Microcomputer Systems 1

Jil Zerndt HS 2025
Lecture Content

Course Information

Course Details

Course Name: Microcomputer Systems 1
Course Code: MC1 (1aekwm6)
Semester: HS2025 (Herbstsemester 2025)
Instructor: A. Rüst
Credits: Institution: ZHAW School of Engineering, InES Institute of Embedded Systems
Lecture Time: Fridays 14:00-15:35, TH 343
Lab Time: Fridays 12:00-13:45 AND 16:00-17:45, TE 502 R. Fritschi - TE 507 A. Rüst

Learning Objectives

By end of course, students should be able to:

- Know design patterns in C and can apply them in their code
- Apply real-time programming techniques to process asynchronous events from several peripheral blocks
- Structure a firmware for a microcontroller system appropriately and partition it into modules
- Understand the interactions of software in C with hardware (memory, peripherals) and can apply this understanding to write efficient and stable software
- Develop, verify and debug microcontroller systems
- Systematically design and implement applications using state event techniques for several cooperating state machines

Exam Details

Exam Format

Type: Written exam on paper (Schriftliche Semesterendprüfung)
Duration: TODO: confirm duration
Date: End of semester
Allowed Materials:

- Open Book (including online resources)
- Use of electronic aids and internet allowed
- No use of generative AI systems (e.g., ChatGPT is forbidden)
- Communication with third parties prohibited (except for exam supervisors)

Exam Regulations:

- Exam must be completed individually and in person
- Further handout of exam questions and solutions is prohibited
- Copying or recording of exam or individual questions is prohibited
- Dishonest behavior results in grade 1 and may have additional legal consequences

Self-declaration of Independence: Students must confirm they have completed the exam independently without help from third parties and without generative AI systems

Grade Composition

Component	Weight
Final Exam (Semesterendprüfung)	60%
Lab Pre-grade (Vornote aus 12 Praktika)	40%
Total	**100%**

Formula: Modulnote = Vornote * 0.4 + Note Semesterendprüfung * 0.6
Passing Grade: 4.0 out of 6.0

High-Priority Topics for Exam

TODO: Identify high-priority topics after analyzing all lecture materials and old exams

Assignments

Laboratory Sessions

Lab Structure

Number of Labs: 12 practical labs
Duration: Varies by lab (students work on-site)
Location: TE 502 R. Fritschi and TE 507 A. Rüst
Attendance: Mandatory
Assessment: 39 possible points total, linear grading scale (39 points = grade 6.0)
Requirements for Credit:

- Code must be cleanly structured and commented
- Program functionality must be successfully demonstrated
- Student must explain the code and answer related questions

Lab Schedule

- **SW1 (19.09):** P_01: Edge detection / debouncing (due 03.10)
- **SW2 (26.09):** P_01: Edge detection / debouncing (due 03.10), P_02: Matrix keyboard (due 10.10)
- **SW3 (03.10):** P_02: Matrix keyboard (due 10.10)
- **SW4 (10.10):** P_04: Power modes (due 17.10)
- **SW5 (17.10):** P_05: Motion Sensor I - Basic access (due 24.10)
- **SW6 (24.10):** P_06: Motion Sensor II - DMA & Power (due 31.10)
- **SW7 (31.10):** P_07: Motion Sensor III - File system (due 07.11)
- **SW8 (07.11):** P_08_09: Cooperating FSMs (due 21.11)
- **SW9 (14.11):** P_08_09: Cooperating FSMs (due 21.11)
- **SW10 (21.11):** P_10: Zephyr basics (due 28.11)
- **SW11 (28.11):** P_11: Zephyr - Real-time operating system (due 05.12)
- **SW12 (05.12):** P_12: Zephyr - Unit Testing (due 12.12)
- **SW13 (12.12):** P_13: Zephyr - Security (due 19.12)
- **SW14 (19.12):** Wrap-up, Finalize labs

Resources and Course Materials

Information Sources

- **Course overview, learning objectives, assessment, schedule:** MC1_T00_introduction.pdf
- **Module description, learning objectives, grading:** Course Overview.pdf

Theory (Lectures)

