Embedded C Programming

02 - Basics of Embedded C

Nitin Chandrachoodan, IIT Madras

Writing Efficient C code

for Embedded Systems

Objective: Understand how to optimize C code for memory and execution efficiency in embedded systems.

C as a programming language

Why?

- Close to hardware - simple constructs that compile well to assembly/ISA
- Control over system resources
- Structured - functions, loops, structs etc. enable higher level of abstraction

C as a programming language

Why?

- Close to hardware - simple constructs that compile well to assembly/ISA
- Control over system resources
- Structured - functions, loops, structs etc. enable higher level of abstraction

Where?

- Operating Systems
- High-performance code: scientific compute, memory intensive
- Databases
- Network applications
- Game development

C: A quick recap - Data Types

- Storing and manipulating data
- `int, char, float, double, short, …`
- Sizes can be implementation dependent
  - Usually int is 32 bit, char is 8 bit.
  - float and double can be expensive without hardware support
  - signed, unsigned variants for integer types
- `int8_t, uint8_t, int16_t` etc. defined in `<stdint.h>`
  - Especially useful in embedded systems needing precise control of memory

C recap: complex data types

- struct
    - encapsulate multiple data as fields of a single type
    - code readability and maintainability
- enum
    - enumerated types
    - portability and modifiability
- union
    - overlapped storage between different data types
    - interesting use cases in embedded context

C recap: conditionals and loops

- `if / else`
  - conditional execution: very common use case. eg. check sensor value and respond
- `switch / case`
  - compact way to express multiple choices
  - can in principle always be replaced by if / else - but more readable
- Loops:
  - `for` – deterministic loops with predetermined iteration counts usually
  - `while` / `do-while` – more flexible loops with varying exit conditions
  - infinite loops: `break, continue` - common scenarios for interaction

C recap: functions

- Reusable code
- Extendable with parameters (function arguments)
- return types
  - compute and return a value
  - update a value in memory through pointers (harder to reason about)
- Modular design
  - readability and maintainability
  - effective use of headers, prototypes etc. needed

C recap: variables, scope, stack vs heap

- Stack
  - specific memory area for each function call
  - recursive function calls possible due to separate stack for each call
  - easy to pass small amounts of data back and forth between caller and callee
- Heap
  - Longer-lived or Larger blocks of memory
  - Globals
- Careful management of scope:
  - Variable lifetime important in embedded systems: long-running codes
  - Globals and permanent allocations can be harmful in resource constrained environments

Pointers - a lot more to say about them…

Embedded C

Why?

- Direct hardware manipulation capabilities
- Constrained resources
- Requirement for highly optimized operations

Where?

- Consumer electronics: TV, kitchen, camera, household
- Automotive
- Medical

Constraints:

- Battery: energy / power
- Size: handheld, portable
- Cost: add-on device

Similarities and Differences

```c
#include <stdio.h>

int main() {
    char data[100];
    printf("Enter some data: ");
    fgets(data, sizeof(data), stdin);
    printf("You entered: %s\n", data);
    return 0;
}
```

Similarities and Differences

```c
#include <stdio.h>

int main() {
    char data[100];
    printf("Enter some data: ");
    fgets(data, sizeof(data), stdin);
    printf("You entered: %s\n", data);
    return 0;
}
```

```c
#include <stdint.h>

#define SENSOR_DATA_REGISTER (*(volatile uint8_t*)0x40001000)

int main() {
    uint8_t sensorData = SENSOR_DATA_REGISTER; // Read from a memory-mapped hardware
    // Process sensor data or take action based on the value
    return 0;
}
```

Summary

- C used as a starting point for embedded programming
  - Close to hardware
  - Easy to appreciate importance of memory and resources
- Manipulating memory - clever (mis)use of addresses to talk to IO
- Basics of C important as background for embedded engineers:
  - Data Types
  - Control
  - Functions and Modular programming
  - Memory

Strategies for Efficiency

- Writing efficient code
- Best practices

“Efficiency”

Key term in the context of embedded systems

- Resource efficiency: memory, CPU, IO bandwidth
- Speed: make good use of available CPU instructions
- Power: sleep states, power modes
- Code maintainability: long-running, infrequent updates, long support cycles

Why Efficiency Matters

- “Fire-and-forget” nature of applications:
  - write once, run forever
- Minimize system cost:
  - development cost
  - deployment
  - maintenance (over lifetime)
- Performance: especially in real-time applications

Memory Efficient Data Types

- **Purpose:**
  - Ensure that the data storage does not consume more memory than necessary
  - Critical in systems with limited memory.
- **Example:**
  - Use uint8_t for storing 8-bit values instead of a standard int (which might be 16 or 32 bits on many platforms)
  - Counter / loop index using smaller range values
- **Benefit:**
  - Reduces the amount of RAM and ROM used
  - Smaller and cheaper microcontrollers

Local Scope instead of Global

- **Purpose:**
  - Global variables stored in a fixed memory location throughout the application's lifecycle
  - Increased memory usage; potential memory corruption if not handled carefully.
- **Example:**
  - Function parameters using locals instead of *global state*
  - Reduce dynamic allocation and deallocation
- **Benefit:**
  - Enhances modularity and reusability of code
  - reduces memory leakage
  - Decreases the chance of bugs related to unintended side-effects on *global state*.

Loops and Function Calls

- **Purpose:**
  - Function calls and loop iterations consume CPU cycles.
  - Critical code sections - identify and optimize
- **Example:**
  - Code optimization: replace multiple calls in loop with single call before loop
- **Benefit:**
  - Reduces CPU cycle consumption
  - Time-sensitive operations

Compiler Intrinsics and Built-in Functions

- **Purpose:**
  - Specific processor features not easily accessible from standard C code.
- **Example:**
  - Bit manipulation, arithmetic operations, and special control instructions.
  - SIMD (Single Instruction Multiple Data) operations
- **Benefit:**
  - More efficient and faster code execution by reducing the overhead of function calls
  - Directly leveraging the hardware capabilities.

