Embedded Systems and Advanced Computing

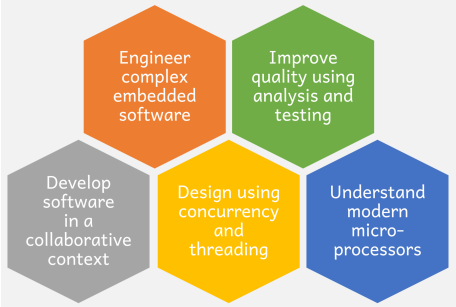
ENCE464

Andy Ming / Quelldateien

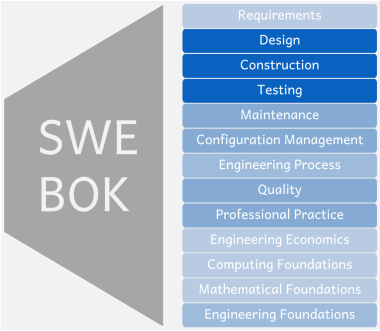
Table of contents

How to work code

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

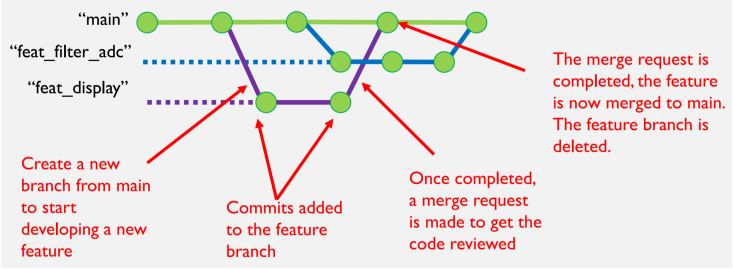


Software engineering has many different aspects (the dark blue ones are focused on here), find out more here.



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

```
// 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
```

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*

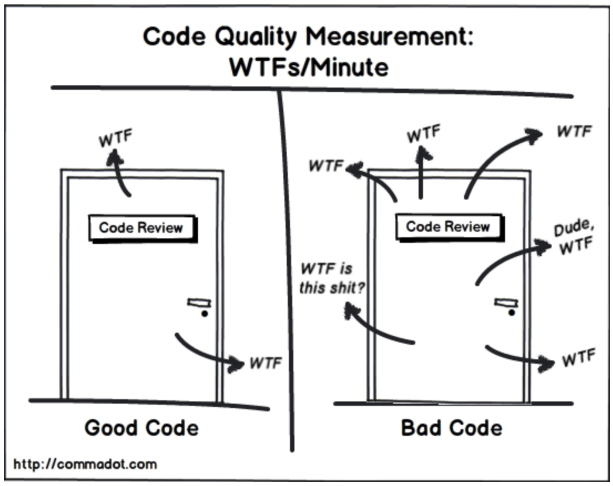
Comments

More comments ≠ better quality. Use comments only to:

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

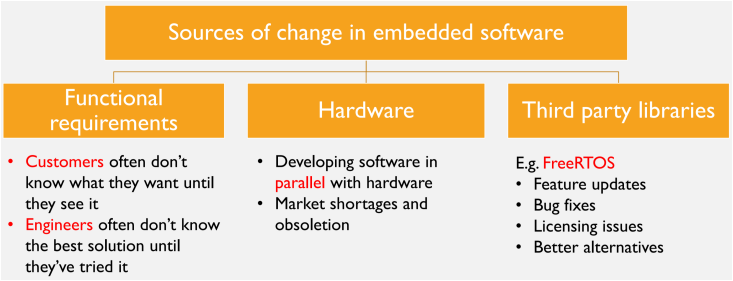
Code Reviews

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

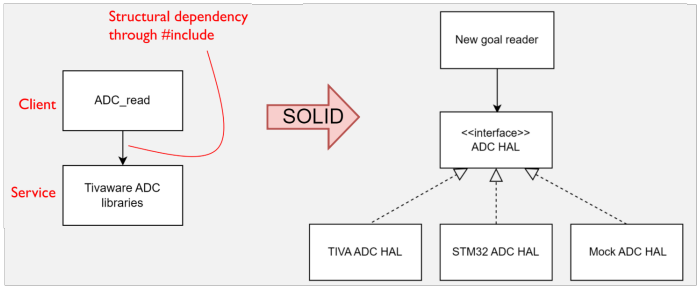


SOLID

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



SOLID is all about **managing dependencies**.



Single Responsibility Principle (SRP)

"A module should only have a single responibility. 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 breack 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 it's 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

Dependecy 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.

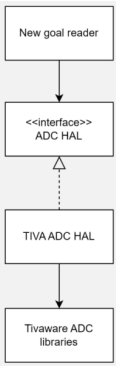
In OOP this is called **Polymorphism**.

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

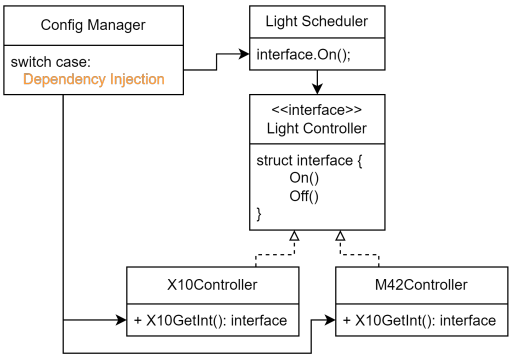


Multiple Instance Model

- 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 struc 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

Design Patterns

23 patterns introduced by the **G**ang **o**f **F**our (GoF), for **Dependency Management**. There are 3 types:

- **Creational**: Create instances of objects
- **Structural**: Set communication pathways
- **Behavioural**: Distribute responsibilities

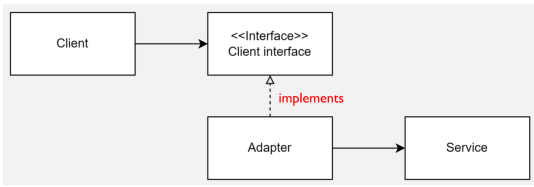
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Recommendations

- Don't overcomplicate, design for todays requirements
- Use pattern if beneficial, maybe simple code is sufficient
- Customise patterns to application

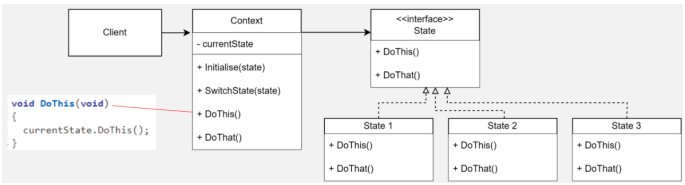
Adapter Pattern

- What: Wrap adapter around another module to give it a more desirable interface
- Hiding ugly interfaces of a 3rd-party service
- Hiding data conversions
- Make incompatible modules compatible
- Fulfills SRP, ISP, LSP



