Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Outline

In this lecture, we will cover:

- Review on the **important points** which were covered in the
- Pointers (Continuation)
- ARM Architecture Overview
- Introduction to Raspberry Pi5
- Pi 5 set distribution
- In Class Lab

Pointers - Recap

- Have to understand these basic operations:
  1. Set the value of a pointer to the address of a variable
  2. Dereference a pointer
  3. Set or Read the value of a dereferenced pointer
  4. Increment a pointer
- Pointers are declared using the * character

Examples:

```
int* ptr1;          // pointer to     type int
int *ptr2;          // alternative    declaration
char* ptr3;         // pointer    to type char
int** ptr4;         // pointer    to an int pointer
```

- & is the address operator
- **&myVariable** is the address of **myVariable**

Primitive Types and Sizes - Recap

Name	Number of Bytes sizeof()	Range
char	1	0 to 255 or -128 to 127 (Depends on Compiler settings)
signed char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	Varies by platform	Varies by platform
int (on TM4C123)	4	-2,147,483,648 to 2,147,483,647
unsigned int (on TM4C123)	4	0 to 4,294,967,295
(pointer)	Varies by platform	Varies by platform
(pointer on TM4C123)	4	Address Space

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, depends on Compiler settings

Primitive Types and Sizes - Recap

Name	Number of Bytes sizeof()	Range
long	4	-2,147,483,648 to 2,147,483,647
signed long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	±1.175e-38 to ±3.402e38
double	Varies by platform	
double (on TM4C123)	8	±2.3E-308 to ±1.7E+308

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, depends on **Compiler settings**

Pointers- Recap

- Incrementing and decrementing a pointer
  - Increments/decrements by the size of the **“sub-type”**
- Example:
  - int* increment by 4 (ints are 4 bytes)
  - char* increment by 1 (chars are 1 byte)

```
 int* ip = 0x1000;      // sizeof(int) == 4
char* cp = 0x1000;      // sizeof(char) == 1
             Ip++;
             cp++;
 // ip == 0x1004 and cp = 0x1001
```

	Address	Value
	0xFFFF_FFFF	0x00
	0xFFFF_FFFE	0x00
	0xFFFF_FFFD	0x10
ip	0xFFFF_FFFC	0x04
	0xFFFF_FFFB	0x00
	0xFFFF_FFFA	0x00
	0xFFFF_FFF9	0x10
cp	0xFFFF_FFF8	0x01

Why Are Pointers Important in Embedded Systems-Recap

- Memory Access Efficiency: Directly manipulate memory locations for performance.
- Peripheral and Register Access: Interact with hardware components efficiently.
- Dynamic Memory Management: Used in RTOS-based applications for heap management.

Common Use Cases

- Accessing Hardware Registers

```
#define GPIO_PORT *(volatile uint32_t*)0x40021000
GPIO_PORT =  0x01;  // Set GPIO high
```

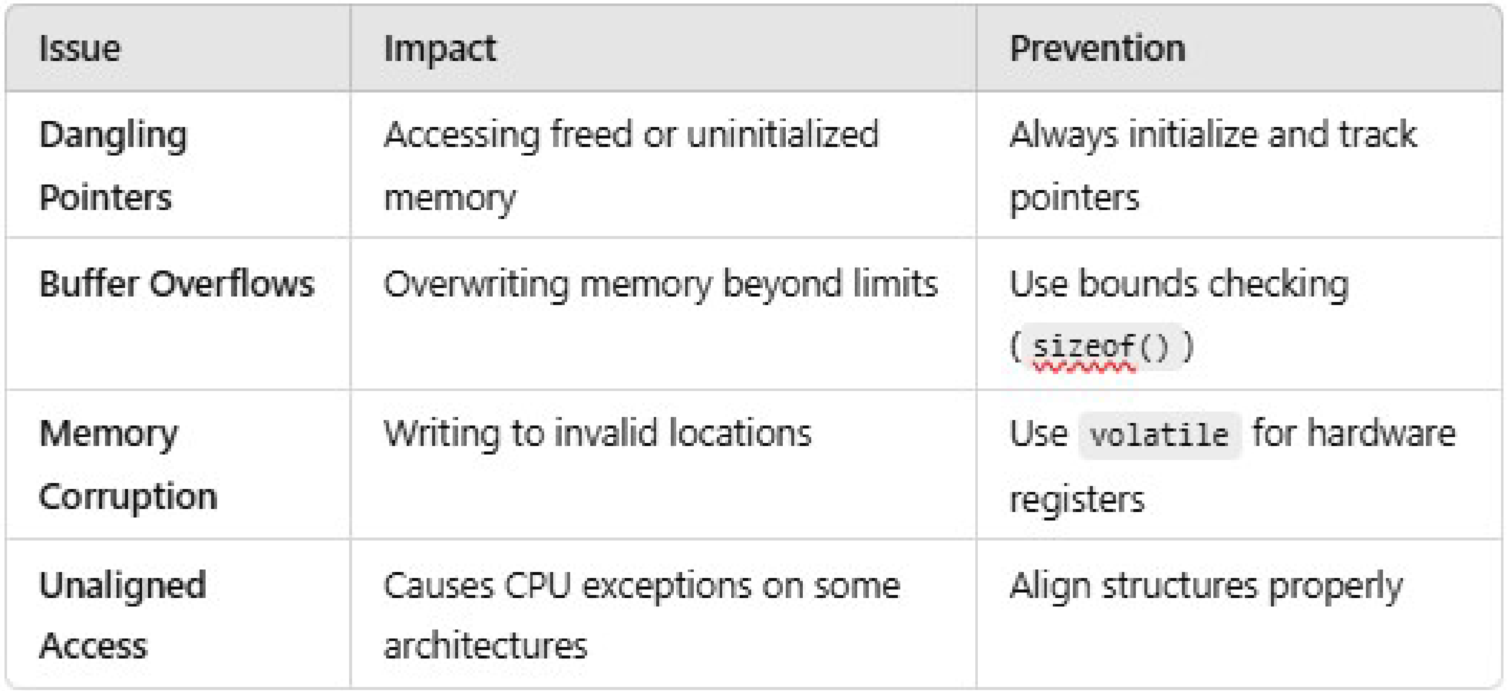
- Pointer Arithmetic for Buffers

```
char buffer[10]; char *ptr = buffer;
ptr++;  // Moves to the next byte
```

- Function Pointers for Callbacks

```
void (*callback)(void);
callback = some_function; callback();
```

Common Pitfalls in Embedded Systems-Recap



Issue	Impact	Prevention
Dangling Pointers	Accessing freed or uninitialized memory	Always initialize and track pointers
Buffer Overflows	Overwriting memory beyond limits	Use bounds checking (`sizeof()`)
Memory Corruption	Writing to invalid locations	Use `volatile` for hardware registers
Unaligned Access	Causes CPU exceptions on some architectures	Align structures properly

Example: Avoiding Buffer Overflow

- void safe_function(char *input)
  { char buffer[8];
  strncpy(buffer, input, sizeof(buffer) - 1); // Ensures safe copying}

Best Practices for Using Pointers in Embedded C

Common Use Cases

1. Use volatile for Hardware Registers

- Prevents compiler optimizations that may remove necessary memory accesses.

```
volatile uint32_t *uart = (uint32_t *)0x40011000;
*uart = 0x01;  // Send data
```

2. Avoid Dynamic Memory Allocation

- Embedded systems often lack dynamic memory safety mechanisms.

```
// Instead of using malloc/free
static uint8_t buffer[256];  // Use static/global allocation
```

3. Monitor Stack Usage

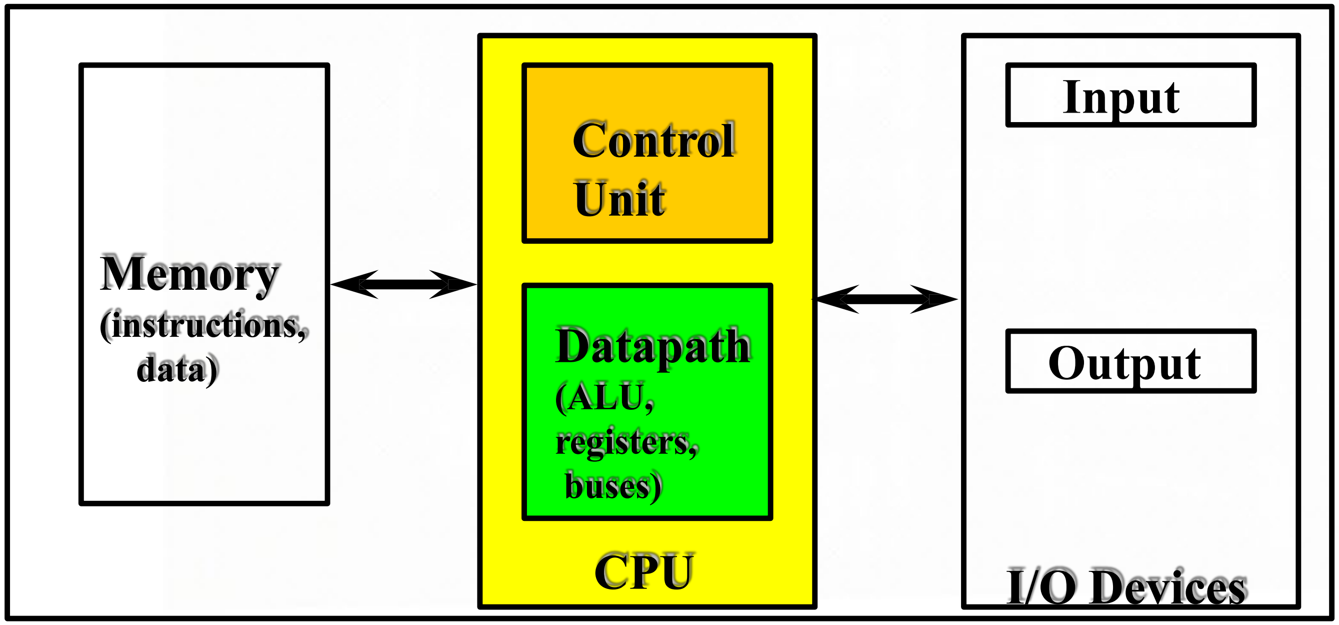
- Prevents stack overflow caused by excessive pointer operations.

```
I f (uxTaskGetStackHighWaterMark (NULL) < 50)
{    printf("Warning: Low Stack Memory!\n");}
```

4. Use Memory Protection Unit (MPU) if Available

- Protects against invalid pointer dereferencing.