“Best Practices”

Collective wisdom of the ancients…

- Common patterns observed to be useful
- May be specific guidelines for security
- Not mandatory - not enforced by compilers
    - Lint checkers (linters) often used for providing suggestions

Avoid Floating Point

- Explanation:
    - Floating point easy for humans to use and understand
    - But resource-intensive - require more CPU cycles to process than integer operations.
    - Many embedded systems do not have dedicated floating-point hardware
- Alternative:
    - Fixed-point computation: integer operations to approximate decimal values by scaling.
    - Harder for humans - overflow, range normalization etc. manually done
- Benefit:
    - Improved performance and reduced computational overhead

Dynamic Memory Management

- Explanation:
  - Memory leak: when dynamically allocated memory is not freed properly.
  - Embedded systems: memory is limited => system instability or crashes.
- Strategies:
  - Pair malloc with free in the same scope or logic block.
  - Use static memory allocation where practical.
- Benefit:
  - Stability
  - Especially critical in embedded applications that are meant to run continuously without reboot.

Compiler Optimizations and Directives

- Explanation:
  - Settings that allow the compiler to modify the generated machine code
  - Improve performance or reduce size.
- Common Flags:
  - -O2 optimizes mainly for speed
  - -Os optimizes for size by reducing the code footprint.
- Benefit:
  - Enhance the performance and efficiency

Key takeaway: learn the capabilities of your tools

Inline functions, Macros - use with care

- Explanation:
  - Inline functions and macros can replace function calls with actual code
  - Eliminate overhead of a call and return sequence.
- Guidelines:
  - Excessive inlining can lead to larger binary size.
  - Macros can have side effects and affect readability.
- Benefit:
  - When used correctly, these tools reduce the runtime overhead and can speed up critical sections of the code.

Test Early and Thoroughly

- Explanation:
  - Catching inefficiencies early can prevent complex bugs and system failures that are harder to diagnose at later stages.
- Strategies:
  - Unit tests, integration tests, and system tests.
  - Use mock objects and simulation environments to replicate and test various system states and edge cases.
- Benefit:
  - Reduced chance of failure “in the field”

Key takeaway: software engineering best practices

Profiling

- Explanation:
    - Identify resource-intensive parts of your code.
- Common Tools:
    - gprof, Valgrind (generic)
    - Specific embedded profiling tools - compiler / tool-chain dependent
    - Measure time spent in functions and memory usage.
- Benefit:
    - Where to focus optimization efforts

Summary

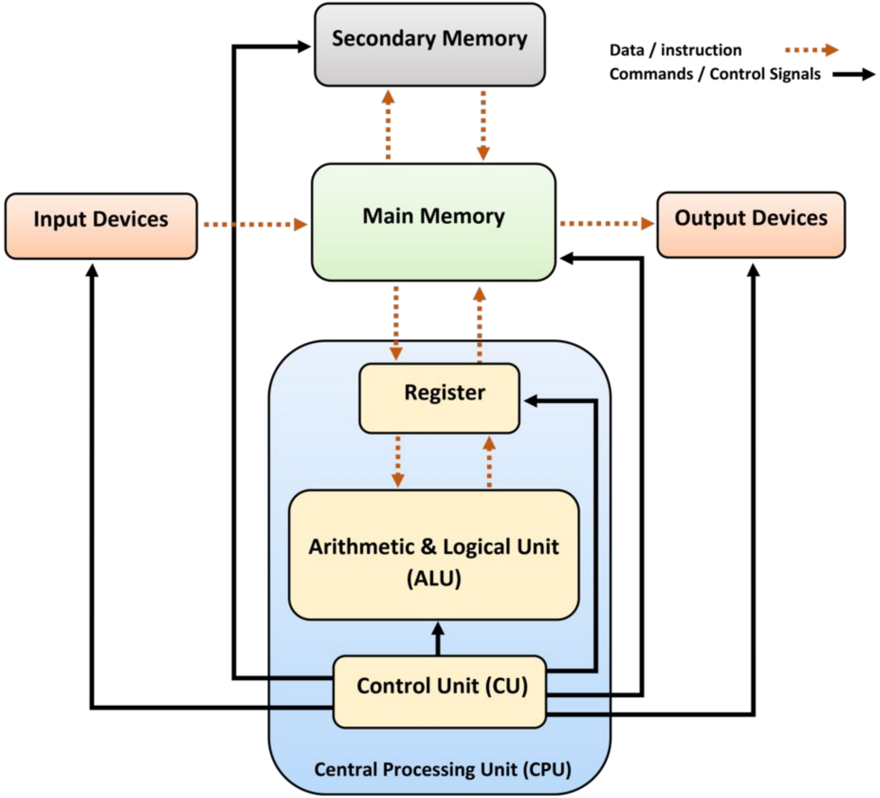
- Embedded C vs. Traditional C
    - Use cases, coding styles
- Efficiency
    - Especially important given constrained nature of designs
- Strategies
    - Specific to embedded systems / software
    - Generic approaches common to software engineering projects

Learn your tools! Make use of them!!

Memory Management

- Types of memory
- Memory management techniques

Memory in Context



- Computation: ALU + decode logic
- I/O: peripherals; interaction with outside world

Memory:

- storage of temporary data, variables
- instruction code, operating system
- configuration data

Memory Types: RAM

- Random-Access Memory
  - Any location can be accessed and read in (approximately) constant time
  - **Volatile:** data lost on power off (not strictly per definition, but common meaning)
- Contrast with:
  - Serial access memory: tape drive; or even a book?
  - Not opposite of ROM (read-only memory): RAM can also be ROM…
- Store:
  - program runtime: stack, heap
  - temporary variables that do not fit in registers

Memory Types: ROM / EEPROM

- Read-Only Memory / Electrically Erasable Programmable ROM
  - Technically also a variant of RAM, but never referred to as such
  - **Non-volatile:** retained even on power off
- Programmability
  - Nil - ROM
  - Limited: EEPROM - meant for infrequent updates
- Store:
  - Configuration information (vendor/product ID, MAC address of ethernet/bluetooth)
  - Limited capacity

