accu 2023

TEST-DRIVEN DEVELOPMENT OF EMBEDDED AND SYSTEM-LEVEL SOFTWARE

VLADIMIR VISHNEVSKII

Motivation

- Further elaboration of the “testing” topic touched in the presentation *“C++ in the world of embedded systems”* (ACCU 2022, CppCon 2022)
- Personal believe that TDD doesn’t have alternatives for production software development

Agenda

- Scope of the talk
- Test-driven development process
- Unit testing of the embedded and system dependent code

Embedded development characteristic

- Software is developed on a host platform and deployed on a target platform using specialized tooling (cross platform development)
- Diagnostics and debugging can be limited (software/hardware debuggers, disabled debugging, limited physical accessibility)
- Software can be dependent on hardware (direct access or specific platform configuration) or system APIs

Embedded development characteristic

- Some applications are resource constrained (operational memory, storage for executable and computational power)
- Some types of software should be designed for runtime efficiency (e.g. device drivers, real-time components)

System code development characteristic

- Software can be dependent on system APIs (e.g. Posix, Win API or platform dependent higher level frameworks)
- Software can require particular target system configuration and environment (hardware, file system (FS), network e.t.c) not available on a host system
- Software can perform actions that should not be allowed on-host
- Run-time efficiency requirements can be applied

Embedded and low-level development characteristics

Characteristic	Embedded	System-level
Direct dependency on hardware	+	-+
Dependency on OS APIs, platform dependent frameworks	+	+
Deployment and testing simplicity	-	-
Run time efficiency requirement	+	+
Resource constraints	+	-

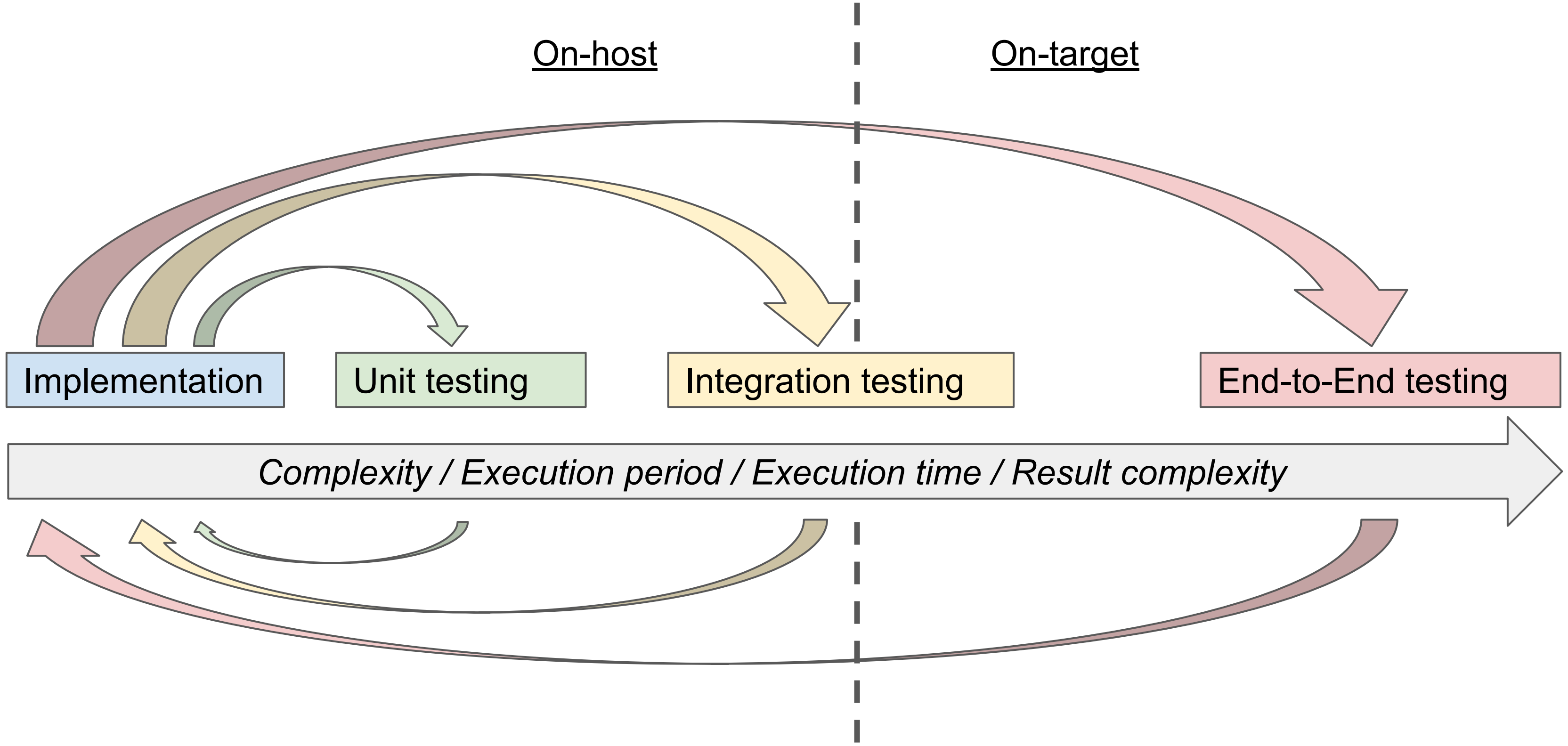
Test-driven development (TDD) process

- Is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases (*Wikipedia*)
- Testing software is implemented before/in parallel with implementation of the production software
- The process is ubiquitous. Testing should be considered during planning, architectural and implementation work
- Software is not introduced into repository without sufficient test coverage

