



Core C++ 2024

Optimizing Embedded Software Infrastructure

Akram Zoabi, Alex Kushnir

Optimizing Embedded Software Infrastructure: Principles and Practices for Platform Selection

Akram Zoabi
Alex Kushnir
27/11/2024

Johnson & Johnson
MedTech

Electrophysiology





1. Who are we ?
2. Evolution of product
3. HW Platform Selection
4. Operating System Selection
5. Programming Language Selection
6. Testing
7. Q&A

Who are we?

How People See Embedded SW Engineers?



Application SW Engineers



HW Engineers



Product Managers



J&J MedTech

Our Managers



Our Day life

Electrophysiology

Who are we:



Alexander Kushnir
Principal SW Engineer
Biosense Webster, J&J MedTech

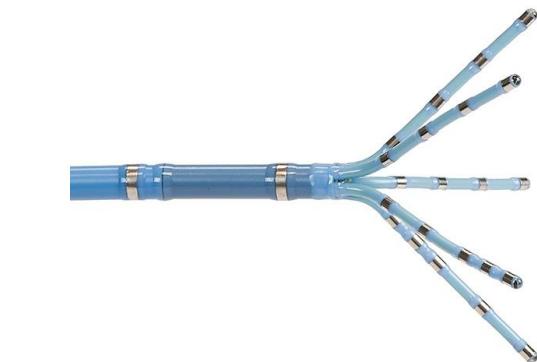
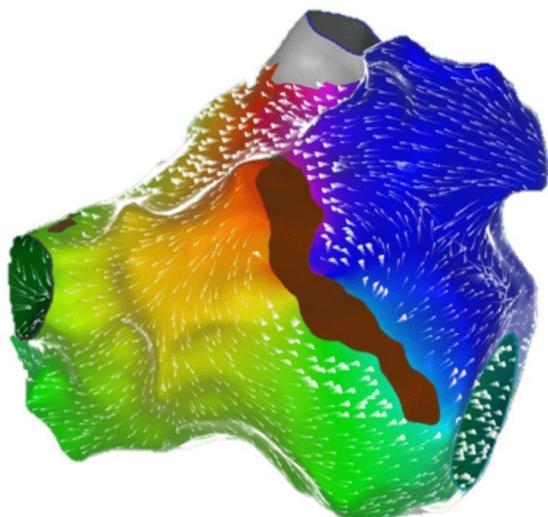


Akram Zoabi
Sr. SW Manager
Biosense Webster, J&J MedTech

Biosense Webster part of J&J Medtech

We're the global leader in delivering innovative solutions in electrophysiology.

The main goal is to ensure those with cardiac arrhythmias can live the lives they want