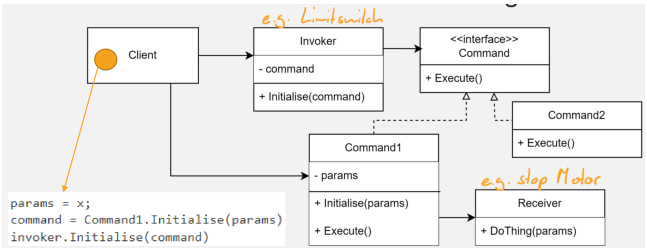
State Pattern

- What: Module will behave differently depending on internal states
- Allows implementation of FSM without several lengthy conditional statements
- State is saved in the private `currentState` variable
- There is a module implementation per state
- The client initialises the context with a state
- Fulfills SRP, OCP



Command Pattern

- What: Turns request into a stand-alone object
- Invoker calls receiver through command object
- Client attaches invoker with its commands
- Multiple things can execute one command
- Command is placed in queue until receiver is ready
- Triggers can invoke series of commands
- Fulfills SRP, OCP



Legacy / Vererbt / 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.

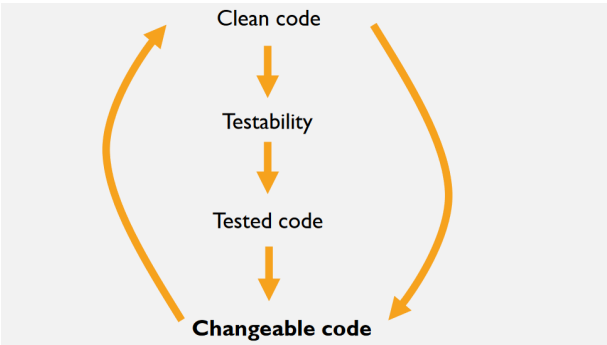
So we try to refactor until it's easy to make changes. To preserve functionality we iterate *in small increments* with **Trageted Tests** *(allows changes and new features to happen at the same time)*:

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.*

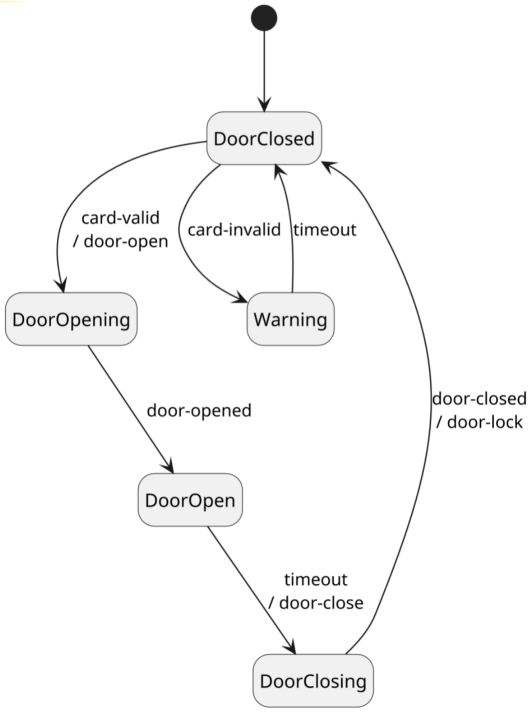
To make sure *key functionality* isn't altered, add **Strategic Tests**. (e.g. control algorithm, safety-critical error detection, ...)

> **Putting Code Under Test**
>
> 1. Identify are of code to test (targeted, characterisation, strategic)
> 2. Find test points (function calls, global variables → encapsulate ASAP, serial)
> 3. Break dependencies (Solitary tests)



Designing with Models

Event Driven State Machine



With a model there are already some flaws which arise. For example: What happens if the door is closing and a valid card is wiped?

Programm Execution:

- State: PC + Variable Values
- Transitions: Instructions

Time Driven State Machine

PID-Controllers are state machines with "near infinite" states

```
errorInt += error;
errorDeriv = error - lastError;
commandSignal = (Kp*error +
                 Ki*errorInt +
                 Kd*errorDeriv);
```

"States"

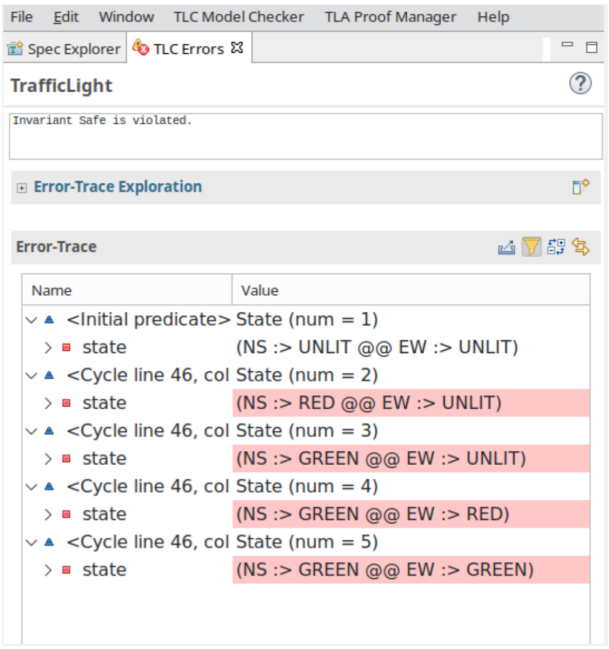
Use modelling languages

The following example is written in PLUSCAL

```
Light == {UNLIT, RED, GREEN, AMBER}
Direction == {NS, EW}

variables
    state = [dir \in Direction |-> UNLIT];

process Lights \in Direction
begin
    Cycle:
        either await state[self] = RED;
```

```
            state[self] := GREEN;
        or await state[self] = GREEN;
            state[self] := AMBER;
        or await state[self] = AMBER;
            state[self] := RED;
        end either;
    goto Cycle;
end process;

// Define a Requirement
define
    Safe =/ ~(state[NS] = GREEN / state[EW] = GREEN)
end define;
```

The model can be checked and we can verify if the requirement is met or not. Add to the model until it meets all requirements



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

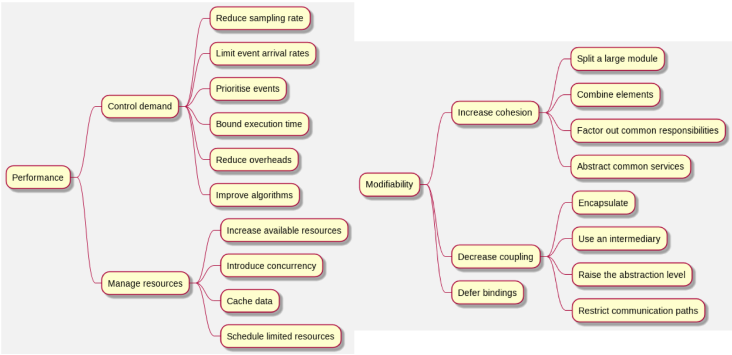
Dynamic Structures: Relationships that exist in executing software

