

Embedded Systems and Advanced Computing

ENCE464

Andy Ming / Quelldateien

Table of contents

How to work code

Feature Branches	1
Clean Code	2
Reveal Intent	2
Don't Repeat Yourself (DRY)	2
Consistent Abstraction	2
Encapsulation	2
Comments	2
Code Reviews	2
SOLID	2
Single Responsibility Principle (SRP)	2
Open-Closed Principle (OCP)	2
Liskov Substitution Principle (LSP)	3
Interface Segregation Principle (ISP)	3
Dependency Inversion Principle (DIP)	3
SOLID Models for C	3
Single Instance Model	3
Multiple Instance Model	3
Dynamic Interface	3
Pre-Type Dynamic Interface	3
Legacy / Vererb / Veralteter Code	3

Embedded Software Design

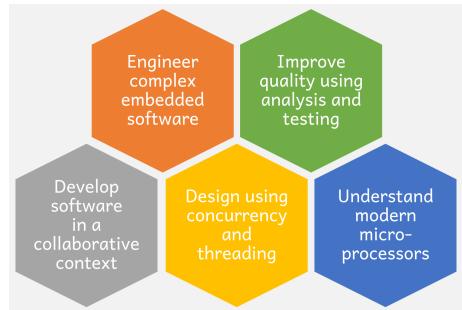
Architecture	4
Layered	4
Ports-and-Adapters (or <i>Hexagonal</i>)	4
Pipes-and-Filters	5
Microkernel	5
RTOS	5
Tasks	6
Concurrency / Gleichzeitigkeit	7
Resources	9
Performance	9

Testing

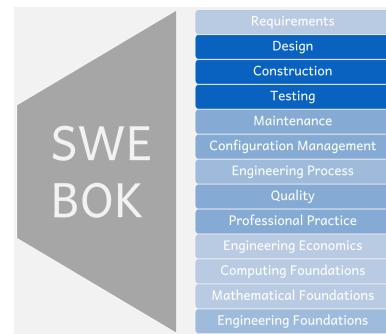
Unit Test	9
Unit Test with Collaborators	10
Test Doubles	10
Continuous Integration	10
Higher Level Testing	10
Automated Acceptance Testing	10
Automated System Tests	10
Manual Testing	10
Test Driven Design TDD	10

How to work code

Remember that software engineering is 50-70% maintenance. Because modern machines heavily rely on microcontrollers there is great demand.

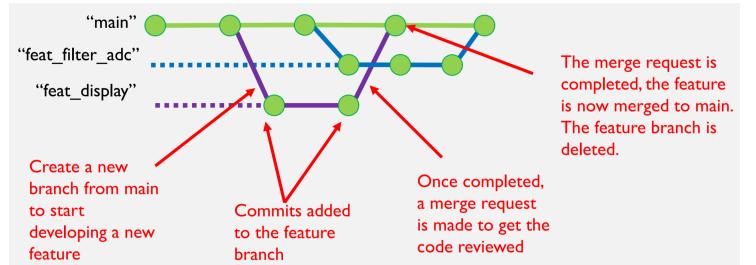


Software engineering has many different aspects (the dark blue ones are focused on here), find out more [here](#).



Feature Branches

To implement different features, use a branch per feature, this guarantees that the main is always in working condition.



! Branching Rules

- Feature branches are **temporary** branches for new features, improvements, bug fixes or refactorings.
- Don't push directly to **master/main**.
- Each feature branch is owned by **one** developer.
- Only do merge requests on **complete** changes i.e. don't break main.
- Thoroughly test your change prior to **starting** AND prior to **completing** a merge request.
- Use your commit messages to tell the **story** of your development process.

To minimise integration issues:

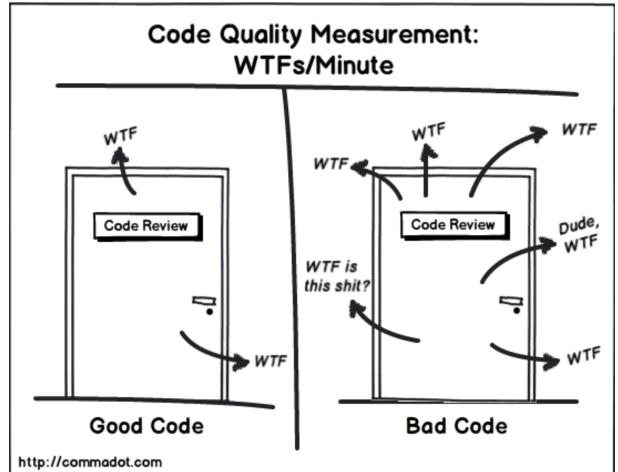
- A feature branch should only hold a small increment of change
- If main is updated during feature development, merge the new main into your feature branch **locally, before** making a

merge request

Clean Code

Smells of Bad Code

- **Rigidity:** Changing a single behaviour requires changes in many places
- **Fragility:** Changing a single behaviour causes malfunctions in unconnected parts
- **Inseparability:** Code can't be reused elsewhere
- **Unreadability:** Original intent can't be derived from code



Reveal Intent

```
// BAD
uint16_t adcAv; // Average Altitude ADC counts
// GOOD
uint16_t averageAltitudeAdc;
```

Don't Repeat Yourself (DRY)

Avoid duplicate code → Put it into a function. Can you put it in a function? Then you should!