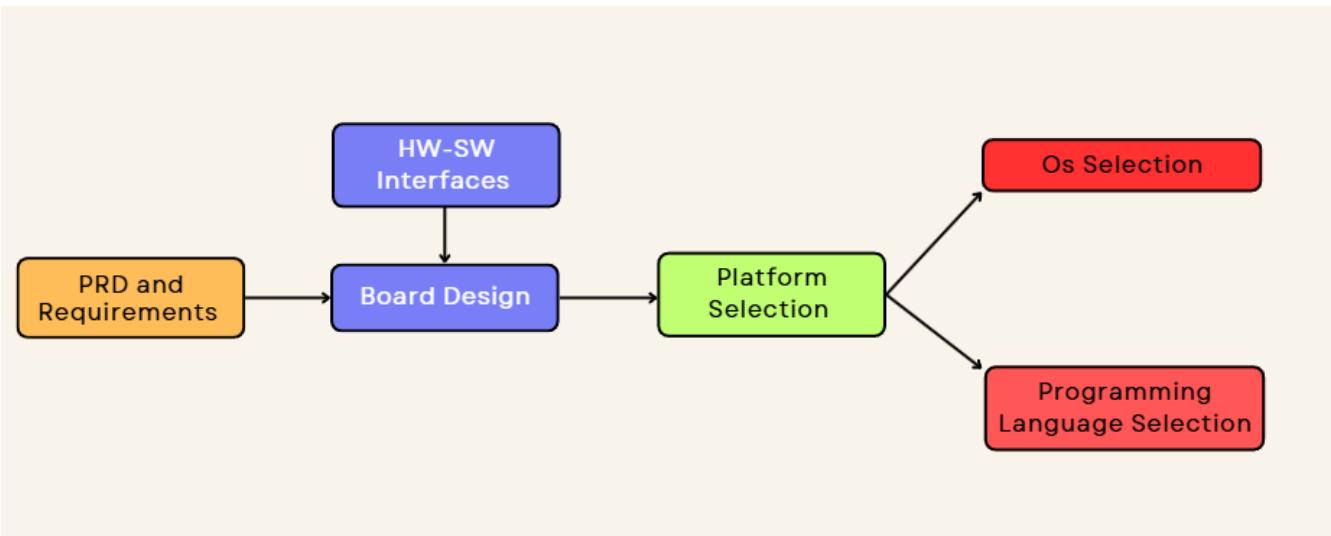


Evolution and architecture

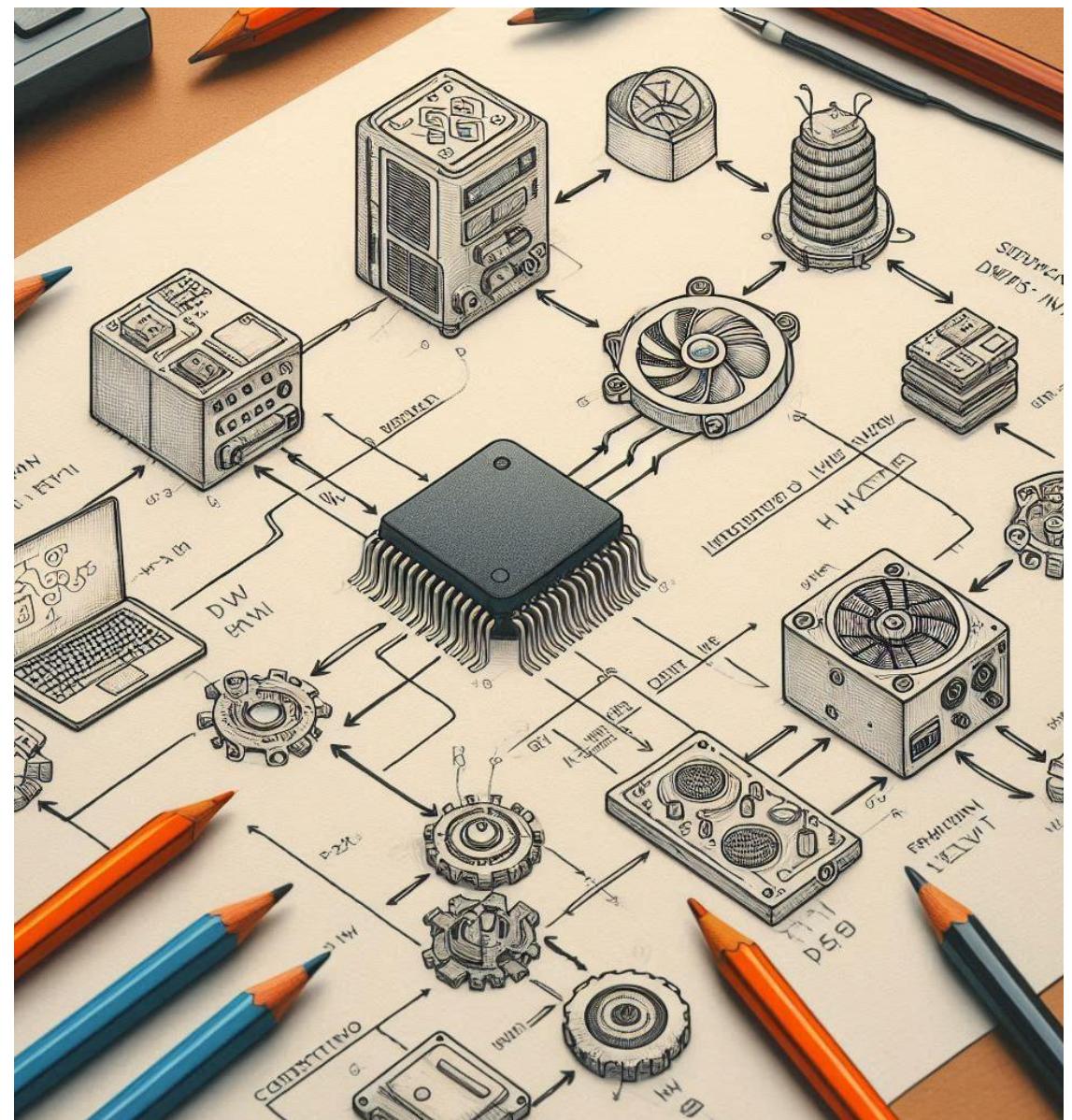


HW Platform Selection

SW requirements from HW



- Computation considerations
- Memory consumption
- FPGA ? Is needed? Integrated? Standalone?
- Communication Protocols
- Storage requirements
- Debugging capabilities