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:

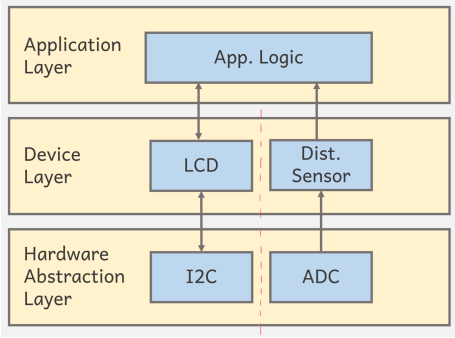
Quality	Often trades with	Notes
Modifiability	Time performance	Modifiability often requires indirection, which introduces overheads. Removing indirection makes modifications more difficult.
Testability	Time performance	Testability, like modifiability, often requires indirection, which introduces overheads.
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.

> **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.

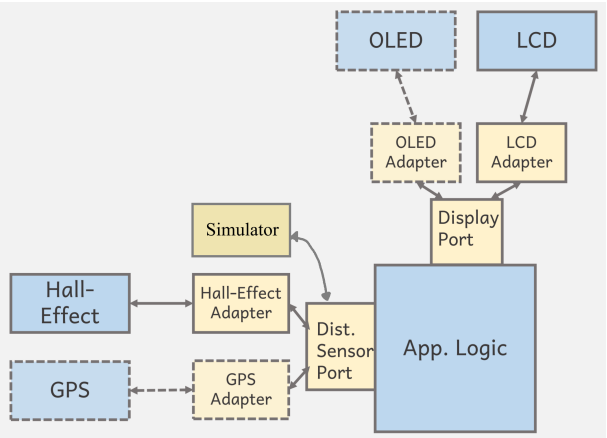
Supports portability and modifiability by allowing internal changes to be made inside a layer without impacting other layers, and isolating changes in layer-to-layer interfaces from more distant layers.



Ports-and-Adapters (or *Hexagonal*)

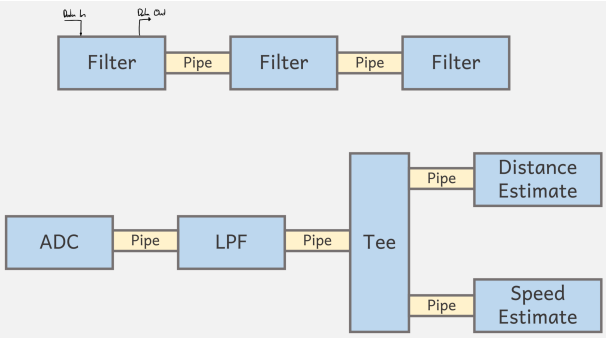
Introduces a single core logic which communicates through abstraction interfaces (**Ports**) to different modules. The **Adapters** map the external interactions to the standard interface of the port.

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

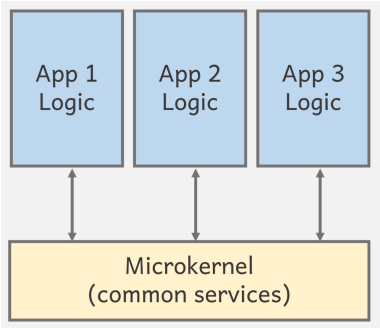
Supports modifiability through loose coupling between components, and performance by introducing opportunities for parallel execution.



Microkernel

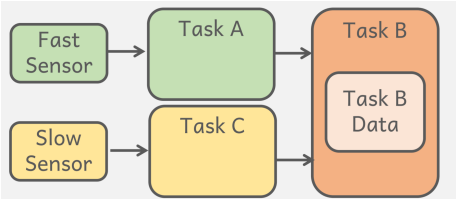
RTOS is a implementation of a Microkernel Architecture. The **Microkernel** includes a set of common core services. Specific services (**Tasks**) can be plugged into the kernel.

Supports modifiability and portability.



Tasks for Priority and Modularity

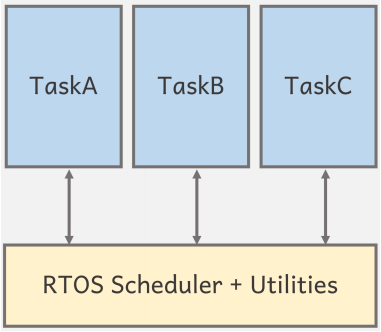
+ Control Priority
+ Control Response Time
+ Modularity
- Concurrency Issues
- Overhead (Scheduler)
- Starvation (Task gets no CPU time)



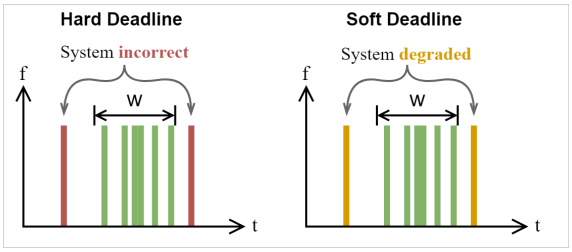
Use **just enough** tasks.

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*



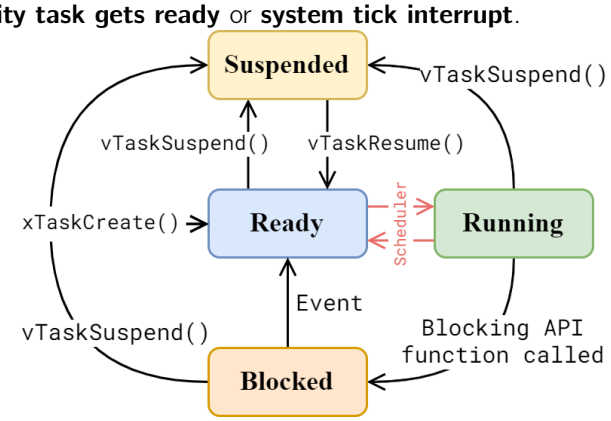
> **RTOS vs. Desktop OS**
>
> - Desktop OSs don't try to achieve *hard* real-time performance

- 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 it's deadline.

1. Highest ← shortest deadline
2. Lowest ← 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). This should improve **Responsiveness**.