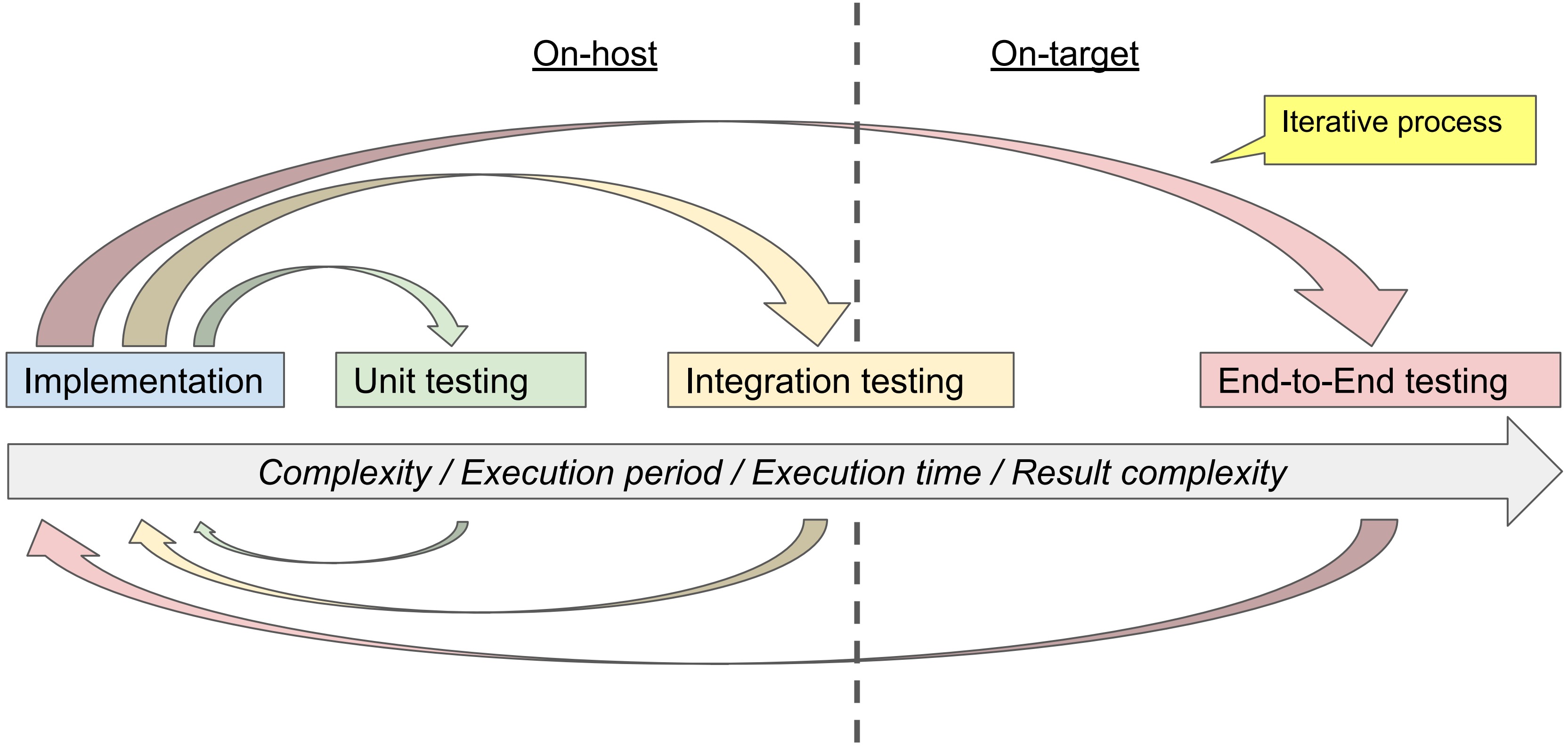
Types of tests (by object granularity)

- End-to-end tests - test of a whole solution on a target platform
- Integration tests - test interaction of several components (units), potentially on a target platform
- Unit tests - test units (classes/functions) without dependencies