Operating System Selection

Considerations in Choosing Operating Systems

- Licensing
- HW interfaces
- Standard Communications
- Scalability/Utilization
- File System
- Hard Real Time Perf.
- OS primitives
- Community and support
- Memory Requirements
- Footprint
- Build configuration



Operating systems to consider:



- No operating system
- The best performance optimization, power, memory
- Limited API package supported by the vendor
- Management of peripherals , scheduling, interrupts..
- distributed under the MIT License
- Small Kernel – very small footprint
- Basic API for tasks, synchronization
- Support more than 40 CPUs/MCUs
- One of the most popular platforms
- Open-source license agreement
- Different custom distributions
- Flexibility and rich development application

Criteria Benchmark

Aspect	Linux	BareMetal	freeRTOS
HW interfaces			
Communications			
Cores Utilization			
File System			
Hard Real Time Perf.			
OS primitives			
Memory Requirements			
Footprint			
Build Configuration			

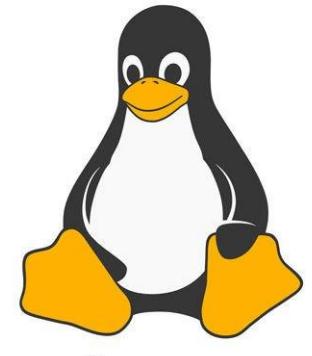
Choose The Right OS – Case 1

- Soft Real Time requirements
- TCP/IP communication
- Low scalability and utilization
- No Filesystem
- Footprint is not an issue
- Limited peripherals
- Multiple tasks and threads
- Integrated CPU and FPGA for data sampling and filtering (SOM with ARM)



Choose The Right OS – Case 2

- **Soft Real Time requirements**
- **TCP/IP communication**
- **High scalability and utilization**
- **Filesystem needed**
- **Complex logic application**
- **Footprint is not an issue**
- **Multiple tasks and threads**
- **Standalone CPU and FPGA for data sampling and filtering**