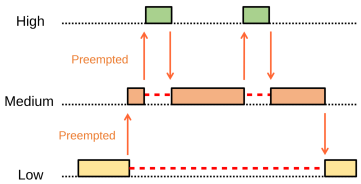


Figure 0.1: Tasks of different priority in a preemptive RTOS

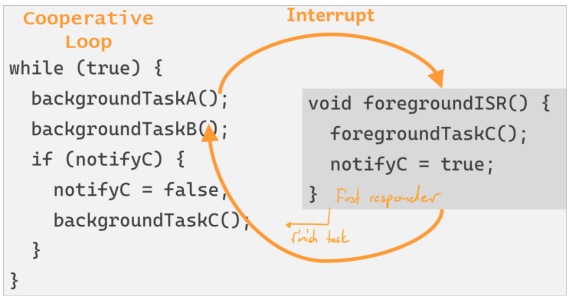
- **Cooperative** multi-tasking: Tasks determine whether they give control back or not
- **Preemptive** multi-tasking: A scheduler takes control of what task gets how much time and also pulls tasks from executing

> **Important Properties**
>
> **Safety**: Nothing bad ever happens
> **Liveness**: Something useful eventually happens
> **Responsiveness**: Eventually is a reasonable amount of time

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*.

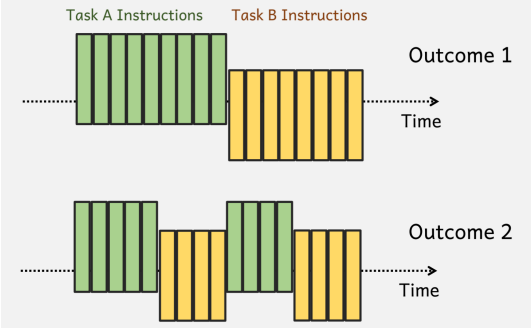
```
// Main Setup
#include <FreeRTOS.h>
#include <task.h>
void main() {
    xTaskCreate(BlinkTask, "BlinkA", STACK_SIZE, NULL,
↪   BLINK_PRIO, NULL);
    xTaskCreate(BlinkTask, "BlinkB", STACK_SIZE, NULL,
↪   BLINK_PRIO, NULL);
    vTaskStartScheduler();
}

// 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

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

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 recource**, use a mutex for protection.

```c
#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);
}
```

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

```c
// counter_manipultor.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, final);
    vTaskSuspend(NULL);
}
```

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.

```c
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.

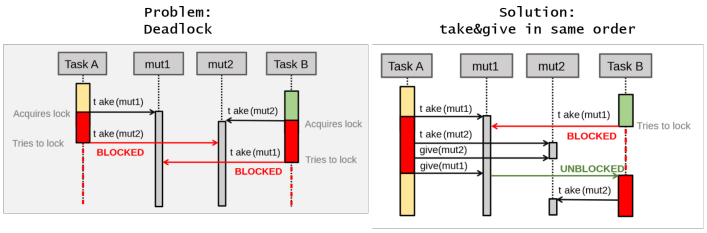
Queues Used to send data from one task to the other. Data is written to a queue **as copy**.

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

```c
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`, veeeeery long.

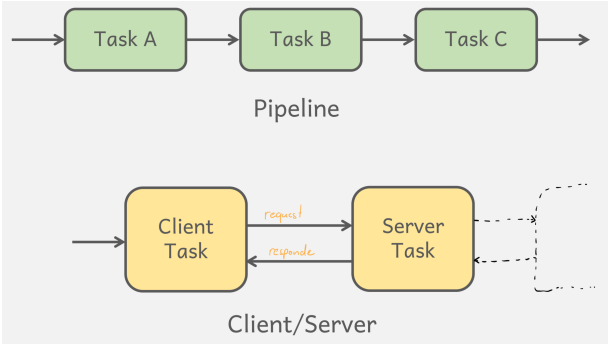
Deadlock .



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

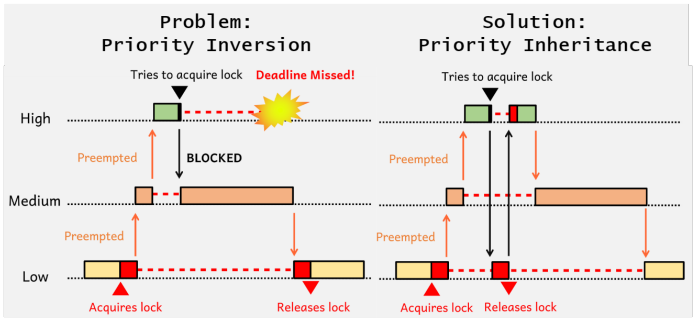
```c
void Task(void* pvParameters) {
    for (;;) {
        xQueueReceive(...); // single block
        switch (received.msgType) {
            case MSG_A: // Handle A events
            case MSG_B: // Handle B events
            case TIMER_1: // Handle timer
            case default: // Assert?
   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

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.

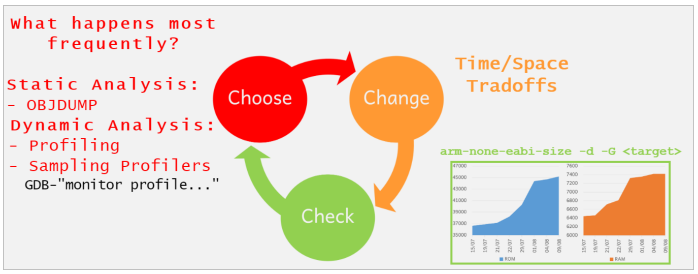
Priority Inversion .



Resources

> **Ausgangslage TIVA**
>
> 32KB RAM, 256KB ROM, Low-Power
> FreeRTOS needs: 5-10KB ROM, RAM: 236bytes (scheduler), 76bytes + storage size (queue), 60 bytes + stack size (task)

> **Premature optimization is the root of all evil.**
>
> Don't optimize before you know your constraints.



Choose

Get the memory map

```
set(CMAKE_EXE_LINKER_FLAGS) {
    ...
    -Xlinker
    -Map=${SMAKE_CURRENT_BINARY_DIR}/%
    ...
}

// output
...
.text 0x00000470 0x280 accl_manager.c.obj
      0x00000470 acclInit
      0x00000488 acclProcess
...
```

The distance between two functions equals their size (mostly).