Generic Processor Model - Recap



CISC and RISC Processors

- CISC - microprocessors with large number of different instructions (100-250)
  more complex processor hardware, easier to program, slower, smaller and more compact programs, less memory required (Z80, Intel 8080/8085, Motorola 6800/6802/6809/68000)
- RISC processors (reduced instruction set) - **smaller set of instructions**, each executes **faster**, **larger memory** requirements and more **complex programs**

ARM ARCHITECTURE OVERVIEW

ARM Powered Products

Why use assembly programming?

- Full access to hardware features
  - Compiler limits programmers’ access to the hardware features that the compiler writer decided to implement
- Writing time-critical portions of code
  - Allows tight control over what the CPU is doing on every clock cycle
- Debugging
  - It is **NOT un**common when trying to debug odd system behavior to have to look at disassembled code

Why learn the ARM Hardware Architecture?

- Helps give intuition to why the **assembly instructions** were created the way the were
- Help understand what **special feature** may be available for you to make use of.

Licencable architecture

- Companies that are currently or formerly ARM licensees include

Alcatel, **Apple Inc.**, **Atmel**, **Broadcom**, **Cirrus Logic**, **Digital Equipment Corporation**, **Freescale**, **Intel** (through DEC), **LG**, **Marvell Technology Group**, **NEC**, **NVIDIA**, **NXP** (previously Philips), **Oki**, **Qualcomm**, **Samsung**, **Sharp**, **ST Microelectronics**, **Symbios Logic**, **Texas Instruments**, **VLSI Technology**, **Yamaha** and **ZiiLABS**

Introduction

- Leading provider of **32**-bit embedded **RISC** microprocessors, 75% of the market
  - High performance
  - Low power consumption
  - Low system cost

- Solutions for
  - Embedded **real-time** systems for mass storage, automotive, industrial and networking applications
  - Secure applications: smartcards and SIMs
  - Open platforms running **complex operating systems**

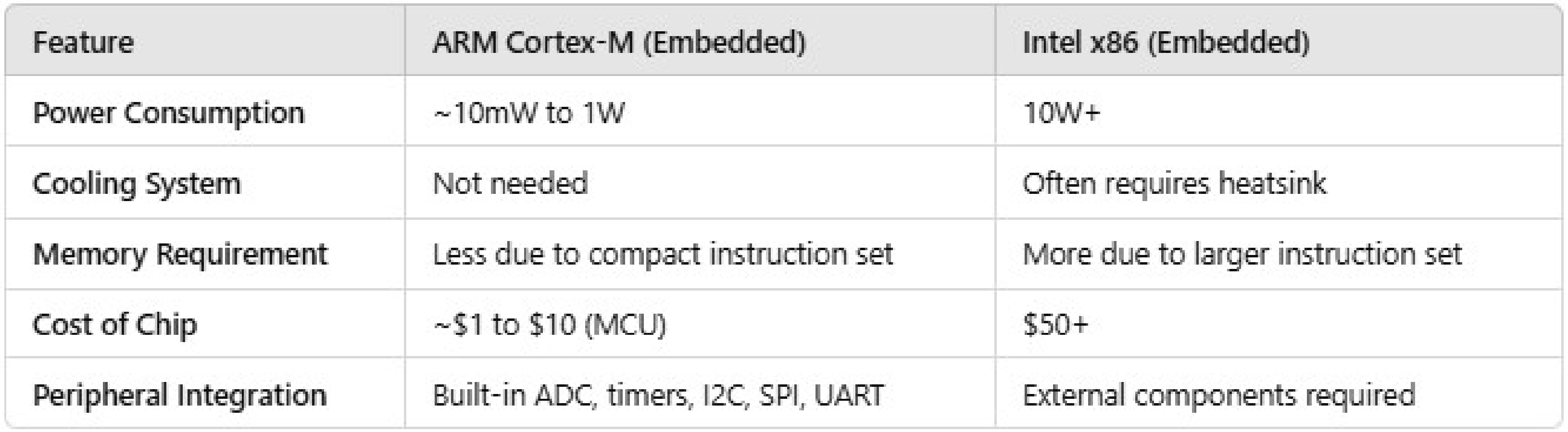
What does "Low System Cost" Mean in Embedded Systems?

- "Low system cost" refers to the **total cost of designing, manufacturing, and deploying an embedded system**. It involves multiple factors, including **hardware, software, power consumption, and production efficiency**. ARM processors are widely used in embedded systems because they help **minimize system costs** in several ways.

- **Why ARM Lowers System Cost**

✅ **License-based design** → Lower chip manufacturing costs.
✅ **Low power consumption** → Smaller batteries, no cooling costs.
✅ **Integrated peripherals** → Fewer external components needed.
✅ **Compact instruction set** → Requires less memory, reducing memory costs.

Real-World Example: ARM vs. x86 in Embedded Systems



Feature	ARM Cortex-M (Embedded)	Intel x86 (Embedded)
Power Consumption	~10mW to 1W	10W+
Cooling System	Not needed	Often requires heatsink
Memory Requirement	Less due to compact instruction set	More due to larger instruction set
Cost of Chip	~$1 to $10 (MCU)	$50+
Peripheral Integration	Built-in ADC, timers, I2C, SPI, UART	External components required

ARM Architecture Version (1/4)

- Version 1
  - The first ARM processor, developed at Acorn Computers Limited 1983-1985
  - **26-bit** address, **no multiply or coprocessor** support
- Version 2
  - Sold in volume in the Acorn Archimedes and A3000 products
  - **26-bit** addressing, including **32-bit result multiply and coprocessor**
- Version 2a
  - Coprocessor 15 as the system control coprocessor to manage **cache**
  - Add the atomic load store (**SWP**) instruction

(https://www.csie.ntu.edu.tw/~cyy/courses/assembly/08fall/lectures/handouts/lec08_ARMisa_4up.pdf)

What is an "Atomic Load-Store Instruction"?

Definition:
An **atomic load-store instruction** is a type of **CPU instruction** that ensures a **load (read) or store (write) operation occurs as an indivisible (atomic) unit**. This means that once the instruction starts executing, **no other process or interrupt can modify the memory location being accessed until the operation is complete**.

Why is it Important?
In **multi-core processors** and **multi-threaded embedded systems**, multiple processes or hardware components may attempt to **read/write to the same memory location** simultaneously. Without atomic operations, race conditions, inconsistent data, or memory corruption can occur.

Understanding Race Conditions in Embedded Systems

What is a Race Condition?

•A **race condition** occurs when **two or more tasks (threads or interrupts) access a shared resource concurrently** and the final result depends on the **timing of execution**.

•This leads to **unpredictable behavior**, data corruption, or system failures.

e.g.

```
volatile int counter = 0;

void Task1() {
    counter++;  // Read-modify-write operation (non-atomic)
}

void Task2() {
    counter++;  // Another thread modifying counter
}
```

◆ **Issue:** If **Task1 and Task2 execute at the same time**, both might **read the same value before incrementing**, causing **data loss**.

Atomic Load-Store in ARM Architecture

ARM introduced an early form of **atomic memory access** with the **SWP (Swap) instruction** in **ARMv2a**. This instruction:

1.Loads a value from memory.

2.Stores a new value in that memory location.

3.Guarantees that no other process modifies the memory during execution.

```
SWP Rd, Rm, [Rn]
// Rd: Destination register (receives old value from memory).
   Rm: Register containing the new value to store.
   [Rn]: Memory address where the swap happens.
   Example: Exchanging two values atomically
   MOV R0, #5      ; Load value 5 into R0
   MOV R1, #10     ; Load value 10 into R1
   SWP R0, R1, [R2] ; Swap R1 with memory at address R2, storing old value in R0
```

Key Feature: Atomicity prevents other cores or threads from modifying the memory during execution. **Essential for multi-threading, locks, and hardware synchronization in embedded systems**.

Modern Atomic Instructions in ARMv6+

Since ARMv6, more advanced atomic load-store instructions have been introduced for multi-core processing and synchronization:

1. **LDREX (Load Exclusive) & STREX (Store Exclusive)**

Used for implementing locks or atomic operations in multi-threaded systems.

- LDREX reads a memory value and marks it as reserved.
- STREX writes a new value only if the reservation is still valid (i.e., no other write has occurred).

Example:

```
LDREX R1, [R0]    ; Load value at R0 into R1 (exclusive access)
ADD R1, R1, #1    ; Increment value
STREX R2, R1, [R0]; Store R1 back to memory, R2 = 0 if successful
```

If STREX fails (another core modified memory), the program retries.

ARM Architecture Version (2/4)

- Version 3
  - First ARM processor designed by ARM Limited (1990)
  - ARM6 (macro cell)
    ARM60 (stand-alone processor)
    ARM600 (an integrated CPU with on-chip cache, MMU, and write buffer)
    ARM610 (used in Apple Newton)
  - **32**-bit addressing, separate current program status register (**CPSR**) and Saved Program Status Register (**SPSR**)s
  - Add the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode

- Version 3M (suitable for multimedia and audio processing)
  - Introduce the signed and unsigned multiply and multiply-accumulate (MAC) instructions, which generate the full **64**-bit result, for DSP applications.

ARM Architecture Development (3/4)

v4	v5	v6	v7
Halfword and signed halfword / byte support System mode Thumb instruction set (v4T)	Improved interworking CLZ Saturated arithmetic DSP MAC instructions Extensions: Jazelle (5TEJ)	SIMD Instructions Multi-processing v6 Memory architecture Unaligned data support Extensions: Thumb-2 (6T2) TrustZone® (6Z) Multicore (6K) Thumb only (6-M)	Thumb-2 Architecture Profiles 7-A - Applications, e.g., smartphones. 7-R - Real-time, e.g., automotive control. 7-M - Microcontroller, e.g., IoT, industrial automation.

- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture **v7-A**, with a **13**-stage pipeline
  - Cortex-A9 - architecture **v7-A**, with an **8**-stage pipeline

ARM Architecture Version (4/4)

Core	Architecture
ARM1	v1
ARM2	v2
ARM2as, ARM3	v2a
ARM6, ARM600, ARM610	v3
ARM7, ARM700, ARM710	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
StrongARM, ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	V4T
ARM9E-S, ARM10TDMI, ARM1020E	v5TE
ARM10TDMI, ARM1020E	v5TE

- ARM1 - ARM3 (Early models).
- ARM6 - ARM10TDMI (Improved power efficiency, added DSP capabilities).
- Cortex series (Modern architectures).