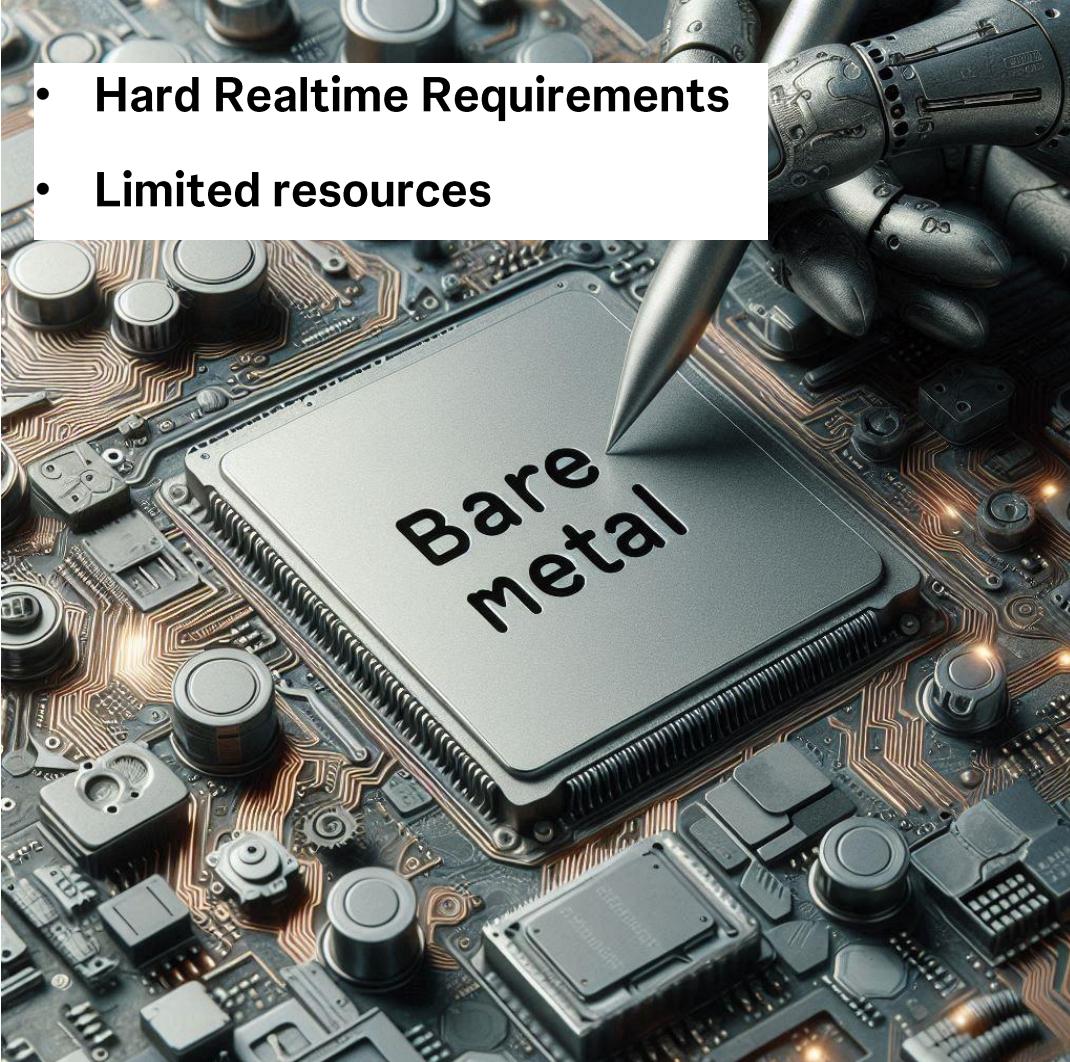
LinuxTM

Choose The Right OS – Case 3

- Hard Real Time requirements (Highly Regulated)
- Serial communication with embedded device
- File System NOT needed
- Simple logic application
- Footprint is an issue
- Small Microcontroller



Make your decision



- Hard Realtime Requirements
- Limited resources

Soft Realtime Requirements

Aspect	Linux	freeRTOS
Cores Utilization	😊	😊
File System	😊	
Build Configuration		😊
Footprint		😊
Device Drivers		😊
Modern C++ Libs	😊	
Off-the-shelf apps	😊	

Programming Language Selection

Language selection

- Development effort
- Maintainability
- Complexity and abstraction
- Ecosystem
- Safety and security
- Portability
- Scalability



The “menu”

- A lot of embedded SW engineers have a strong C background
- Very slim (and therefore powerful) language
- “Fear” of C++ - performance, footprint, etc.



- Abstraction vs. explicitness – just at the right level
- Great ecosystem
- Performance aspect in latest standards
- All the advantages of OOP

New kids on the block

- Rust
 - Limited commercial support
 - Steep learning curve
 - Interoperability with existing codebases
 - Lack of standardization
 - Limited pool of experienced engineers
- Carbon, Zig
 - Not production-ready
 - Uncertain future