Use `objdump -d` to disassemble executable

```
00000798 <readADC>:
     798:       b580            push    {r7, lr}
     79a:       b082            sub     sp, #8
     79c:       af00            add     r7, sp, #0
     79e:       2300            movs    r3, #0
     7a0:       607b            str     r3, [r7, #4]
     7a2:       2300            movs    r3, #0
     7a4:       807b            strh    r3, [r7, #2]
     7a6:       2300            movs    r3, #0
     7a8:       807b            strh    r3, [r7, #2]
     7aa:       e00c            b.n     7c6
     7ac:       4b0c            ldr     r3, [pc, #48
     7ae:       681b            ldr     r3, [r3, #0]
     7b0:       4618            mov     r0, r3
     7b2:       f000 fa67       bl      c84 <readCircBuf>
     7b6:       4603            mov     r3, r0
```

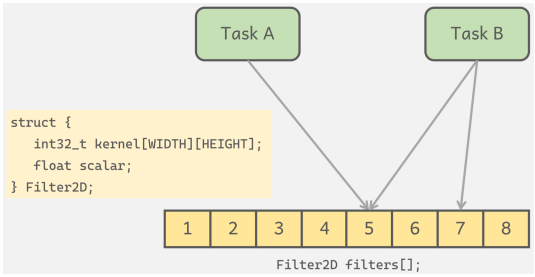
Profiling

- Use GPIO and oscilloscope to profile how long certain parts of a function run
- Use a DIY sampling profiler

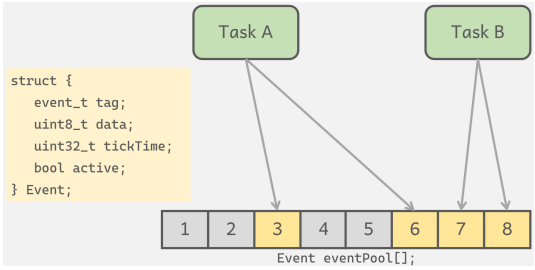
```
(gdb) continue Continuing.
^C Program received signal SIGINT, Interrupt.
↪  prvIdleTask (pvParameters=0x0
↪  <vPortValidateInterruptPriority>)
    at FreeRTOS/Source/tasks.c:3487
3487                vApplicationIdleHook();
```

Change

Flyweight pattern Task refers to same immutable data over and over again



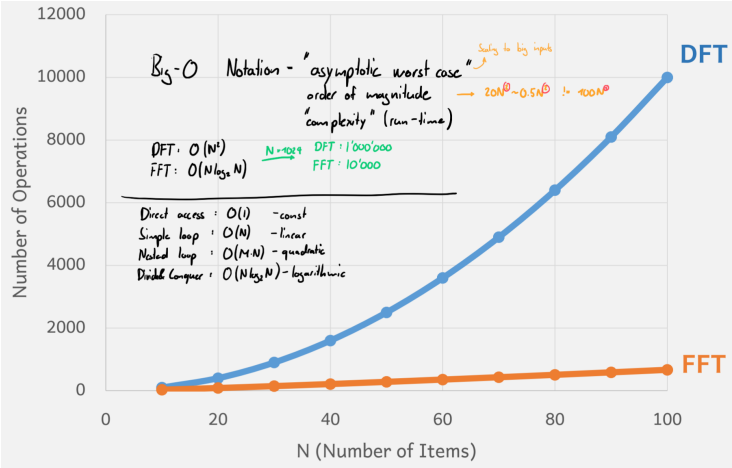
Memory Pool pattern Prebuilt datapool for mutable data → Control over size



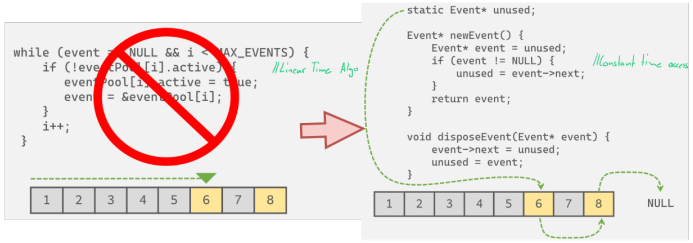
Compiler and Linker help

```
// Optimize for size
gcc -Os <your file>.c
// Use link-time optimizations
gcc -flto
// Use linker to remove unused code and data
gcc -ffunction-sections -fdata-sections
Link with --gc-sections
// Optimize for speed
gcc -O2 <your file>.c
gcc -O3 <your file>.c
```

Use better algorithms .



Write better algorithms .



Cache Data .

```
...
int degrees = 2*PI*freq*time;
output = amplitude*sin((float)degrees);
...
```

```
static float sinLookup[360] = {
    0.0, 0.017, 0.034, ...
};

...
int degrees = (2*PI*freq*time) % 360;
output = amplitude*sinLookup[degrees];
...
```

More

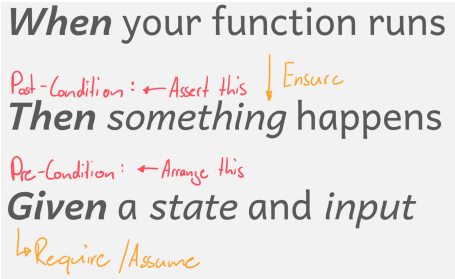
- Minimize data passing by **passing by reference**
- **Datacompression** with Differencing (store difference of two values in sequence), or Run length encoding
- Task notifications instead of queues
- Limit scope of local variables
- **Reduce Overhead** through removing layers

Engineering in the face of uncertainty:

1. Use margins (e.g. RAM 200%-300%)
2. Estimate early
3. Check often

Performance

Preemptive Debugging




```c
float squareRoot(float x) {
    // Require:
    // Failure equals happened before function gets
    ↪  called
    assert(x >= 0);
    y = ... ;
    // Ensure:
    // Failure equals error in this function
    assert((y*y) == x);
    return y;
}
```

With FreeRTOS `configAssert( x )` can be called. The behaviour is user dependent (standard `while(true);` → Fail-Safe.

```c
/* Custom implementation */
void vAssertCalled( const char * pcFile, unsigned long
↪  ulLine ) {
 (void)pcFile; // unused
 (void)ulLine; // unused
 while (true); // LED, EEPROM,...
}
```

Don't **R**epeat **Y**ourself - **DRY**

```c
ASSUME("event in set of handled events")

switch (event) {
  case ev1: handleEv1(); break;
  case ev2: handleEv2(); break;
  case ev3: handleEv3(); break;
}
```

```c
switch (event) {
  case ev1: handleEv1(); break;
  case ev2: handleEv2(); break;
  case ev3: handleEv3(); break;
  default:
    // Should never get here
    ASSERT(false);
}
```

Don't do this

```c
ASSERT(kbPress != 'j');
```

Avoid asserting **expected errors** → Handle directly in code

```c
ASSERT(i++ < 5);
```

Avoid **Sideeffects**

Static Analysis

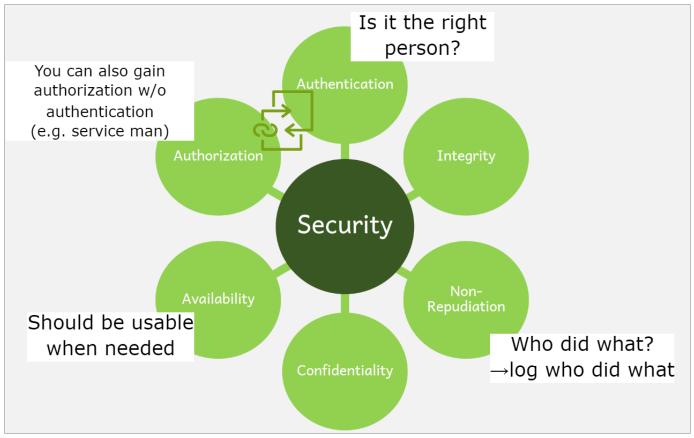
Happens **at compile-time**:

```c
_Static_assert(BLINK_STACK_SIZE >
↪  configMINIMAL_STACK_SIZE, "Stack size too small");
```

