SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

19ECT312 – EMBEDDED SYSTEM DESIGN

III YEAR/ VI SEMESTER

TOPIC : Memory Management in Embedded system

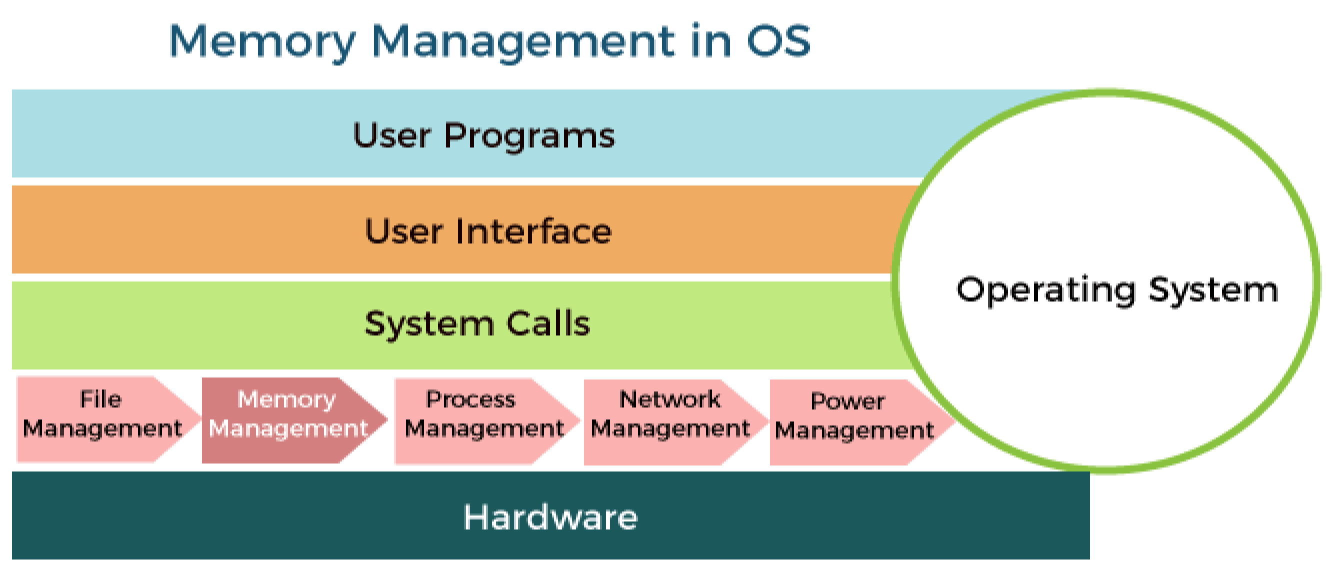
Memory Management

What is an Memory Management ?

