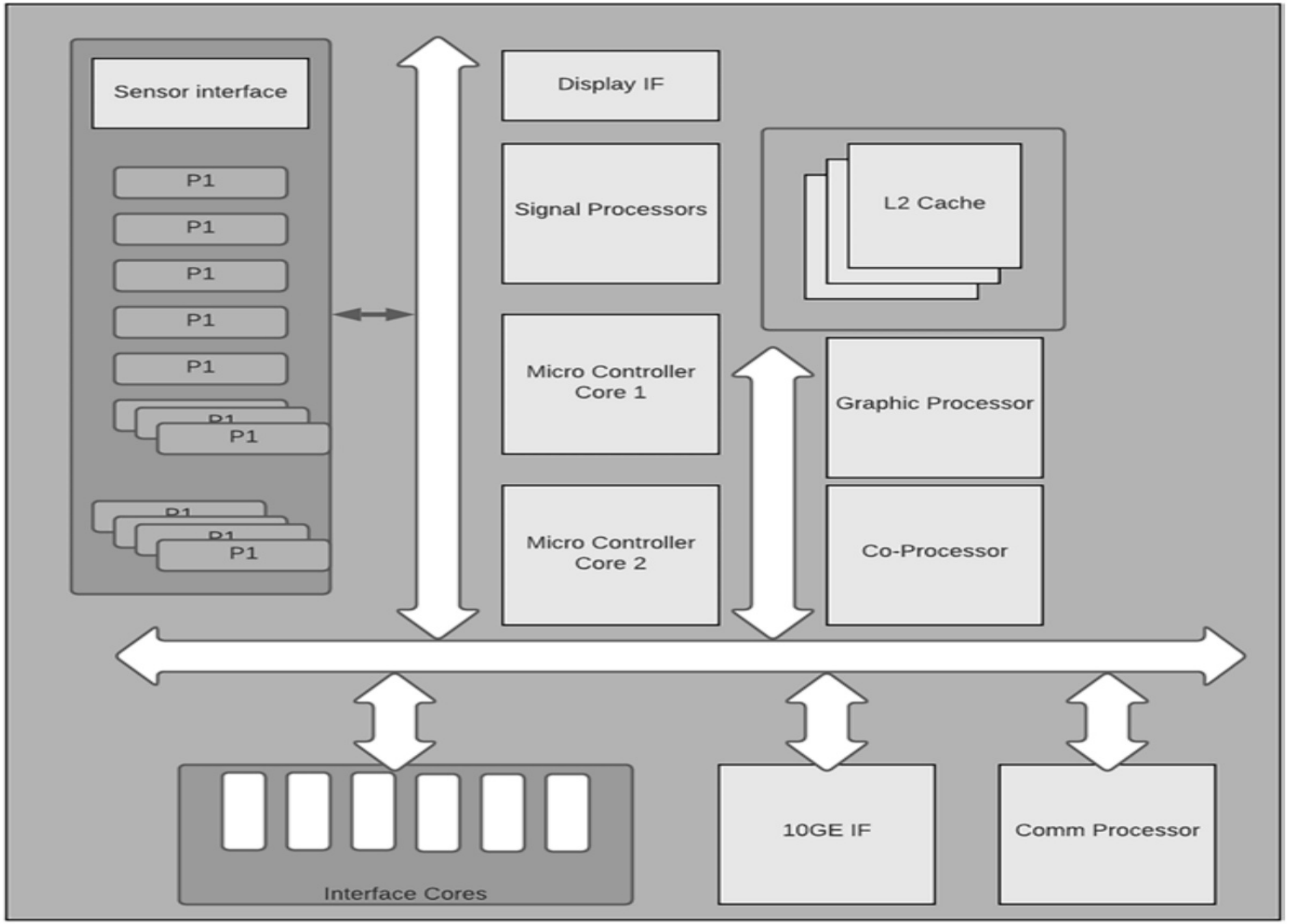
HPC Embedded Systems





Embedded System Key Components

- **Processor**
- **ISA**
- Internal Bus
- Memory Unit
- Power
- Scheduling
- Input / Output
- DMA



Processor

- RISC (Reduced Instruction Set Computing):
- Common ISAs include ARM and RISC-V.
- Simplified instructions for efficient execution and low power consumption.

Internal Bus: System on Chip (SoC)

- **Integrated Components:**
- Combines CPU, memory, I/O ports, and other peripherals on a single chip.
- Reduces physical space and power consumption.
- **Peripheral Integration:**
- Includes components such as ADCs, DACs, timers, PWM controllers, and communication interfaces (UART, SPI, I2C, etc.).

Memory Unit

- **On-Chip Memory:**
- Typically includes SRAM and ROM/Flash memory.
- Fast access times for real-time performance.
- **External Memory Interfaces:**
- Support for connecting to external memory modules like DRAM or NOR/NAND Flash.

SRAM (Static RAM)

Purpose: Temporary storage for variables and data during execution.
Characteristics: Volatile, fast access.
Usage: Stores variables, stack, and temporary data.

EEPROM

Purpose: Stores data that must persist across power cycles.
Characteristics: Non-volatile, byte-addressable.
Usage: Saves user settings and calibration data.

Flash Memory (Program Memory)

Purpose: Stores the firmware or program code.
Characteristics: Non-volatile, reprogrammable.
Usage: Holds the application code and bootloader.

Scheduling

- **Real-Time Operating System (RTOS) Support:**
- Features for deterministic execution and low-latency interrupts.
- **Dedicated Timers and Counters:**
- Hardware support for precise timing operations.
- **Interrupt Handling:**
- Fast and efficient interrupt processing capabilities.

- Power Consumption

- **Power Management Features:**
- Multiple power modes (active, idle, sleep, deep sleep).
- Dynamic voltage and frequency scaling (DVFS).
- **Efficient Instruction Execution:**

Instructions optimized for minimal power use per operation.

I/O and Communication Interfaces

- **Integrated Communication Peripherals:**
- Support for serial communication protocols like UART, SPI, I2C, CAN, and USB.
- **GPIO (General-Purpose Input/Output) Pins:**
- Configurable pins for direct hardware interfacing and control.

External Buses Low Performance

Increasing demand for high-speed data transfer rates

- Growing need for low latency and high throughput

- Scalability and flexibility requirements

- Advancements in technology and cost reductions

- **UART (Universal Asynchronous Receiver-Transmitter)**
  - - Theory: UART is a serial communication protocol that uses asynchronous transmission, meaning that data is transmitted one bit at a time, without a clock signal.
  - Pros:
    - Simple and easy to implement
    - Low power consumption
    - Widely used in serial communication applications
  - Cons:
    - Limited data transfer rate (typically up to 115.2 kbps)
    - No built-in error detection or correction
    - Not suitable for high-speed or real-time applications

SPI (Serial Peripheral Interface)

SPI is a serial communication protocol that uses synchronous transmission, meaning that data is transmitted with a clock signal.

- Pros:
  - Full-duplex communication (simultaneous read and write)
  - High data transfer rates (up to 100 Mbps or more)
  - Simple and easy to implement
- Cons:
  - Requires four wires (MOSI, MISO, SCK, SS)
  - No built-in error detection or correction
  - Can be prone to noise and interference issues

I2C (Inter-Integrated Circuit)

- Theory: I2C is a serial communication protocol that uses synchronous transmission, meaning that data is transmitted with a clock signal.