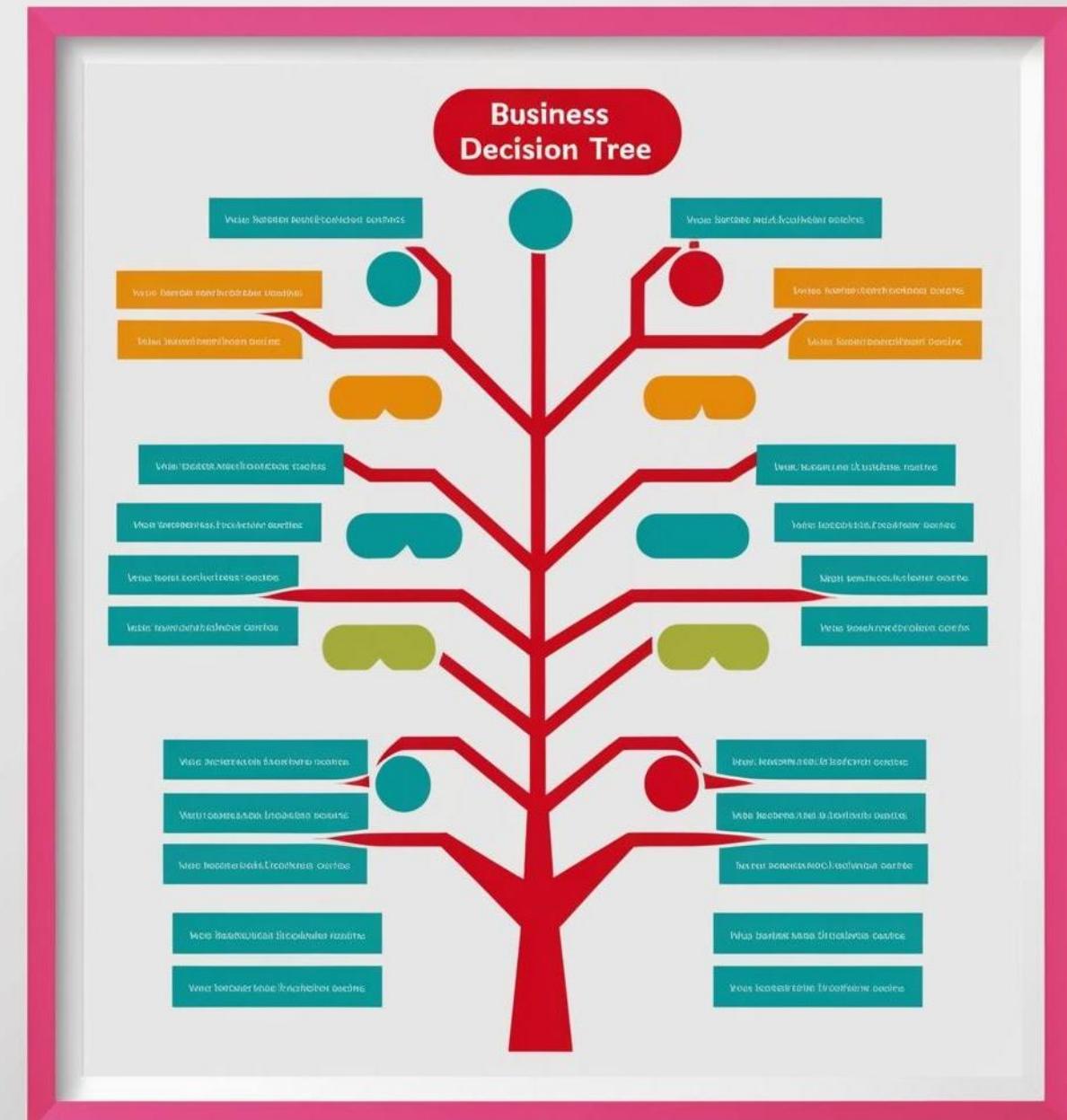
The fear of C++

- Unexpected heap allocations
- “Not invented here” – ready building blocks I don’t trust
- Possible performance issues – why vector when I can use old good C-array?
- Virtual functions overhead
- Debugging **TEMPLATES!**



Factors to consider

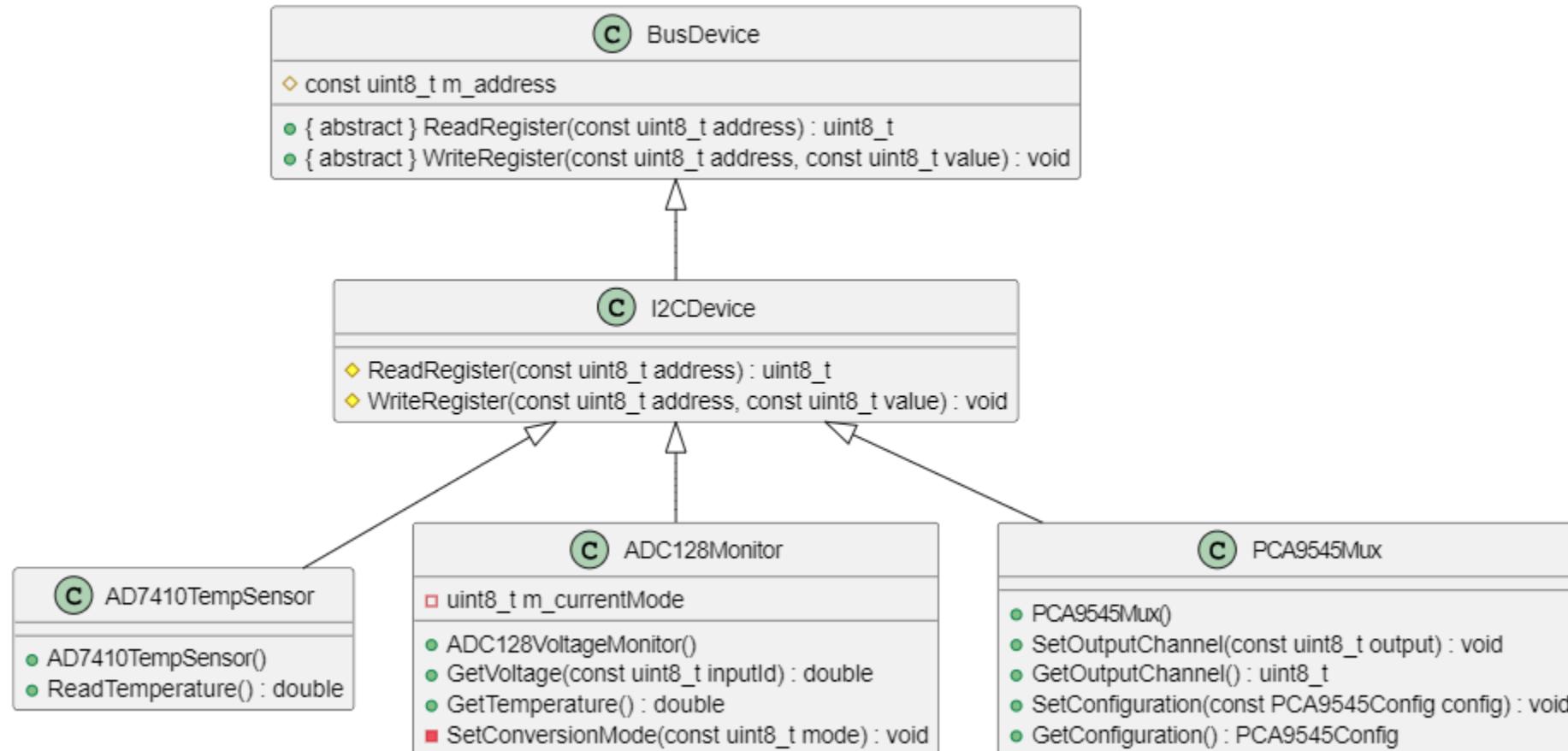
Aspect	C	C++
Memory Footprint	Smaller	Larger
Performance Predictability	Higher	Can be less predictable
Language Complexity	Lower	Higher
Hardware Control	More direct	Abstracted
Code Reusability	Limited	Extensive
Object-Oriented Features	Very limited	Comprehensive