Use a framework like Frama-C:

```c
/*@
    requires y > 0
    ensures \result > y
*/
int makeLarger(int y) {
    int x = y*2;
    return x;
}
```

Security



C is not a "safe" language

Array overflow is not prevented, you could access the `state` variable in the following example:

Code

```c
uint32_t inputBuffer[128];
uint8_t state;
```

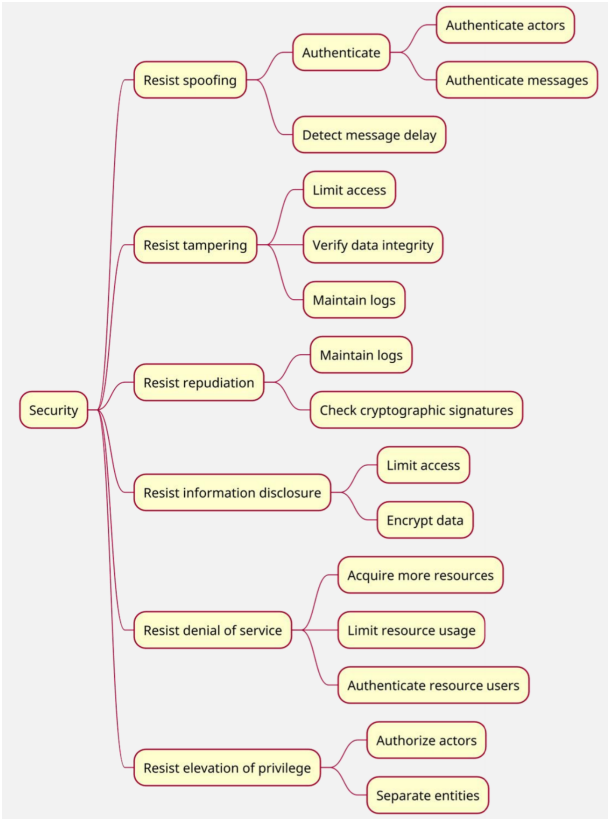
Memory map

```
.bss    0x00009040      ...
        0x00009040      inputBuffer
        0x00009240      state
```

Security failures are often design failures

Use the checklist **STRIDE** and design a architecture to avoid running into problems

- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation Disclosure
- **D**enial of Service
- **E**levation of Privilige



Evaluate Threads

This is time consuming → **in Projektmanagement beachten**
https://emb3d.mitre.org/

Element	Interaction	Threat	Mitigation
Fitness Monitor	Transmit data to app	Tamper with data in transmit	Cryptographically sign data
	Transmit data to app and website	Information disclosure	End-to-end encryption of data
...	...	...	...
Mobile App	Poll FM for data	Spoofing of FM packets	Explicitly paired Bluetooth connection
...	...	...	...

Follow "secure" coding standards

https://securecoding.cert.org/

EXP12-C. Do not ignore values returned by functions
EXP19-C. Use braces for the body of an if, for, or while statement

CON35-C. Avoid deadlock by locking in a predefined order
CON43-C. Do not allow data races in multithreaded code
CON01-C. Acquire and release synchronization primitives in the same module, at the same level of abstraction

Use static analysis to find problems

```
// Turn on all warnings
gcc -Wall -Wextra <your file>.c
// Use static analyzer
gcc -fanalyzer <your file>.c
```

Consider exceptional CHILDREN

- **C**omputation
- **H**ardware
  - Transient Faults
  - Memory Corruption
- **I**/O
  - Running out of file space?
- **L**ibrary
  - Handle error returns
- **D**ata input
  - Buffer input overflows
  - More / Less data than expected
- **R**aces and deadlocks
- **E**xternal user
  - Wrong, Late, Other input
- **N**ull pointer and memory

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

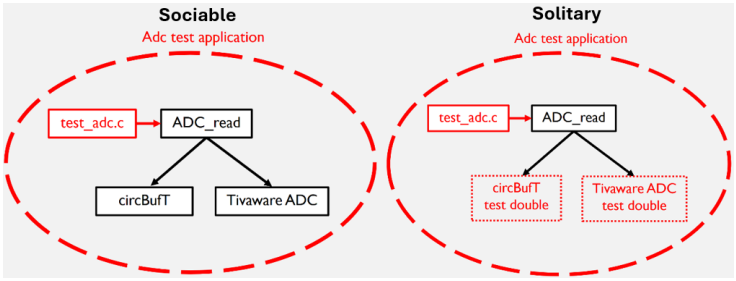
A good test case checks **one behaviour** under **one condition**, this makes it easier to localise errors.

```
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);
}
```

i 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

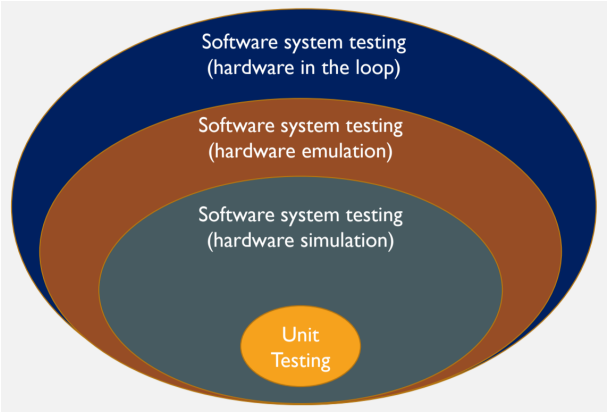
Unit tests only verify small elements of a system in isolation.