- Pros:
  - Multi-master, multi-slave communication
  - Built-in error detection and correction (ACK/NAK protocol)
  - Widely used in microcontroller and sensor applications
- Cons:
  - Limited data transfer rate (up to 400 kbps in standard mode, 3.4 Mbps in fast mode)
  - More complex than UART, requiring more pins and logic
  - Can be prone to noise and interference issues

Direct Memory Access (DMA) in Microcontrollers

Offload CPU:
DMA allows the CPU to delegate data transfer tasks to the DMA controller, freeing up the CPU to perform other processing tasks.

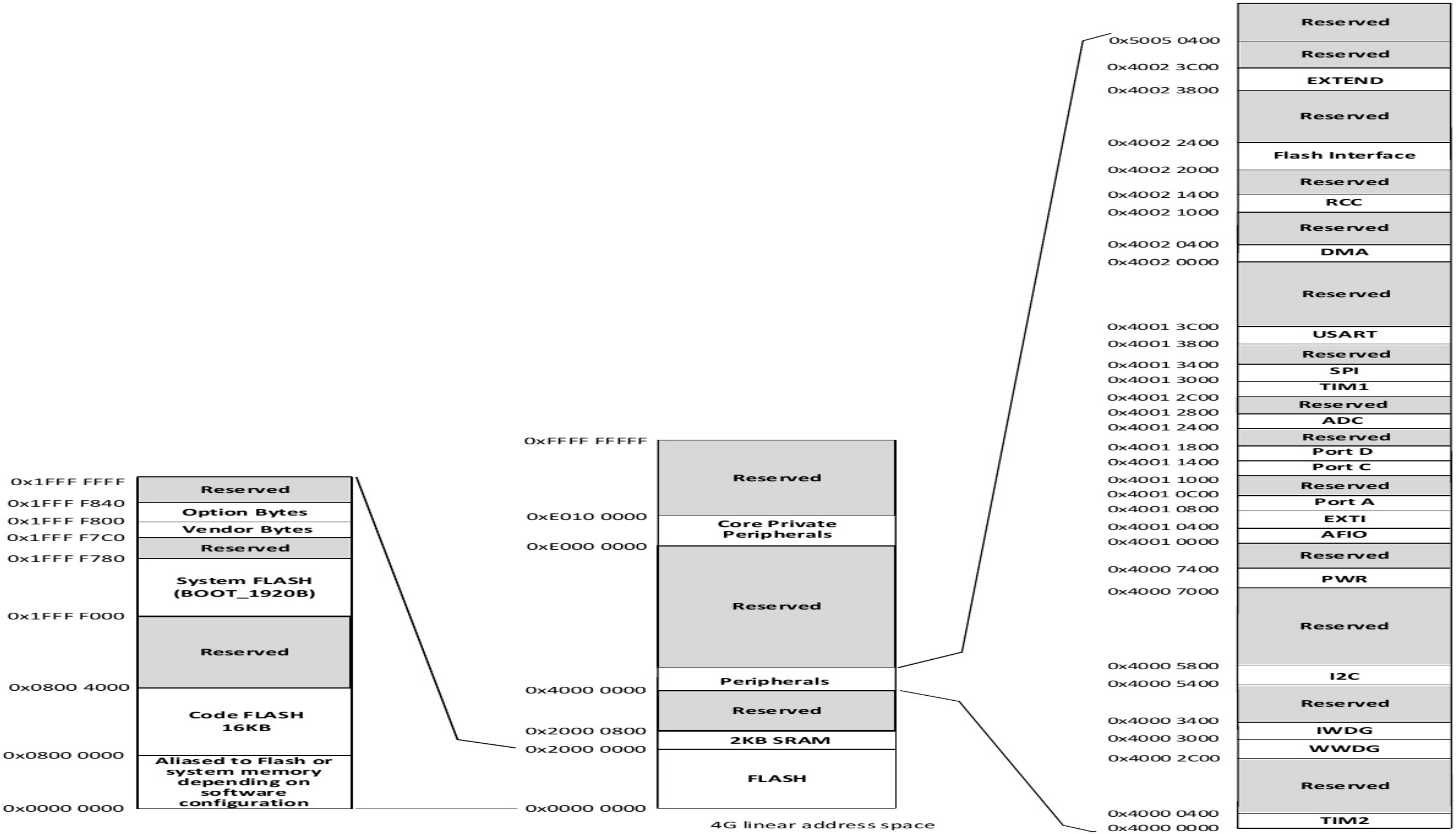
Efficient Data Transfer:
DMA enables high-speed data transfers directly between memory and peripherals without CPU intervention for each data byte or word, improving overall system efficiency.

Extended Address Space:
DMA can handle block transfers using a single address setup, efficiently moving large amounts of data and effectively increasing the addressable data space.

Contents

Memory Mapping

- Memory mapping is a crucial aspect of System on Chip (SoC) architecture. It refers to the way different components of the SoC are allocated addresses in the memory space. This mapping allows the CPU and other components to access and interact with various parts of the system's memory and peripherals.
- **Key Aspects of Memory Mapping in SoCs.**
  - Bus System
  - Address Space Allocation
  - Memory Regions
  - Memory Mapping Techniques:
  - Memory Map Tables
  - Access Mechanisms
  - Virtual Memory Mapping
  - Address Decoding:
- **Example of Memory Mapping in an SoC**
  - 0x0000_0000 - 0x1FFF_FFFF: RAM (1 GB of addressable RAM)
  - 0x2000_0000 - 0x3FFF_FFFF: ROM or Flash memory
  - 0x4000_0000 - 0x5FFF_FFFF: Peripheral registers (e.g., GPIO, UART)
  - 0x6000_0000 - 0x7FFF_FFFF: External memory or memory-mapped I/O space


0x1FFF FFFF
0x1FFF F840
0x1FFF F800
0x1FFF F7C0
0x1FFF F780
0x1FFF F000
0x0800 4000
0x0800 0000
0x0000 0000
Reserved
Option Bytes
Vendor Bytes
Reserved
System FLASH (BOOT_1920B)
Reserved
Code FLASH 16KB
Aliased to Flash or system memory depending on software configuration
0xFFFF FFFF
0xE010 0000
0xE000 0000
0x4000 0000
0x2000 0800
0x2000 0000
0x0000 0000
Reserved
Core Private Peripherals
Reserved
Peripherals
Reserved
2KB SRAM
FLASH
4G linear address space
0x5005 0400
0x4002 3C00
0x4002 3800
0x4002 2400
0x4002 2000
0x4002 1400
0x4002 1000
0x4002 0400
0x4002 0000
0x4001 3C00
0x4001 3800
0x4001 3400
0x4001 3000
0x4001 2C00
0x4001 2800
0x4001 2400
0x4001 1800
0x4001 1400
0x4001 1000
0x4001 0C00
0x4001 0800
0x4001 0400
0x4001 0000
0x4000 7400
0x4000 7000
0x4000 5800
0x4000 5400
0x4000 3400
0x4000 3000
0x4000 2C00
0x4000 0400
0x4000 0000
Reserved
Reserved
EXTEND
Reserved
Flash Interface
Reserved
RCC
Reserved
DMA
Reserved
USART
Reserved
SPI
TIM1
Reserved
ADC
Reserved
Port D
Port C
Reserved
Port A
EXTI
AFIO
Reserved
PWR
Reserved
I2C
Reserved
IWDG
WWDG
Reserved
TIM2

Bus System Components

- Address Bus: Carries address information from the CPU to memory and peripherals. The address bus width determines the range of addresses that can be used in memory mapping.
- Data Bus: Transfers data between components based on the address specified on the address bus.
- Control Bus: Carries control signals that manage the read and write operations and other control functions.
- Functions:
  - Memory Map Configuration
  - Interconnects and Buses
  - Address Decoding
  - Memory-Mapped I/O

Example:

On-Chip Memory and Peripheral Mapping: Within the bus system, the memory map determines the layout of on-chip memory, peripheral registers, and I/O devices. The bus system ensures that the CPU and other components access the correct addresses based on this map.