- **L01: Understanding Memory and I/O - The Basics of Embedded Software - SW1 (19.09)**
  MC1_T01_understanding_memory_and_IO.pdf
- **L02: Power Management and Low Power Applications - SW2 (26.09)**
  MC1_T02_power_management.pdf
- **L03: Digital Sensors - Direct Memory Access (DMA) - SW4 (10.10)**
  MC1_T03_DMA.pdf
- **L04: From Hardware Timer to Scheduler - SW5 (17.10)**
  MC1_T04_from_timer_to_scheduler.pdf
- **L05: Structuring Embedded Software - part 1 - SW6 (24.10)**
  MC1_T05_embedded_SW_structures_part1.pdf
- **L06: Partitioning Reactive Systems - Concurrent Operating Finite State Machines - SW7 (31.10)**
  MC1_T06_FSM_basics.pdf
- **L07: RTOS - Real-Time Operating Systems - SW8 (07.11)**
  MC1_T07_RTOS.pdf
- **L08: RTOS - Real-Time Operating Systems (continued) - SW9 (14.11)**
  MC1_T08_RTOS_continued.pdf
- **L09: Structuring Embedded Software - part 2 - SW10 (21.11)**
  MC1_T09_embedded_SW_structures_part2.pdf
- **L10: Verification - SW11 (28.11)**
  MC1_T10_verification.pdf
- **L11: Verification / Watchdog - SW12 (05.12)**
  MC1_T11_watchdog.pdf
- **L12: Security for Embedded Systems - SW13 (12.12)**
  MC1_T12_security.pdf
- **TODO:** Check if L13 exists for SW14 (19.12): Wrap-up

Labs

- **P01: Edge detection / debouncing - SW1 (19.09), SW2 (26.09)** **due 03.10. - 3pts**
  MCP_edge_detection_debouncing.pdf
- **P02: Matrix keyboard - SW2 (26.09), SW3 (03.10)** **due 10.10. - 3pts**
  MCP_matrix_keyboard.pdf
- **P04: Power modes - SW4 (10.10)** **due 17.10. - 3pts**
  MCP_power_modes.pdf
- **P05: Motion Sensor I - Basic access - SW5 (17.10)** **due 24.10. - 3pts**
  MCP_motion_sensor_I.pdf
- **P06: Motion Sensor II - DMA & Power - SW6 (24.10)** **due 31.10. - 3pts**
  MCP_motion_sensor_II.pdf
- **P07: Motion Sensor III - File system - SW7 (31.10)** **due 07.11. - 3pts**
  MCP_motion_sensor_III.pdf
- **P08_09: Cooperating FSMs - SW8 (07.11), SW9 (14.11)** **due 21.11. - 6pts**
  MCP_cooperating_FSMs.pdf
- **P10: Zephyr basics - SW10 (21.11)** **due 28.11. - 3pts**
  MCP_zephyr_basics.pdf
- **P11: Zephyr - Real-time operating system - SW11 (28.11)** **due 05.12. - 3pts**
  MCP_zephyr_RTOS.pdf
- **P12: Zephyr - Unit Testing - SW12 (05.12)** **due 12.12. - 3pts**
  MCP_zephyr_unit_testing.pdf
- **P13: Zephyr - Security - SW13 (12.12)** **due 19.12. - 3pts**
  MCP_zephyr_security.pdf
- **Wrap-up - SW14 (19.12)** Finalize labs
- **TODO:** Add remaining lab PDFs if available

Exercises

- **TODO:** Add exercises if available

Exam preparation

- **TODO:** Add old exams if available

Setup

- **TODO:** Add setup instructions if available

Extra Materials

- MC1 Box - Hardware kit to be returned at end of semester

Online Resources:

- **Course Website (Moodle):** https://moodle.zhaw.ch/course/view.php?id=25748
- **TODO:** Add any additional online resources if available

Recommended Reading:

- **TODO:** Check course materials for recommended textbooks

Timeline

Complete Course Overview and Timeline
Course Duration: 19.09.2025 (SW1) - 19.12.2025 (SW14)
Exam Date: January 2026

Preparation and Introduction

- L00: Course Introduction (19.09.)
- Setup development environment for labs

SW1 (19.09.)	L01: Understanding Memory and I/O	P01: Edge detection/debouncing **(03.10)**
SW2 (26.09.)	L01: Understanding Memory and I/O L02: Power Management and Low Power Applications	P01: Edge detection/debouncing **(03.10)** P02: Matrix keyboard **(10.10)**
SW3 (03.10.)	L02: Power Management and Low Power Applications	P02: Matrix keyboard **(10.10)**
SW4 (10.10.)	L03: Digital Sensors - Direct Memory Access (DMA)	P04: Power modes **(17.10)**
SW5 (17.10.)	L04: From Hardware Timer to Scheduler	P05: Motion Sensor I - Basic access **(24.10)**
SW6 (24.10.)	L05: Structuring Embedded Software I	P06: Motion Sensor II - DMA & Power **(31.10)**
SW7 (31.10.)	L06: Partitioning Reactive Systems - Concurrent Operating Finite State Machines	P07: Motion Sensor III - File system **(07.11)**
SW8 (07.11.)	L07: RTOS - Real-Time Operating Systems	P08_09: Cooperating FSMs **(21.11)**
SW9 (14.11.)	L08: RTOS - Real-Time Operating Systems	P08_09: Cooperating FSMs **(21.11)**
SW10 (21.11.)	L09: Structuring Embedded Software II	P10: Zephyr basics **(28.11)**
SW11 (28.11.)	L10: Verification	P11: Zephyr - Real-time operating system **(05.12)**
SW12 (05.12.)	L11: Verification / Watchdog	P12: Zephyr - Unit Testing **(12.12)**
SW13 (12.12.)	L12: Security for Embedded Systems	P13: Zephyr - Security **(19.12)**
SW14 (19.12.)	Wrap-up, Finalize labs	