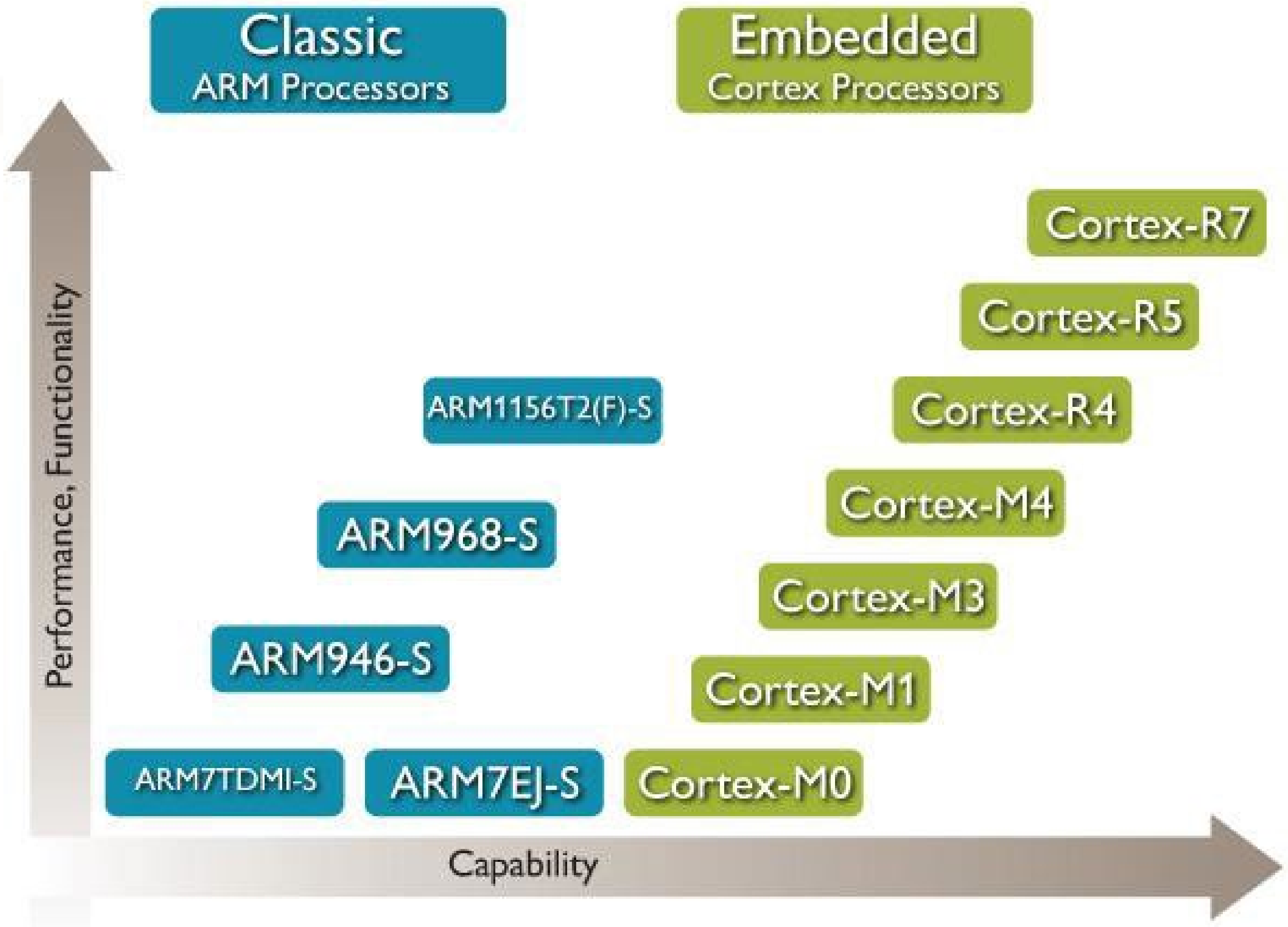
Terminology

- Application processor
  - The application processor moniker came out of the cellular industry.
  - It is a special kind of microprocessor for the primary processing of cellular phones and other smart functions.
  - It is NOT for handling background functions such as running the display, handling wireless communications and managing power drain.
- Embedded processors
  - It is simple.
  - Lack of performance-tweaking extras such as integer divide instructions, branch prediction logic, floating-point instructions or instruction/data caches.
  - They are aimed at running smaller fixed-in-place programs.

Application processor vs. Embedded processor

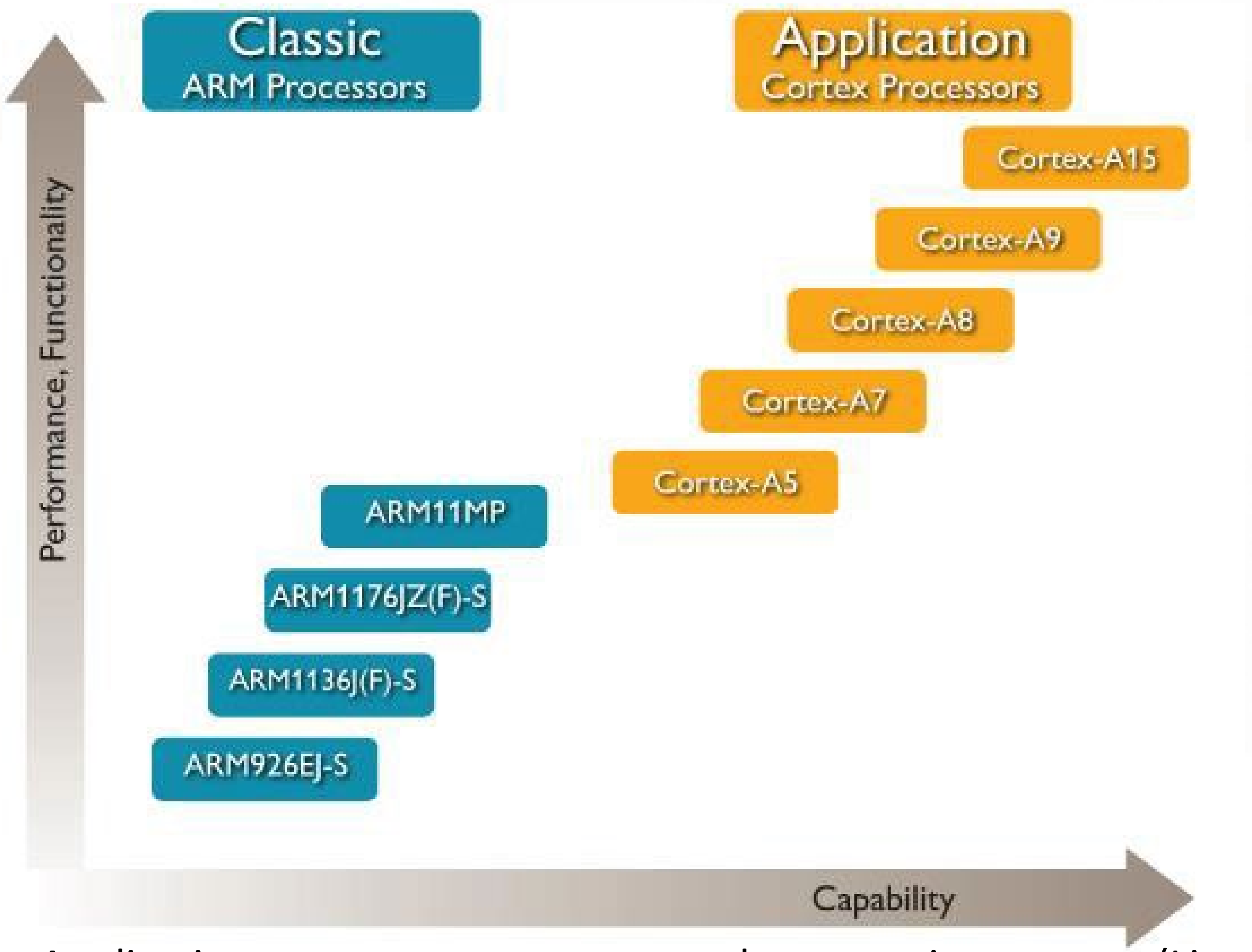
- ❑ Cost - Embedded Processors are generally cheaper.
- ❑ Frequency - Application processors are faster.
- ❑ Power - Application processors are more power hungry.

Embedded Processors



- Used in control systems, industrial automation, automotive applications.
- Designed for efficiency rather than raw performance.