Address Map/Space Allocation

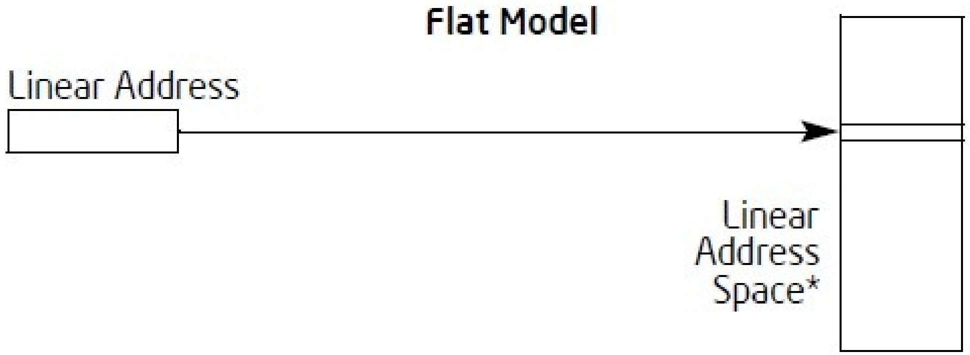
- Memory Address Space: Defines the range of addresses used to access different types of memory, including RAM, ROM, and external memory.
- Peripheral Address Space: Allocates addresses for various peripherals and I/O devices.

Memory Regions

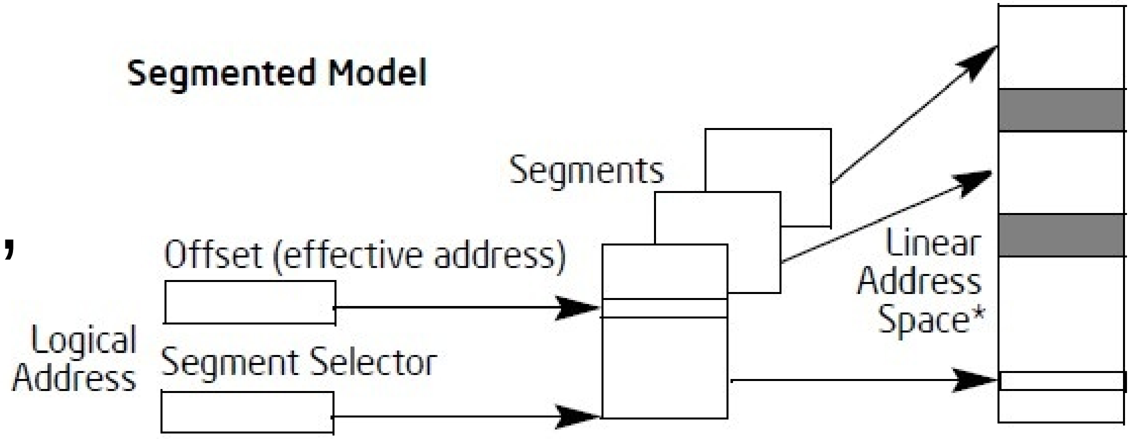
- Boot Memory: Often used to store the bootloader or initial firmware.
- Code Memory: Stores executable code and program instructions.
- Data Memory: Used for storing variables, stack, and heap data.
- Peripheral Registers: Memory-mapped addresses used to control and interact with peripheral devices (e.g., timers, UARTs, GPIOs).

Memory Mapping Techniques:

- Flat Memory Model: All memory and peripherals are mapped into a single, linear address space.

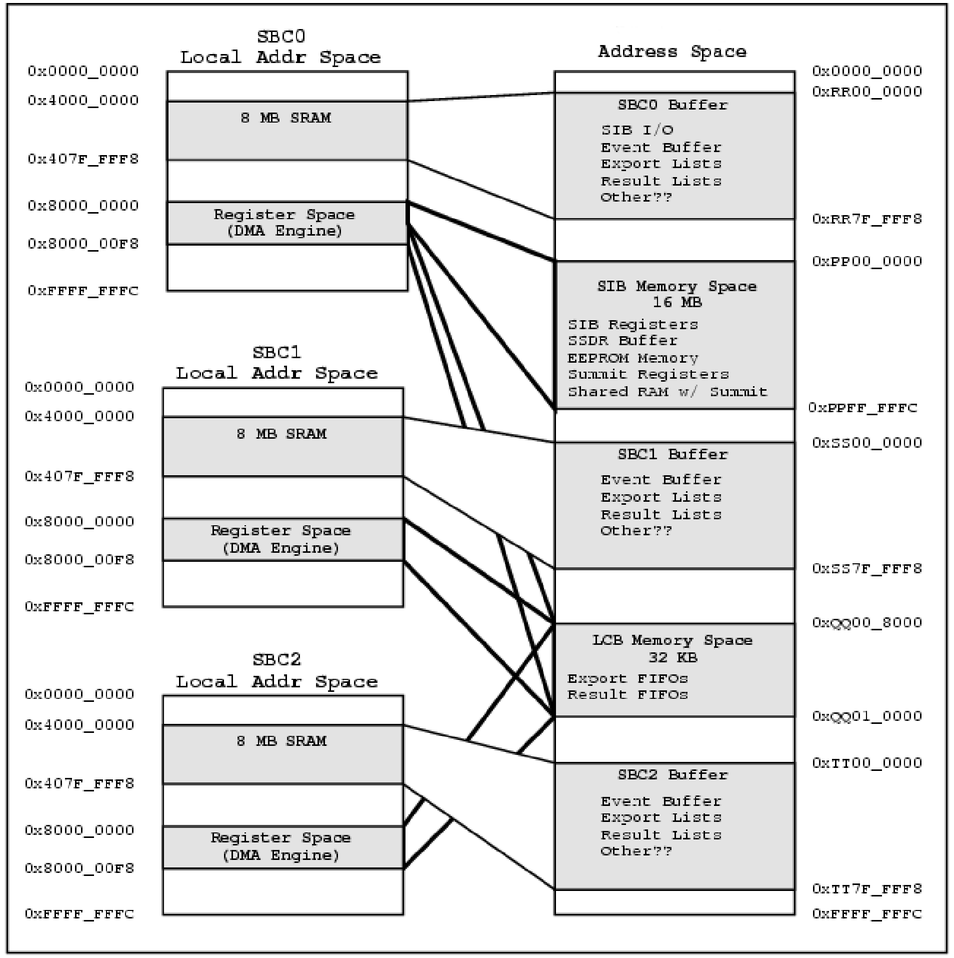


- Segmented Memory Model: Memory and peripherals are divided into segments or blocks, each with a specific address range.



Memory Map Tables and Access Mechanisms

- Memory Map Table: A detailed table that outlines the starting address, size, and type of each memory region and peripheral.
- Memory-Mapped I/O: Peripherals are accessed by reading from or writing to specific memory addresses.
- Access Mechanism:
  - Linker Script: Define how different code and data sections are placed in memory.
  - Direct Memory Access (DMA): Allows peripherals to directly access memory without CPU intervention, reducing latency and improving performance.