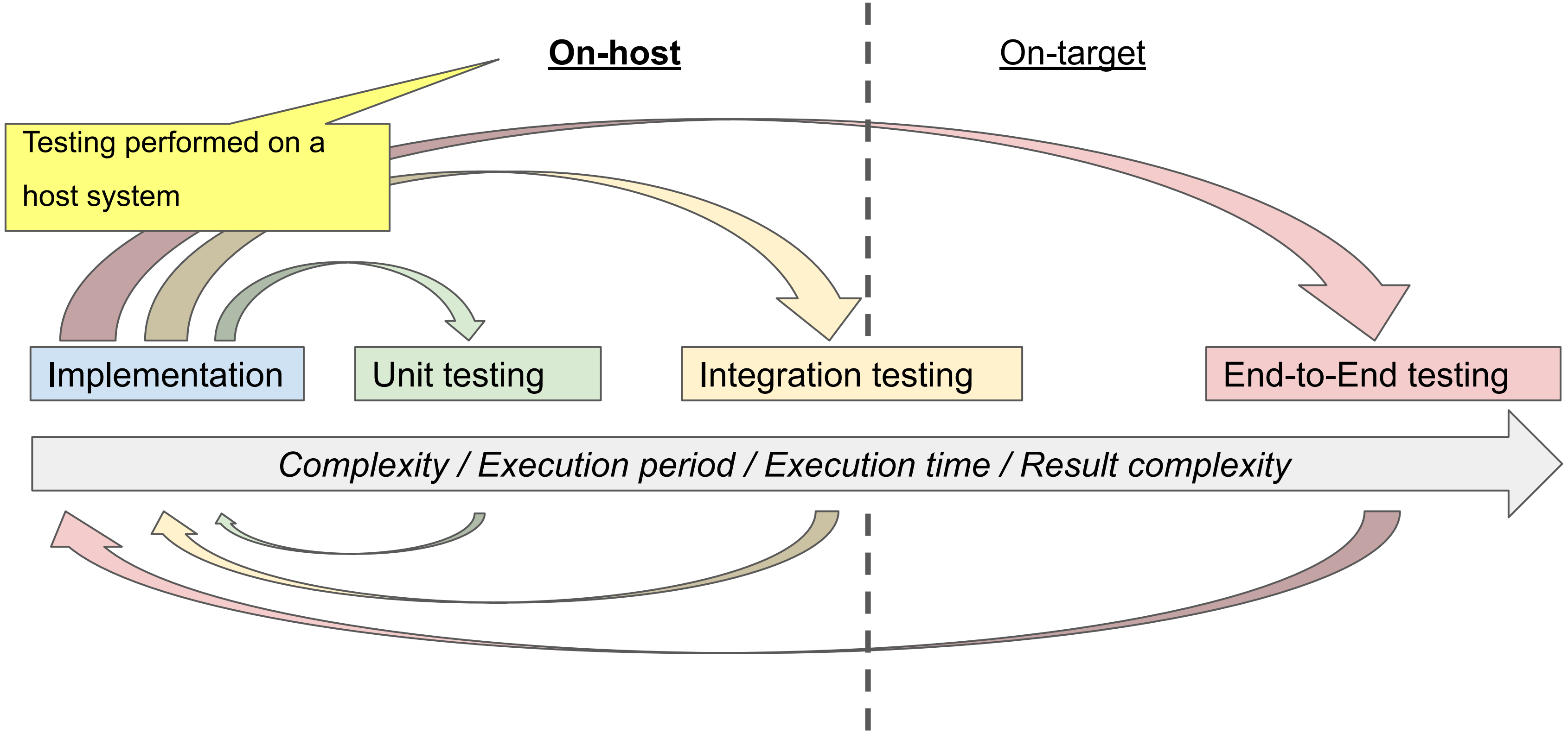
Typical TDD cycles



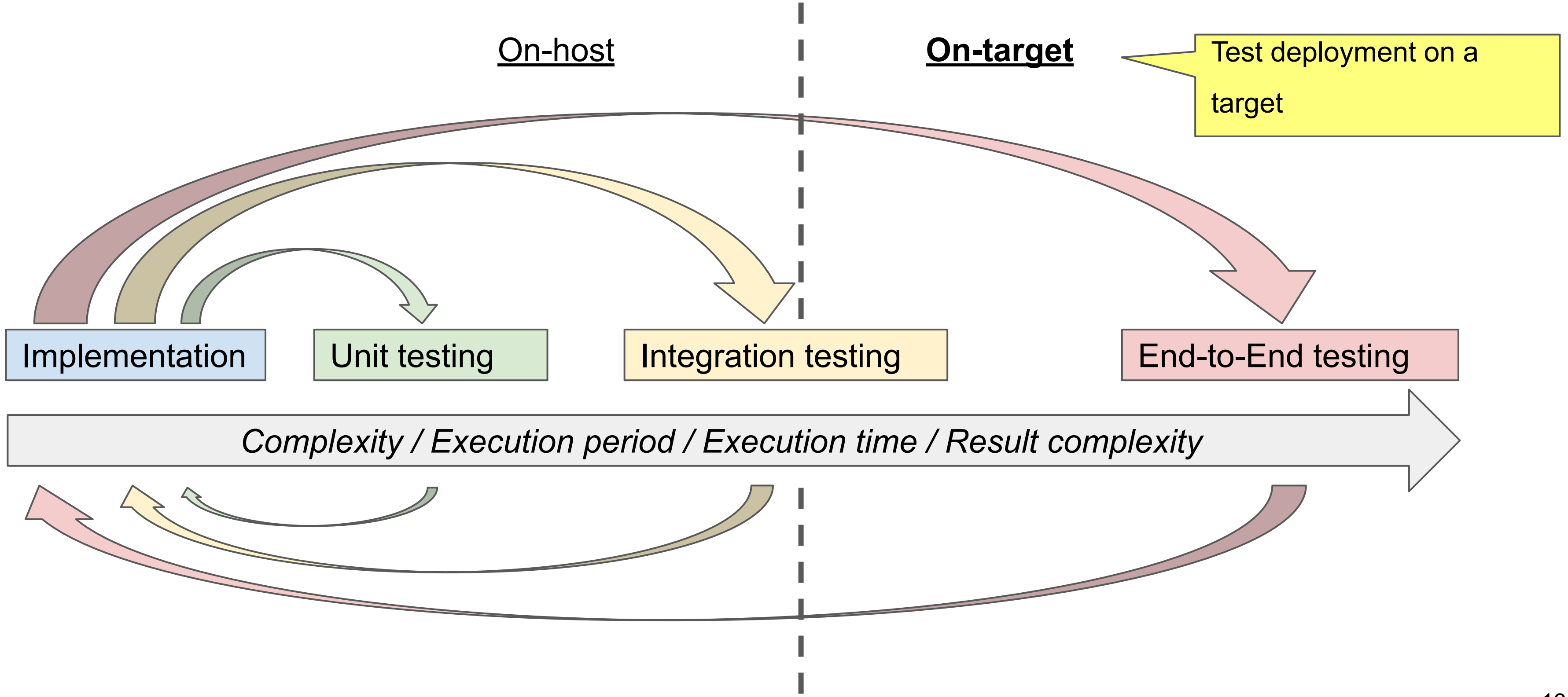
Typical TDD cycles



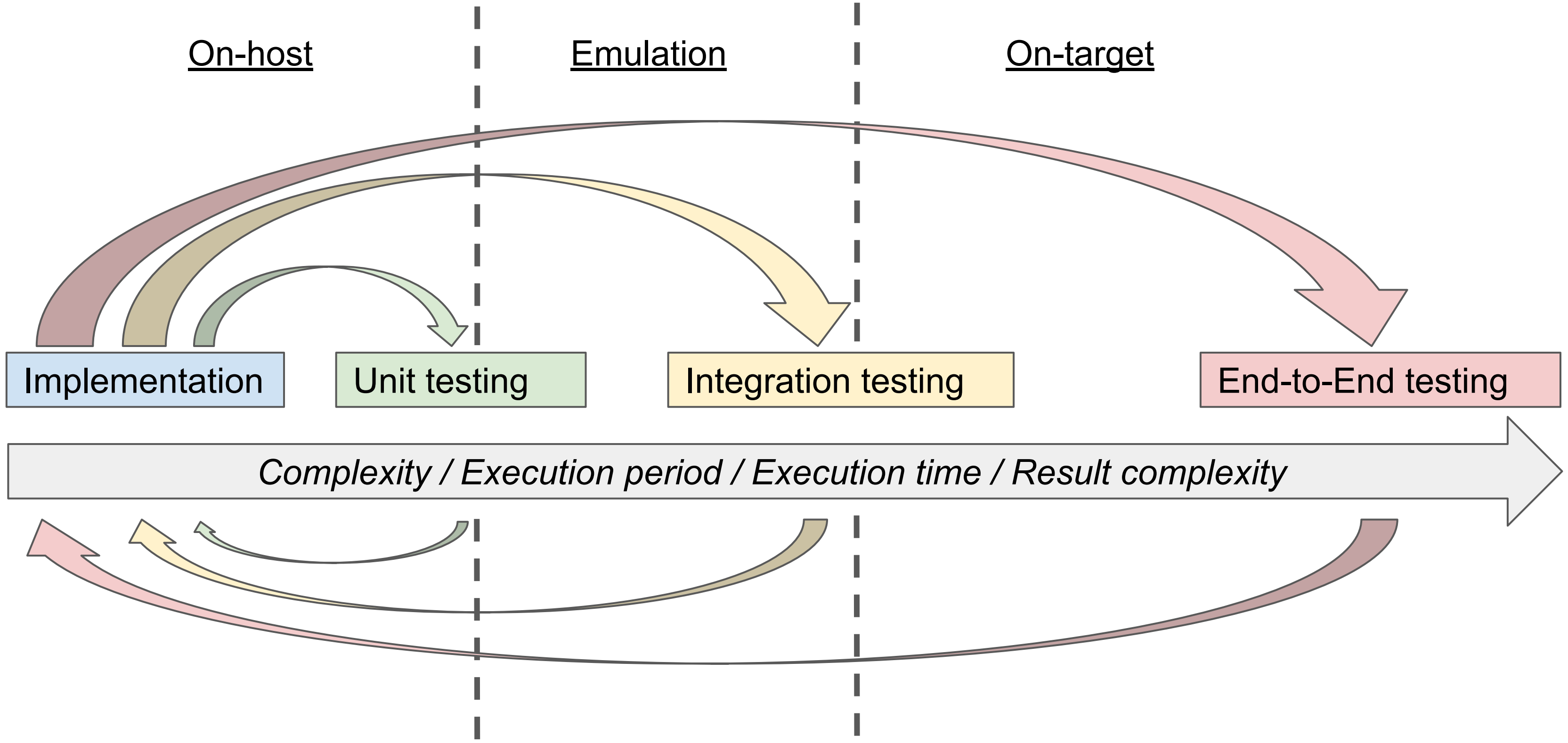
Typical TDD cycles


On-host
On-target
Testing performed on a host system
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity

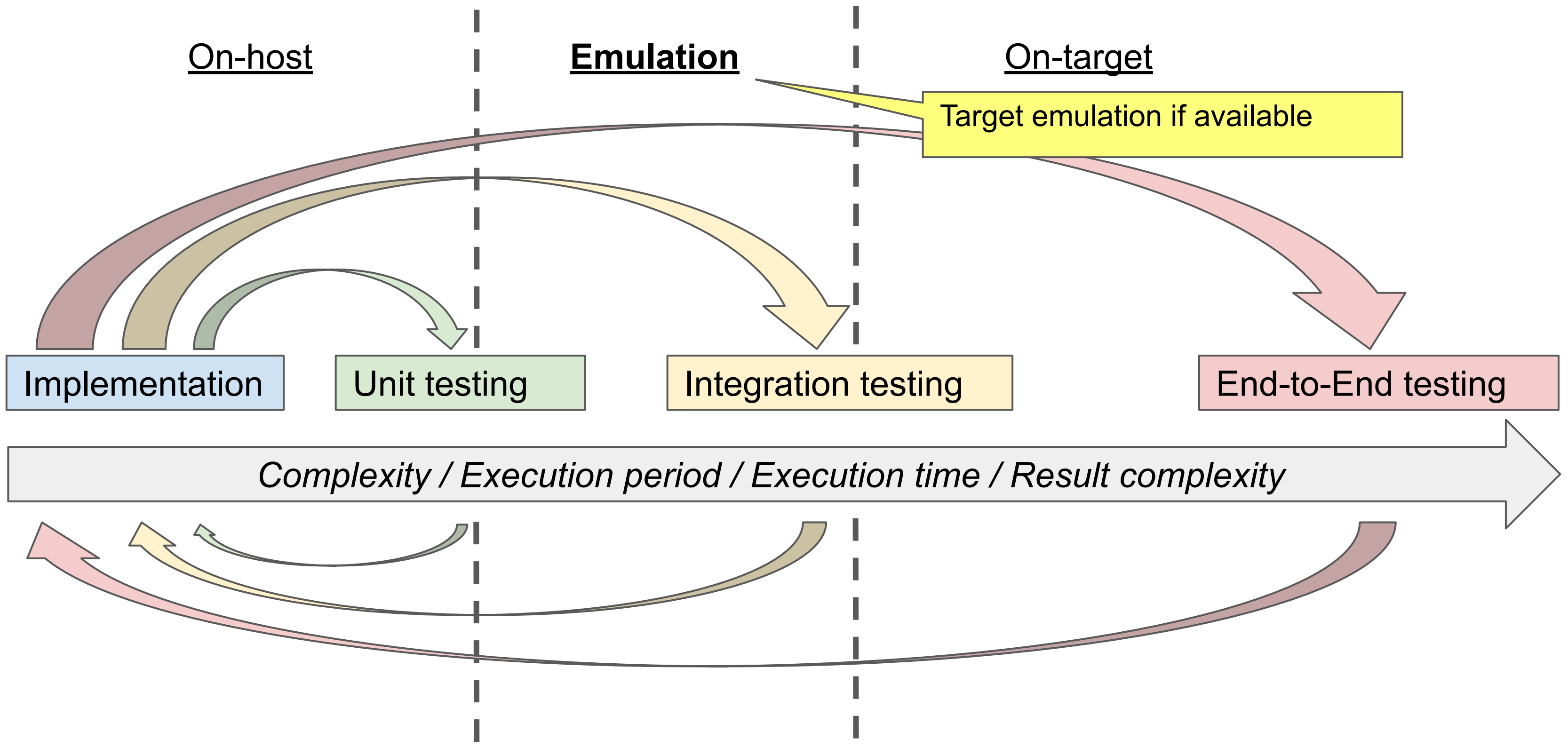
Typical TDD cycles


On-host
On-target
Test deployment on a target
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity

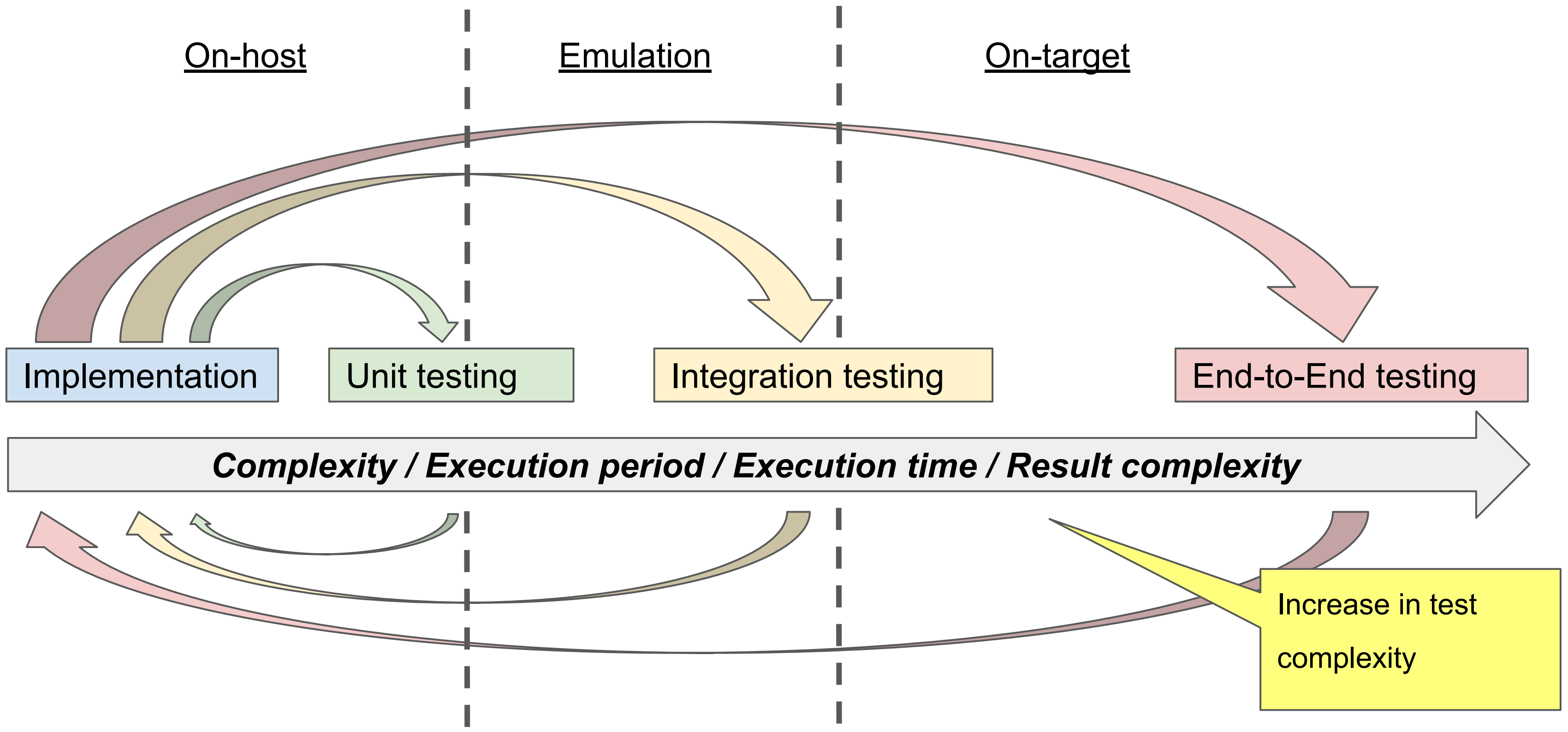
Typical TDD cycles for cross platform development