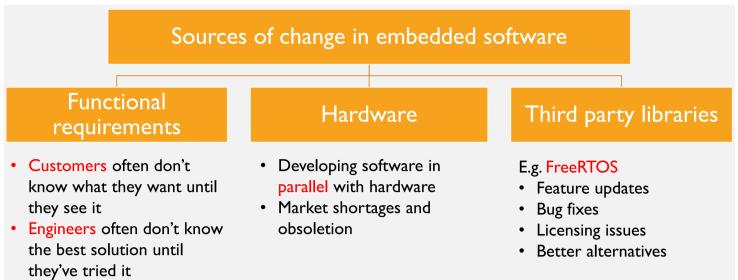
Consistent Abstraction

High-Level ideas shouldn't get lost in **Low-Level** operations.

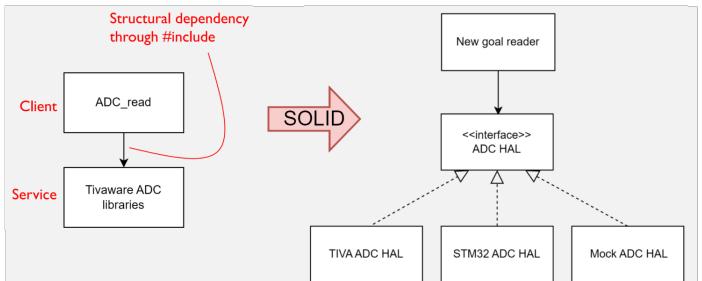
```
// Bad Example
deviceState.newGoal = readADC() * POT_SCALE_COEFF;
↪ // low-level
deviceState.newGoal = (deviceState.newGoal /
↪ STEP_GOAL_ROUNDING)*STEP_GOAL_ROUNDING; // 
↪ high-level
```

SOLID

How to make designs **flexible**, as requirements change all the time (Agile).



SOLID is all about **managing dependencies**.



Encapsulation

- **Hide** as much as possible
- **Public Interface:** Header File, only declare what other modules need to know
- **Private / Inner Workings:** Source File
- **Avoid** global variables → Use *getter & setter*

Comments

More comments ≠ better quality. Use comments only to:

1. Reveal intent after you tried everything else
2. Document public APIs - sometimes

Code Reviews

Use merge requests, label feedback as *bug*, *code*, *quality*, *preference*.

Single Responsibility Principle (SRP)

"A module should only have a single responsibility. It should only have one reason to change"

Instead of `ADC_read` which handles the ADC reading, averaging and the setting of the new goal, we break it up into a module `ADC_HAL` which handles the ADC reading and averaging and the module `new_goal_reader` which just handles the setting of the new goal.

- A module does **one thing** and does it **well**
- Use good names, reveal intent
- Don't access numerous data structures and globals

Open-Closed Principle (OCP)

"Software entities (modules, functions) should be **open to extension**, but **closed for modification**"

The new_goal_reader doesn't have to be modified in case of a hardware change. But its functionality can be extended through swapping out the HAL implementation.

- Changes through adding code instead of modifying
- aka USB standard: Plug-n-Play, no hardware change needed
- OOP use "interface" or "abstract class"
- C use "header files" or "function pointers"

Liskov Substitution Principle (LSP)

"Subtypes should be substitutable with their base types"

TIVA ADC HAL is perfectly substitutable with its base type of the ADC HAL interface.

Interface Segregation Principle (ISP)

"Clients should not be forced to depend on functions that they do not use"

The interface ADC HAL is defined on the needs of new_goal_reader (the client). Don't show unneeded functions.

- Write "small" interfaces
- Allows to limit dependencies

Dependency inversion Principle (DIP)

"High-level modules should not depend upon low-level modules. Both should depend on abstraction"

The interface ADC HAL is the abstraction and both, new_goal_reader and TIVA ADC HAL, are implementing this. new_goal_reader doesn't know (and care) which implementation is called.

In OOP this is called **Polymorphism**.

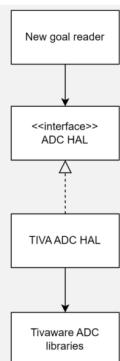
SOLID Models for C

💡 How much design is enough?

- Use simplest flexible design for today's requirements
- Changing requirements → Change **Design**
- Tests ensure functionality is retained

Single Instance Model

- HAL function names same as Interface
- Linker binds interface & implementation
- Choice defined in **CMake**
- **Single** instance, **Static** binding



Multiple Instance Model

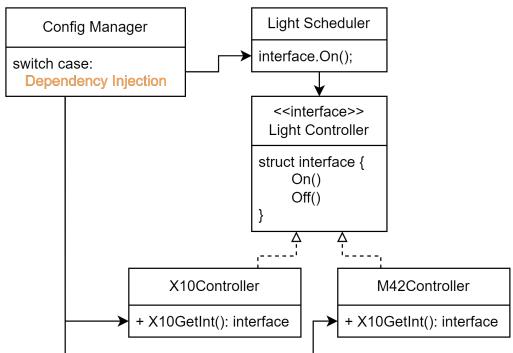
- Create multiple instances
- Use abstract data type (ADT) → object definitions and details are **encapsulated**
- Public functions to operate on the abstract data type f(0, x)
- **Multiple** instance, **Static** binding

```

// In header file
typedef struct circBuf circBuf_t;

// In source file
struct circBuf {
    uint32_t size;
    uint32_t windex;
    uint32_t rindex;
    int32_t *data;
}
  