Memory Types: Flash

- Flash: specific type of semiconductor technology
  - **High density** possible - store large amounts of data (esp. NAND flash)
  - **Non-volatile**
- Programmability:
  - 1000s to millions of “write cycles” possible
  - Largely replace PROM except for specific niche use cases
- Store
  - Program code
  - Configuration
  - **Cannot** be used for runtime data

Memory Types: Others

- FRAM or FeRAM
  - Ferroelectricity principle: non-volatile
  - Unified memory for storing both code and non-volatile data (eg. logging applications)
  - eg. TI microcontrollers
- Phase-Change memory
  - Higher durability and more write cycles than Flash
  - More expensive
- MRAM
  - Magneto-resistive tech
  - Expensive

Microcontrollers very cost conscious market

Memory Management

What?

- Run-time storage of temporary values, arrays, data
- Either for processing or transmission: buffering
- Cannot fit in registers, but needed for rapid update

Memory Management

What?

- Run-time storage of temporary values, arrays, data
- Either for processing or transmission: buffering
- Cannot fit in registers, but needed for rapid update

Why?

- Limited amount of storage available
- Make best use of available storage
- Decision taking:
    - Run-time
    - Compile-time

Static Memory Allocation

- Before program runs: decided by compiler
- Allocate space for all variables that program will ever use
  - Buffers, temporary storage
  - Not for registers etc. - these are anyway mapped by compiler
- Typically remain in memory for lifetime of execution
  - Most embedded programs never terminate: so this means till restart

Advantages of Static Allocation

- Predictability:
  - Memory allocated at compile time
  - Does not change at run-time
- No fragmentation:
  - compiler can place arrays as close together as possible
  - No possibility of resizing or changing array type or location
- Lower overhead
  - No run-time checks needed
  - Out of bounds access etc can be checked at compile time: all sizes known

Usage in Embedded Systems

- Simplicity:
  - Fixed memory layout: addressing, access through simple computations
- Reliability
  - Stable and predictable: no memory allocation failures possible
  - Especially useful in medical, automotive: safety critical applications
- Real-time
  - No runtime overhead for allocating
  - No defragmentation, garbage collection

Possible drawbacks?

- Inflexible:
  - All data structures and variables must be known at compile time
- Wastage:
  - alignment or safety margins - allocate excess that is not used
- Scalability:
  - dynamic data structures like trees can handle growing data efficiently, but are hard to implement with static allocation
- Development complexity and maintenance overhead:
  - all data storage must be planned for ahead of time
  - changes in future could require significant changes in large parts of code

Dynamic Memory Allocation

- Allocate memory at run-time
- Dynamic data:
    - data logging, buffering: array sizes not known till system runs
    - network buffering: can hold data if downstream not ready to receive
- malloc, calloc, realloc, free
    - Pointers assigned at run-time
- Requires a run-time allocator

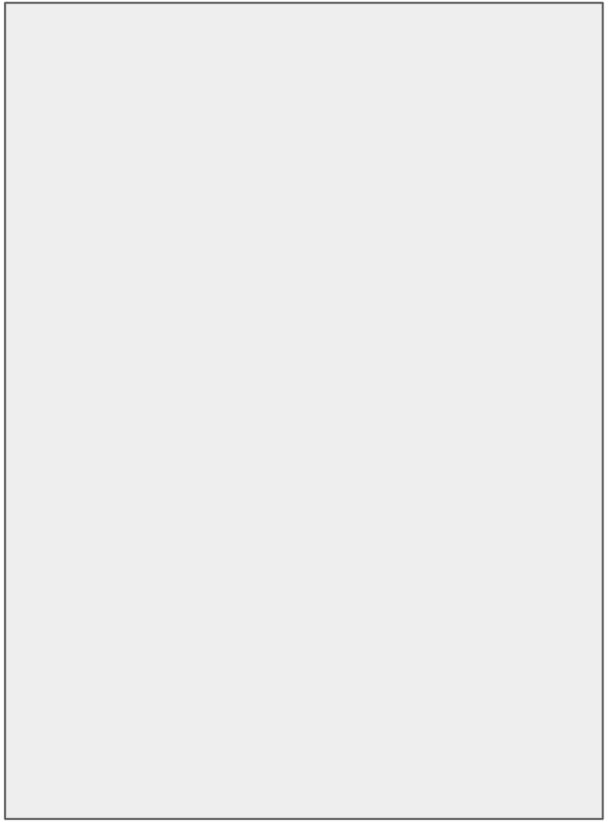
Advantages of dynamic allocation

- Adapt to actual usage scenario
- Single code base can easily adapt to different microcontroller systems
  - Amount of memory need not be fixed ahead of time
  - More memory will automatically be used as needed
- Average case instead of worst case buffer usage possible
  - Handle worst case by dropping packets etc.: still useful