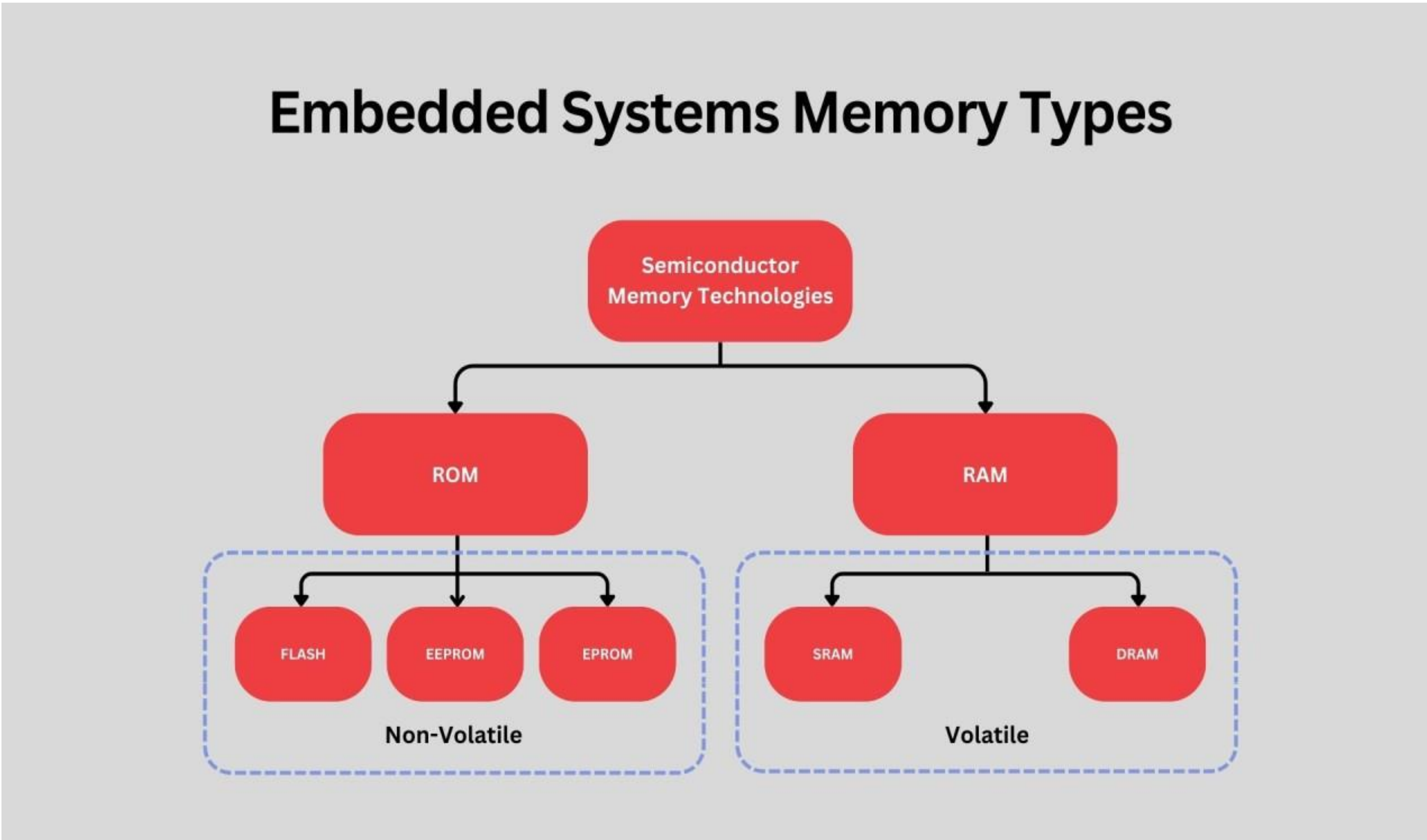
- Memory management in embedded systems is a crucial aspect of software development, especially considering the limited resources typically available in such systems.

- Embedded systems often have constraints in terms of memory size, processing power, and energy consumption.

- Efficient memory management is essential to ensure optimal utilization of resources and to meet performance requirements.