```

Dynamic Interface



- Configuration determined in runtime
- Interface is a *public struct of function pointers*
- **Single** instance, **Dynamic** binding

Pre-Type Dynamic Interface

- Support any number & combination of drivers
- Each type has its own constructor
- Objects cast to abstract interface
- Have a array of abstract instances
- **Multiple** instance, **Dynamic** binding

Legacy / Vererb / Veralteter Code

If handed bad quality (no tests, no encapsulation, no good practices, ...) and you have to add features you can either (1) *add to the mess by hacking in new features* or (2) *rewrite code from scratch*.

The issues are (1) will *reduce productivity* and it's *easy to introduce bugs* but (2) is *very time consuming*, it's *difficult to maintain two versions* (there might still be old versions in the field which have to be maintained) and there will be a *new set of bugs* to deal with.

So we try to refactor until it's easy to make changes. To preserve functionality we iterate *in small increments* with **Trageted Tests** (*allows changes and new features to happen at the same time*):

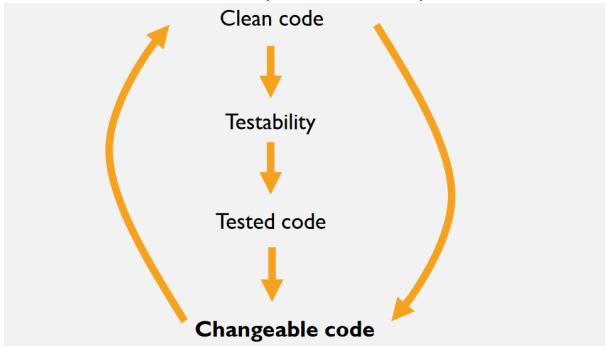
1. Write tests of the code you need to change
2. Test drive changes to legacy code
3. Test drive new code
4. Refactor tested area

Also add **Characterisation Tests** where understanding is required.
Tests are a living documentation.

To make sure key functionality isn't altered, add **Strategic Tests**.
(e.g. control algorithm, safety-critical error detection, ...)

i Putting Code Under Test

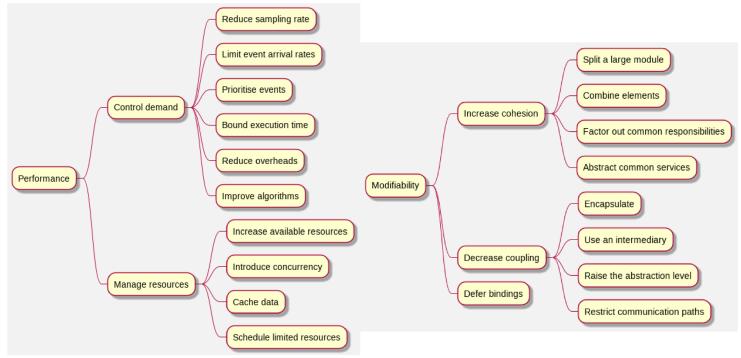
1. Identify areas of code to test (targeted, characterisation, strategic)
2. Find test points (function calls, global variables → encapsulate ASAP, serial)
3. Break dependencies (Solitary tests)



Allocation Structures: Assignment of software elements to external things

Structure	Elements	Relationships
Memory Map	Data, addresses	Allocated to
Implementation	Modules, files	Allocated to
Deployment	Software, hardware	Allocated to

Patterns are always a combination of tactics, depending on what you're trying to achieve.



! Trade-Offs

Quality attributes can conflict with each other. For example:

Quality	Often trades with	Notes
Modifiability	Time performance	Modifiability often requires indirection, which introduces overheads. Removing indirection makes modifications more difficult.
Testability	Time performance	Testability, like modifiability, often requires indirection, which introduces overheads.
Execution speed (performance)	Memory (resource use)	The classic trade-off in software design. Reduce computation time by using more memory, or vice versa.
Responsiveness (performance)	Power (resource use)	Responding quickly may involve avoiding the latency involved in waking from sleep.
Determinism	Responsiveness (performance)	Ensuring deterministic timing of actions may involve polling instead of responding to events as they happen.

i Keep Record of Decisions

To keep record of decisions and to not lose the overview use tools like:

- **Architecture Haikus:** A onepager overview of your document [see here or in the appendix folder](#).
- **Architecture Decision Records:** A incremental document to record decisions on the go [either in a tool or a markdown file](#).

Layered

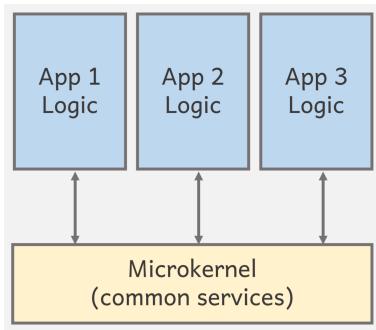
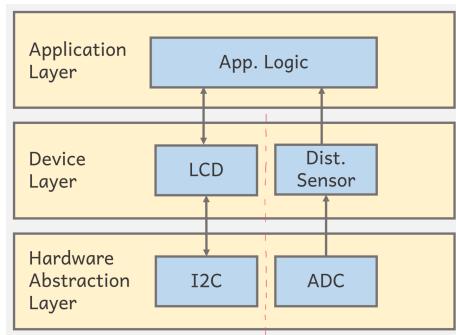
Each layer is providing services to the above layer through well-defined interfaces. Each layer can only interact with the layer directly above or below.