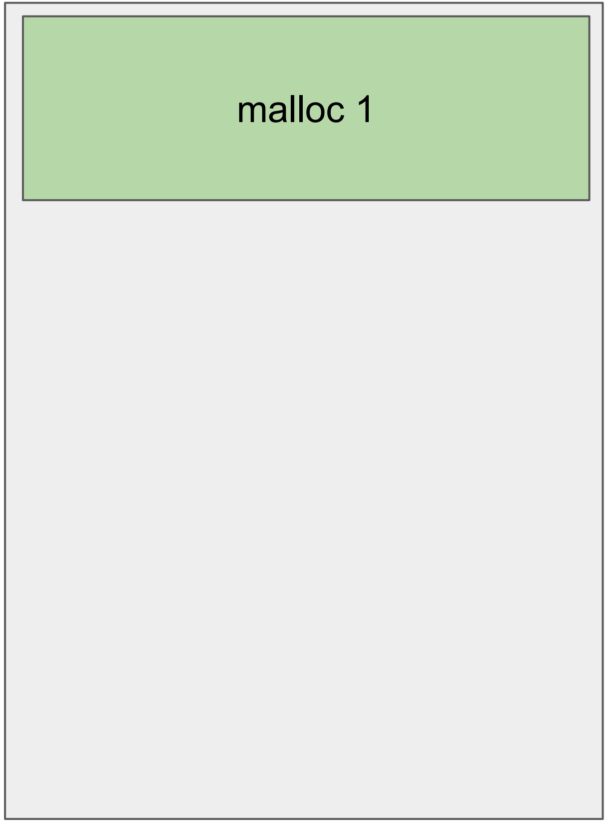
Problems with dynamic allocation

- Fragmentation
  - Chunks of data allocated from heap

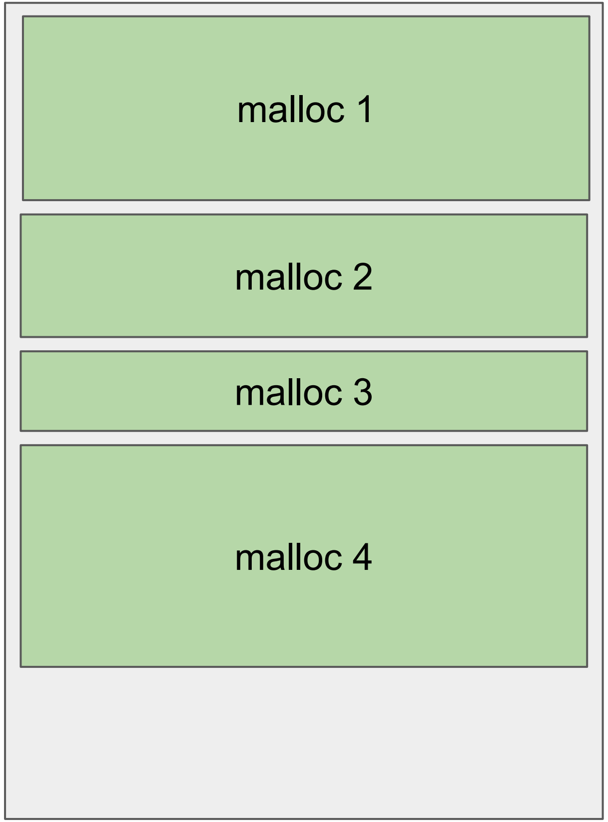
Total available memory



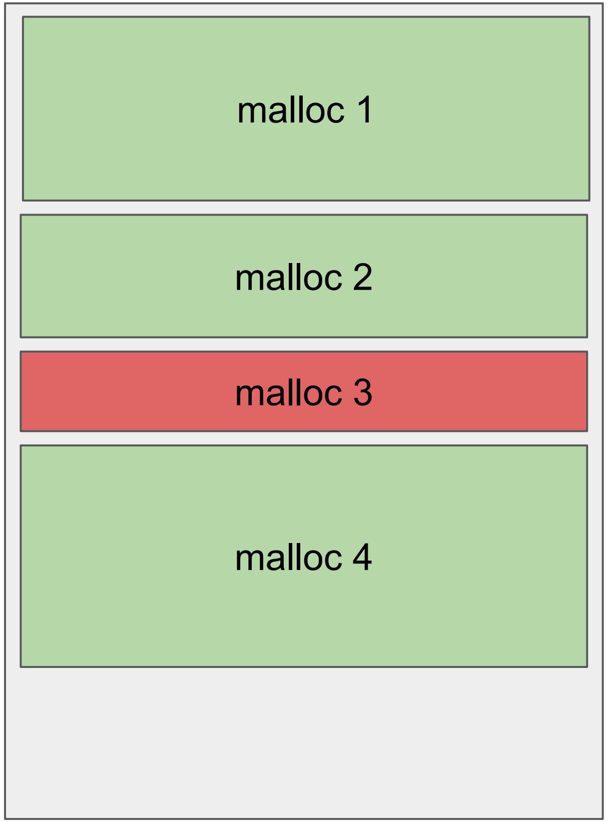
Total available memory



Total available memory

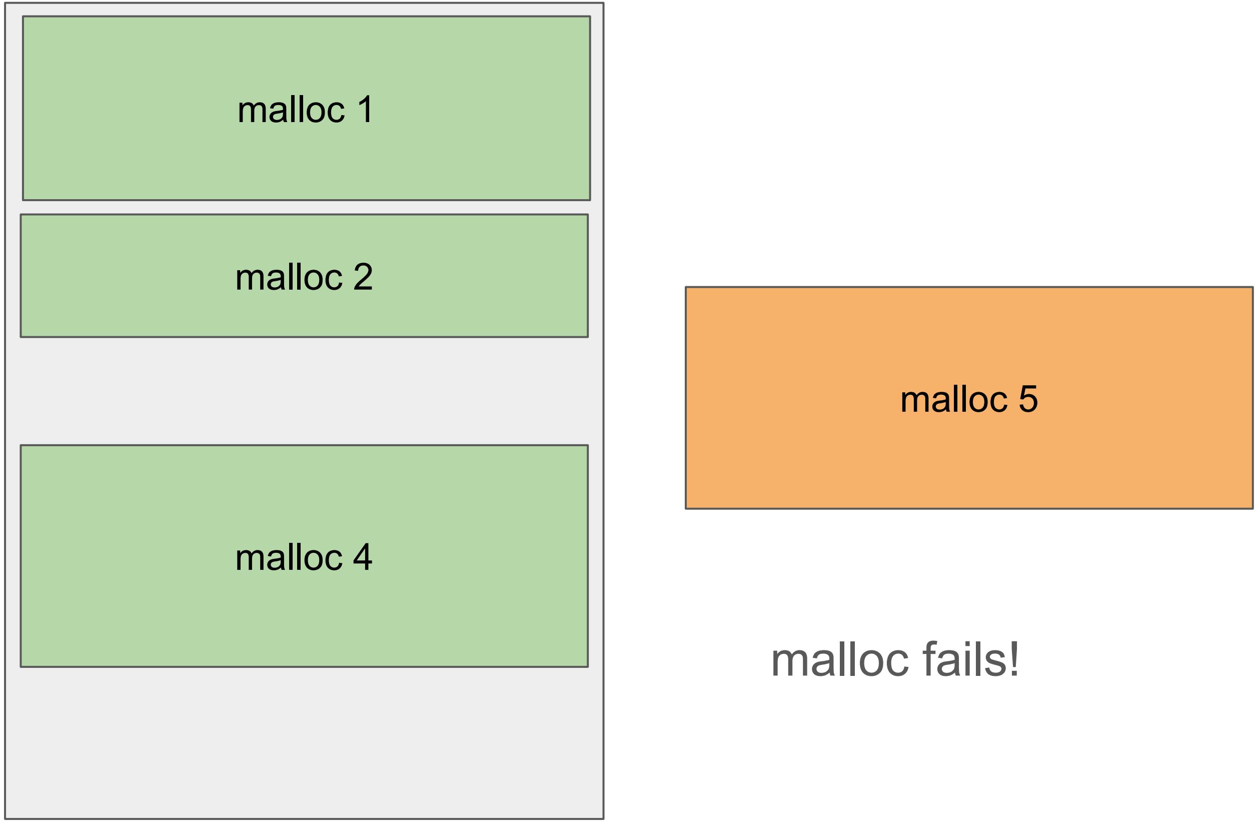


Total available memory



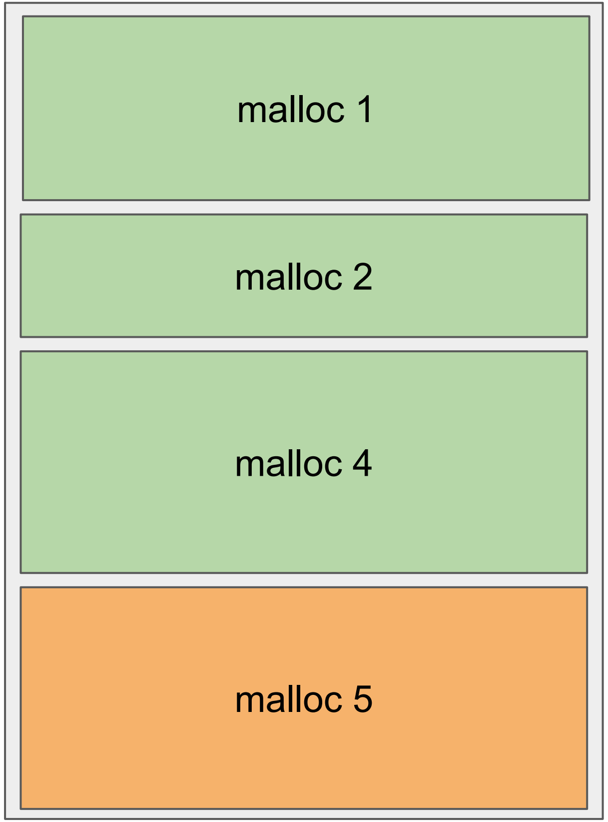
Free unit 3

Total available memory



malloc fails!

Total available memory



Defragment: total memory was sufficient

malloc succeeds!

Problems with dynamic allocation

- Fragmentation
  - Chunks of data allocated from heap
- Unpredictable memory usage
  - Sudden loss of network leads to buffer growth
- Memory leaks
  - malloc should be followed by free as soon as possible
  - else memory is allocated and not released though unused

Impact of improper dynamic memory management

- Runtime failures
  - Failure to allocate memory: crash system
  - Overuse of memory: insufficient memory for OS functions
  - Priority of tasks impacted
- Non-determinism
  - Garbage collection or memory defragmentation
  - Harder to guarantee real-time performance

Overall Recommendation

- Preferred: use static allocation for embedded systems where possible
- Minimize use of dynamic variables / structures
  - Preferably in non-critical portions
  - Allow failure modes: dropping packets etc.
- Separate memory allocation
  - critical (OS, real-time)
  - non-critical (data logging, signal processing)

Memory Optimization - Data Type selection

```
uint8_t sumValues(uint8_t values[], size_t numValues) {
    uint8_t sum = 0;
    for (size_t i = 0; i < numValues; i++) {
        sum += values[i];
    }
    return sum;  // Return the sum as uint8_t
}
```

- Arrays of appropriate size and accumulator as required

Memory Optimization - Data Type selection

```
void delay_ms(uint32_t milliseconds) {
    for (uint32_t i = 0; i < milliseconds; i++) {
        for (volatile int j = 0; j < 1000; j++) {
            // Busy wait loop for roughly 1 millisecond
        }
    }
}
```

- Arrays of appropriate size and accumulator as required
- Temporary variables only marginally affected: register size dominates
  - Maximum possible value of milliseconds?
  - j < 1000: is int needed? uint16_t sufficient