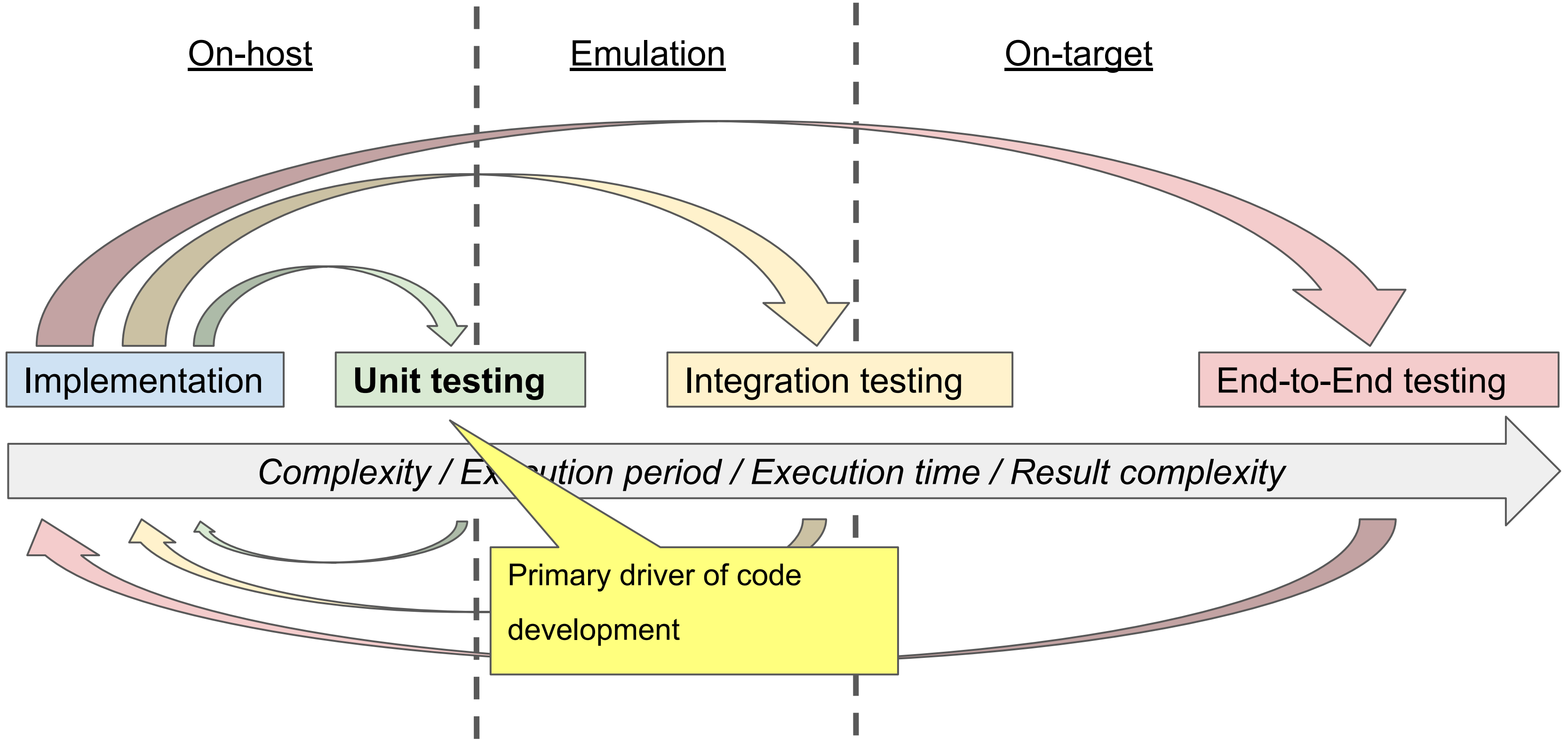
Typical TDD cycles for cross platform development


On-host
Emulation
On-target
Target emulation if available
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity

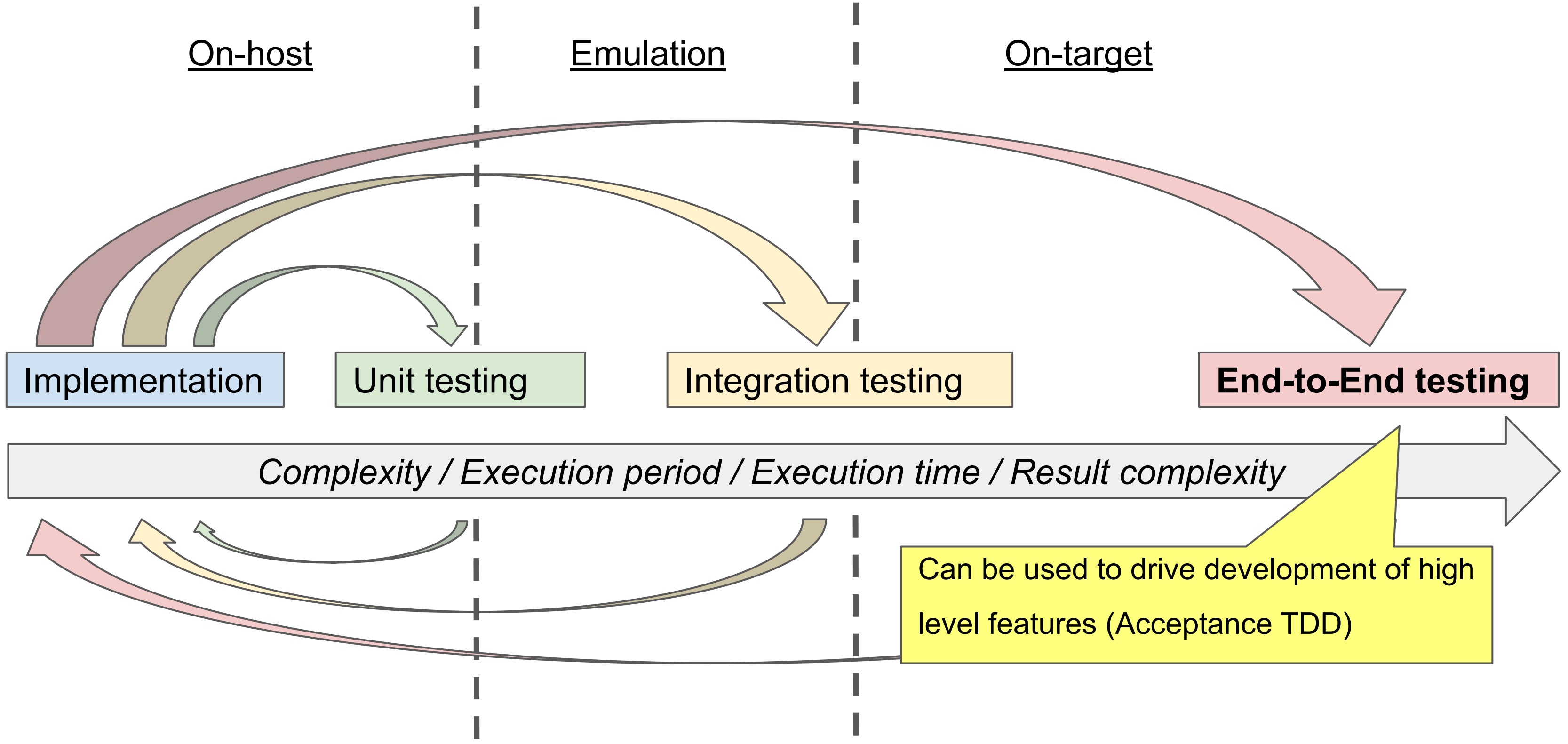
Typical TDD cycles


On-host
Emulation
On-target
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity
Increase in test complexity

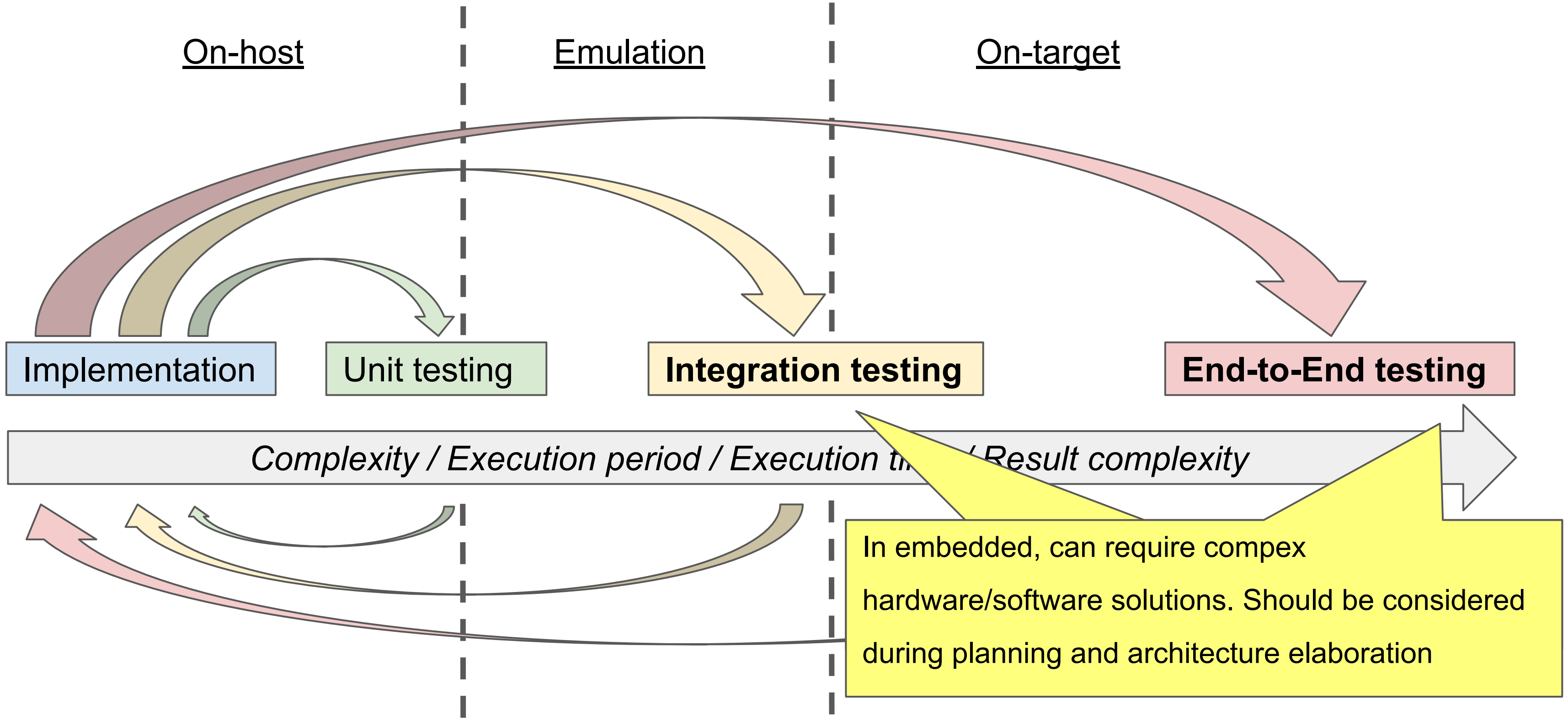
Typical TDD cycles for cross platform development


On-host
Emulation
On-target
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity
Primary driver of code development

Typical TDD cycles for cross platform development


On-host
Emulation
On-target
Implementation
Unit testing
Integration testing
End-to-End testing
Complexity / Execution period / Execution time / Result complexity
Can be used to drive development of high level features (Acceptance TDD)

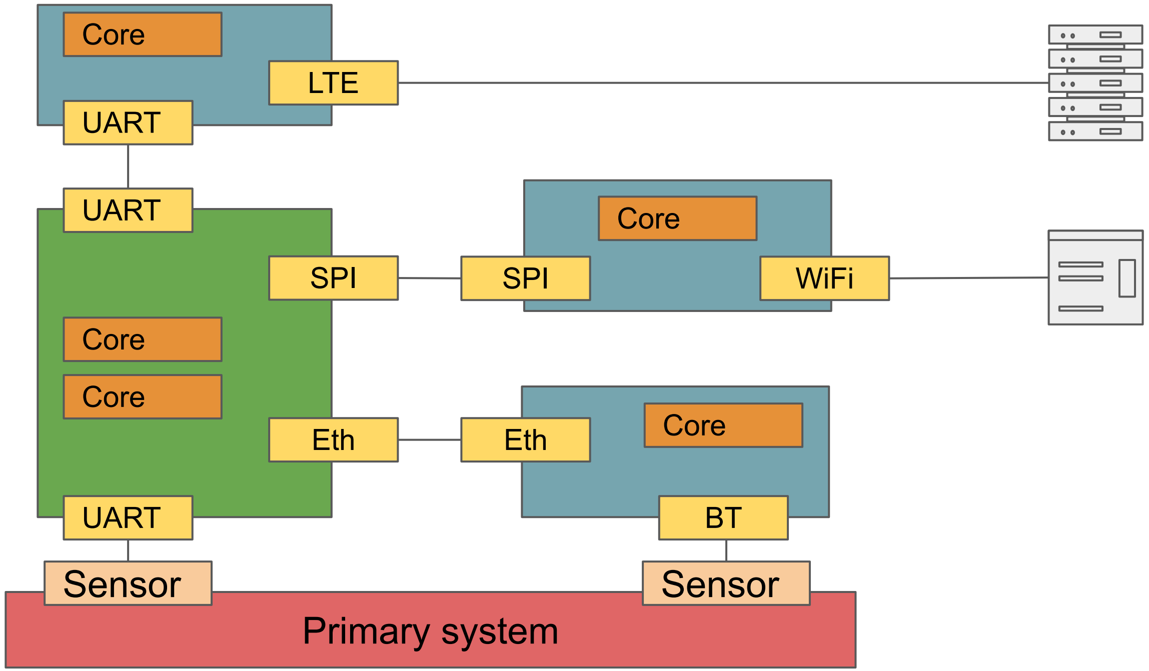
Typical TDD cycles for cross platform development