Supports portability and modifiability by allowing internal changes to be made inside a layer without impacting other layers, and isolating changes in layer-to-layer interfaces from more distant layers.

Structure	Elements	Relationships
Decomposition	Modules, functions	Submodule of
Dependency	Modules	Depends on
Class	Classes	Inheritance, association

Static Structures: Conceptual abstraction a developer works with

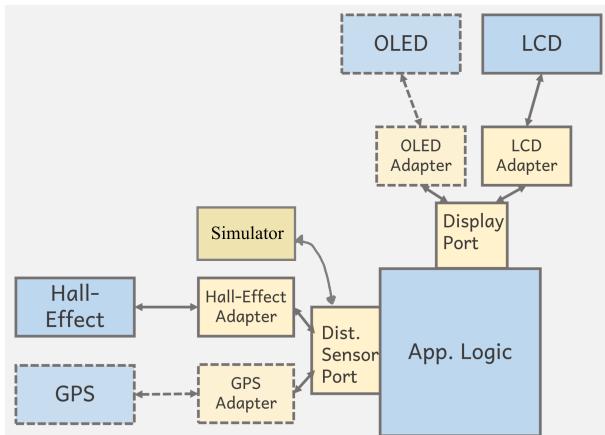
Structure	Elements	Relationships
Collaboration	Components	Connections
Data-flow	Processes, stores	Flows of data
Task	Tasks, objects	Interactions



Ports-and-Adapters (or Hexagonal)

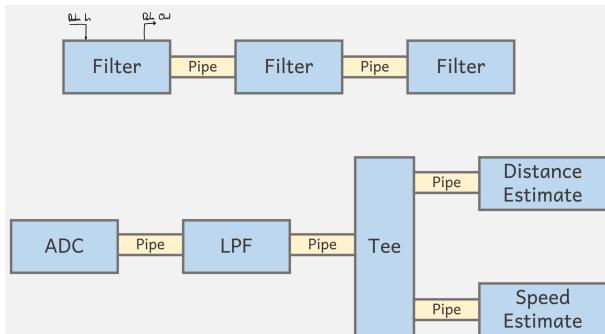
Introduces a single core logic which communicates through abstraction interfaces (**Ports**) to different modules. The **Adapters** map the external interactions to the standard interface of the port.

Supports portability and testability by making the inputs to the ports independent of any specific source, and supports modifiability by creating a loose coupling between components.



Pipes-and-Filters

Supports modifiability through loose coupling between components, and performance by introducing opportunities for parallel execution.



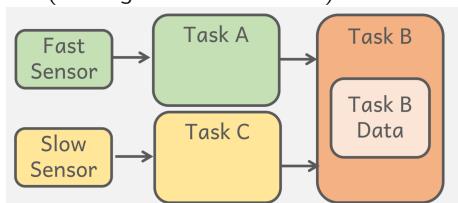
Microkernel

RTOS is a implementation of a Microkernel Architecture. The **Microkernel** includes a set of common core services. Specific services (**Tasks**) can be plugged into the kernel.

Supports modifiability and portability.

Tasks for Priority and Modularity

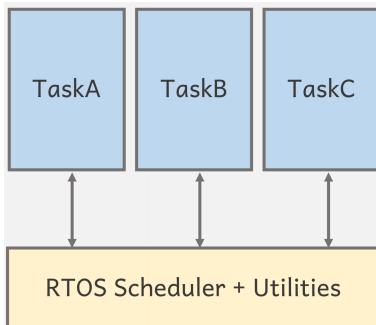
- + Control Priority
- + Control Response Time
- + Modularity
- Concurrency Issues
- Overhead (Scheduler)
- Starvation (Task gets no CPU time)



Use **just enough** tasks.

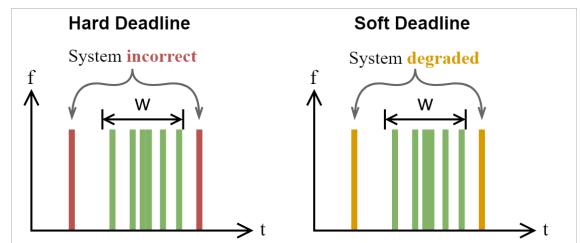
RTOS

To improve **Performance** we introduce **Concurrency** (Run tasks in parallel).