Table of Contents

Course Structure Overview
This summary covers embedded systems programming across 12 main chapters:

Total KRs: [TBD - fill in after all chapters complete]
Total Examples: [TBD - fill in after all chapters complete]

1. Understanding Memory and I/O

Chapter 1: Understanding Memory and I/O - The Basics of Embedded Software
Topics Covered:
- Memory sections and allocation
- Scope and lifetime of variables
- I/O operations in embedded systems
- Hardware-software interaction

Key Concepts and KR
- ... TODO: Fill in key concepts and knowledge requirements after chapter completion

LateX Files
- `lecture01A_memory.tex` - Memory concepts
- `lecture01B_io.tex` - I/O concepts

Relevant Course Materials
Lecture Coverage: Lecture T01 (SW1, 19.09) **Related Labs:** P_01 (Edge detection/debouncing)

2. Power Management

Chapter 2: Power Management and Low Power Applications
File: `02_power_management.tex`
Topics Covered:
- Power consumption basics
- Low power modes (Sleep, Stop)
- Power optimization techniques
- Wake-up mechanisms (WFI, WFE)

Key Concepts: Energy efficiency, sleep modes, power management strategies
Lecture Coverage: Lecture T02 (Week 2, 26.09)
Related Labs: P_04 (Power modes)

3. Direct Memory Access (DMA)

Chapter 3: Digital Sensors - Direct Memory Access
File: `03_dma.tex`
Topics Covered:
- DMA basics and configuration
- DMA channels and priorities
- Peripheral-to-memory transfers
- DMA for sensor data acquisition

Key Concepts: Direct memory access, efficient data transfer, sensor interfacing
Lecture Coverage: Lecture T03 (Week 4, 10.10)
Related Labs: P_06 (Motion Sensor II - DMA & Power)

4. From Hardware Timer to Scheduler

Chapter 4: Scheduling with Hardware Timers

File: `04_timer_to_scheduler.tex`
Topics Covered:

- Hardware timer basics
- Interrupt-driven scheduling
- Task scheduling with timers
- Multi-task scheduling approaches

Key Concepts: Timer-based scheduling, interrupt handling, task management
Lecture Coverage: Lecture T04 (Week 5, 17.10)
Related Labs: P_05 (Motion Sensor I - Basic access)

5. Structuring Embedded Software - Part 1

Chapter 5: Software Architecture Fundamentals

File: `05_sw_structure_part1.tex`
Topics Covered:

- Module structure and interfaces
- Design patterns in C
- Header file organization
- Encapsulation and abstraction

Key Concepts: Modular design, clean architecture, software patterns
Lecture Coverage: Lecture T05 (Week 6, 24.10)
Related Labs: P_06 (Motion Sensor II), P_02 (Matrix keyboard)

6. Finite State Machines (FSMs)

Chapter 6: Partitioning Reactive Systems with FSMs

File: `06_fsm_basics.tex`
Topics Covered:

- FSM fundamentals
- Event-driven programming
- Cooperating FSMs
- FSM implementation in C
- Event queues

Key Concepts: State machines, event handling, reactive systems
Lecture Coverage: Lecture T06 (Week 7, 31.10)
Related Labs: P_08_09 (Cooperating FSMs)

7. Real-Time Operating Systems - Basics

Chapter 7: RTOS Fundamentals

File: `07_rtos_basics.tex`
Topics Covered:

- RTOS concepts
- Tasks and threads
- Scheduling algorithms
- Inter-task communication

Key Concepts: Real-time scheduling, multitasking, RTOS architecture
Lecture Coverage: Lecture T07 (Week 8, 07.11)
Related Labs: P_10, P_11 (Zephyr RTOS)

8. Real-Time Operating Systems - Advanced

Chapter 8: Advanced RTOS Topics

File: `08_rtos_advanced.tex`
Topics Covered:

- Synchronization primitives
- Priority inversion
- Resource management
- Real-time constraints

Key Concepts: Concurrency, synchronization, real-time guarantees
Lecture Coverage: Lecture T08 (Week 9, 14.11)
Related Labs: P_11 (Zephyr - Real-time operating system)

9. Structuring Embedded Software - Part 2

Chapter 9: Advanced Software Architecture

File: `09_sw_structure_part2.tex`
Topics Covered:

- Advanced design patterns
- Software layering
- Driver architecture
- System integration

Key Concepts: Layered architecture, driver development, system design
Lecture Coverage: Lecture T09 (Week 10, 21.11)
Related Labs: P_07 (Motion Sensor III - File system)

10. Verification

Chapter 10: Software Verification Techniques

File: `10_verification.tex`
Topics Covered:

- Testing strategies
- Debugging techniques
- Code verification
- Quality assurance

Key Concepts: Testing, debugging, verification methods
Lecture Coverage: Lecture T10 (Week 11, 28.11)
Related Labs: P_12 (Zephyr - Unit Testing)

11. Watchdog and System Monitoring

Chapter 11: Watchdog Timers and System Reliability

File: `11_watchdog.tex`
Topics Covered:

- Watchdog timer basics
- System monitoring
- Fault detection
- Recovery mechanisms

Key Concepts: System reliability, watchdog implementation, fault tolerance
Lecture Coverage: Lecture T11 (Week 12, 05.12)
Related Labs: P_12 (Unit Testing)

Chapter 12: Embedded Systems Security

File: `12_security.tex`
Topics Covered:

- Security fundamentals
- Cryptography basics
- Secure boot
- Threat mitigation

Key Concepts: Security principles, cryptography, secure design
Lecture Coverage: Lecture T12 (Week 13, 12.12)
Related Labs: P_13 (Zephyr - Security)

Understanding Memory and I/O - Part A: Memory Objects

Learning Objectives

At the end of this lesson, you will be able to:
- Enumerate the attributes of a variable in C
- Explain the effect a chosen data type has on memory usage
- Differentiate between a variable declaration and a variable definition
- Write C-modules that distinguish between interface and implementation
- Explain how memory contents are interpreted based on the attributes
- Analyze a piece of C-code and identify where (in which section) memory will be allocated
- Indicate the scope and the lifetime of a variable in C:
  - Where in code is a variable visible
  - When will memory be allocated, and when will memory be deallocated
  - How and when will a variable be initialized
- Apply the qualifier `static` to provide encapsulation for private module data
- Apply the qualifier `const` to pass read-only data structures
- Apply the critical take-aways in your own C-code
- Allow the reuse of functions by extracting the state from a function

Motivation

Understanding Code-Hardware Interactions

This lecture summarizes known aspects from previous courses and adds microcontroller-specific details. The key motivation is to understand interactions between source code, compilation, and hardware memory layout in embedded systems.

Memory Objects in C

Six Attributes of Memory Objects

Every memory object in C has six fundamental attributes:
1. **Type** - Data type of the object
2. **Name** - Identifier used to reference the object
3. **Value** - The data stored in the object
4. **Location** - Which memory section the object resides in
5. **Scope** - Where in the code the object is visible
6. **Lifetime** - When the object exists in memory

Type

Data Type

The type attribute specifies:
- How much memory space is allocated
- How the bit pattern in memory is interpreted
- Which operations are valid on the object

Basic Data Types in Embedded Systems

Basic data types and their typical sizes:

```c
uint8_t  var1;  // 1 byte, unsigned
int16_t  var2;  // 2 bytes, signed
uint32_t var3;  // 4 bytes, unsigned
float    var4;  // 4 bytes, floating point
```

Platform-dependent pointer sizes

- **8051/Keil:** `char *` has variable size
- **Cortex-M/Keil:** `void *` is 4 bytes (32-bit)
- **x86-64 (i7)/gcc:** `void *` is 8 bytes (64-bit)

typedef

The `typedef` facility creates new data type names for better code readability and maintainability.

```c
typedef int length_t;      // length_t is now an alias for int
typedef char *string_t;    // string_t is now an alias for char*

int i;
length_t length;  // Same as: int length;
string_t name;    // Same as: char *name;
```

Name

Definition vs Declaration

Definition:
- Introduces name/type **and** allocates storage space
- Includes function bodies for functions

Declaration:
- Introduces name/type only
- Does **not** allocate storage space
- Uses `extern` to specify object is defined elsewhere

Definition vs Declaration

```c
// DEFINITION - allocates memory
uint8_t var1;

// DEFINITION - provides function body
uint16_t foo(void) {
    // what the function does
}
```

```c
// DECLARATION - no memory allocation
extern uint8_t var1;

// DECLARATION - no function body
extern uint16_t foo(void);
```

The `extern` keyword specifies that the declared object is defined elsewhere (i.e., in another module). This is optional for functions but required for variables.