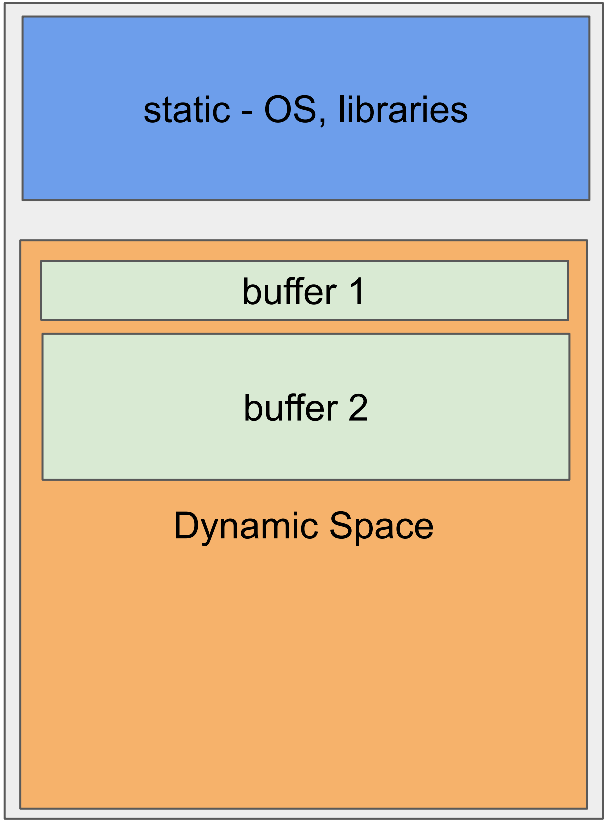
Memory Optimization - Global vs Local

- Global variables on heap:
  - lifetime of program - no deallocation / out of scope
  - Compete with dynamic space
- Local variables:
  - allocate on stack - only when function called
  - potential runtime overhead with allocation, but memory used only when needed
- Static variables:
  - heap allocated - similar to global - allows retaining state
  - similar benefits to global, similar drawbacks
  - avoids other problems with globals being accessible / modifiable anywhere

Memory Optimization - Data Packing

```c
// Function to pack RGB values into a 16-bit value
uint16_t packRGB(uint8_t r, uint8_t g, uint8_t b) {
    uint16_t packed = 0;
    packed |= (uint16_t)r >> 3; // 5 bits for red
    packed |= (uint16_t)g >> 2 << 5; // 6 bits for green
    packed |= (uint16_t)b >> 3 << 11; // 5 bits for blue
    return packed;
}
```

Memory Optimization - Pooling



Dynamic space available as pool for buffers and non-critical blocks

Summary

- Memory: one of the most important limited resources in embedded systems
- Volatile vs. Non-volatile
  - OS, configuration etc. non-volatile
  - Runtime data
- Static vs Dynamic memory allocation and management
- Strategies and best practices for efficiency

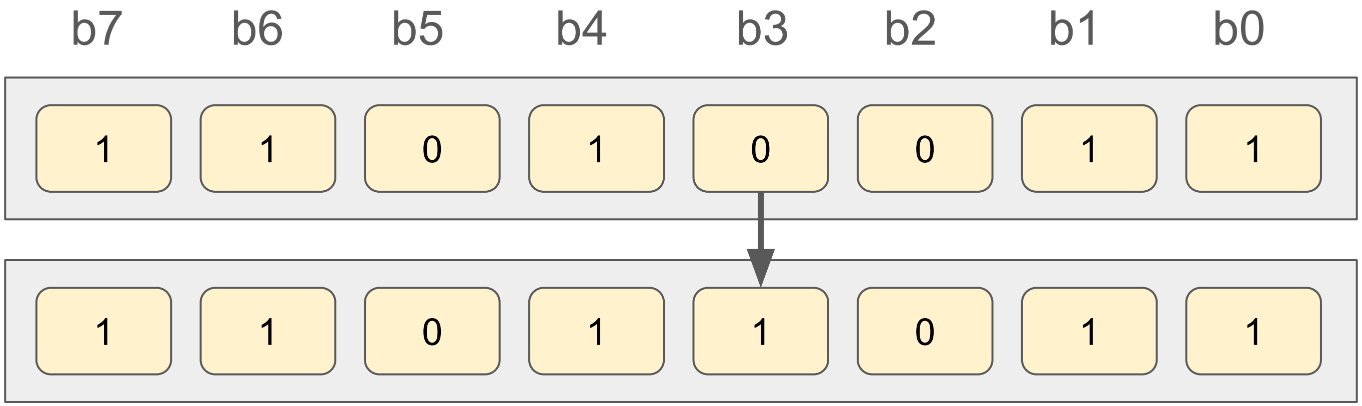
Bit Manipulation

- Direct control of hardware
- Atomic operations

Why bit manipulation

- Embedded systems: direct control of hardware units
  - Switches / control signals
- CPU word size: 8, 16, 32 bits etc.
  - Not designed for accessing / modifying individual bits
- Compromise
  - Multiple related bits often grouped together
  - Common constructs used to test values or set/reset individual bits

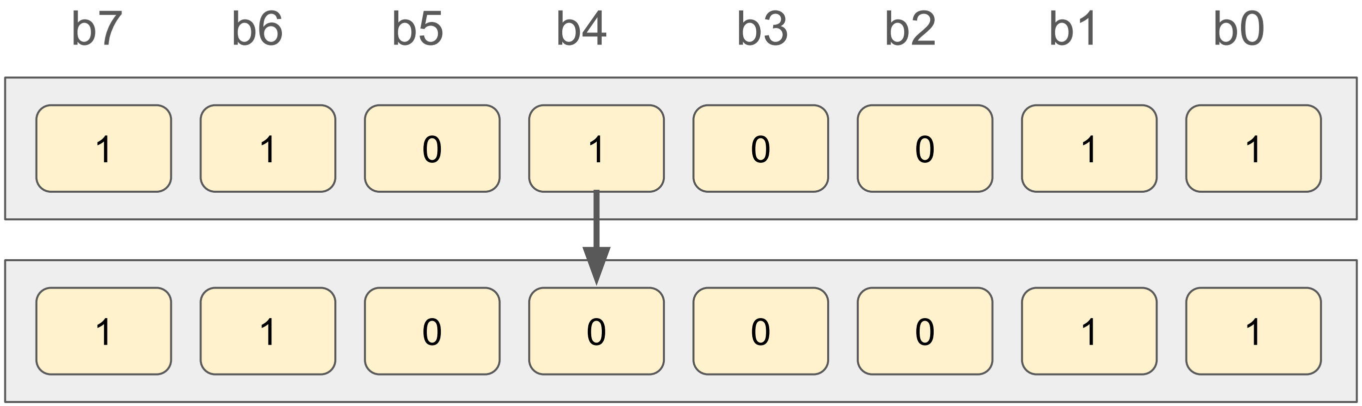
Bit manipulation: Set a bit



Requirement: Set bit b3 to 1
Bitmask for setting: **(1 << bit_position) = 00001000**

data = data | (1 << 3)