Virtual Memory Mapping

- Virtual Address Space: Some SoCs use virtual memory systems to abstract physical memory addresses, providing flexibility in memory management.
- Virtual Address: It is an address of a program's memory space.
- Page Table: The table contains mappings from virtual addresses to physical addresses. Each entry in the page table corresponds to a "page" of memory.
- Page Size: Memory is divided into fixed-size pages, typically ranging from 2 KB to 16 KB (though sizes like 4 KB or 8 KB are common). The virtual address is split into two parts:
  - Page Number: Identifies the page within the virtual address space.
  - Offset: Identifies the specific location within the page.
- Translation: When a virtual address is used, the page number is looked up in the page table to find the corresponding physical page. The offset is then added to this physical page to get the final physical address.
- Physical Address: The final physical address points to the exact location in the system's memory (RAM) where the data is stored.

Contents

Embedded System Clock Tree

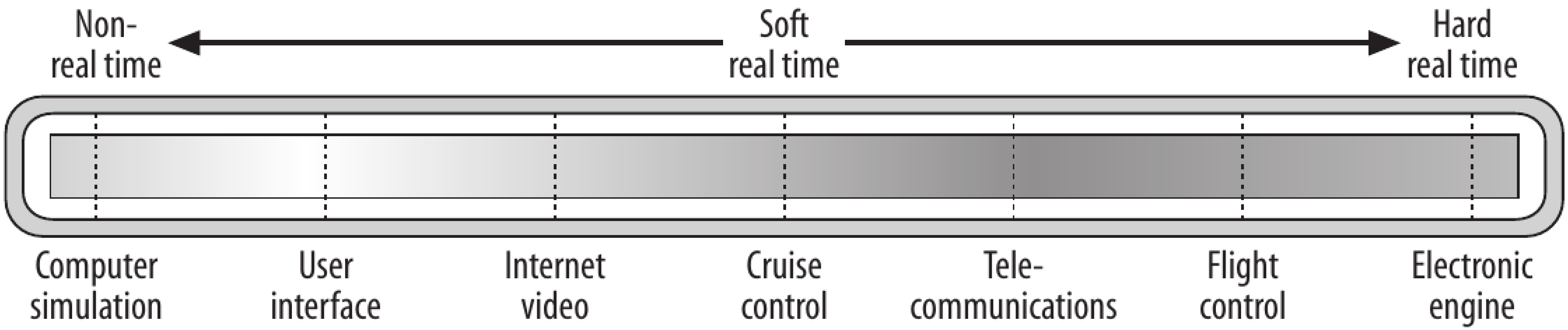
- It is responsible for distributing clock signals throughout the system. It ensures that all components receive accurate and synchronized timing signals necessary for proper operation. Here's a detailed look at the clock tree and its role in embedded systems:
- Purpose of the Clock Tree:
- Timing Distribution: The clock tree distributes clock signals from a central oscillator or clock source to various components and subsystems within the embedded system.
- Synchronization: Ensures that different parts of the system operate in sync, which is crucial for reliable and predictable system performance.

Components of a Clock Tree:

- Clock Source: The primary oscillator or clock generator that provides the initial clock signal.
- Clock Distributors: Distributes the clock to various parts of the system. This may include clock buffers, drivers, and multiplexers.
- Clock Dividers: Reduce the frequency of the clock signal to provide lower frequency clocks for different subsystems.
- Clock Multipliers: Increase the frequency of the clock signal if higher frequencies are required for certain components.
- Phase-Locked Loops (PLLs) and Delay-Locked Loops (DLLs): Used to generate stable, high-frequency clock signals from a lower-frequency reference clock, or to align the phase of clocks.
- Clock Gating: Mechanism to enable or disable the clock signal to specific parts of the system to save power when those parts are not in use.

Contents

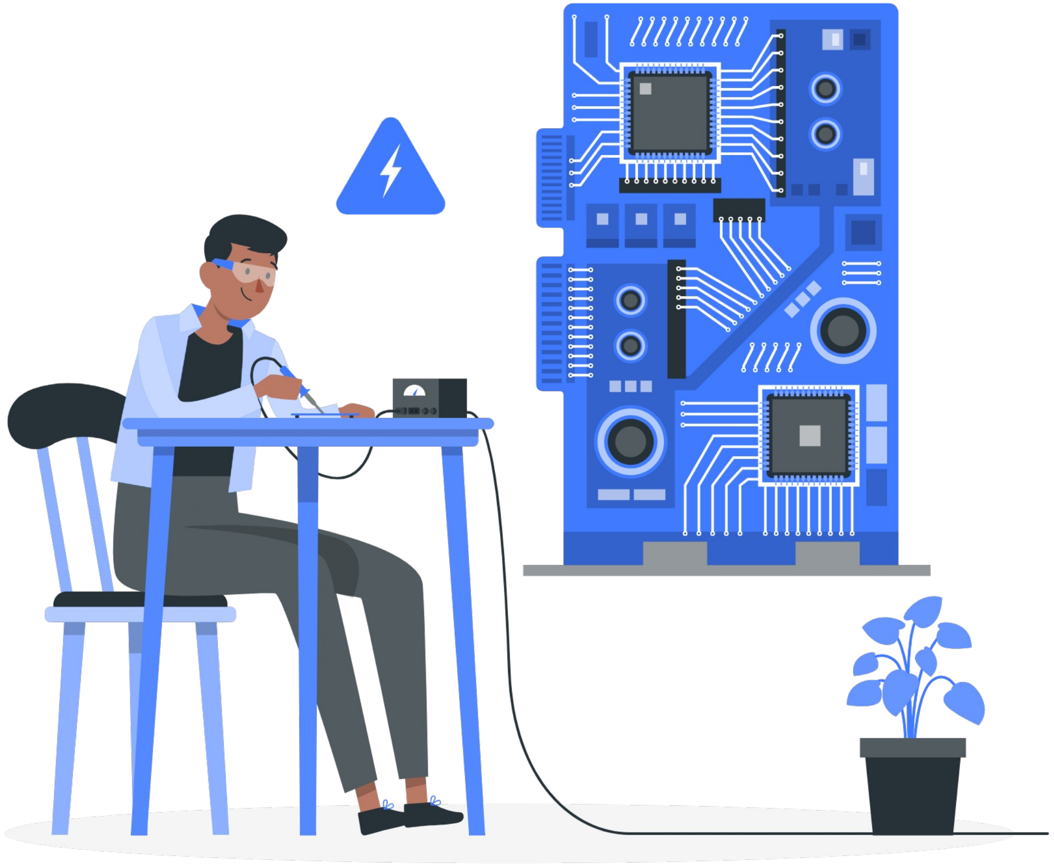
- Range of Applications



Introduction to Embedded C Programming

Embedded C

- Embedded C and standard C (often just called "C") are both programming languages used to write software, but they differ in their target environments, constraints, and some aspects of functionality.
- Embedded C can be considered as the subset of C language. It uses same core syntax as C.
- Embedded C programs need cross-compliers to compile and generate HEX code
- Embedded C is designed for embedded system programming with specific constraints, hardware interaction requirements, and specialized development tools.