In embedded, can require compex hardware/software solutions. Should be considered during planning and architecture elaboration

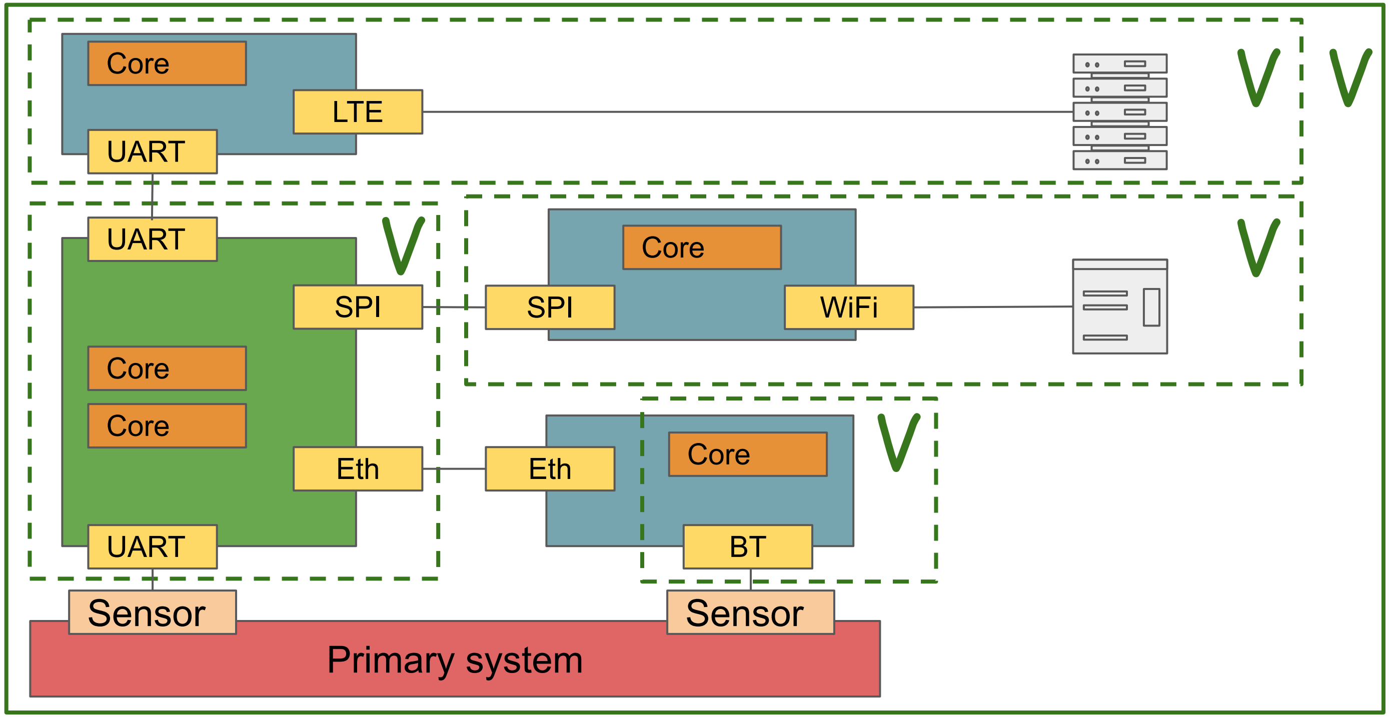
Integration and end-to-end testing development

- Complex embedded solution can consist of multiple hardware/software blocks:



Integration testing and end-to-end testing development

- Gradual sustainable development of a complex project requires automated testing strategy including integration and end-to-end testing

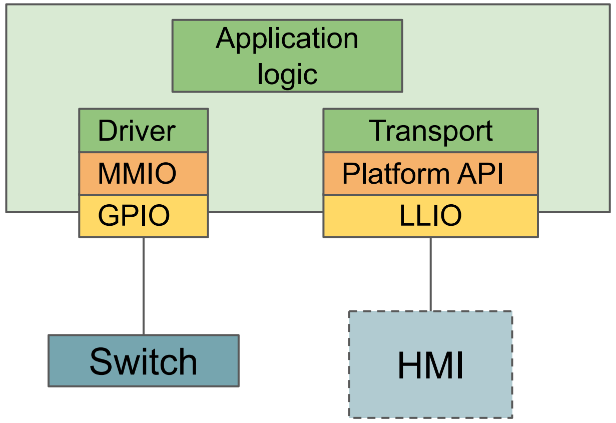


Integration and end-to-end non-functional testing

- Allows to perform reliability testing
- Enables continuous acquisition and monitoring of performance metrics

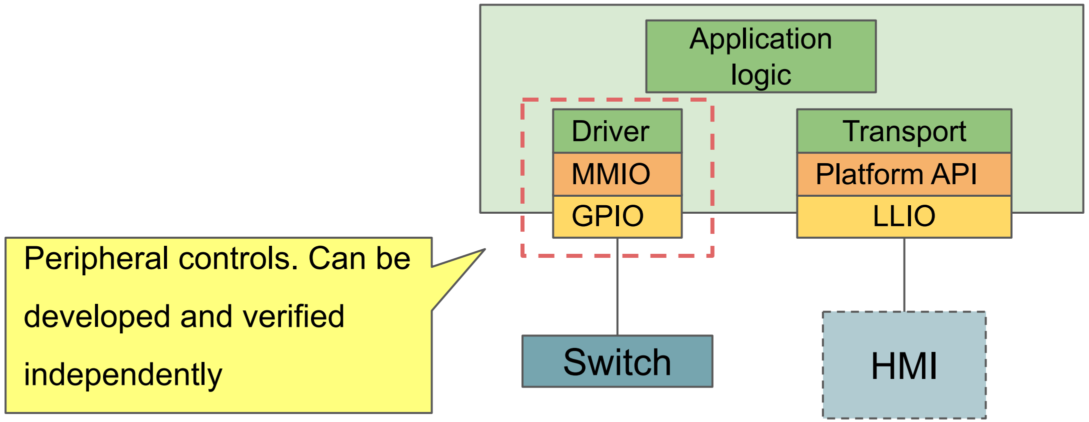
Model project

- Greenfield project supposed to visualize state of physical control elements
- Feature to implement: transfer state of a physical switch (on/off) to an HMI
- Details are not important, project used for illustration purpose only