Module Structure: Interface and Implementation

Header Files (Interface)

Header files contain:
- Only declarations → **NO** definitions
- Function prototypes (declarations)
- Type definitions (`typedef`, `struct`, `enum`)
- No other `#include` statements (except for required typedefs)

Key principle: Interface → header file

Implementation Files (.c files)

Implementation files provide:
- Definitions of items declared in the header file
- Include their own header file
- Private (static) helper functions
- Private (static) module variables

Key principle: Implementation → .c file

Module Interface Example

Example of a properly structured module interface (bar.h):

```c
// bar.h - Header file contains only declarations
typedef enum {
    RED   = 0x01,
    BLUE  = 0x02,
    GREEN = 0x03
} colors_t;

/* Set specified color */
void bar_set(colors_t color);

/* global --> very bad style */
extern uint32_t bar_x;
```

Implementation file (bar.c):

```c
// bar.c - Implementation file
#include "bar.h"

void bar_set(colors_t color) {
    // what the function does
}

uint32_t bar_x;  // Definition of the extern variable
```

Usage in another file (foo.c):

```c
#include "bar.h"

void foo_ex(void) {
    bar_set(BLUE);
    bar_x = 0x27;
    ...
}
```

Location - Memory Sections

Memory Sections in Embedded Systems

Memory for variables is allocated in different sections depending on their characteristics:

- **CODE RO / .text** - Program code (read-only)
- **DATA RO / .rodata** - Read-only data (constants)
- **DATA ZI / .bss** - Zero-initialized data (static allocation without initial value)
- **DATA RW / .data** - Read-write data (initialized static variables)
- **HEAP** - Dynamic allocation (malloc/free)
- **STACK** - Automatic variables and function call data

These sections are managed by the linker/loader and have different memory characteristics.

Linker and Loader Process

Compilation process:

1. Each .c source file produces a combination of CODE RO and DATA (RO, ZI, RW)
2. Linker/Loader combines object files and allocates actual memory addresses
3. Final memory layout has distinct sections in Flash and RAM

Memory Allocation by Section

Source: Lecture 01, Slide 23

The following code demonstrates where different variables are allocated:

```c
// Global variables
uint32_t bss_a;                      // -> DATA ZI (.bss)
uint32_t data_a = 0x12;              // -> DATA RW (.data)
static uint32_t bss_b;               // -> DATA ZI (.bss)
static uint32_t data_b = 0x34;     // -> DATA RW (.data)
const uint32_t read_only_a = 0x35; // -> DATA RO (.rodata)

void foo(void) {
    uint32_t *p;                     // -> STACK (or register)
    uint32_t stack_a;                // -> STACK (or register)
    static uint32_t bss_c;           // -> DATA ZI (.bss)
    static uint32_t data_d = 0x56; // -> DATA RW (.data)
    const uint32_t read_only_b = 0x78; // -> STACK or DATA RO

    // Dynamic allocation
    p = (uint32_t*) malloc(sizeof(uint32_t)); // -> HEAP
    if (p == NULL) { /* handle error */ }
    ...
    free(p);
}
```

Memory Allocation by Section Solution

- Produced object code → CODE RO / .text
- Global uninitialized variables → DATA ZI / .bss
- Global initialized variables → DATA RW / .data
- `const` global variables → DATA RO / .rodata
- Local variables → STACK (or registers if optimized)
- `static` local variables → DATA ZI or DATA RW
- `malloc()` allocated memory → HEAP

Note: Requires `#include <stdlib.h>` for `malloc()` and `free()`. Compiler optimizations may place stack objects in registers. Depending on the compiler, `const` local variables might be placed on the stack instead of DATA RO.

Scope

Scope

Scope determines the visibility of names in source code. It defines where in the code a variable or function can be accessed. Scope is an arrangement between the compiler/assembler and programmer, and is not relevant for object code.

Scope Levels in C

- **Function local** - Visible only within the function
- **Module local** - Visible throughout the module (.c file), uses `static`
- **Global** - Visible across all modules that include the declaration

Terms "local variablesänd äutomatic variablesäre used interchangeably.