Bit manipulation: Clear a bit



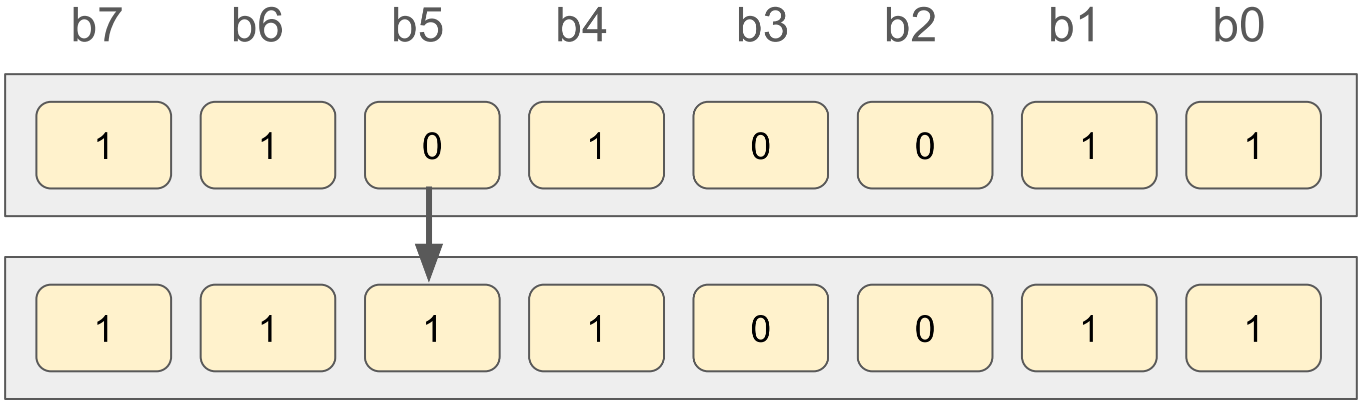
Requirement: Set bit b4 to 0

Bitmask for setting: **~(1 << bit_position) = ~00010000**

= 11101111

data = data & ~(1 << 4)

Bit manipulation: Toggle a bit



Requirement: Set bit b3 to 1
Bitmask for setting: **(1 << bit_position) = 00001000**
Operation to toggle: **XOR**
data = data ^ (1 << 5)

Bit manipulation: Check if a bit is set

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	1	0	0	1	1

Requirement: Check if bit b6 is set to 1
Bitmask for setting: **(1 << bit_position) = 00001000**
Operation to test: **AND and check result**
result = (data & (1 << 6) != 0)
= (11010011 & 01000000) -> 01000000 != 0

Example: turn LED on or off

```
// Set or reset the LED
// state: 0 -> Turn off the LED, 1 -> Turn on the LED
void set_led_state(int state) {
    volatile uint32_t *odr = (volatile uint32_t *)(GPIO_PORT_BASE + GPIO_ODR_OFFSET);
    if (state) {
        *odr |= (1 << LED_PIN);  // Set the bit corresponding to the LED pin to turn
    } else {
        *odr &= ~(1 << LED_PIN);  // Clear the bit corresponding to the LED pin to tu
    }
}
```

Example: check if button pressed or not

```c
// Read the state of the button
// Returns 1 if the button is pressed, 0 if it is not
int read_button_state() {
    volatile uint32_t *idr = (volatile uint32_t *)(GPIO_PORT_BASE + GPIO_IDR_OFFSET);
    return (*idr & (1 << BUTTON_PIN)) ? 1 : 0;  // Check if the button pin is high
}
```

Atomic Operations

- Race conditions:
  - Difference between time a value is read and when it is written / updated
- **data = data | (1 << 3)**
  - First read the value of data
  - Then create a temporary value where bit 3 is set
  - Write that value to data
- Inbetween step 1 and 3 it is possible that the value of data changed
  - Only possible when multiple sources can cause a value to change
  - Multi-processor systems, multiple IO drivers for a pin etc.
- Atomic operation: read-modify-write guaranteed to be done as one operation
  - Requires hardware support on CPU side

Bit Banding

- Specific to ARM-Cortex processors
- Specific 1MB area of memory is mapped to a 32MB address space
  - 32 addresses in bitband memory correspond to 1 address in original memory
  - Each bit has a separate address and can be directly accessed
  - Set or Clear operations are atomic: will complete without interruption
  - Need not look at other bits of word

```
#define BITBAND_PERIPH(addr, bit)
    ((BITBAND_PERIPH_BASE + (
        (addr - PERIPH_BASE) * 32) + (bit * 4)))
```

Direct bit manipulation

```
// Using bit-banding to set and clear a bit atomically
void set_pin() {
    *(volatile uint32_t *)(BITBAND_PERIPH((uint32_t)gpio_odr, ODR_PIN_5)) = 1;
}

void clear_pin() {
    *(volatile uint32_t *)(BITBAND_PERIPH((uint32_t)gpio_odr, ODR_PIN_5)) = 0;
}
```

Advantages of bit-banding

- Atomic:
  - Simplified access for set/clear/toggle bits in control registers
  - Reduces need for locking mechanisms while bits being manipulated
- Real-time performance:
  - Race conditions avoided
  - Deterministic (related to Races)
- Code clarity
  - Readability and Maintainability

Summary

- Bit manipulation essential part of most embedded systems programs
- Interaction with peripheral devices
  - Control of hardware
  - Receiving inputs
- Standard “templates” for bitmasks and modification
- Atomic operations
- Bit-banding in ARM processors

Pointers

- Use of pointers in embedded programming
- Best practices

Pointers in C

A pointer is a variable that stores the address of another variable

- Variables in C are just storage locations for data
- Stored either in registers or in memory
    - Anything non-trivial likely to be in memory
- Can use the address of the variable in memory to manipulate it