Introduction to Embedded C Programming

VS

A structural and programming language used by developers to create desktop-based applications	**Target Environment**	An extension of C primarily used to develop microcontroller based applications.
Typically used on systems with more resources.	**Memory Constraint**	Often used in environments with limited resources (memory, processing power).
Hardware interactions are managed by operating system or libraries, unless used in system-level programming.	**Hardware Interaction**	Interacts directly with hardware components, such as registers, I/O ports, and peripheral devices.
Uses standard libraries provided by the C standard library (e.g., stdio.h, stdlib.h) and other platform-specific or third-party libraries.	**Libraries and Extensions**	Uses specialized libraries and extensions for embedded systems (e.g., specific APIs for handling hardware interrupts, timers, and serial communication).

Introduction to Embedded C Programming

VS

Development Tools

Typically uses general-purpose IDEs (e.g., Visual Studio, Eclipse) and compilers (e.g., GCC, Clang).

Specific Integrated Development Environments (IDEs), compilers, and debuggers designed for embedded system development (e.g., Keil, IAR, MPLAB).

Real-Time Constraint

It can be used in real-time applications, but it is not inherently designed for real-time constraints and may rely on external real-time extensions or operating systems.

Often used in real-time systems where meeting timing constraints is crucial. It may include real-time operating systems (RTOS) or bare-metal programming.

Code Portability

Code is generally more portable across different platforms, adhering to the C standard.

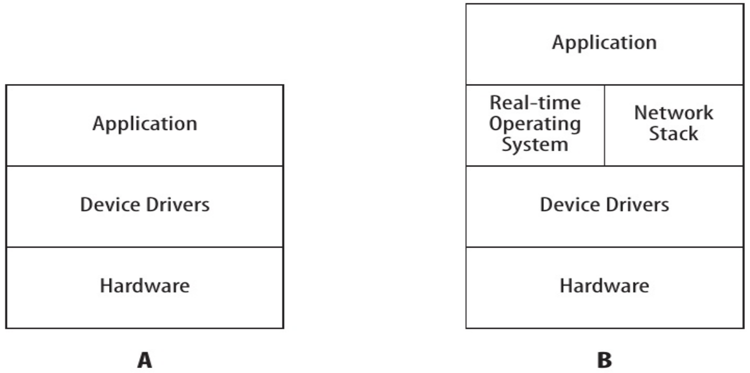
Code is often less portable due to hardware-specific dependencies and optimizations. Porting code between different embedded platforms can be challenging.

- Target Hardware Architecture:
  - Processor and Specifications:
  - Program Memory and Data Memory Size:
  - Peripherals and Components
- Memory Mapping
- Software Development
  - GCC Compiler: Compiler: riscv32-unknown-elf-gcc or riscv64-unknown-elf-gcc.
  - Debugger: GDB with RISC-V support.
  - ELF Loader: OpenOCD or RISC-V Proxy Kernel.

Stress Checking and Profiling Tools for RISC-V:

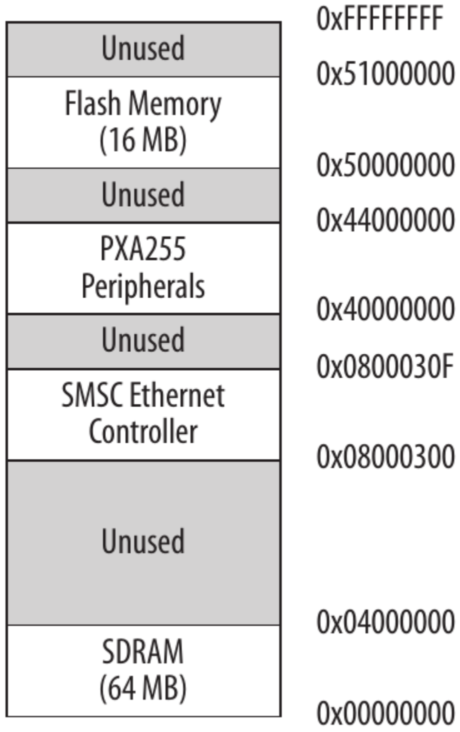
- RISC-V Performance Monitor or Perf.