Application Processors

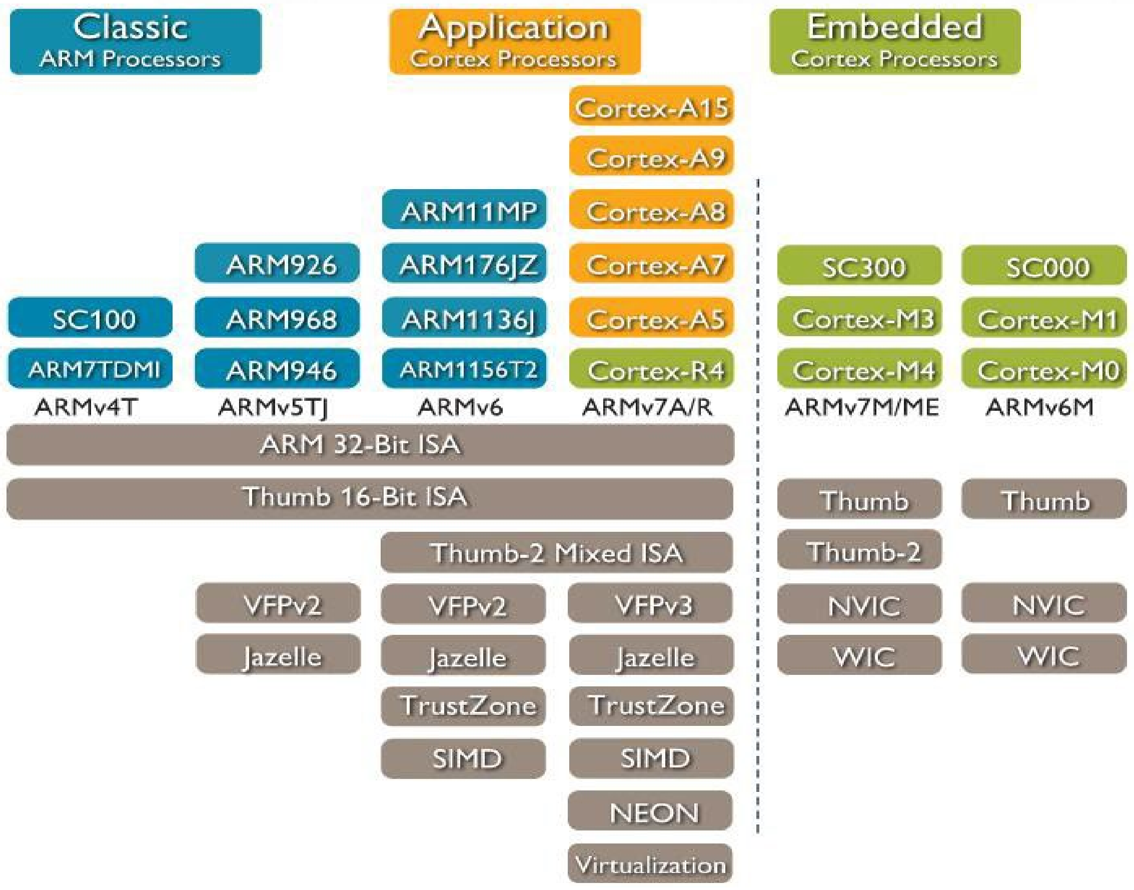


- Application processors support complex operating systems (Linux, Android).
- Application processors integrate GPUs, memory controllers, and security modules for advanced features.

Which Architecture is My Processor? （1）

Processor core	Architecture
• ARM7TDMI family	v4T
– ARM720T, ARM740T	
• ARM9TDMI family	v4T
– ARM920T,ARM922T,ARM940T	
• ARM9E family	v5TE, v5TEJ
– ARM946E-S, ARM966E-S, ARM926EJ-S	
• ARM10E family	v5TE, v5TEJ
– ARM1020E, ARM1022E, ARM1026EJ-S	
• ARM11 family	v6
– ARM1136J(F)-S	
– ARM1156T2(F)-S	v6T2
– ARM1176JZ(F)-S	v6Z
• Cortex family	
– ARM Cortex-A8	v7A
– ARM Cortex-R4	v7R
– ARM Cortex-M3	v7M

Which Architecture is My Processor? （2）



Identifying whether your processor follows ARM v4, v5, v6, v7, or v8.
Understanding whether your system is 32-bit or 64-bit.
Different ARM-based processors have unique instruction sets and capabilities.
Knowing your CPU’s architecture helps in optimizing embedded software.

ARM: RISC CPU Architecture

- What is RISC?
  - Reduced Instruction Set Computing (with respect to CISC, Complex Instruction set Computing)
- Typical RISC
  - LD/Store based: ALU to memory transaction via registers
  - **Most instruction** are the **same length**
  - Typically many **less** instructions than a CISC architecture
  - Typically many **more** registers than CISC since Data must be moved into a register before it can be operated on
  - **Low number of instruction typically makes hardware design simpler** (as compared to CISC)

ARM: RISC vs CISC example

Size / time	**CISC**
1 byte, 1clk	**mov R1, 10**
1 byte, 1clk	**mov R2, 5**
4 byte, 30 clk	**mul R2, R1**

Size / time	**RISC**
1 byte, 1clk	**mov R1, 0**
1 byte, 1clk	**mov R2, 10**
1 byte, 1 clk	**mov R3, 5**
1 byte, 1clk	**Begin: add R1, R2**
1 byte, 1 clk	**loop Begin**

- **CISC**: Instructions often **variable** length and **variable** time
- **RISC**: Instruction typically **constant** length and time
  - **Simpler hardware logic** for decoding instructions (thus typically **faster**)

ARM CPU Core Summary

Instructions are 16-bit or 32-bit

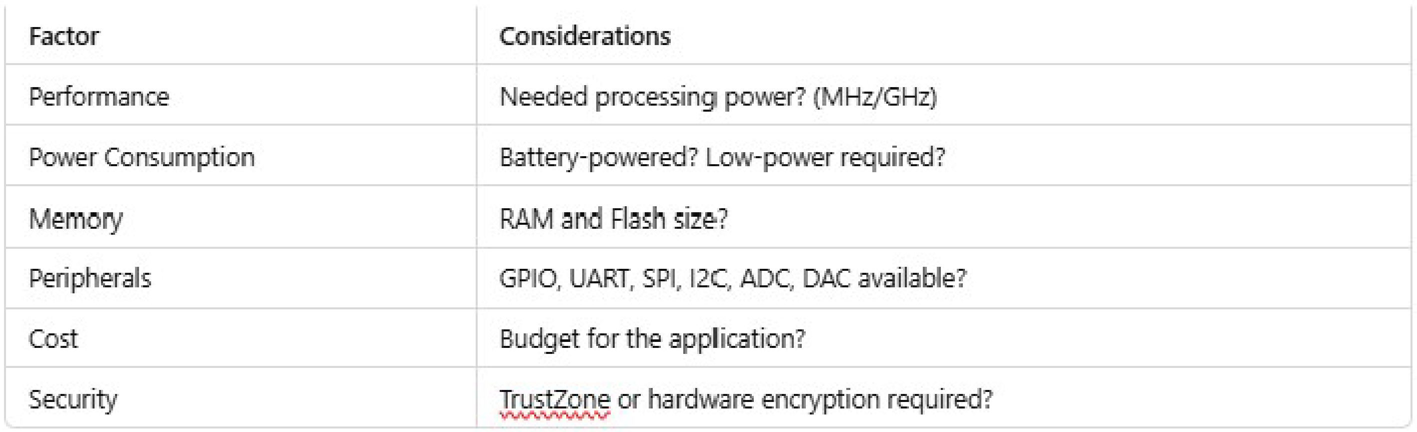
Simple three-stage **pipeline**

Registers are 32-bit and **addresses** are 32-bit

- ARM's simple pipeline design enhances performance and efficiency.
- The combination of 16-bit and 32-bit instructions improves code density, making it ideal for low-power embedded devices.

Summary: Choosing the Right ARM Processor for Embedded Systems (1)

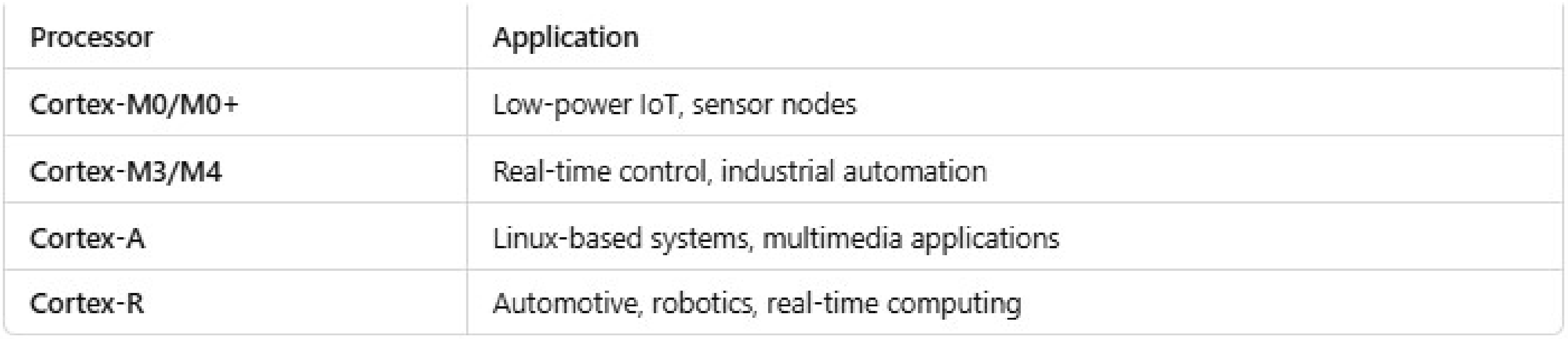
Key Factors to Consider



Factor	Considerations
Performance	Needed processing power? (MHz/GHz)
Power Consumption	Battery-powered? Low-power required?
Memory	RAM and Flash size?
Peripherals	GPIO, UART, SPI, I2C, ADC, DAC available?
Cost	Budget for the application?
Security	TrustZone or hardware encryption required?

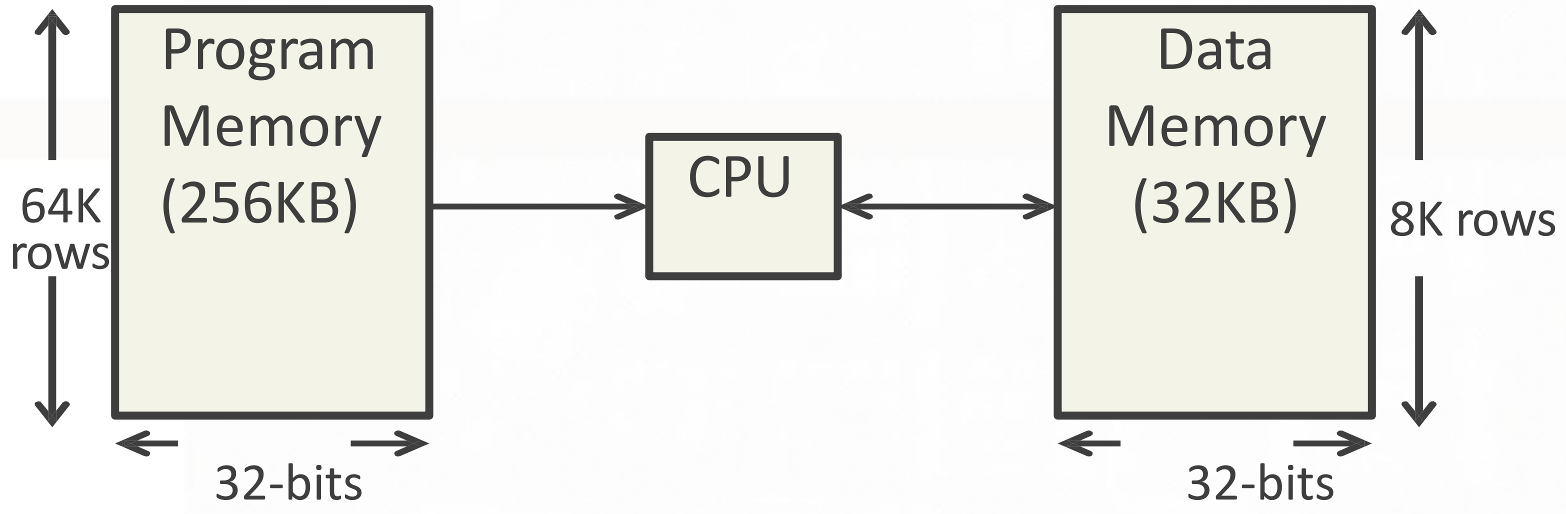
Summary: Choosing the Right ARM Processor for Embedded Systems (2)