Indirect access to a variable

Syntax

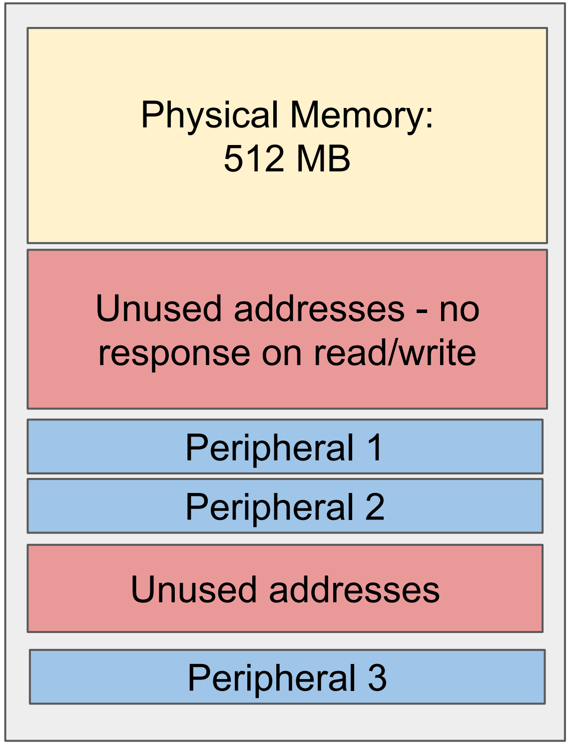
```
int *ptr;    // Declare a pointer to an integer
ptr = &x;    // Get the address of 'x' and store in 'ptr'
*ptr = 42;   // Indirectly update the value stored in 'x'
```

- **Usage:** array manipulation, dynamic memory allocation and access, linked lists, trees and other complex data structures

Memory Map

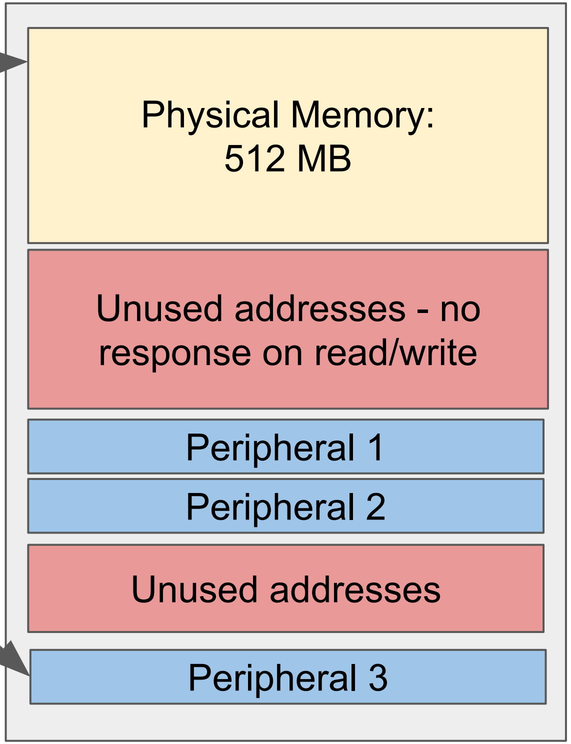
Pointer on 32-b arch: 32-b wide

- Up to 2^{32} addresses possible.
  - ~ 4×10^9
- Actual memory less: say 512MB
- Remaining addresses unused
  - Map some of them to peripherals
  - Use digital logic to decode the address - activate some hardware
  - Response comes from hardware
  - CPU cannot distinguish



Memory Map

```
int *ptr1 = 0xFF000000;
*ptr1     = 42; // write mem
int *ptr2 = 0x00000000;
*ptr2     = 1;  // write IO
```



Embedded Systems Peripheral Access

- Common scenario: peripheral mapped to an address or address range
- Example:

```
#define SENSOR_DATA_REGISTER (*(volatile uint8_t*)0x40001000)

int main() {
    uint8_t sensorData = SENSOR_DATA_REGISTER; // Read from a memory-mapped hardware
    // Process sensor data or take action based on the value
    return 0;
}
```

Volatile pointers

`volatile` keyword in C

- Explicitly mark a pointer as NOT to be optimized or cached
- Ensures that compiler will not silently remove it as not useful
- Essential for proper behaviour with memory mapped systems

```c
// Define a pointer to the GPIO port output register and mark it as volatile
volatile uint32_t* const gpio_port_out = (volatile uint32_t*) GPIO_PORT_OUT_REGISTER;

void toggle_led(uint32_t pin) {
    *gpio_port_out ^= (1 << pin);  // Toggle the specific pin
}
```

More pointers in embedded systems

- Direct interfacing with peripheral devices
  - eg. Read analog-to-digital converted values
- Efficient data handling
  - Buffers used in UART, SPI etc.
  - Higher speed/size: video memory, network buffers etc.

Pointer Safety

- Dangling Pointers
  - Pointers just hold a value: not checked for validity
  - Can point to a location that has been freed or gone out of scope
- Wild pointers
  - Uninitialized pointers
  - Use before allocation
  - No guaranteed defaults: NULL pointers or random pointers both possible

Best Practices and Safety

- Initialization:
  - Always explicitly set to NULL on declaration, until initialized
- Check validity
  - Ensure NOT NULL before dereferencing
  - eg. result of `fopen()` in regular C, or for memory map
  - What about explicitly using address location 0x0000??? (Hint: don't write code like this)
- malloc / free
  - Prevent memory leaks and dangling pointers
- **`const`** qualifiers: let compiler know modification through this pointer is NOT expected
- Use pointer arithmetic (**`*(p+1)`** etc.) with extreme caution

Summary

- Memory Map is essential part of embedded programming
- Pointers used to manipulate peripheral devices directly
  - Immensely powerful
  - Immensely dangerous
- Many pitfalls and traps possible
  - Careful use of best practices needed to avoid problems
  - Some memory management sanitization tools available to catch and prevent memory errors