Automatic Variable Scope

Automatic variables have local scope:

- Scope begins at point of definition
- Scope ends at end of containing block (closing brace)

```c
void foo(void) {
    int a;  // Scope begins here
    {
        int b;  // Scope of b begins here
        a = 5;  // a is visible here
        b = 10; // b is visible here
    }  // Scope of b ends here
    a = 15;  // a still visible
    // b is NOT visible here
}  // Scope of a ends here
```

Lifetime

Lifetime

Lifetime is the duration of a memory object's existence. It defines:

- When memory is allocated (Creation)
- When initial values are assigned (Initialization)
- When memory is deallocated (Destruction)

Three Types of Lifetime

C has three distinct types of memory objects based on lifetime:

1. **Automatic** - Local variables (stack/registers)
2. **Static** - Global and static variables
3. **Dynamic** - Heap-allocated memory

Automatic Variables

Automatic Lifetime

Storage: Local variables in registers or on the stack

Creation: Each time the program enters the function in which the variable is defined. If the function is called recursively, several instances may exist simultaneously.

Initialization:

- Default: No automatic initialization (contains garbage)
- If defined with assignment: Initialized each time function is entered

Destruction: On each return of the function, the memory on the stack will be released (not deleted/zeroed)

Automatic Variables Example

```c
void foo(void) {
    int a;        // No initialization - contains garbage!
    int b = 5;    // Initialized to 5 each time foo() is called

    for (int i = 0; i < 10; i++) {
        int c = i;  // c is created and initialized each iteration
        // Use c
    }  // c is destroyed here

}  // a and b are destroyed here
```

Automatic Variables Example Key Points:

- Memory for a, b, and c allocated on stack (or registers) each time `foo()` is entered
- If `foo()` is called recursively, several instances of a, b, and c may exist
- Variable `a` is NOT automatically initialized (undefined behavior)
- Variable `b` is initialized each time `foo()` is entered
- Variable `c` is initialized each time the for loop block is entered
- Memory is released on return, but not actively cleared

Static Variables

Static Lifetime

Storage: Memory objects in DATA sections (DATA ZI or DATA RW)

Types of static variables:

1. Global variables
2. Module-wide variables with qualifier `static`
3. `static` variables within functions

Creation: Once, at start of program (when program is loaded into memory)

Initialization: Once, just before program starts:

- Variables in DATA section (.data) initialized to specified values
- Variables in BSS section (.bss) initialized to zero

Destruction: When the program terminates

Static Variables Example

```c
// Global static variables
static int s;           // Module-wide, initialized to 0 (.bss)
static int t = 10;      // Module-wide, initialized to 10 (.data)

int u;                  // Global, initialized to 0 (.bss)
int v = 20;             // Global, initialized to 20 (.data)

void foo(void) {
    static int w;      // Function static, initialized to 0 (.bss)
    static int x = 30; // Function static, initialized to 30 (.data)

    w++;  // Persists across function calls
    x++;  // Persists across function calls
}
```

Static Variables Example Key points:

- Memory for all variables (s, t, u, v, w, x) allocated when program loads
- Initialization happens just before program starts:
  - t, v, x (data section) initialized to specified values
  - s, u, w (bss section) initialized to zero automatically
- Although s, u, w have no initial value in source code, they are initialized to 0
- Memory persists for entire program execution
- Function-local `static` variables retain their value between function calls
- Memory released only when program terminates

Dynamic Variables

Dynamic Lifetime

Storage: Memory objects on the heap
Creation: By calling `malloc()` or similar allocation functions

- Use `sizeof()` operator to specify size
- ALWAYS check return value (NULL indicates failure)
- Failure can occur due to insufficient memory or heap fragmentation

Initialization:

- `malloc()` does NOT initialize memory (contains garbage)
- `calloc()` provides zero-initialized memory
- Programmer responsible for initialization

Destruction: By calling `free()`

- Responsibility of programmer
- Failure to free causes memory leaks

Dynamic Memory Example

```c
#include <stdlib.h>

void bar(void) {
    uint32_t *p;

    // Allocate memory on heap
    p = (uint32_t*) malloc(sizeof(uint32_t));

    // CRITICAL: Always check for allocation failure
    if (p == NULL) {
        // Handle error: out of memory or fragmentation
        return;
    }

    // Use the allocated memory
    *p = 42;  // Assign value

    // CRITICAL: Free the memory when done
    free(p);
    p = NULL;  // Good practice: avoid dangling pointer
}
```

Dynamic Memory Example

Key points:

- Memory allocated on heap persists until explicitly freed
- ALWAYS check return value of `malloc()` for NULL
- Memory is NOT automatically initialized (use `calloc()` for zero-init)
- Programmer has full responsibility for allocation and deallocation
- Forgetting to `free()` causes memory leaks
- Setting pointer to NULL after `free()` prevents dangling pointers
- Requires `#include <stdlib.h>`