Preemptive approach: *Separation of concerns, Scalability, State is Managed.*

Do the **right thing** at the **right time W**



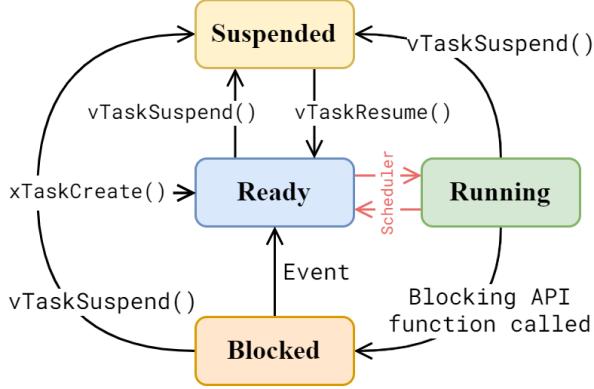
RTOS vs. Desktop OS

- Desktop OSs don't try to achieve *hard* real-time performance
- In a Desktop OS, programs can be loaded in runtime
- RTOS is compiled as part of the application, to add a new "program" the application has to be recompiled

Tasks

```
xTaskCreate(
    BlinkTask,      // Function that defines task
    "Blink Task",  // Task name (used in debugging)
    STACK_SIZE,    // No. of 4-byte words for stack
    NULL,          // Optional pointer to task argument
    PRIORITY,      // Higher number = higher priority
    NULL);         // Optional pointer to task handle
```

Taskswitches happen at *scheduling points* which occur when a **task is blocked**, **interrupt causes a task**, **priority change**, **higher priority task gets ready** or **system tick interrupt**.



Stack Size

! Stack Size Value

The stack size value passed in `xTaskCreate` is measured in **4-Byte words**.

Set high margins, something like **300%**

- Minimal: `configMINIMAL_STACK_SIZE`
- Maximal: Device **RAM**
- Actually: Analyse
 - Dynamic: Set something and see if it works / use `uxTaskGetStackHighWaterMark` to measure
 - Static: Use tools (e.g. GCC `-fstack-usage`) to attempt reading on how much stack is needed per function

Priority Task priority has a strong influence on when a task is run and thus on the overall behaviours of the application.

Assign priority based on importance

1. Separate tasks into "critical" (hard deadline) and "non-critical" (soft deadline)
2. Assign low priority to non-critical tasks
3. To be sure about critical tasks meeting their deadlines, apply scheduling theory

Assign non-critical tasks to low priorities

1. Either apply the same priority for all non-critical tasks
2. Or prioritise by *importance*, which depends on
 - a. Shortness of Deadline
 - b. Frequency of Execution
 - c. Need for Precessor time

Assign critical tasks deadline monotonic priorities

Apply priority based on the size of its deadline.

1. Highest \leftarrow shortest deadline
2. Lowest \leftarrow longest deadline

💡 Deadline / Rate Monotonic Priorities

Deadlines D_i and Period T_i for each task i

Deadline Monotonic: $D_i \leq T_i$

Rate Monotonic: $D_i = T_i$

Furthermore, following assumptions are made:

- Fixed-priority preemptive scheduling
- Hard-Deadline tasks are either:
 - *Periodic* (fixed interval)
 - *Sporadic* (known minimum time between triggering events)

Check Schedulability of Critical Tasks To check if deadlines can be met (schedulable) we calculate the **response time upper bound** R_i^{ub} for each task i . This has to be less than the task deadline D_i :

$$R_i^{ub} \leq D_i$$

The tasks are ordered after priority from $i = 1$ (highest priority) and so on. Then Calculate the upper bound for every task through

$$R_i^{ub} = \frac{C_i + \sum_{j=1}^{i-1} C_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j}$$

C_i : worst case execution time (WCET)

U_i : utilisation $U_i = \frac{C_i}{T_i}$

T_i : task period

Thus $R_1^{ub} = C_1$ and each lower priority task has a response time that depends on the utilisation of the tasks above it.

🔥 Response Time Upper Bound

- If task $R_i^{ub} \leq D_i$ checks, task is practically schedulable
- If task fails test, there is still chance for it to work, as there've been many assumptions
- Response times tests don't account for task interactions and os overhead
- Tests depend on some kind of worst case execution time per task

Task	i	Deadline	Period	WCET	U
A	1	1 ms	10 ms	0.5 ms	0.05
B	2	4 ms	30 ms	3 ms	0.1
C	3	7 ms	25 ms	4 ms	0.16

The response time upper bounds for each task are

$$R_1^{ub} = C_1 = 0.5ms$$

$$R_2^{ub} = \frac{C_2 + C_1(1 - U_1)}{1 - U_1} = 3.66ms$$

$$R_3^{ub} = \frac{C_3 + C_1(1 - U_1) + C_2(1 - U_2)}{1 - (U_1 + U_2)} = 8.44ms$$

Comparing R^{ub} to the deadline:

Task	i	Deadline	R^{ub}
A	1	1 ms	0.5 ms
B	2	4 ms	3.66 ms
C	3	7 ms	8.44 ms

- Task A is schedulable ($R_1^{ub} < D_1$)
- Task B is schedulable ($R_2^{ub} < D_2$)
- Task C is not schedulable according to this test ($R_3^{ub} > D_3$)

Estimating WCET To estimate **Worst Case Execution Time**, there are two basic approaches

Static Analysis (Analysis of the source code)

- Relies on good processor model
- Good for simple code & MCU
- Difficult for complex code & MCU

Dynamic analysis (Measurement at runtime)

- Common in industry
- Must be able to exercise worst-case path
- Simple: Toggle GPIO

Concurrency / Gleichzeitigkeit

Tasks "logically happen" at the same time, either physically (multi-core) or through context switches (single-core). This should improve **Responsiveness**.