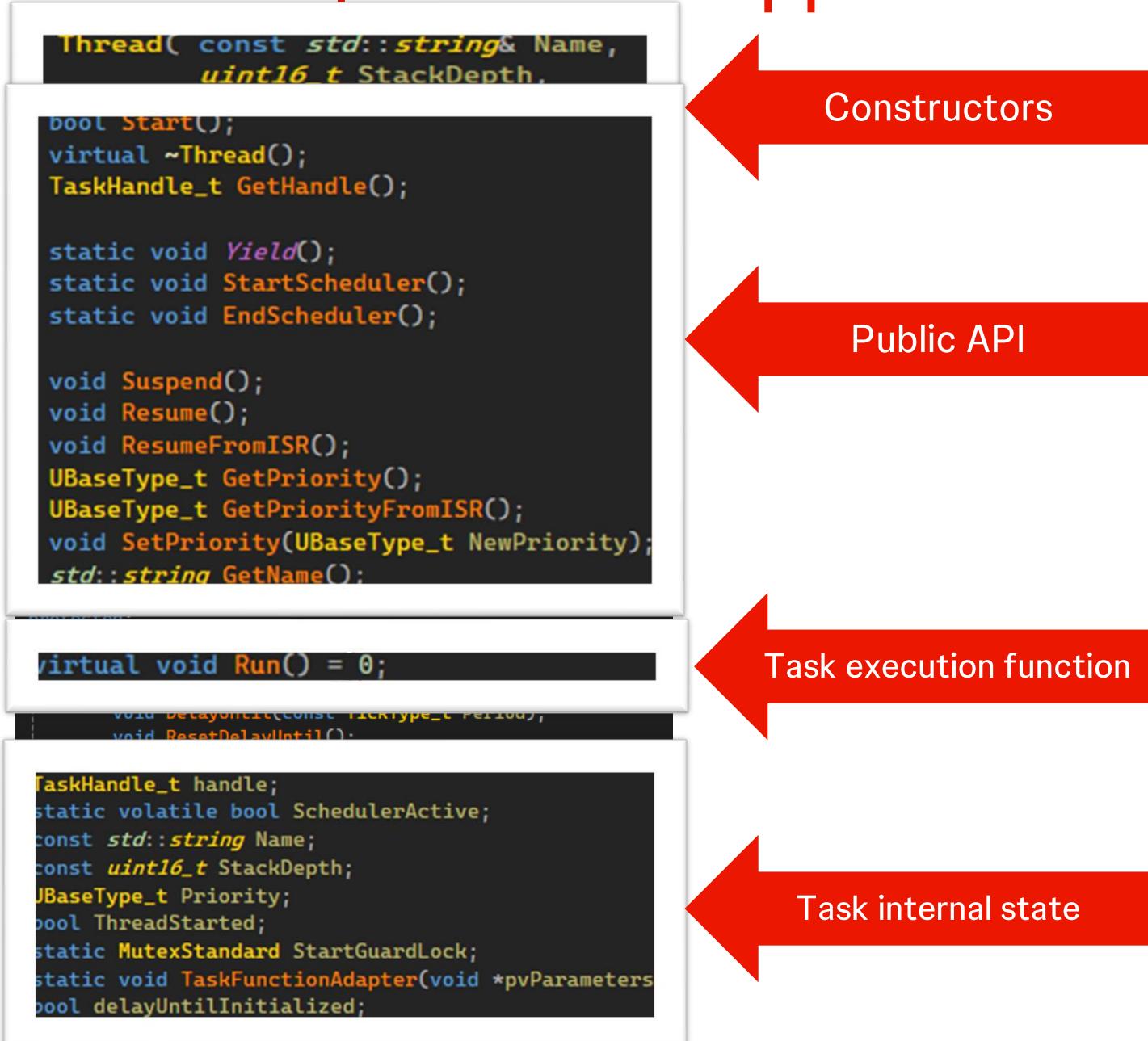
And the winner is...