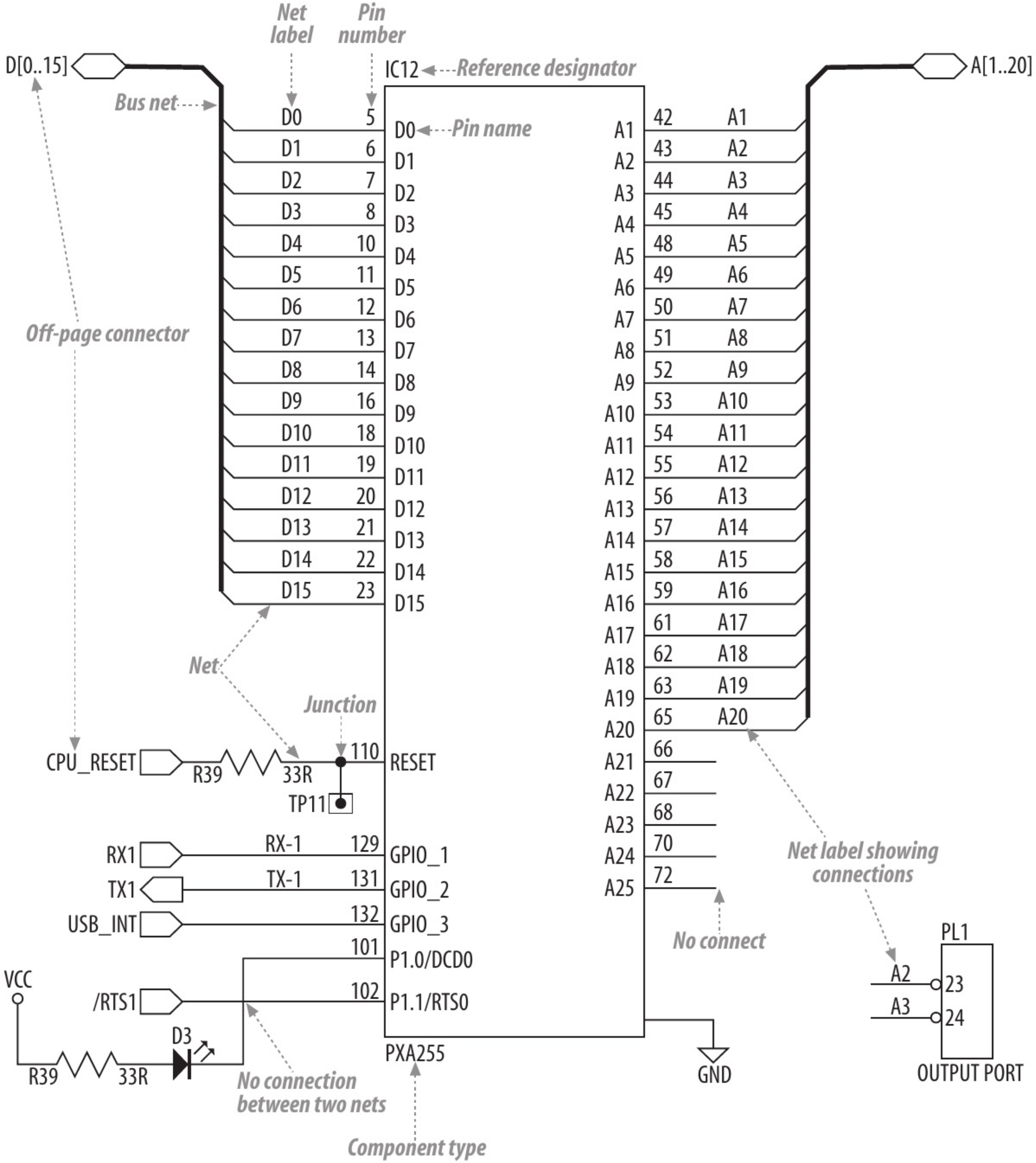
Requirements: Basic and Complex



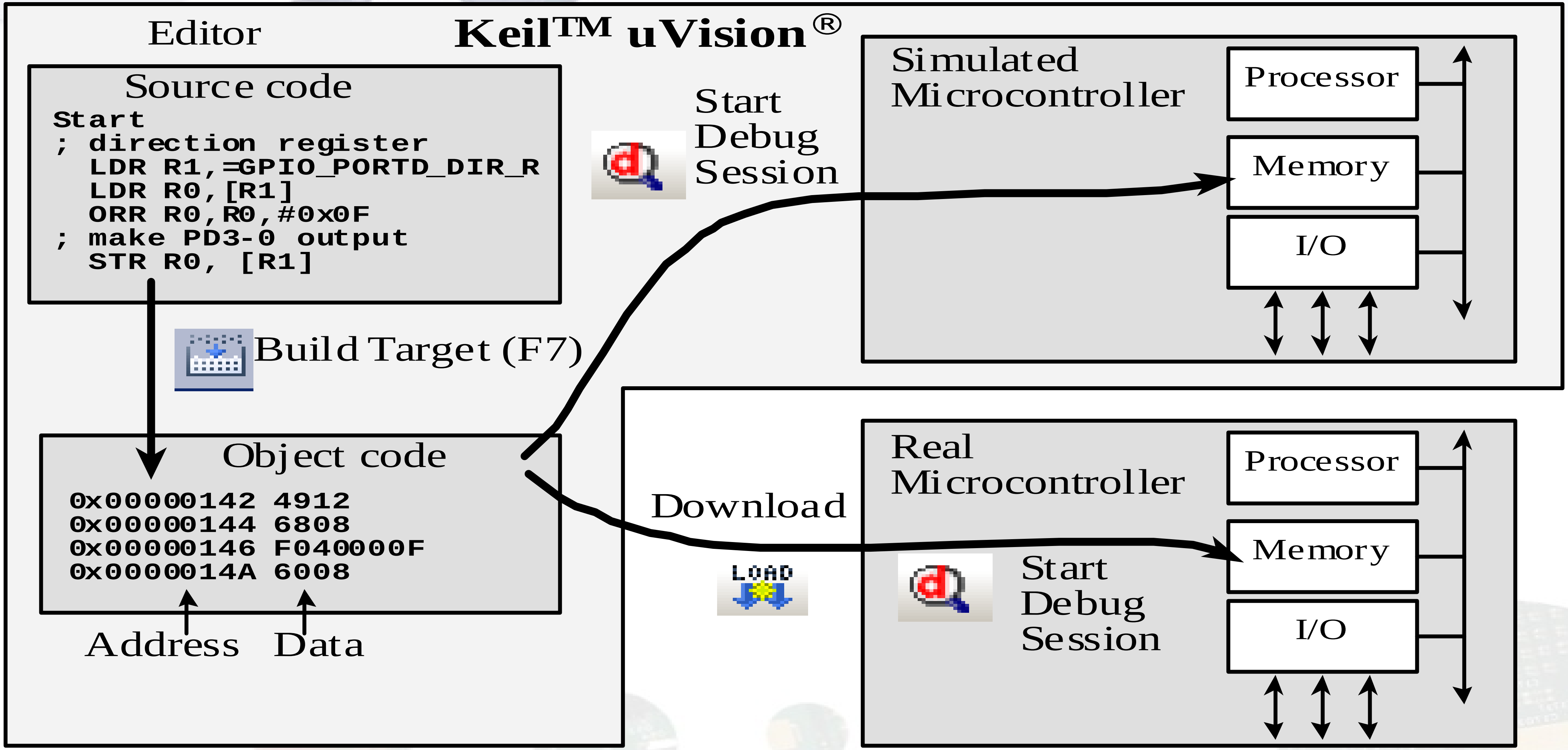
Criterion	Low	Medium	High
Processor	4- or 8-bit	16-bit	32- or 64-bit
Memory	< 64 KB	64 KB to 1 MB	> 1 MB
Development cost	< $100,000	$100,000 to $1,000,000	> $1,000,000
Production cost	< $10	$10 to $1,000	> $1,000
Number of units	< 100	100 to 10,000	> 10,000
Power consumption	> 10 mW/MIPS	1 to 10 mW/MIPS	< 1 mW/MIPS
Lifetime	Days, weeks, or months	Years	Decades
Reliability	May occasionally fail	Must work reliably	Must be fail-proof

Embedded System Schematic and Memory Mapping





SW Development Environment



Compiler Options

- riscv32-unknown-elf-gcc //
  -march=rv32imac // Architecture and ISA Extensions:
  -mabi=ilp32 // ABI (Application Binary Interface: Int, long, pointer):
  -O2 // Optimization Levels:
  -mtune=sifive-e31 // Code Genartion for specific RISCV core
  -g // Debugging and Profiling -pg
  mhard-float // Floating Point Options: Hard/Soft Floting point:
- -T linker_script.ld // -T: Specify a linker script.
  -I/path/to/include // Include Paths and Libraries
  -L/path/to/li //
  -o output.elf // Output file
  source.c // source file
  -lm // -lm (math library)
- -funroll-loops // Loop Unrolling option