Memory Management

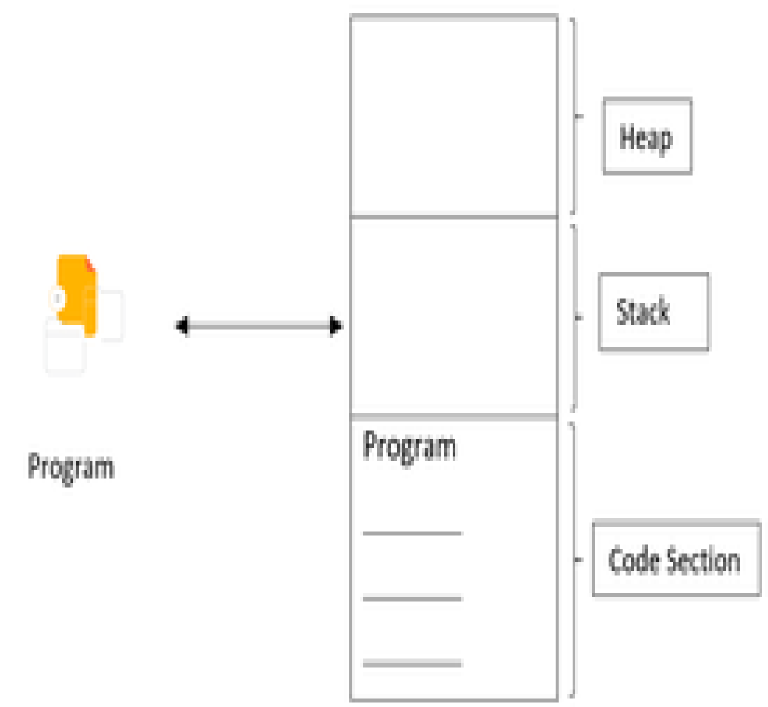


What is an Memory Allocation ?

Memory allocation in memory management for embedded systems involves the process of efficiently assigning memory resources to different components of the embedded software.

Static Allocation:

- In many embedded systems, memory allocation is primarily static, meaning that memory usage is determined at compile time.
- Memory for variables, data structures, and buffers is allocated statically, typically based on predefined requirements and constraints.
- This approach helps avoid the overhead and unpredictability associated with dynamic memory allocation and deallocation.



Memory Management

Stack-Based Allocation:

- Stack memory is commonly used for local variables and function call frames in embedded systems.
- As functions are called and return, the stack pointer is adjusted accordingly to allocate and deallocate memory for local variables and function parameters.
- Stack-based allocation provides deterministic behavior and efficient memory management, particularly for short-lived data.

Static Data Allocation:

- Global variables and statically allocated data structures are placed in memory regions determined at compile time.
- The size and location of these variables are known during compilation, allowing the compiler/linker to assign memory addresses accordingly.
- Static data allocation is well-suited for data that persists throughout the program's execution.

Memory Management

Heap-Based Allocation:

- Although less common in embedded systems due to its dynamic nature, heap-based memory allocation is sometimes used for allocating memory at runtime.
- Embedded systems may employ custom memory allocation schemes, such as memory pools or fixed-size allocators, to mitigate fragmentation and overhead associated with traditional heap management.
- Heap-based allocation requires careful management to prevent memory fragmentation and ensure efficient memory usage.

Memory Constraints and Optimization:

- Memory allocation in embedded systems must consider stringent constraints such as limited memory size and real-time performance requirements.
- Optimization techniques, including code and data compression, memory pooling, and alignment, are applied to minimize memory usage and maximize resource utilization.
- Trade-offs between memory usage, performance, and complexity are carefully evaluated to meet the specific requirements of the embedded system.

Memory Profiling Techniques

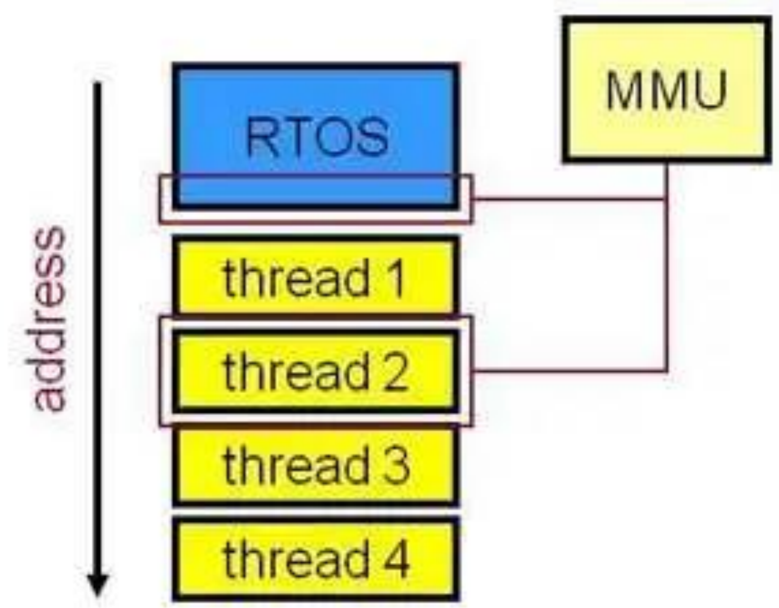
Introducing various profiling techniques for analyzing memory usage in embedded systems.