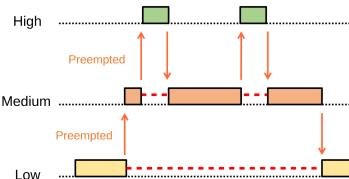


Figure 0.1: Tasks of different priority in a preemptive RTOS

- **Cooperative** multi-tasking: Tasks determine whether they give control back or not
- **Preemptive** multi-tasking: A scheduler takes control of what task gets how much time and also pulls tasks from executing

Important Properties

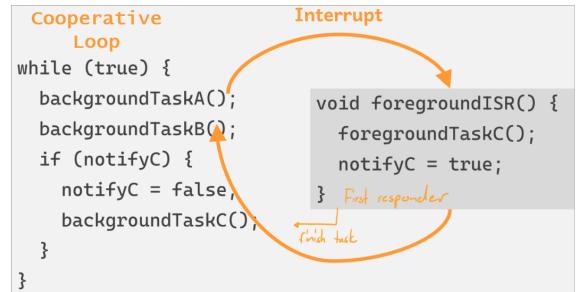
Safety: Nothing bad ever happens

Liveness: Something useful eventually happens

Responsiveness: Eventually is a reasonable amount of time

- + Simple
- No priorities
- Worst case response = sum of all task times
- Scheduling can be deterministic, but task periods must be harmonic
- Must manually manage state of long-running tasks
- Any change may alter response times

Preemptive: Fore-/Background



- + Prioritise tasks
- + Separation of tasks and scheduling eases change
- Worst case response = interrupt time + longest task time
- Time-triggered scheduling deterministic, task harmonic
- Complex task handling / 3rd-party microkernel
- Race conditions for interrupts
- Manual managing of long-running tasks

Preemptive: RTOS Implementation Each task is written as if it is a *single main loop*.

```

// Main Setup
#include <FreeRTOS.h>
#include <task.h>
void main() {
    xTaskCreate(BlinkTask, "BlinkA", STACK_SIZE, NULL,
    → BLINK_PRIO, NULL);
    xTaskCreate(BlinkTask, "BlinkB", STACK_SIZE, NULL,
    → BLINK_PRIO, NULL);
    vTaskStartScheduler();
}

// The Task
void BlinkTask(void* pvParameters) {
    while(true) {
        ledInvert();
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

```

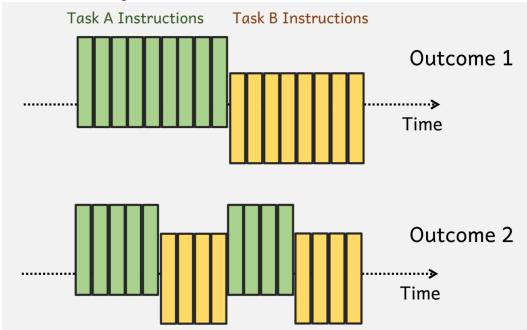
- + Prioritise tasks and responses
- + Separation of tasks and scheduling eases change
- + Long-running tasks are scheduler managed
- + Scheduling is flexible
- + Useful features (timing-services, protocol stacks, multi-processors,...)
- Worst-case response = interrupt time + scheduler context switch
- Depending on 3rd-party microkernel
- Must manage raceconditions on resources

Cooperative Round-Robin

- OS overhead costs resources

⚠️ Concurrency Issues

Race Condition: Outcome depends on timing → Occur when task modify **shared data**



Containment: Keep data within a task

Immutability: Use unchanging data

Atomic Data: Only share data which can be changed atomically

Critical Section: Section of code must execute *atomically* (in one run)

Synchronisation: Concurrency Control

```
// counter_manipulator.c
static SemaphoreHandle_t mutex;
static int32_t counter = 0;

void counterAdd(int32_t value) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    counter = counter + value;
    xSemaphoreGive(mutex);
}

int32_t counterGetValue() {
    xSemaphoreTake(mutex, portMAX_DELAY);
    int32_t local = counter;
    xSemaphoreGive(mutex);
    return local;
}

// counter_task.c
void CounterTask(void* pvParameters) {
    for (int32_t i = 0; i < 100; i++) {
        counterAdd(1);
    }
    int32_t final = counterGetValue();
    printf("%d, %d", final);
    vTaskSuspend(NULL);
}
```

Mutex (Mutual Exclusion) Only one task can take / lock the mutex at a time. Other task trying to acquire the mutex are blocked until the mutex is released. A mutex can only be **released** by the task that **acquired** it.

If two or more tasks **share a resource**, use a mutex for protection.

```
#include <semphr.h>
SemaphoreHandle_t mutex = xSemaphoreCreateMutex();

...
// in a task
for (int32_t i = 0; i < 1000000; i++) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    counter = counter + 1;
    xSemaphoreGive(mutex);
}
```

Semaphore A semaphore can be **given** by any task. To receive and wait for a signal use `xSemaphoreTake(...)`.

If two or more tasks need to **coordinate actions**, use a semaphore to send signals.

```
SemaphoreHandle_t signal = xSemaphoreCreateBinary();
void TaskA(void* pvParameters) {
    for (;;) {
        printf("Ready!");
        xSemaphoreGive(signal);
        vTaskSuspend(NULL);
    }
}
void TaskB(void* pvParameters) {
    for (;;) {
        xSemaphoreTake(signal, portMAX_DELAY);
        printf("Go!");
        vTaskSuspend(NULL);
    }
}
```

Task Notification Task notifications are FreeRTOS specific and offer a *light weight* alternative to a semaphore.