Automated Acceptance Testing

- Verifys 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: **B**ehaviour-**D**riven **D**evelopment **BDD**

Automated System Tests

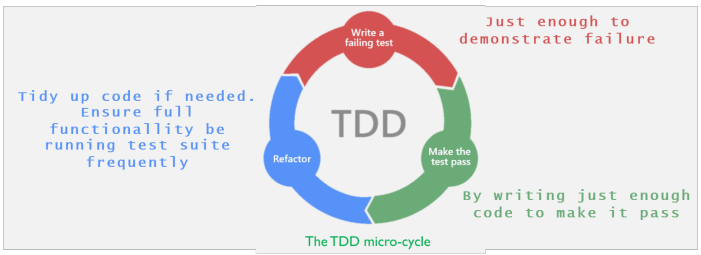
Hardware Simulation: Developed on PC, no need to know specific hardware implementation yet, limitations with hardware peripherals.
Hardware Emulation: Emulate processor on PC, needs resources for emulator. Tools: QUEMU
Hardware in the Loop: Runs on target, test scripts on a test enclosure to manipulate 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

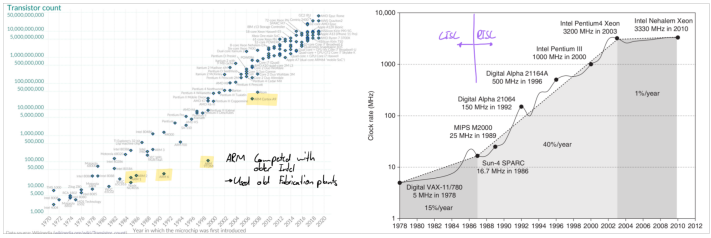
Use **ZOM** to come up with tests:

- **Z**ero case(s): simplest scenario → build interface
- **O**ne case: simplest scenario to transition from **Z**ero to **O**ne
- **M**any cases: generalise design, each test case adds a scenario

Computer History

Moor's Speculation

The *prediction* of Gordon Moore states "doubling the number of components on a IC each year". This is also largely true.



The exponential growth turned out to be true, for how much longer?

Early History of Digital Computers

The early history of digital computers and microprocessors highlights the progression from large, power-hungry machines to more efficient, smaller designs:

- **1950s**: The largest computer, IBM AN/FSQ-7, used 55,000 vacuum tubes and consumed 3 MW of power.
- **1964**: DEC introduced the first minicomputer, the PDP-8, followed by the PDP-11, which played a key role in developing Unix and C.
- **Intel 4004 (1971)**: The first microprocessor with a 4-bit CPU and 2300 transistors, originally designed for calculators.
- **Intel 8008 (1972)**: A more advanced 8-bit CPU with 5000 transistors.
- **Intel 8080 (1974)**: Improved performance (10x over 8008) and used in the first PC (Altair).
- **Texas Instruments TMS1000 (1974)**: First microcontroller, used in calculators and appliances.
- **MOS Technology 6502 (1975)**: Popular in early PCs (Commodore, Apple, Atari) for $25.
- **Intel 8086/8088 (1978)**: Intel's first 16-bit processor, chosen for the IBM PC despite flaws.
- **Motorola 68000 (1979)**: A 32-bit processor used in the Apple Macintosh.
- **Intel 386 (1985)**: Introduced linear addressing and virtual memory support, popular in PCs by 1990.

> **Why x86?**
>
> The **IBM PC** is the first personal computer, the engineers wanted to use the *Motorola 68000* chip, but management choose the *Intel 8088*, that's the dawn of Intels dominance and thus the dominance of the *x86-Architechture*

Fundamentals of Microprocessors and Architectures

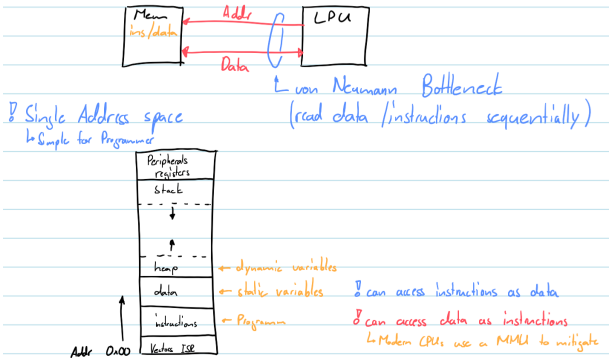
Microprocessor architectures

Flynn's Classification processor architectures into subgroups

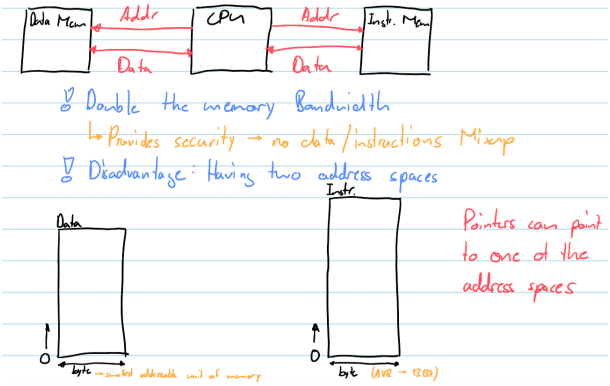
SISD - Single Instruction / Single Data

Processors with a SISD-Architecture are **Uniprocessors (von Neumann)**, they consist of a single processor. They are partitioned into *input devices, output devices, memory, ALU and control unit

Princeton Memory stores *data and instructions*, thus the **von Neumann Bottleneck** appears. There is also the risk of **accessing data as instructions** and vice verca. This can stall the CPU and is mitigated through memory management units (MMU) in modern processors.



Harvard The memory is seperated as data / instruction. This adds security, but harder for the programmer (What memory space does a pointer point to?).



Modified Harvard There are separate instruction and databuses, but share the same memory space. Application: **DSP**

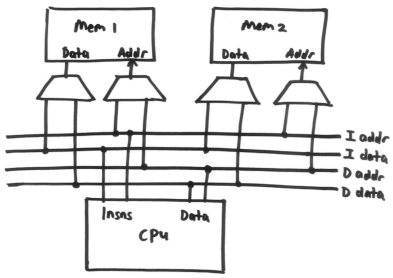
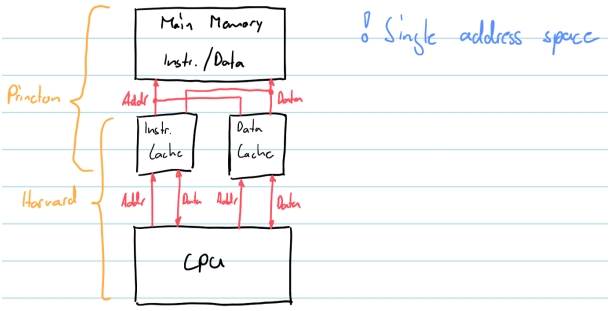


Figure 2.3: The modified Harvard architecture uses bridges to link the buses. These are implemented using multiplexers and bidirectional tri-state buffers (transceivers).

Hybrid There is a single memory space and data/insn-bus, but to gain speed, caches are used to mimic a harvard architecture.



SIMD - Single Instruction / Multiple Data

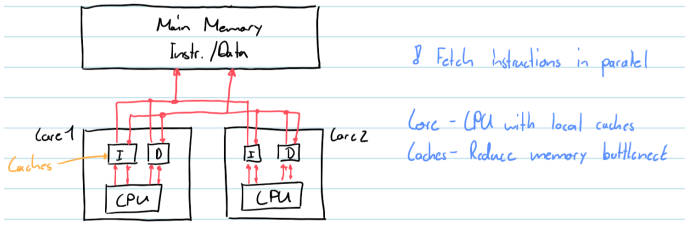
Typical SIMD-Processors are **Array-/Vector-Processors** and are often used in *GPUs* or *special CPUs*. Each processor simultaneously performs the same instruction on different data.