ARM Processor Families & Use Cases



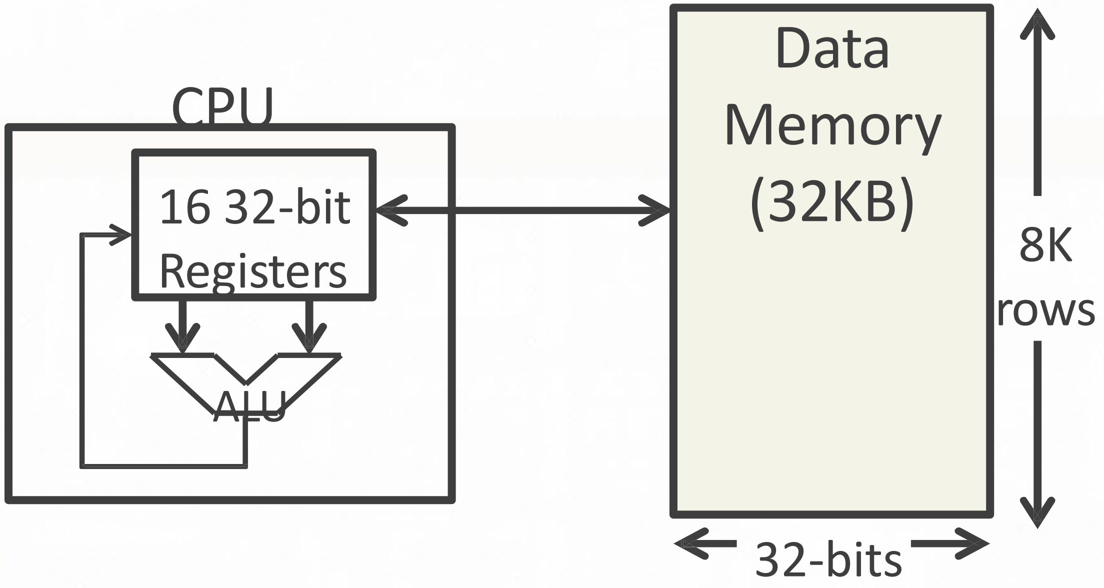
Processor	Application
Cortex-M0/M0+	Low-power IoT, sensor nodes
Cortex-M3/M4	Real-time control, industrial automation
Cortex-A	Linux-based systems, multimedia applications
Cortex-R	Automotive, robotics, real-time computing

ARM: Harvard Architecture



- Program memory
  - **Flash** based: **Program stays** even if **power turned off** (non-volatile
  - 16-bits wide, instructions are 16-bit or 32-bit wide.
- Data Memory
  - **SRAM** based: Data **disappears** if power is **turned off** (volatile)
  - 32-bits wide
- **In embedded systems**, this allows efficient **real-time processing**, reducing delays caused by memory bottlenecks.

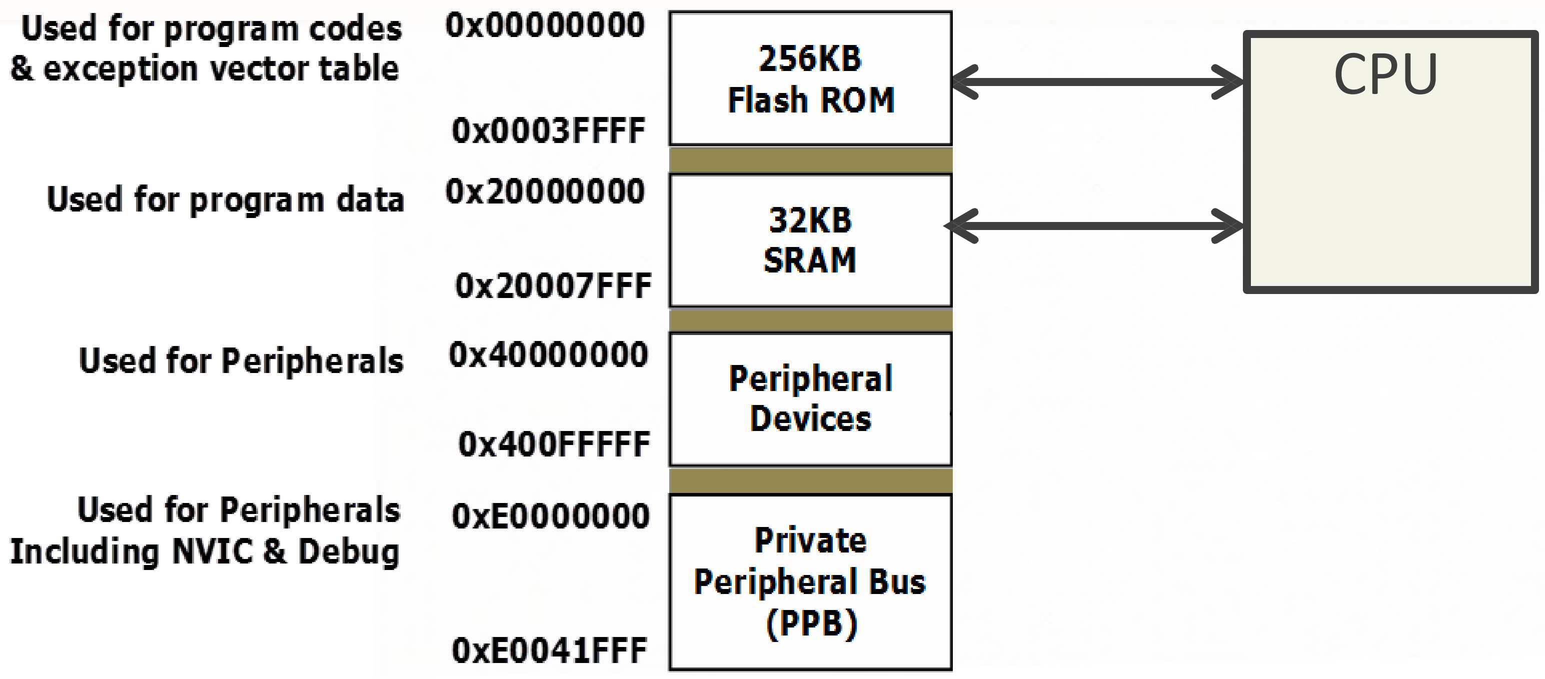
ARM: Logical Data Memory organization



- Registers:
  - 32-bits wide
  - Directly accessible by ALU
- Data Memory:
  - 32-bit wide
  - Must use a register to move to/from the ALU

* When adding two numbers, the values must first be loaded into registers, operated on, and then stored back into memory.

ARM: Memory Map organization



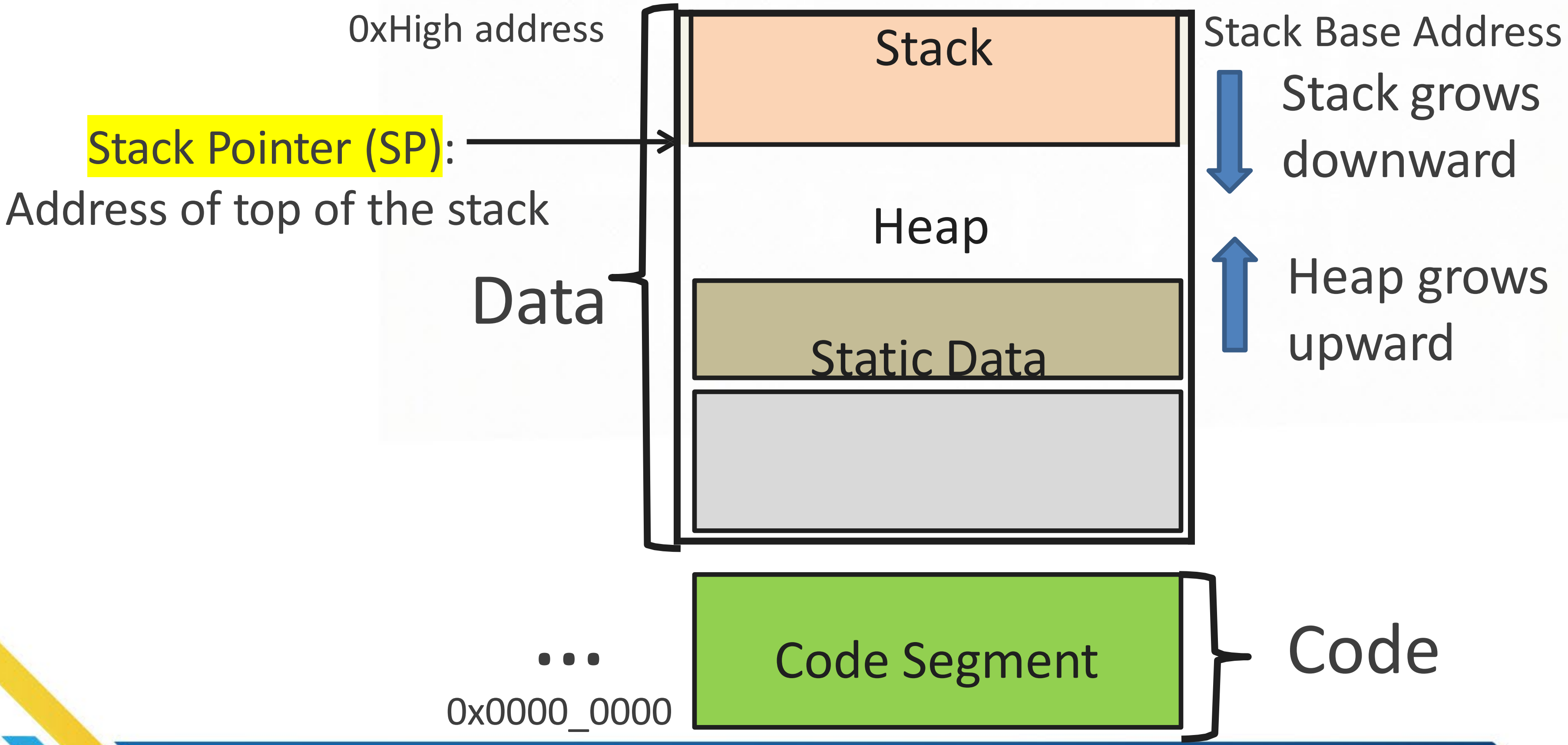
- Embedded systems engineers must understand memory mapping to efficiently interact with hardware components.
- Misconfigured memory maps can cause incorrect data access, system crashes, or security vulnerabilities.

