Embedded Systems and Advanced Computing

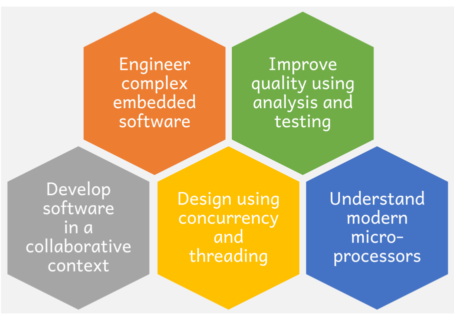
ENCE464

Andy Ming / Quelldateien

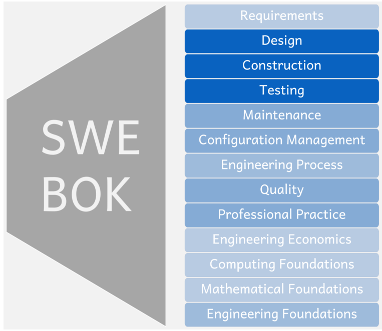
Table of contents

How to work code

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

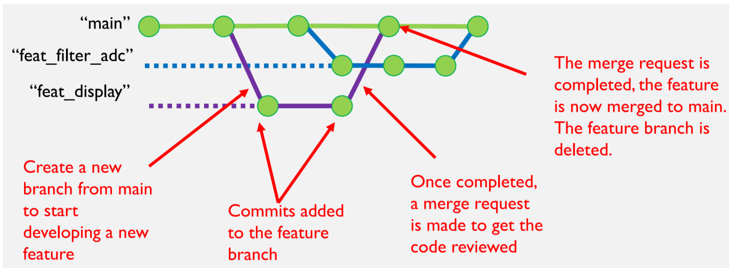


Software engineering has many different aspects (the dark blue ones are focused on here), find out more here.