Example 1 – I2C drivers abstraction



Example 2 - Thread portable wrapper



The phantom leak: a C++ horror story

- Goal: manage a queues of std::shared_ptr<T> using freeRTOS queues
- Mysterious memory leaks start haunting our system
- Plot twist: freeRTOS queue uses memcpy for enqueueing
- Hero of the hour: std::queue swoops in to save the day

Lesson learned: Even the smartest pointers can't outsmart a mismatched API!

Testing

The testability challenge

How to test?

Sometimes the embedded machine has no CLI/UI, and even can run only 1 image

What to test?

Do we have to test hardware? Or device drivers? 3rd party code?

Testing level

Unit testing? Integration testing? When we know that it is enough?

Portability

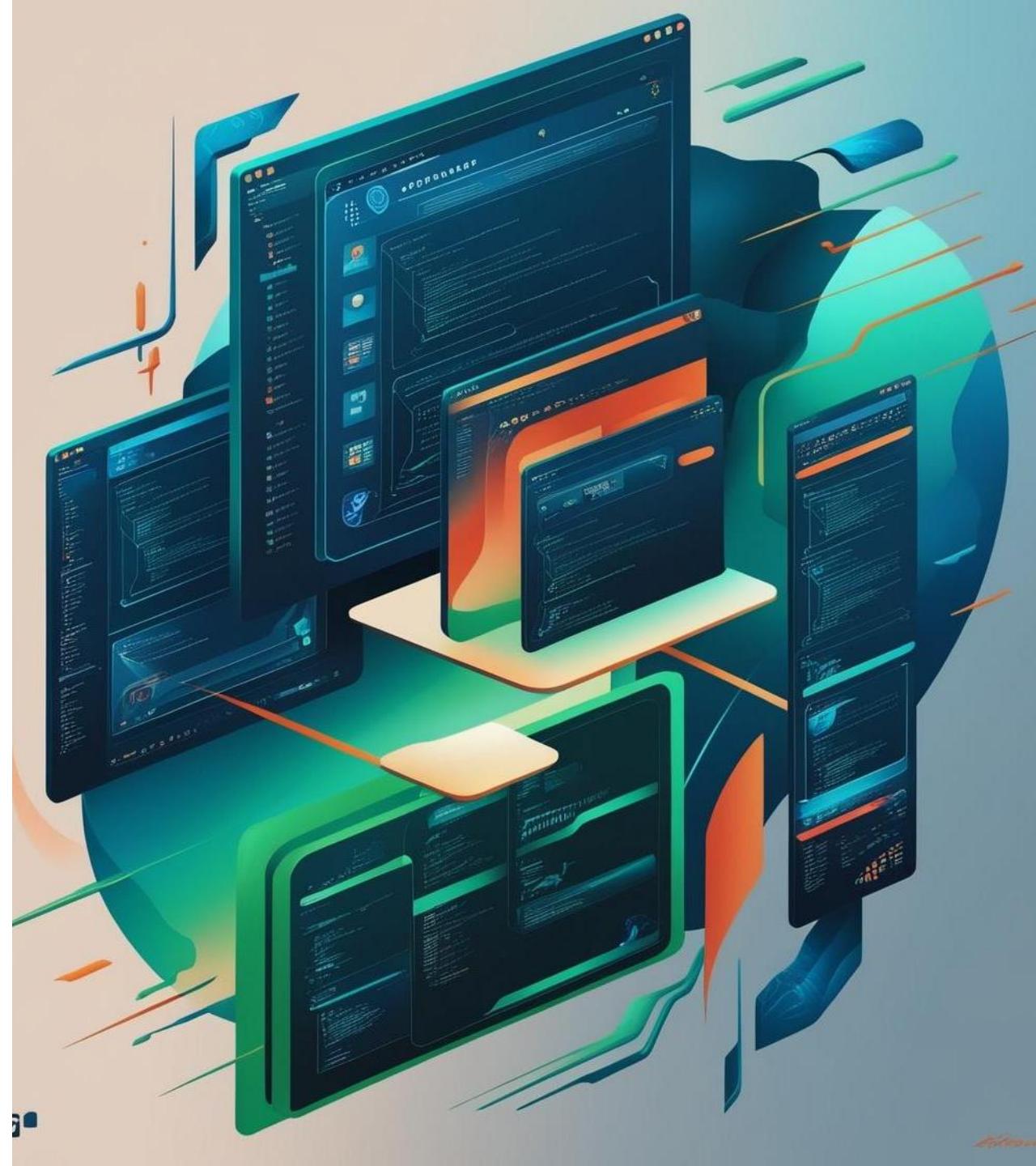
What if we decide to choose different platform? Should we rewrite our unit tests also?