Understanding Data

- **Stack**
  - Stores data related to **function variables, function calls, parameters, return variables**, etc.
  - Data on the stack can go "out of scope", and is then automatically deallocated
  - **Starts at the top of the program's data memory space**, and addresses move down as more variables are allocated
  - Stack accesses **local variables only**
- **Heap**
  - Stores **dynamically** allocated data
  - Dynamically allocated data usually calls the functions *alloc* or *malloc* (or uses *new* in C++) to allocate memory, and *free* to (or *delete* in C++) deallocate
  - There's no garbage collector!
  - **Starts at bottom of program's data memory space**, and addresses move up as more variables are allocated
  - Manually managed (must be freed with free()).
  - Access variables **globally**

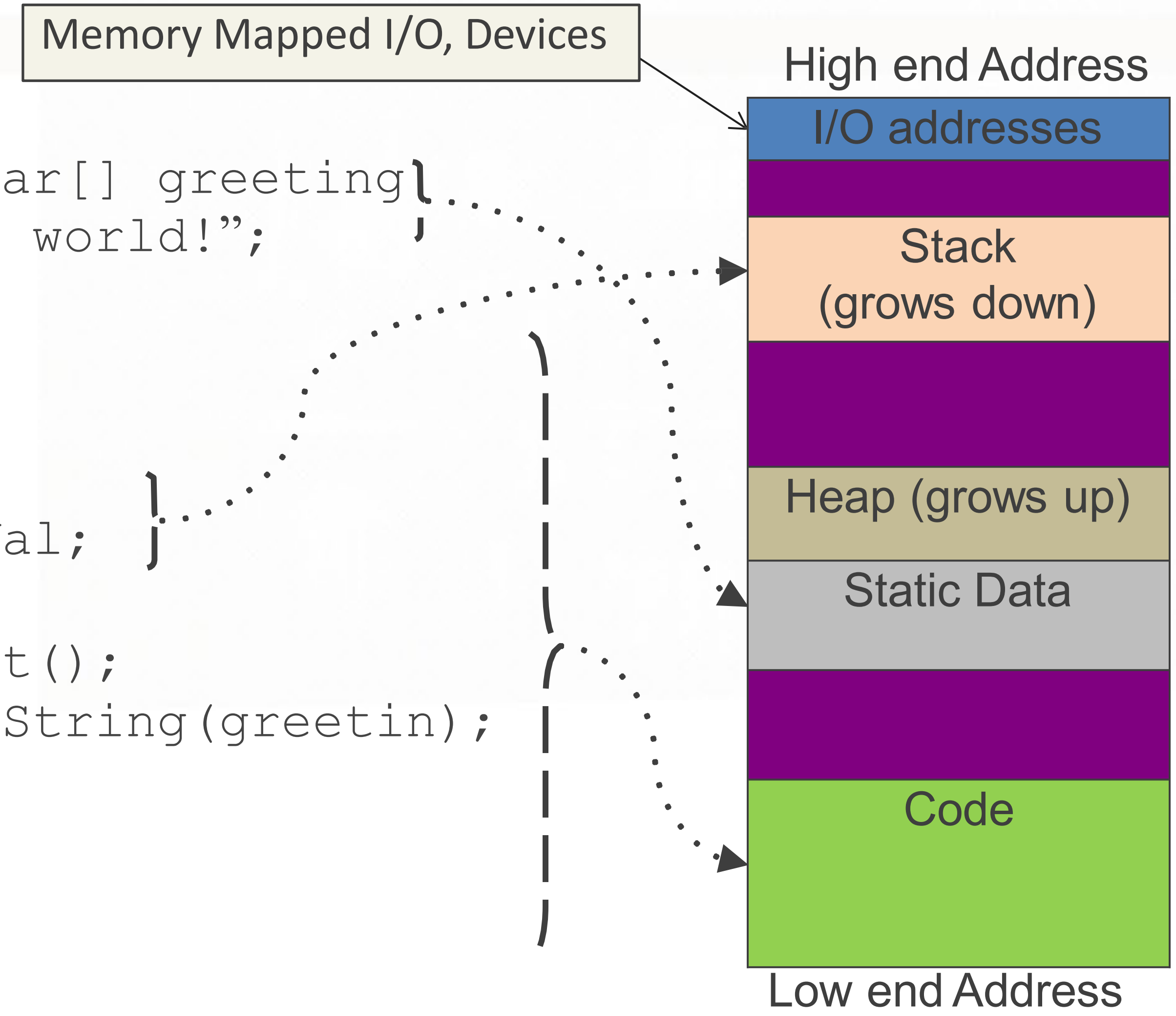
ARM: Typical Logical Memory Layout

- **Static** Data (e.g. **Global** vars)
- Stack (e.g. local vars, function args & return value, return address)
- Heap (e.g. dynamically allocated memory: malloc, new)
- Code (Code may be in the same or a separate memory device)



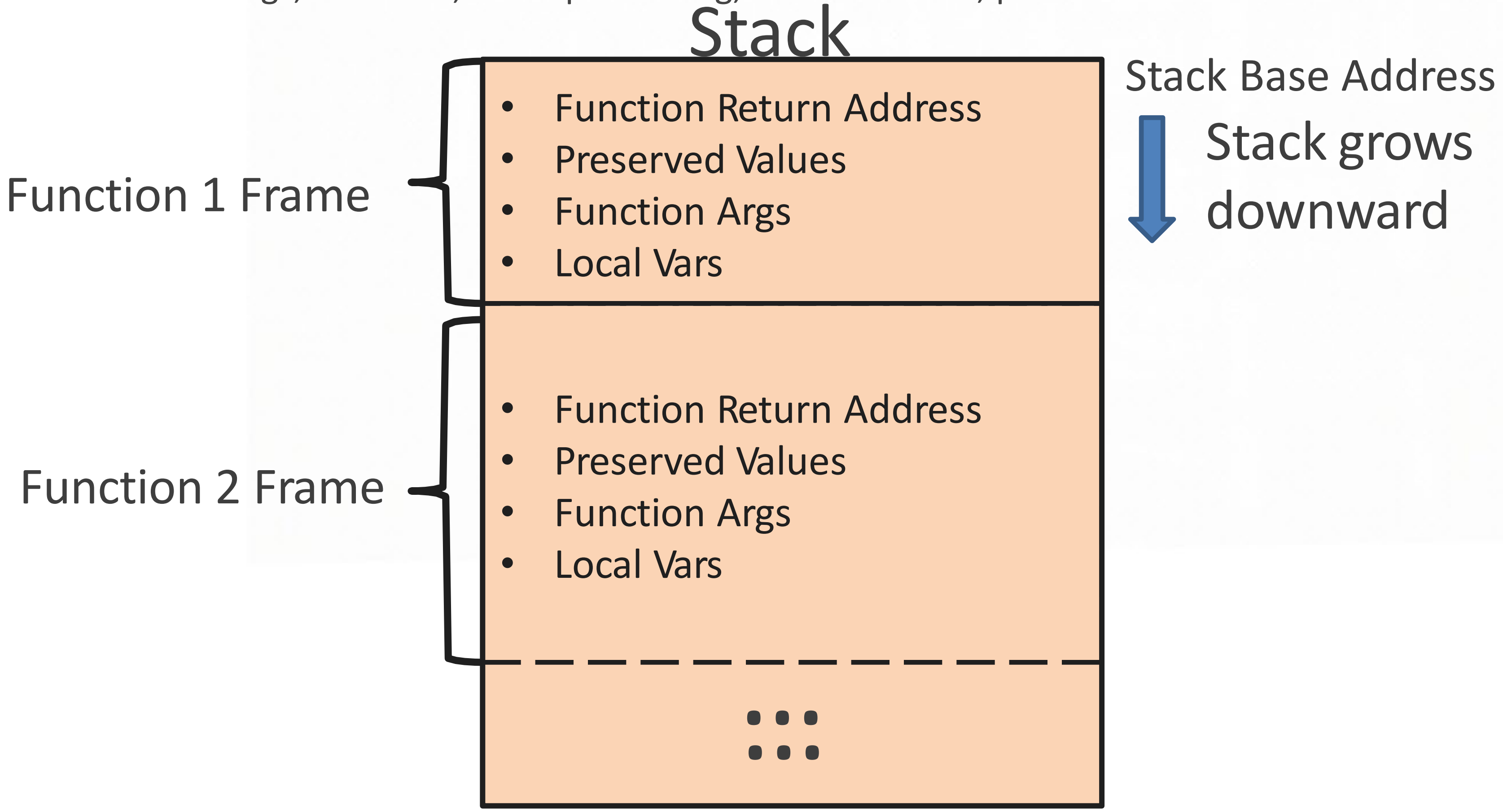
Example Memory Layout

```
static char[] greeting
   ="Hello world!";

main()
{
   int i;
   char bVal;

   LCD_init();
   LCD_PutString(greetin);
   …
}
```