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

```c
// 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.

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

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*

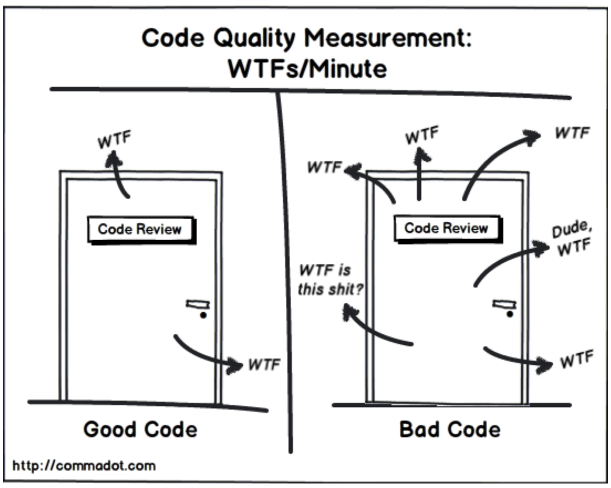
Comments

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 feeback as *bug, code, quality, preference*.



SOLID

How to make designs **flexible**.

Legacy Code

Embedded Software Design

Architecture

Architecture are "important" structures, every structure is important for a specific part of the software. There are several different structures in embedded software systems.

> **Architecture Goals**
>
> - *Understandability* - In Development & Maintenance
> - *Modifiability* - Through "best practices"
> - *Performance* - Reduce Overheads
>
> Other possible requirements: Portability, Testability, Maintainability, Scalability, Robustness, Availability, Safety, Security

Static Structures: Conceptual abstraction a developer works with

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

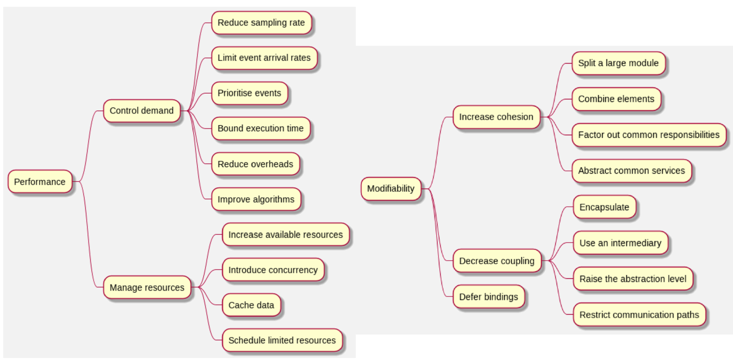
Dynamic Structures: Relationships that exist in executing software

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

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

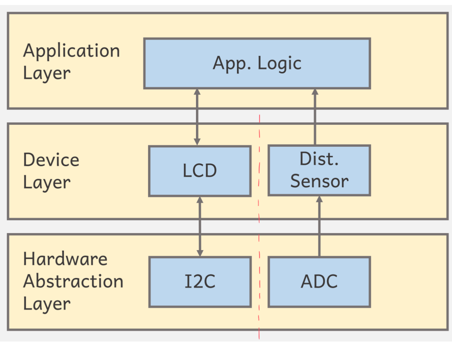
To keep record of decisions and to not loose the overview use tools like:

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

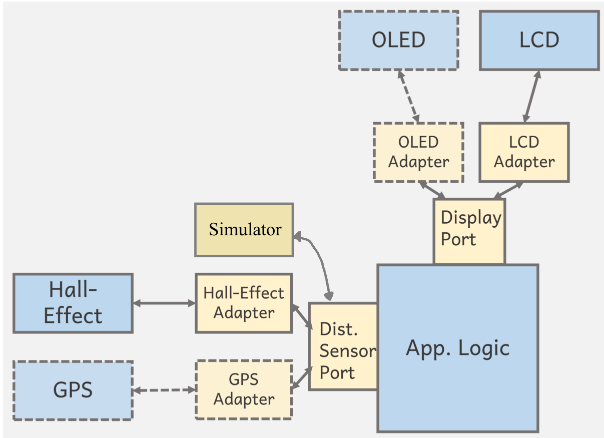
Supports portability and modifiability by allowing internal changes to be made inside a layer without impacting other layers, and isolating changes in layer-to-layer interfaces from more distant layers.



Ports-and-Adapters (or *Hexagonal*)

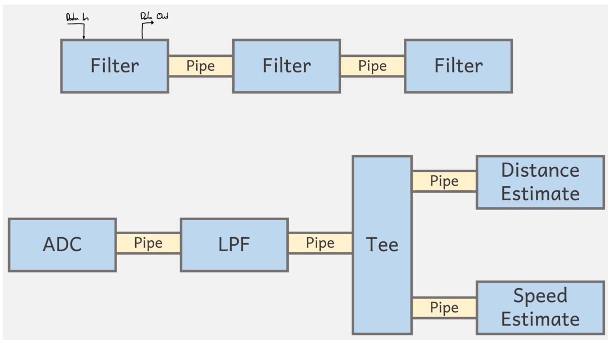
Introduces a single core logic which communicates through abstraction interfaces (**Ports**) to different modules. The **Adapters** map the external interactions to the standard interface of the port.

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

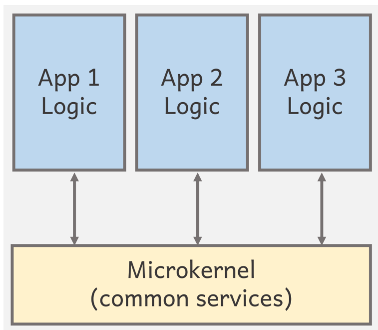
Supports modifiability through loose coupling between components, and performance by introducing opportunities for parallel execution.