Function Reusability

Extracting State from Functions

Functions with `static` variables are not reusable for multiple instances. To make functions reusable:
Problem: Function with internal state can only handle one instance:

```c
uint8_t debounce_switches(void) {
    static uint8_t samples[NR_SAMPLES];  // Only one instance!
    // ... processing
}
```

If you need to debounce switches on two different ports, you cannot reuse this function.
Solution: Move state out of function, pass pointer to state:

```c
// Caller allocates state
uint8_t port_A_samples[NR_SAMPLES];
uint8_t port_B_samples[NR_SAMPLES];

// Function accepts state as parameter
uint8_t debounce_switches(uint8_t *samples) {
    // Use samples array for processing
    // ... processing
}

// Now reusable for multiple ports
debounce_switches(port_A_samples);
debounce_switches(port_B_samples);
```

Design principle

Functions should be stateless when possible. If state is needed, pass it as a parameter to enable reuse. The caller manages the state, and the function operates on it. This is a fundamental principle of reusable software design.

Understanding Memory and I/O - Part B: Digital I/O Processing

Hardware Aspects

GPIO Architecture on STM32F4xx

General Purpose Input/Output (GPIO) pins on the STM32F4xx microcontroller have configurable hardware features including Schmitt triggers, pull-up/pull-down resistors, and edge detection capabilities.

GPIO Voltage Levels

VDD Configuration:

- User-defined voltage level through VDD pin
- Range: 1.2V to 3.6V
- CMOS voltage levels
- Most GPIOs are 5V tolerant (check datasheet for individual pins)

Voltage compatibility:

- Output stage VOL/VOH must match input stage VIL/VIH of external device
- STM32 and external device must have compatible logic levels
- Some interfaces require external level shifters or drivers (e.g., RS-232)

Schmitt Trigger

A Schmitt trigger is a comparator circuit with hysteresis that provides noise immunity on digital inputs. It has two threshold voltages:

- Upper threshold for rising edge detection
- Lower threshold for falling edge detection
- Typical hysteresis: 200mV on STM32F4xx

This prevents multiple transitions when the input signal slowly crosses the threshold or has noise.

Hardware Best Practices:

1. **Avoid floating inputs** - Noise causes floating inputs to change randomly. Connect unused inputs to VDD or VSS through pull-up/pull-down resistors.
2. **Avoid oscilloscope probes on high-impedance nodes** - Probe capacitance can cause unwanted behavior. Use buffered test points for measurement.

Software Techniques for I/O

Edge Detection Challenge

Detecting signal transitions (edges) on GPIO inputs is a fundamental task in embedded systems. However, mechanical switches introduce bounce, which creates multiple false edges during a single button press. Software must handle both edge detection and debouncing.

Edge Detection by Polling

Edge Detection by Polling

Edge detection can be implemented by comparing current and previous button states in a polling loop.

Algorithm:

1. Read current GPIO state
2. Compare with previous state
3. Detect transitions:
   - Rising edge: was 0, now 1 → `(~last) & current`
   - Falling edge: was 1, now 0 → `last & (~current)`
4. Store current state for next iteration

Limitation: This basic approach assumes no bouncing. Real mechanical switches bounce!

Basic Edge Detection Implementation

Source: Lecture 01, Slide 47

```c
#include <reg_ctboard.h>

#define CT_BUTTON (*(uint8_t *)0x60000210)

int main(void) {
    uint8_t levels_of_buttons;
    uint8_t last_levels_of_buttons = 0;
    uint8_t buttons_rising_edge;

    while (1) {
        // Read current button state
        levels_of_buttons = CT_BUTTON;

        // Detect rising edges
        // Rising edge = was 0, now 1
        buttons_rising_edge = (~last_levels_of_buttons) &
            levels_of_buttons;

        if (buttons_rising_edge) {
            // ... do something on button press
        }

        // Store current state for next iteration
        last_levels_of_buttons = levels_of_buttons;

        // ... other code
    }
}
```

Key concepts:

- Polling: Repeatedly read button state in main loop
- Edge detection: Compare current reading to previous reading
- Rising edge formula: `(~last) & current`
- Falling edge formula: `last & (~current)`
- Bitwise operations enable parallel processing of multiple GPIO pins

Hardware Edge Detection

Interrupt-Based Edge Detection

STM32 microcontrollers have built-in hardware edge detection in GPIO peripherals. Each GPIO can generate an interrupt (EXTI) on rising edge, falling edge, or both.