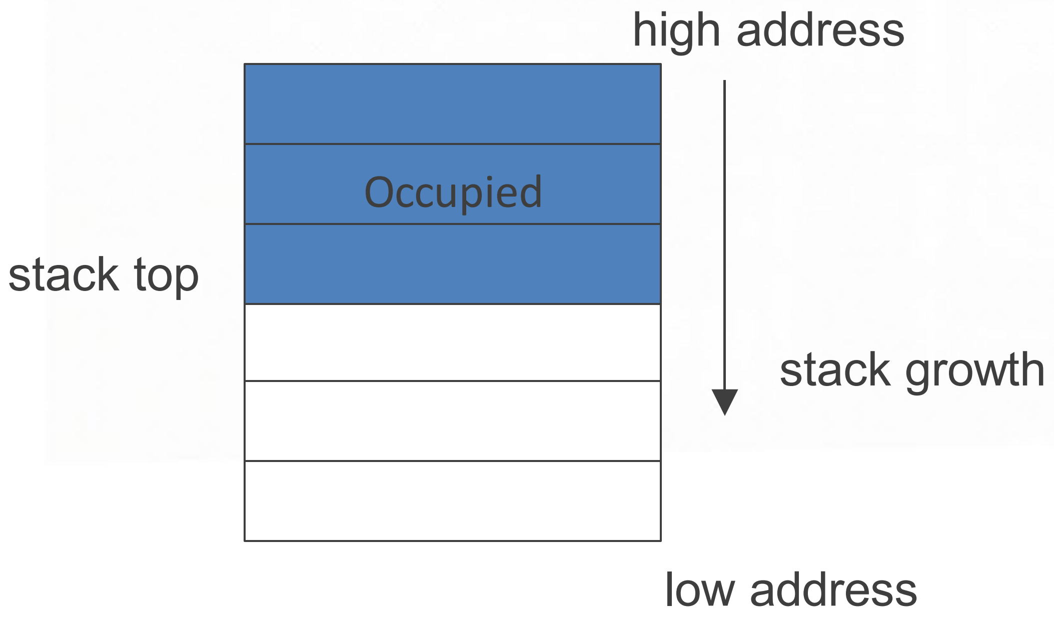
ARM: Typical Function Stack-Frame

- Stack is typically divided into a number of function stack-frames.
- Stack-Frame: Contains information associated with a function call
  - Function args, local vars, interupt handing, return address, preserved values



Function and Stack

Conventional program stack grows downwards: New items are put at the top, and the top grows down



Function and Stack

Auto, local variables have their storage in stack

Why stack?

- The Last-In-First-Out (LIFO) order matches perfectly with functions call/return order
  - LIFO: Last In, First Out
  - Function: Last called, first returned
- Efficient memory allocation and de-allocation
  - Allocation: Decrease SP (stack top): PUSH
  - De-allocation: Increase SP: POP

Function and Stack

Function Frame: Local storage for a function

Example: 1. A is called; 2. A calls B; 3. B calls C; 4. C returns

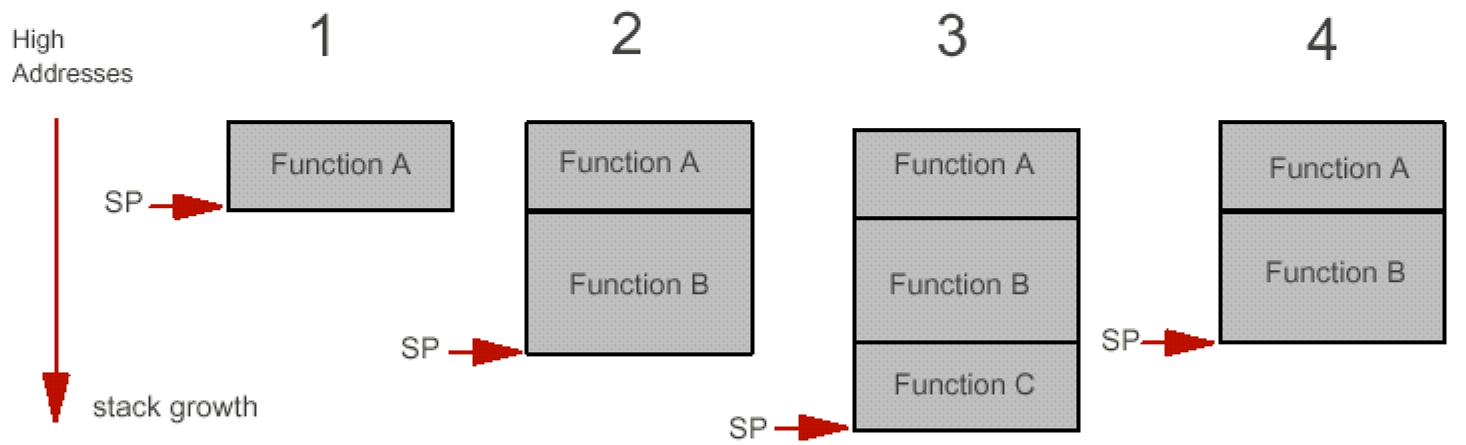


Figure 1 - Stack Frame creation and destruction

*

- The **stack pointer (SP)** keeps track of the top of the stack.
- When a function completes, its stack frame is **popped off**.
- **Excessive recursion leads to stack overflow**, causing system crashes.

Function and Stack

What can be put in a stack frame?

- Function return address
- Parameter values
- Return value
- Local variables
- Saved register values

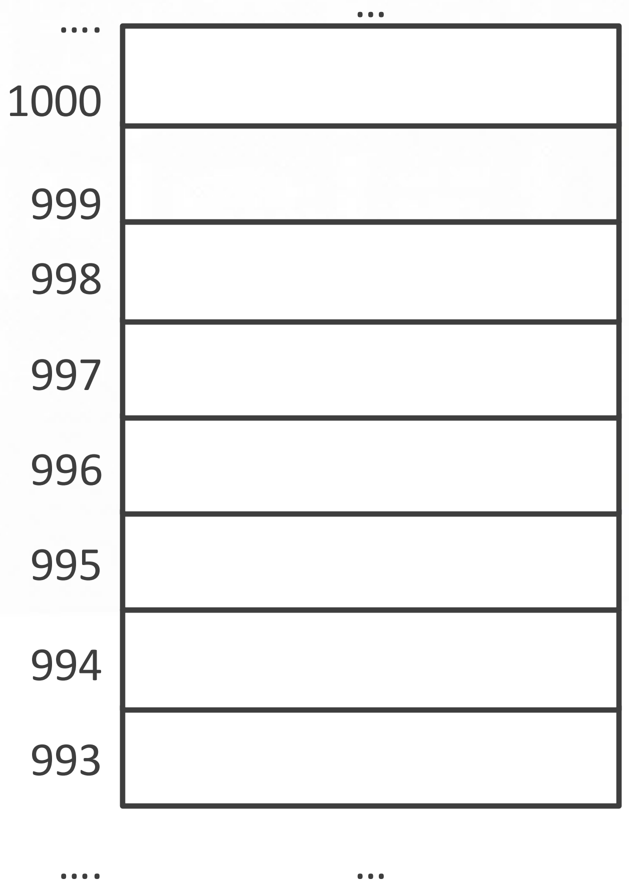
Example: Stack

- The following example shows the execution of a simple program (left) and the memory map of the stack (right)

Example: Stack

- void doNothing() {
    - char c;
- }

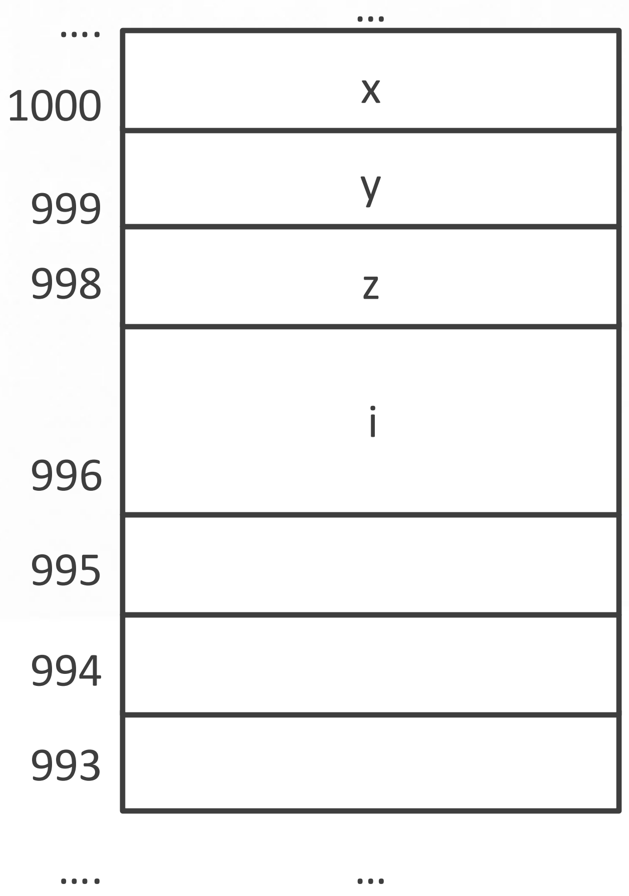
- int main() {
    - char x, y, z;
    - short i;
    - for (i = 0; i < 10; i++) {
        - doNothing();
    - }
    - return 0;
- }



Example: Stack

- void doNothing() {
  - char c;
- }

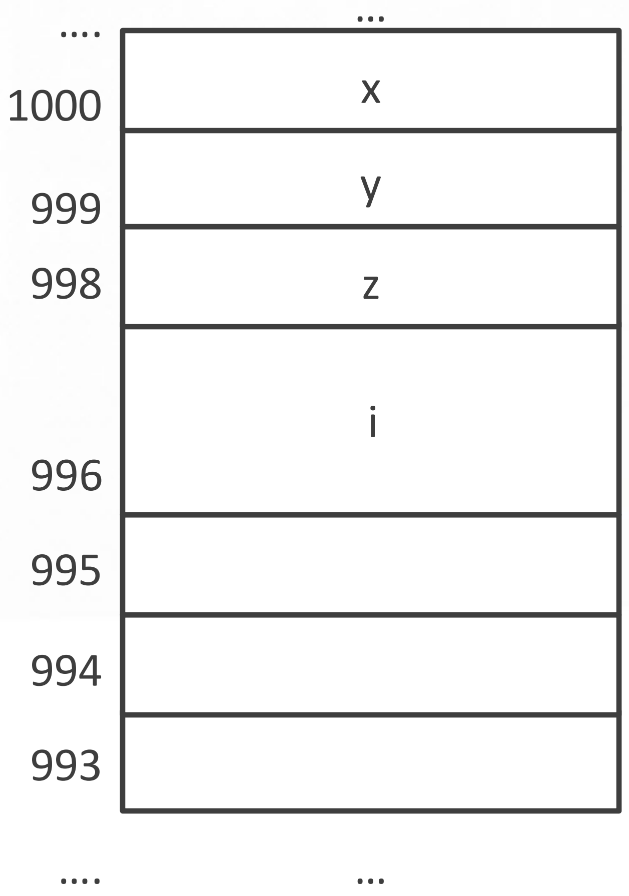
- int main() {
  - char x, y, z;
  - short i;
  - for (i = 0; i < 10; i++) {
    - doNothing();
  - }
  - return 0;
- }