- **Memory Usage Graphs**: Visual representations of memory consumption over time, showing trends and peaks in memory usage.
- **Memory Heatmaps**: Color-coded representations of memory usage across different memory regions or components, highlighting areas of high and low usage.
- **Memory Allocation Tree**: Hierarchical diagram illustrating memory allocation relationships, showing how memory is divided among different components and data structures.
- **Memory Leak Detection**: Diagrams or tables indicating potential memory leaks, including information such as allocated memory blocks, their sizes, and references to the code causing the leaks.
- **Heap and Stack Usage**: Graphs or diagrams showing the usage of heap and stack memory over time, helping identify potential issues such as stack overflow or excessive heap fragmentation.
- **Function Call Memory Profiles**: Visualizations depicting memory usage associated with specific functions or modules, aiding in identifying memory-intensive areas of code.
- **Memory Access Patterns**: Visual representations of memory access patterns, including read and write operations, helping optimize memory usage and access efficiency.

Memory Management

Modeling Memory Usage

- Memory usage modeling is a way to organize and define how memory behaves.
- It provides structure and rules for how to access and use addresses in a system.
- Memory models are important for concurrent programs because they define the possible values that a read operation can return based on the write operations performed by the program.
- They also provide the basic semantics of shared variables, which are crucial for reasoning about programs and programming language



Memory Management

Memory Optimization Strategies:

1.Code Size Reduction:
1. Minimize the size of executable code by eliminating redundant or unnecessary instructions.
2. Use compiler optimization flags to reduce code size without sacrificing functionality.
3. Employ techniques like function inlining, loop unrolling, and dead code elimination to streamline code execution.

2.Data Compression:
1. Compress data stored in memory to reduce memory footprint.
2. Apply compression algorithms such as run-length encoding, Huffman coding, or delta encoding to efficiently store and retrieve data.
3. Balance compression ratio with decompression overhead to ensure acceptable performance.

3.Memory Pooling:
1. Allocate fixed-size memory blocks from a pre-allocated pool instead of using dynamic memory allocation.
2. Reduce memory fragmentation and overhead associated with dynamic memory management.
3. Implement custom memory allocators tailored to specific application requirements.

4.Memory Alignment:
1. Align data structures to memory boundaries to optimize memory access.
2. Improve performance by ensuring that data structures are accessed efficiently without unnecessary padding.
3. Minimize memory waste and enhance cache utilization by aligning data structures appropriately.

5.Selective Compilation:
1. Use conditional compilation to include or exclude features based on system requirements.
2. Enable/disable optional features or modules to reduce memory usage.
3. Customize build configurations for different target platforms or deployment scenarios to optimize memory allocation.

Memory Management

Real Time case study:

In an automotive embedded system, memory management is critical for real-time performance and reliability. For instance, in an engine control unit (ECU), memory allocation ensures efficient storage of sensor data, control algorithms, and diagnostic routines. Static allocation reserves memory for critical functions like ignition timing, while dynamic allocation handles variable-sized data streams from sensors. Memory pooling optimizes resource usage, reducing fragmentation and ensuring timely responses to engine events. By carefully managing memory, the ECU maintains real-time responsiveness, enhances fuel efficiency, and ensures safe operation, illustrating the pivotal role of memory management in embedded systems for automotive applications.

.

Memory Management

Advantages:

- Resource Optimization
- Reliability and Stability
- Real-Time Responsiveness
- Space Efficiency
- Security Enhancement
- Ease of Maintenance
- Optimized Performance

Memory Management

Disadvantages:

- Complexity Overhead
- Overhead in Real-Time Systems
- Fragmentation
- Memory Leaks
- Resource Constraints
- Security Risks
- Performance Degradation

SUMMARY & THANK YOU