Microkernel

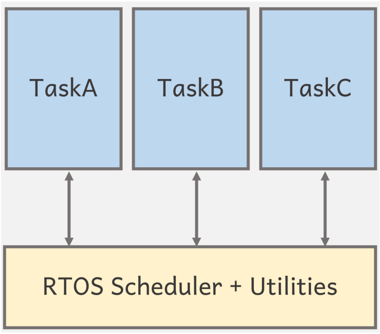
RTOS is a implementation of a Microkernel Architecture. The **Microkernel** includes a set of common core services. Specific services (**Tasks**) can be plugged into the kernel.

Supports modifiability and portability.



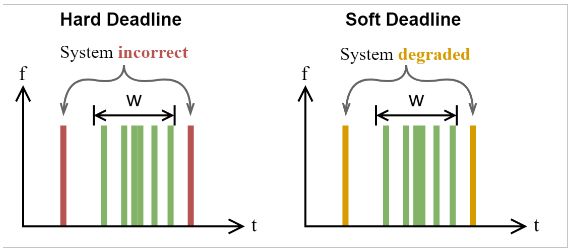
RTOS

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*

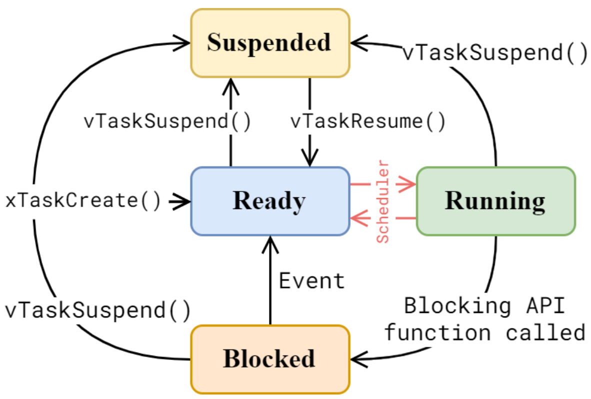


> **RTOS vs. Desktop OS**
>
> - Desktop OSs don't try to achieve *hard* real-time performance
> - In a Desktop OS, programs can be loaded in runtime
> - RTOS is compiled as part of the application, to add a new "program" the application has to be recompiled

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 it's 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**: $D_i = T_i$
> Futhermore, 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 priotity) 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 $R_1^{ub} = C_1$ ans 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 **W**orst **C**ase **E**xecution **T**ime, 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).

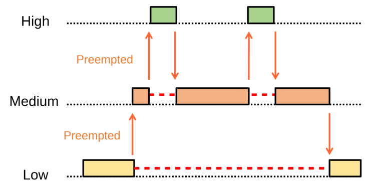


Figure 0.1: Tasks of different priority in a preemptive RTOS

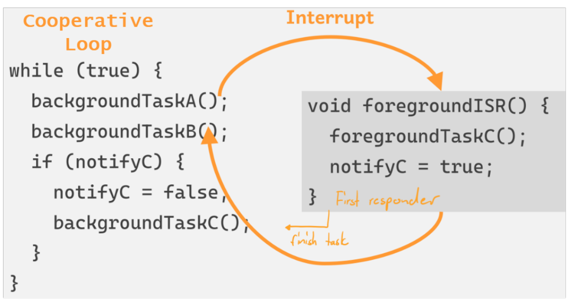
- **Cooperative** multi-tasking: Tasks determine whether they give control back or not
- **Preemptive** multi-tasking: A scheduler takes control of what task gets how much time and also pulls tasks from executing

Cooperative Round-Robin

\+ 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 Impelementation 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
+ Usefull features (timing-services, protocol stacks, multi-processors,...)
- • Worst-case response = interrupt time + scheduler context switch
- Depending on 3rd-party microkernel
- Must manage raceconditions on recourses
- OS overhead costs recources

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*: Evalutate 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**.

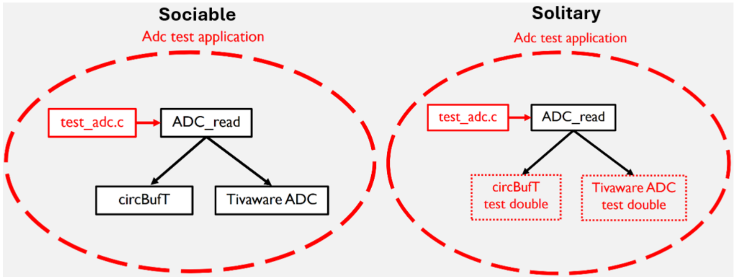
Unit Test

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.