Queues Used to send data from one task to the other. Data is written to a queue **as copy**.

```
// create queue
#include <queue.h>
```

ℹ️ Encapsulate Synchronization

- Avoid scattering mutexes around the code
- Prevent client tasks of accessing mutexes directly
- Ensure only a single mutex is hold at a time

```

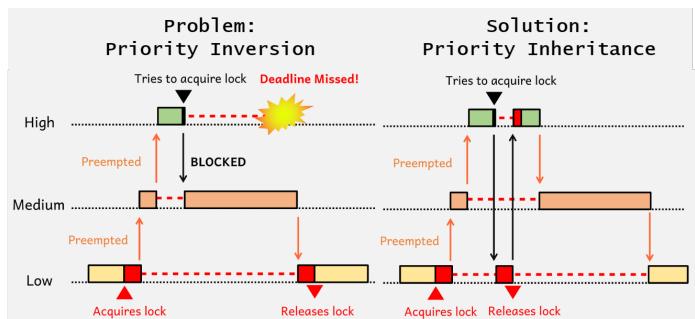
QueueHandle_t msgQueue = xQueueCreate(QUEUE_SIZE,
→ sizeof(msg_t));

// send a message
xQueueSend(msgQueue, &toSend, 0);

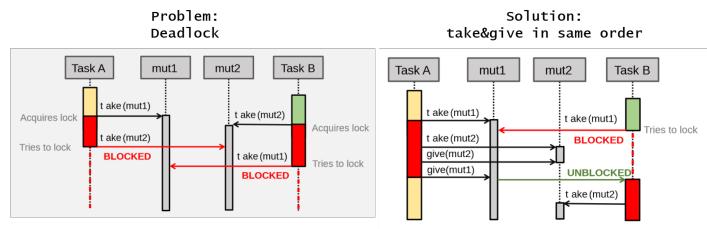
// receive a message
xQueueReceive(msgQueue, &received, portMAX_DELAY);

```

xQueueReceive(..., portMAX_DELAY) is blocking until something is put into the queue. The time can be adjusted by the last argument, usually portMAX_DELAY, veeeery long.



Deadlock



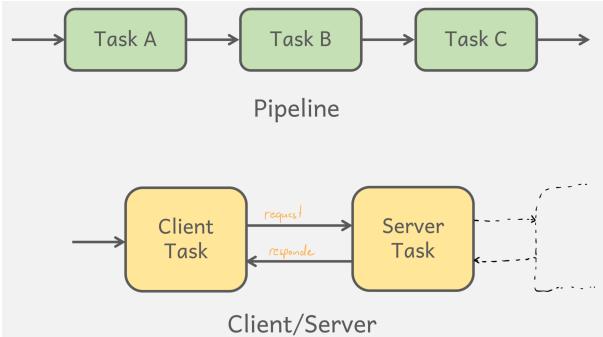
We can also design tasks to only block in one place and thus deadlocks are less likely

```

void Task(void* pvParameters) {
    for (;;) {
        xQueueReceive(...); // single block
        switch (received.msgType) {
            case MSG_A: // Handle A events
            case MSG_B: // Handle B events
            case TIMER_1: // Handle timer
            case default: // Assert?
        }
    }
}

```

Or use a structure which prevents deadlocks generally



Use a **Pipeline** to minimise circular dependencies and a **Client-Server** structure to restrict the directionality of connections.

Priority Inversion

Resources

Performance

Testing

Testing is for *Finding Bugs*, *Reduce risk to user and business*, *reduce development costs*, *keep code clean*, *improve performance* and to *verify that requirements are met*. There are different test which can be performed:

- **Unit Testing**: Verify behaviour of individual units (modules)
- **Integration Testing**: Ensure that units work together as intended
- **System Testing**: Test **end-to-end** functionality of application
- **Acceptance Testing**: Verify that the requirements are met (whole system)
- **Performance Testing**: Evaluate performance metrics (e.g. execution time)
- **Smoke Testing**: Quick test to ensure major features are working

To make testing efficient, we implement automatic testing routines. They act as a **live documentation**. Allows for **refactoring with confidence**.

Unit Test

A good test case checks **one behaviour** under **one condition**, this makes it easier to localise errors.

```

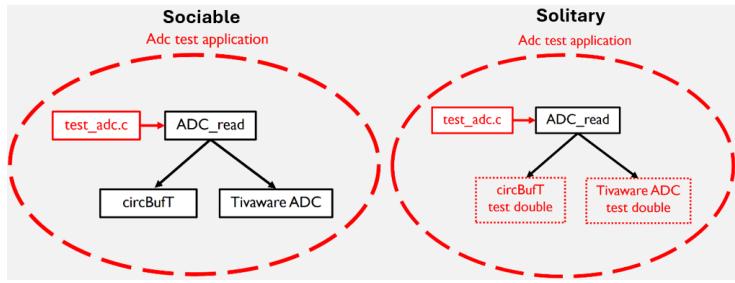
void test_single_element_in_single_element_out(void)
{
    // Arrange: given buffer has a single element
    writeCircBuf(&buff, 11);
    // Act: when buffer is read
    uint32_t value = readCircBuf(&buff);
    // Assert: then the same value is returned
    TEST_ASSERT_EQUAL(11, value);
}

```

Testing Frameworks

- Unit Test Framework Unity
- Test Double Framework fff

Unit Test with Collaborators



Test Doubles

Implement test doubles through the fake function framework ([fff](#)).