Our solution

- Mock all HW/platform dependent behavior
- Compile the application logic on the development platform (Windows/Linux)
- Run the tests on your development machine
- System automated testing

Mocking HW-dependent SW

- Replacing platform- or HW-dependent SW with “mock”
- Allows to isolate logic
- Run tests on another platform
- Injection, compile time
- Googlemock, fff, Typemock



Example of driver injection

```
TxProcessor::TxProcessor(const Json & _config, LoggerPtr _log)
    : m_logger(std::move(_log))
    , m_configurator{ std::make_unique<AclTxConfigurator>(
        CreateDriver(json::Get<string>(_config, "dev-name"),
                     json::Get<string>(_config, "mock-type")) ) }
    , m_electrodeMap(CreateElectrodeMap(_config))
    , m_frequencyMap(CreateFrequencyMap(_config))
{}
```

```
ICharDriverPtr CreateDriver(std::string _driverPath, const std::string& _mockType)
{
    if (_mockType == "acltx")
    {
        return std::make_shared<AclTxMockupDriver>(_driverPath);
    }
    else // not a mock - the real driver
    {
        return std::make_shared<HWDriver>(_driverPath);
    }
}
```

```
{
  "modules": [
    {
      "id": 0,
      "name" : "AclTx",
      "dev-name" : "/dev/acltx"
    }
  ]
}
```

Unit tests vs. system tests

Unit tests

- Smallest possible unit is tested
- Focus on return value/exceptions
- OS and HW specifics are mocked
- Fast to run
- Easier to write
- Easily integrated into CI

System tests

- The system is tested as black box
- Focus on a specific scenario
- Real components
- Runs may take longer – depending on the scenario
- System-wide level – complex scenarios can be tested

Unit tests vs. system tests

Unit tests

```
TEST_F(MessageRouterTest, HandleRegisteredMessage)
{
    MessageRouter mr;
    mr.RegisterCommandHandler(MessageID::MEB_UNKNOWN, &m_handler);

    IncomingMessage message;
    message.m_id = MessageID::MEB_UNKNOWN;

    mr.HandleInboundMessage(message);

    ASSERT_EQ(m_handler.GetHandledCounter(), 1);
}

// The test does not register a handler
// Expected result - no handler is invoked
TEST_F(MessageRouterTest, HandleUnregisteredMessage)
{
    MessageRouter mr;

    IncomingMessage message;
    mr.HandleInboundMessage(message);

    ASSERT_EQ(m_handler.GetHandledCounter(), 0);
}
```

System tests

```
TEST_F(PacingModuleTest, ConfigValidInMaintenance)
{
    TCPServer srv{ [this](boost::asio::ip::tcp::socket& _sock) {
        SetSystemState(_sock, statemachine::SystemState::Maintenance);

        auto receivedChannels = SetAndGetStimRoute(_sock, VALID_CHANNELS_2);
        ASSERT_NE(receivedChannels, VALID_CHANNELS_2);

        SetSystemState(_sock, statemachine::SystemState::Operative);
    } };
}

TEST_F(PacingModuleTest, DefaultRoutingOperative)
{
    TCPServer srv{ [this](boost::asio::ip::tcp::socket& _sock) {
        SetSystemState(_sock, statemachine::SystemState::Operative);

        auto receivedChannels = SetAndGetStimRoute(_sock, VALID_CHANNELS);
        ASSERT_EQ(receivedChannels, VALID_CHANNELS);

        receivedChannels = SetAndGetStimRoute(_sock, DEFAULT_LOGICAL_CHANNELS);
        ASSERT_EQ(receivedChannels, DEFAULT_INVALID_CHANNELS);

        receivedChannels = SetAndGetStimRoute(_sock, VALID_CHANNELS_2);
        ASSERT_EQ(receivedChannels, VALID_CHANNELS_2);
    } };
}
```

The benefit – case study

- **Time to market**
 - The product was released ahead of time
 - Reusable infrastructure and OOP boosted development time
- **Quality**
 - No regression was introduced during development cycles
 - Managed to simulate many scenarios without available hardware
 - Reduced number of bugs from field



Thank you