Example: Stack

- void doNothing() {
  - char c;
- }

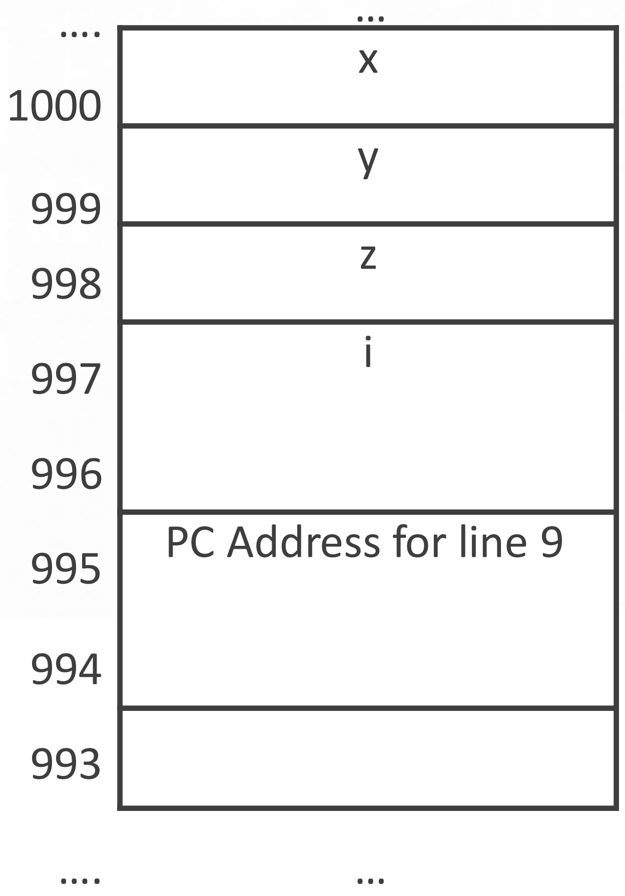
- int main() {
  - char x, y, z;
  - short i;
  - for (i = 0; i < 10; i++) {
    - doNothing();
  - }
  - return 0;
- }



Example: Stack

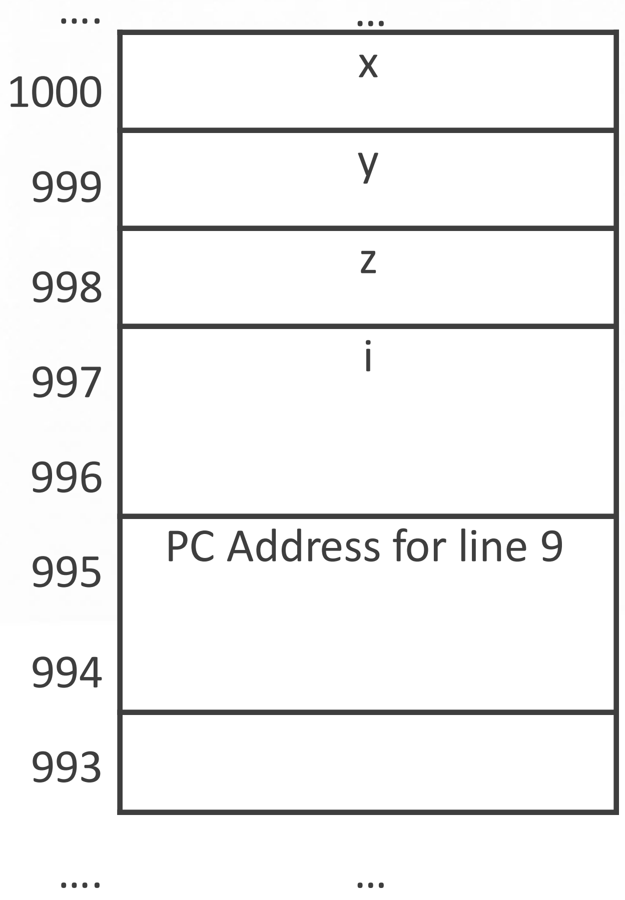
- void doNothing() {
  - char c;
- }

- int main() {
  - char x, y, z;
  - short i;
  - for (i = 0; i < 10; i++) {
    - doNothing();
  - }
  - return 0;
- }



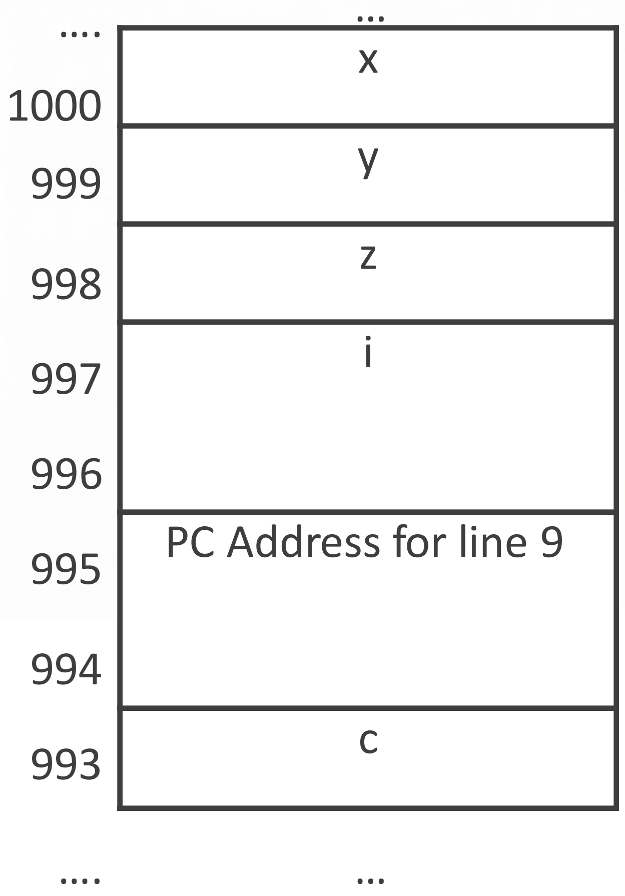
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        short i;
        for (i = 0; i < 10; i++) {
                •   doNothing();
        }
        return 0;
}
```



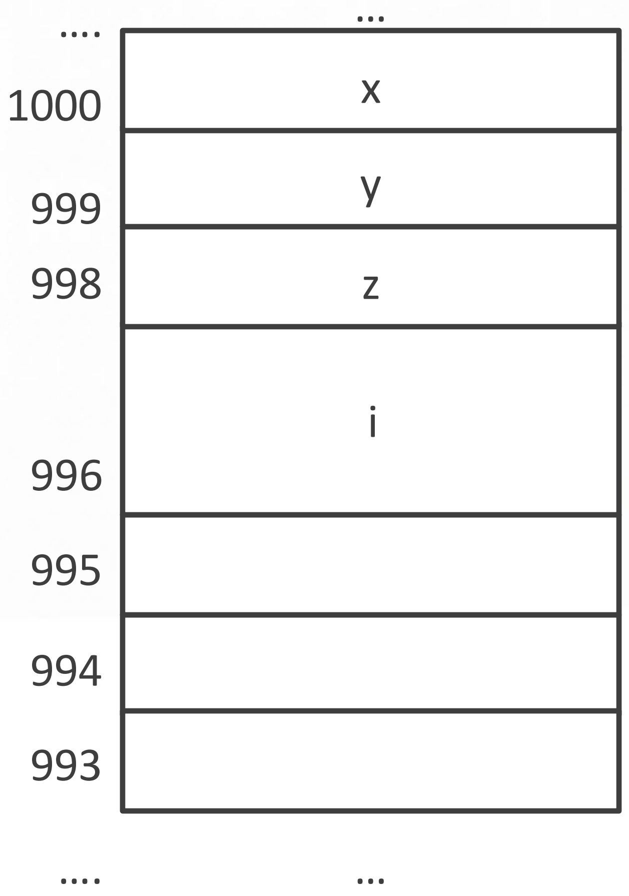
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        short i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



Example: Stack

```
void doNothing() {
        • char c;
    }

int main() {
        char x, y, z;
        short i;
        for (i = 0; i < 10; i++) {
            doNothing();
    }
    return 0;
}
```



Question 1: Stack vs Heap Memory Allocation in Embedded C_(1)

You are developing an embedded application for a real-time system with limited RAM (16KB). You need to decide whether to use stack or heap memory allocation.

```
#include <stdio.h>
#include <stdlib.h>
void allocate_stack() {
    int arr[1024];  // Allocate 4KB on stack
    arr[0] = 100;
    printf("Stack allocated value: %d\n", arr[0]);
}
void allocate_heap() {
    int *arr = (int *)malloc(1024 * sizeof(int));  // Allocate 4KB on heap
    if (arr) {
        arr[0] = 200;
        printf("Heap allocated value: %d\n", arr[0]);
        free(arr);  // Free allocated memory
    }
}
int main() {
    allocate_stack();
    allocate_heap();
    return 0;
}
```

Question 1: Stack vs Heap Memory Allocation in Embedded C_(2)

(a) Explain the differences between stack and heap allocation in the given code.
(b) If the program runs on an embedded system with 16KB RAM, which function (allocate_stack() or allocate_heap()) is more dangerous? Why?
(c) What will happen if malloc() fails in allocate_heap()? How should it be handled properly in an embedded system?

Question 2: Stack Overflow Debugging in Embedded C_(1)

Your team is developing firmware for an ARM Cortex-M microcontroller with 8KB of stack memory. During execution, the system randomly resets or crashes.

```
void recursive_function(int n) {
    int local_array[256];  // Large local array
    if (n > 0) {
        recursive_function(n - 1);
    }
}

int main() {
    recursive_function(10);
    return 0;
}
```

Question 2: Stack Overflow Debugging in Embedded C_(2)

(a) Identify the potential stack-related issue in this program and explain why it occurs.
(b) If you were debugging this on an embedded system, how would you detect and prevent stack overflow?
(c) Rewrite the function to prevent stack overflow using a safer approach.