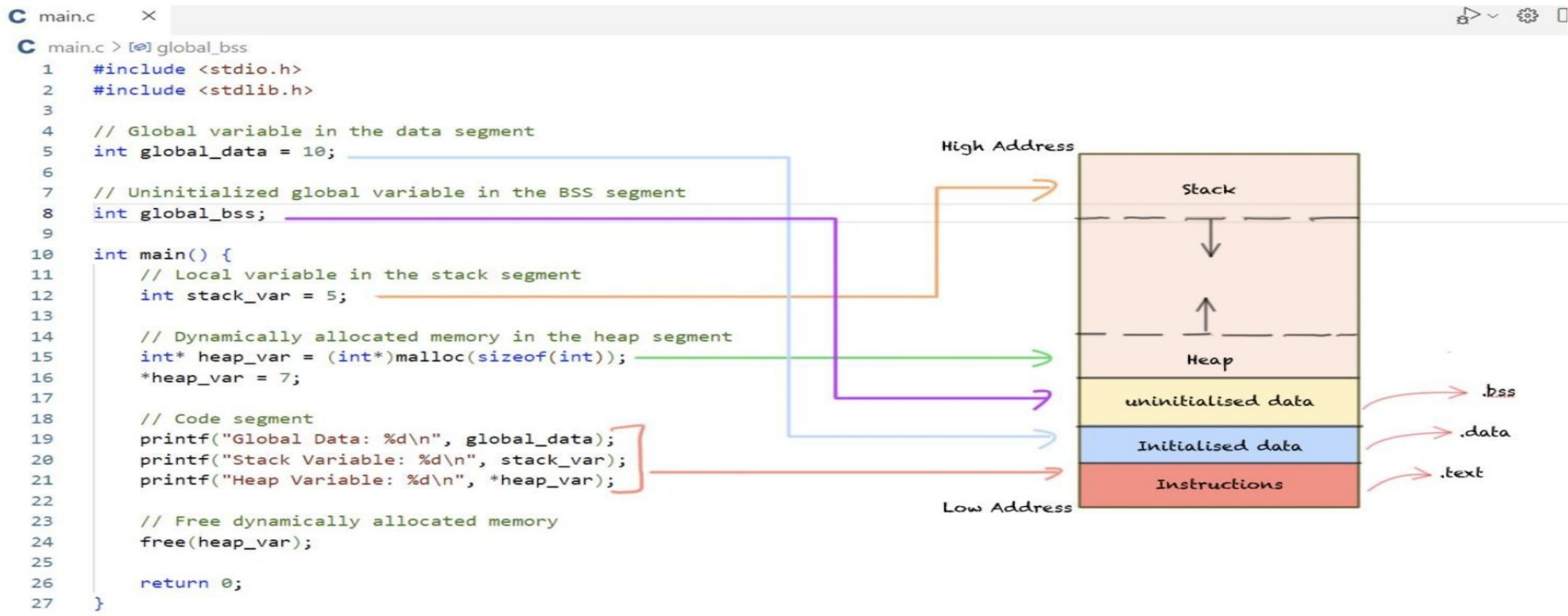
Define Memory Address

```
/* Timer Registers */
#define TIMER_0_MATCH_REG          (*((uint32_t volatile *)0x40A00000))
#define TIMER_1_MATCH_REG          (*((uint32_t volatile *)0x40A00004))
#define TIMER_2_MATCH_REG          (*((uint32_t volatile *)0x40A00008))
#define TIMER_3_MATCH_REG          (*((uint32_t volatile *)0x40A0000C))
#define TIMER_COUNT_REG            (*((uint32_t volatile *)0x40A00010))
#define TIMER_STATUS_REG           (*((uint32_t volatile *)0x40A00014))
#define TIMER_INT_ENABLE_REG       (*((uint32_t volatile *)0x40A0001C))

/* Timer Interrupt Enable Register Bit Descriptions */
#define TIMER_0_INTEN              (0x01)
#define TIMER_1_INTEN              (0x02)
#define TIMER_2_INTEN              (0x04)
#define TIMER_3_INTEN              (0x08)

/* Timer Status Register Bit Descriptions */
#define TIMER_0_MATCH              (0x01)
#define TIMER_1_MATCH              (0x02)
#define TIMER_2_MATCH              (0x04)
#define TIMER_3_MATCH              (0x08)

/* Interrupt Controller Registers */
#define INTERRUPT_PENDING_REG      (*((uint32_t volatile *)0x40D00000))
#define INTERRUPT_ENABLE_REG       (*((uint32_t volatile *)0x40D00004))
#define INTERRUPT_TYPE_REG         (*((uint32_t volatile *)0x40D00008))

/* Interrupt Enable Register Bit Descriptions */
#define GPIO_0_ENABLE              (0x00000100)
#define UART_ENABLE                (0x00400000)
#define TIMER_0_ENABLE             (0x04000000)
#define TIMER_1_ENABLE             (0x08000000)
#define TIMER_2_ENABLE             (0x10000000)
#define TIMER_3_ENABLE             (0x20000000)
```

```
/* General Purpose I/O (GPIO) Registers */
#define GPIO_0_LEVEL_REG           (*((uint32_t volatile *)0x40E00000))
#define GPIO_1_LEVEL_REG           (*((uint32_t volatile *)0x40E00004))
#define GPIO_2_LEVEL_REG           (*((uint32_t volatile *)0x40E00008))
#define GPIO_0_DIRECTION_REG       (*((uint32_t volatile *)0x40E0000C))
#define GPIO_1_DIRECTION_REG       (*((uint32_t volatile *)0x40E00010))
#define GPIO_2_DIRECTION_REG       (*((uint32_t volatile *)0x40E00014))
#define GPIO_0_SET_REG             (*((uint32_t volatile *)0x40E00018))
#define GPIO_1_SET_REG             (*((uint32_t volatile *)0x40E0001C))
#define GPIO_2_SET_REG             (*((uint32_t volatile *)0x40E00020))
#define GPIO_0_CLEAR_REG           (*((uint32_t volatile *)0x40E00024))
#define GPIO_1_CLEAR_REG           (*((uint32_t volatile *)0x40E00028))
#define GPIO_2_CLEAR_REG           (*((uint32_t volatile *)0x40E0002C))
#define GPIO_0_FUNC_LO_REG         (*((uint32_t volatile *)0x40E00054))
#define GPIO_0_FUNC_HI_REG         (*((uint32_t volatile *)0x40E00058))
```

Embedded System Programming and Memory Layout

- Understanding C memory layout is crucial for debugging, optimizing performance, security and interfacing with low-level systems.
- **Text (Code) Segment**:
- **Data Segment**:
- **BSS Segment**:
- **Heap Segment**:
- **Stack Segment**:



• Text (Code), Data and BSS Segment:

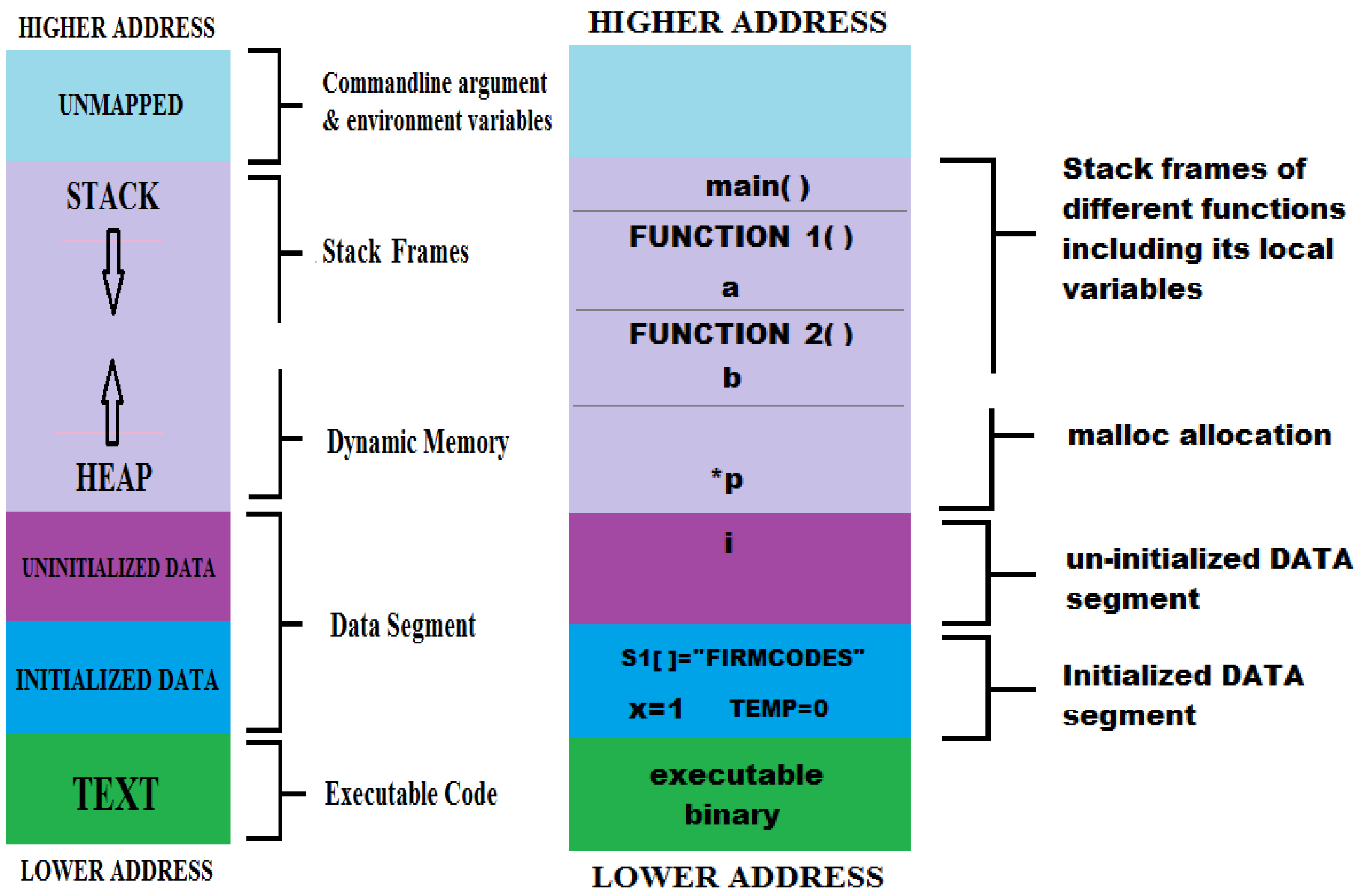
- The text segment contains the executable code of the program. It is read-only and holds the instructions for the program.
- The data segment contains initialized global and static variables. In the example code, global_data is an initialized global variable with value 10.
- The BSS (Block Started by Symbol) segment contains uninitialized global and static variables. The BSS segment is set to zero during program startup. In the example code, global_bss variable will be added to the bss section by linker.
- The Text, Data, and BSS segments collectively form the static part of the program that contains fixed-sized instructions and data that persists throughout its execution. These should be kept in a non-volatile memory to ensure successful execution of code following a power cycle.
- You can use the size utility that comes with the compiler to get the size of the executable. Below is the output for the example code:

- text data bss dec hex filename
- 1585 600 8 2193 891 main.out

Heap and Stack Segments

- **Heap Segment**:
- The heap segment is used for dynamic memory allocation during the program's runtime. In the example, we allocate memory for an integer using malloc(), and heap_var points to the newly allocated memory location.
- It's important to free the allocated memory after it is no longer needed.
- Over time, repeated memory allocation without freeing memory can cause the program's memory usage to grow unnecessarily leading to poor performance and runtime allocation failures.

- **Stack Segment**:
- The stack segment is used for managing function calls, local variables, and function call frames. In the example, stack_var is a local variable that will be allotted on the stack during the execution of the main() function.
- The stack and heap memory share the dynamic memory area of the program. The stack typically starts from the end address of the memory and grows downward, while the heap starts from the end of the BSS segment.




```c
#include<stdio.h>
#include<malloc.h>

void FUNCTION_1();
void FUNCTION_2();

char S1[]="FIRMCODES";  //initialized read-write area of DATA segment
int i;                  //uninitialized DATA segment
const int x=1;          //initialized read-only area of DATA segment

int main()
{
    static int TEMP=0;  //uninitialized DATA segment

    char *p=(char*)malloc(sizeof(char)); //Heap segment

    FUNCTION_1();       //FUNCTION_1 stack frame

    return 0;
}

void FUNCTION_1()
{
    int a;          //initialized in stack frame of FUNCTION_1

    FUNCTION_2();   //FUNCTION_2 stack frame
}

void FUNCTION_2()
{
    int b;          //initialized in stack frame of FUNCTION_2
}
```

Steps: Code Compilation to Execution

- riscv32-unknown-elf-gcc -march=rv32i -S -o riscv.s ./code.c
- riscv32-unknown-elf-as -march=rv32i -S -o riscv.o ./riscv.s
- riscv32-unknown-elf-as -march=rv32i -o riscv.o ./riscv.s
- riscv32-unknown-elf-ld -o riscv ./riscv.o
- riscv32-unknown-elf-objcopy -O binary --only-section=.text riscv instr.mem
- riscv32-unknown-elf-objcopy -O binary --only-section=.data riscv data.mem
- riscv32-unknown-elf-objdump -D -b binary -m riscv:rv32i instr.mem

Debugging

- # Compile with debugging information
- riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -g -o my_program ./for_loop.c

Start GDB and load program

- riscv64-unknown-elf-gdb my_program
- # Run program in GDB
- (gdb) target sim
  - (gdb) break linenumber
  - (gdb) print variable_name

Profiling

- **# Compile for performance analysis with perf**
- riscv32-unknown-elf-gcc -march=rv32i -o my_program ./code.c
- # Run program with QEMU and collect profiling data
- qemu-riscv32 -cpu rv32, my_program -perf my_program
- # Analyze profiling data with perf
- // Not yet configured in cluster

Stress Testing

- riscv32-unknown-elf-gcc -march=rv32i -o stress-ng stress-ng.c
- **# Run stress tests with stress-ng**
- qemu-riscv32 -L /path/to/riscv/rootfs ./stress-ng --cpu 4 --io 2 --vm 2 --vm-bytes 128M --timeout 60s
- **Custom Stress Checking**
- riscv32-unknown-elf-gcc -march=rv32i -o stress_test ./stress_test.c
- **# Run custom stress test program**
- qemu-riscv32 ./stress_test

Performance Analysis

- riscv32-unknown-elf-gcc -march=rv32i -o my_program ./code.c
- qemu-riscv32 -L /path/to/riscv/rootfs valgrind --tool=cachegrind ./my_program
- # Run program with QEMU for performance analysis
- qemu-riscv32 -d in_asm,cpu ./my_program > qemu_log.txt
- # Analyze QEMU log
- grep -E 'IN:|CPU:|Cycle:' qemu_log.txt

Testing Spike

```
/opt/riscv-gnu32/bin/spike --isa=RV32IMAC -d /opt/riscv/riscv32-unknown-elf/bin/pk ./heap32
until reg 0 pc 0x1000  # Stop execution when program counter of core 0 reaches 0x1000
mem 0 0x80000000  # View memory content at address 0x80000000 for core 0
freg 0 f0  # Display floating-point register f0 for core 0
run 1000  # Resume execution for 1000 instructions
reg 0  # View all registers for core 0
pc 0    # View the program counter of core 0
until pc 0 0x1000  # Stop execution when PC of core 0 reaches address 0x1000
while reg 0 sp 0x80000000  # Continue running while stack pointer (sp) of core 0 is 0x80000000
dump 0x80000000 0x80001000  # Dump memory from address 0x80000000 to 0x80001000
quit
mtime
mtimecmp 0
```

QEMU Debuging

- qemu-system-riscv32 -gdb tcp::1234 -S -kernel ./hello32.o
- riscv32-unknown-elf-gdb ./hello32.o #Sperate window open
- Debug Commands
- (gdb) target remote :1234   # Connect to the QEMU GDB server

```
(gdb) load                  # Load the binary into QEMU
(gdb) b main                # Set a breakpoint at the main function
(gdb) c                     # Continue execution until the breakpoint is hit
(gdb) info reg              # Display registers
(gdb) step                  # Step through code line by line
(gdb) next                  # Step over functions
(gdb) continue              # Continue execution until the next breakpoint
(gdb) quit                  # Exit GDB
```

Profiling QEMU

- qemu-system-riscv32 -d exec,int -kernel ./hello32.o
- perf record -e cycles -a -- qemu-system-riscv32 -kernel ./hello32.o
- perf report

Hands-on Embedded C for RISCV