> **i** SIMD Instructions
>
> TO operate on **vector registers**...
> - ... *ARM* Processors have **NEON** instructions
> - ... *Intel* Processors have **SEE** instructions

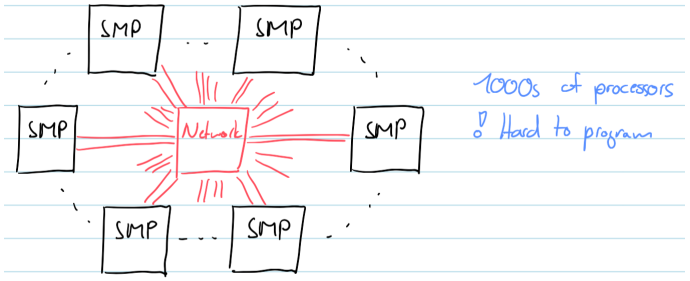
MIMD - Multiple Instruction / Multiple Data

MIMD-Processors are multi-core processors, with several cores, often accessing the same memory. This is done to improve computer performance through **parallelism**.

SMP - Symmetrical Multiprocessor Few identical processors share a common memory. Caches are used to mitigate the *von Neumann Bottleneck*.



MPP - Massively Parallel Processor Many processors using distributed memory and communication through a network *(many topologies, e.g., star, ring, ...)*.



Accessing Program Memory on AVR

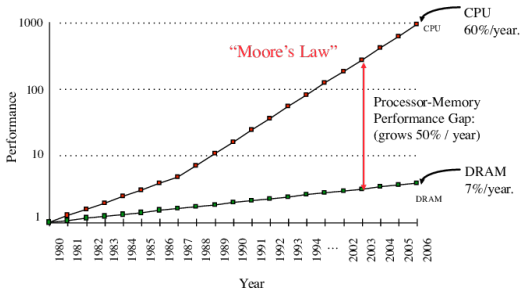
AVR is a manufacturer who uses the *Harvard*-Architecture. To access the program memory (flash) the `PROGMEM` macro from `avr/pgmspace` is used. *(support of that is introduced in GCC 4.8)*

```
#include <avr/pgmspace.h>

// for flash access
PROGMEM const char flash_str[] = "Hello world";
char flash_read_byte (const char *p) {
    return pgm_read_byte(p);
}

/* translates to following assembly */
flash_read_byte:
    movw r30 , r24
    lpm r24 , Z    // load program memory
    ret

// for SRAM access
const char sram_str[] = "Hello world";
char sram_read_byte (const char *p) {
    return *p;
}

/* translates to following assembly */
sram_read_byte:
    movw r30 , r24
    ld r24 , Z
    ret
```

Instruction Set Architectures (ISA)

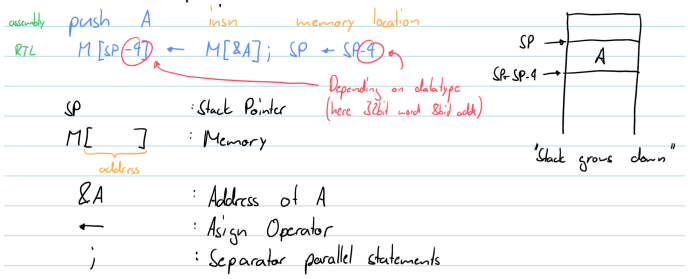
The *instruction set architecture* (ISA) is the assembly language of a specific processor. The ISA includes things such as:
- Instruction Set (NOP, BRA, LOAD, ADD, ...)
- Data Types (u/s-int, float, addresses)

- Registers
  - PC: programm counter
  - SP: stack pointer
  - R0, R1, R2, . . . : general purpose registers
  - W, Z, V, C: status registers
- Addreassing nodes, for accessing memory



Register Transfer Language (RTL)

Describes how a cpu instruction behaves



Stack ISA

Stack ISAs operands are pushed onto a stack before operators are applied. Source operands are popped off the stack, and destination operands are pushed back onto it, resulting in very short instructions. However, the stack can become a bottleneck, as it is not randomly addressable.

```
Assembler   RTL
PUSH  A     ; SP <- SP - 4; M[SP] <- M[&A];
PUSH  B     ; SP <- SP - 4; M[SP] <- M[&B];
ADD         ; M[SP + 4] <- M[SP + 4] + M[SP]; SP <- SP + 4
POP   C     ; M[&C] <- M[SP]; SP <- SP + 4
```

Accumulator ISA

Accumulator ISAs use a single accumulator register for storing the results of all ALU operations, leading to short instruction lengths where only one operand is specified. Memory traffic is high since the accumulator is the primary storage, but this design was popular in early microprocessors when memory and CPU speeds were comparable.

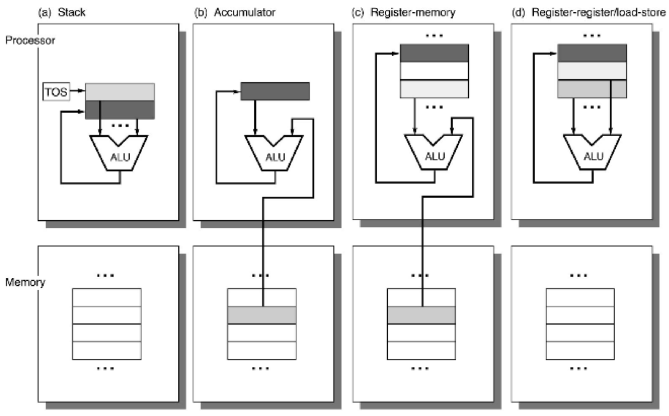
```
Assembler   RTL
LOAD  A     ; ACC <- M[&A]
ADD   B     ; ACC <- ACC + M[&B]
STORE C     ; M[&C] <- ACC
```

General Purpose Register

> Issue with frequent memory access
>
> Because the speed of **DRAM** didn't keep up with the speed of **CPUs**, memory is a bottleneck of performance.

Most high-performance computers use general-purpose register ISAs because registers are faster than memory. These ISAs have many registers, explicit operands, and longer instructions. Load/store ISAs, where ALU instructions only operate on registers, are popular despite requiring more instructions because they are simpler and more suited to pipelining. ISAs can be classified into memory-memory, register-memory, and register-register, based on how many memory operands an ALU instruction can have.

CISC		RISC
memory-memory	**register-memory**	**register-register**
`ADD A,B,C`	`LOAD A,R0`	`LOAD A,R0`
	`ADD B,R0,R1`	`LOAD B,R1`
	`STORE R1,C`	`ADD R0,R1,R2`
		`STORE R2,C`
IC = 1	IC = 3	IC = 4



Pipeline and Parallelism

Memory Systems and Optimization

Advanced Topics and Future Technologies