There are different variations of test doubles:

Stub: Specify a return value - *Arrange*

```
// Set single return value
i2c_hal_register_fake.return_val = true;
// Set return sequence
uint32_t myReturnVals[3] = { 3, 7, 9 };
SET_RETURN_SEQ(readCircBuf, myReturnVals, 3);
```

Spy: Capture Parameters - *Arrange / Assert*

```
// Arrange, e.g. get passed function
adc_hal_register(ADC_ID_1, dummy_callback);
void (*isr) (void) = ADCIntRegister_fake.arg2_val;
// Assert Parameter
TEST_ASSERT_EQUAL(3,
→ ADCSequenceDataGet_fake.arg1_val);
```

Mock: Can act as a *Stub*, *Spy*, and much more (from [fff](#)). Implemented as follows:

```
// in some_mock.h
VALUE_FUNC(uint32_t *, initCircBuf, circBuf_t *,
→ uint32_t);
VOID_FUNC(writeCircBuf, circBuf_t *, uint32_t);
```

Fake: Provide a custom fake function - *Arrange*

```
// Define Fake Function
int32_t ADCSequenceDataGet_fake_adc_value(uint32_t
→ arg0, uint32_t arg1, uint32_t *arg2) {
*arg2 = FAKE_ADC_VALUE;
return 0;
}
// Apply Fake Function - Arrange
ADCSequenceDataGet_fake.custom_fake =
→ ADCSequenceDataGet_fake_adc_value;
```

Continuous Integration

CI is used to automate the integration of code changes. These are automated scripts running all the tests. This is usually implemented in the code hoster (e.g. *GitLab*) and is executed after

every push. It also runs before every merge and **blocks a merge** if one of the tests fails.

Higher Level Testing

Unit tests only verify small elements of a system in isolation.

Automated Acceptance Testing

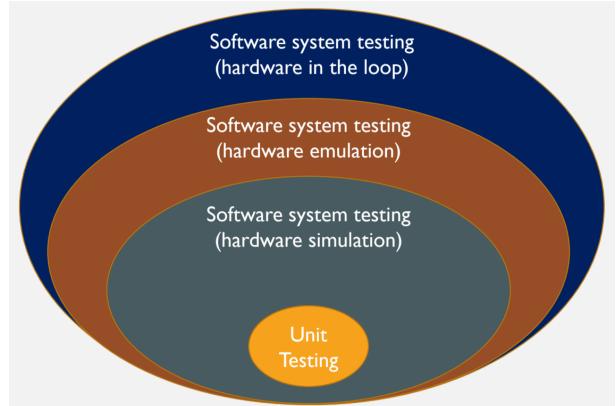
- Verifies system requirements
- Live documentation of high-level requirements
- Understandify behaviour
- Acceptance test pass → requirement met
- Written by PM or QA (≈ customer)
- Written in natural scripting language
- Non-Technical stakeholder in the loop
- Called: **Behaviour-Driven Development BDD**

Automated System Tests

Hardware Simulation: Developed on PC, no need to know specific hardware implementation yet, limitations with hardware peripherals.

Hardware Emulation: Emulate processor on PC, needs resources for emulator. Tools: QEMU

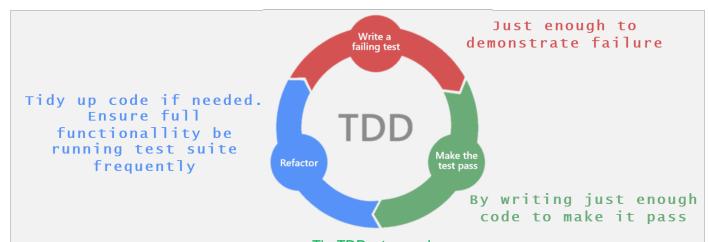
Hardware in the Loop: Runs on target, test scripts on a test enclosure to manipulate hardware, expensive setup. Tools: NI DAQ, Labview



Manual Testing

Sometimes automated test setups are more expensive. Manual testing can involve **user interaction**, **Debugger**, **direct Signal Probing** (Oscilloscope, Multimeter, Logic Analyzer).

Test Driven Design TDD



Applying TDD through writing *unit tests* during development, benefits:

- **Reduce Debug Time:** small feedback loop

- **Courage to make changes:** tested code is changeable code
- **Tests are Reliable:** high level of coverage
- **Good Architecture:** Writing test implies decoupling

For each new unit:

1. Come up with a **set of requirements**
2. Generate a **rough test list**
3. Implement unit by going through the list with **TDD**
4. Tick off, remove, or add items to/from the list in the process

Use **ZOM** to come up with tests:

- **Zero case(s):** simplest scenario → build interface
- **One case:** simplest scenario to transition from **Zero** to **One**
- **Many cases:** generalise design, each test case adds a scenario