Hardware Edge Detection Process

1. GPIO pin has built-in edge detection circuit
2. Edge detected → Interrupt request sent to NVIC (Nested Vectored Interrupt Controller)
3. NVIC prioritizes interrupt and triggers corresponding ISR (Interrupt Service Routine)
4. CPU executes ISR to handle the event

Advantages:

- No polling required - CPU can sleep
- Immediate response to events
- Multiple GPIO lines can generate interrupts

Challenge: Hardware still detects every bounce!

Software Debouncing

Button Bounce

Mechanical switches do not transition cleanly between states. When pressed or released, the contacts bounce, creating multiple rapid transitions before settling. Bounce duration is typically 5-20ms but can be longer for poor-quality switches.

Low Sampling Rate Debouncing

Sample the input at a rate slower than the bounce frequency. The low sample rate acts as a filter.
Problem: Timing is critical. Sample too early during bounce and detect wrong state. Sample rate must be carefully chosen relative to bounce duration.

High Sampling Rate Debouncing

Sample input at high frequency and detect every transition, including bounces.
Problem: Every bounce is detected as a separate edge, causing multiple false triggers per button press.

Sliding Window Filter for Debouncing

The sliding window (moving average) filter is an effective debouncing technique:
Algorithm:
1. Maintain a buffer of recent input samples (e.g., 8-20 samples)
2. At each sampling time:
   - Shift buffer contents (discard oldest, add newest)
   - Read new input value and store in buffer
3. Detect edges only when buffer shows stable transition:
   - Rising edge: All zeros followed by one (newest bit = 1)
   - Falling edge: All ones followed by zero (newest bit = 0)

Implementation approach:
- Use `static` array to store samples across function calls
- Use bitwise operations (&, ~) for parallel edge detection on multiple pins
- Choose buffer size based on bounce duration and sample rate

Buffer size selection:
- Bounce duration: typically 5-20ms
- Sample rate: e.g., 1ms (1kHz)
- Buffer size: 10-20 samples usually sufficient
- Test with actual hardware to verify

Sliding Window Debouncing Implementation Concept

Source: Lab P_01, MCP_Debouncing.pdf
Conceptual implementation of sliding window debouncing:

```c
#define NR_SAMPLES 10  // Adjust based on bounce characteristics

uint8_t detect_switch_change_debounce(void) {
    // Static array persists between function calls
    static uint8_t switch_samples[NR_SAMPLES];

    // Shift all samples one position (discard oldest)
    for (int i = NR_SAMPLES - 1; i > 0; i--) {
        switch_samples[i] = switch_samples[i-1];
    }

    // Read new sample from GPIO
    switch_samples[0] = read_gpio_inputs();

    // Detect rising edges: all samples were 0, now newest is 1
    uint8_t all_zero = 0xFF;  // Start assuming all pins have all zeros
    for (int i = 1; i < NR_SAMPLES; i++) {
        all_zero &= ~switch_samples[i];  // AND with inverted samples
    }
    uint8_t rising_edges = all_zero & switch_samples[0];

    // Detect falling edges: all samples were 1, now newest is 0
    uint8_t all_ones = 0xFF;  // Start assuming all pins have all ones
    for (int i = 1; i < NR_SAMPLES; i++) {
        all_ones &= switch_samples[i];  // AND with samples
    }
    uint8_t falling_edges = all_ones & ~switch_samples[0];

    return rising_edges;  // Or return falling_edges, or both
}
```

Key implementation details:
- `static` array maintains state between function calls
- Parallel processing using bitwise operations handles multiple pins simultaneously
- Buffer size `NR_SAMPLES` determines debouncing strength
- Function must be called at regular intervals (e.g., every 1ms from timer interrupt)

Optimization: Use circular buffer with index pointer instead of shifting all elements (more efficient, covered in lab).

Key Takeaways

Critical Takeaways from This Lecture

Memory Management:

- Understand the six attributes: Type, Name, Value, Location, Scope, Lifetime
- Know where variables are allocated: Stack, Heap, BSS, Data, RO Data, Code
- Use `static` for module encapsulation
- Use `const` for read-only data and function parameters
- Always initialize automatic variables
- Always check `malloc()` return values

Module Design:

- Header files: declarations only (interface)
- Implementation files: definitions (implementation)
- Use `static` for private module functions and variables
- Use `extern` for cross-module access

Digital I/O:

- Schmitt triggers provide noise immunity
- Mechanical switches bounce - must debounce in software
- Sliding window filter is effective debouncing technique
- Extract state from functions for reusability
- Sample rate and buffer size affect debouncing performance

See also: Labs P_01 (Edge detection and debouncing) and P_02 (Matrix keyboard) for hands-on practice.