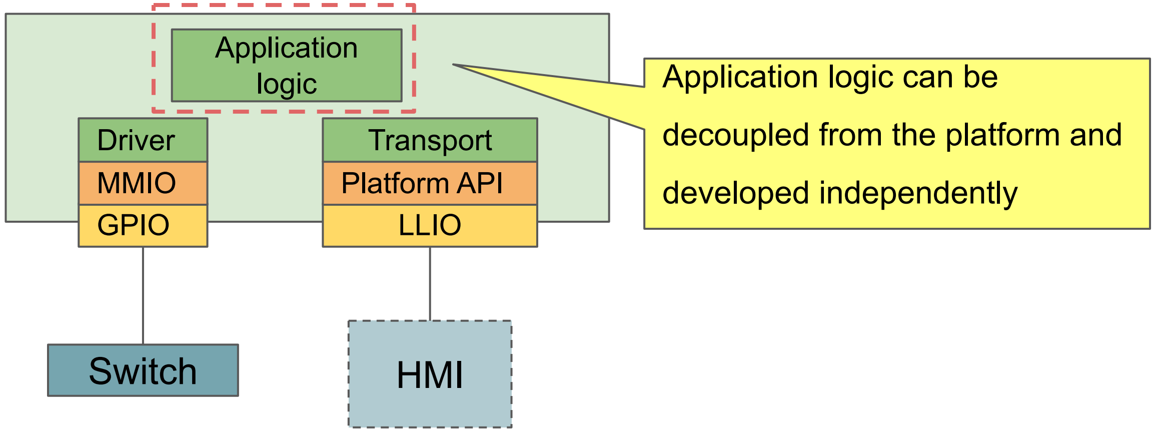
Model project

- Physical switch has on/off states
- The state is transferred via GPIO line



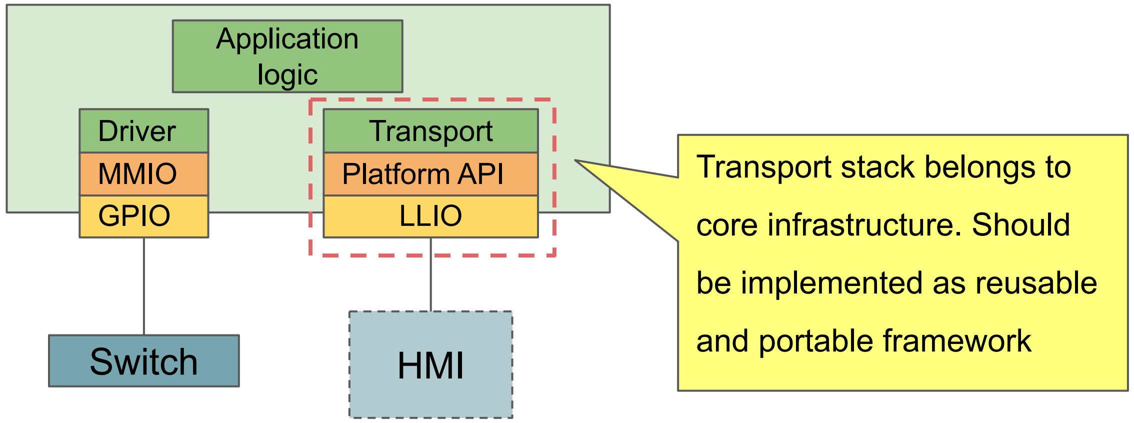
Model project

- Application logic uses driver to access switch state and forwards it to the transport stack


Application logic
Driver
MMIO
GPIO
Transport
Platform API
LLIO
Switch
HMI
Application logic can be decoupled from the platform and developed independently

Model project

- Low level physical interface (LLIO) connects board to the future HMI platform
- Platform SDK for platform offers API to communicate via LLIO
- HMI communication protocol is specified
- HMI is a source of synchronized time used by communication stack


Application logic
Driver
MMIO
GPIO
Transport
Platform API
LLIO
Switch
HMI
Transport stack belongs to core infrastructure. Should be implemented as reusable and portable framework

Project plan. Solution breakdown

- ☐ Transport stack
- ☐ Peripheral controls
- ☐ Application logic

Transport stack

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

Transport stack

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

Data to send is the only parameter

Transport stack

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

Result code as an output

On-target integration testing. Initial example

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

- Executable to perform positive test case:

```
int main(int /*argc*/, char ** /*argv*/)
{
    std::array<uint8_t, 2> const data{0x10, 0x20};
    TEST_ASSERT(transport::send_message(data) == result_t::ok);

    return 0;
}
```

On-target integration testing. Initial example

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

- Executable to perform positive test case:

```
int main(int /*argc*/, char ** /*argv*/)
{
    std::array<uint8_t, 2> const data{0x10, 0x20};
    TEST_ASSERT(transport::send_message(data) == result_t::ok);

    return 0;
}
```

Can be executed only on a target. Requires test setup. Defect analysis can be challenging.

On-target testing. Initial example

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

- Test for the error case:

```
int main(int /*argc*/, char ** /*argv*/)
{
    std::array<uint8_t, 2> const data{0x10, 0x20};
    TEST_ASSERT(transport::send_message(data) == result_t::link_layer_error);

    return 0;
}
```

On-target testing. Initial example

- Transport stack client interface:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
    link_layer_error,
};

auto send_message(data_t const &data) -> result_t;
```

- Test for the error case:

```
int main(int /*argc*/, char ** /*argv*/)
{
    std::array<uint8_t, 2> const data{0x10, 0x20};
    TEST_ASSERT(transport::send_message(data) == result_t::link_layer_error);

    return 0;
}
```

Can be caused by platform issue with rare occurrence. Handling and recovery logic verification requires long test runs

Issues with the on-target test. Summary

- Target should be available
- Test should contain enough functionality to be executed on-target
- The test is dependent on the hardware/software platform that limits testability:
    - Some conditions (errors) can not be intentionally provided
    - Test can not track intermediate state and side effects. Only preconditions and postconditions can be checked
    - Test results can be ambiguous due to interference of dependency issues

On-host verification

- To minimize on-target risks code quality should be maximized prior to deployment and on-target integration testing:
  - Off-target verification of application logic
  - Verification of the software/hardware platform contract conformance
- On-host development and verification increase productivity (shorter feedback time)
- On-host development can be driven by unit testing

Unit testing

- Unit tests verify a compliance of independent units to a contract, comparing an observable behaviour with expectations under a predefined input
- Unit testing can be a primary driver of a code development

Unit testing principles

- Unit behaviour can be verified through the “input and observation points”:

Direct input	Function arguments, setters	Can be checked by the conventional assertions. Sufficient for trivial functions and value objects
Direct output	Return values, output parameter, getters, exceptions	
Indirect (implicit) input/output	Interaction with other entities the unit is dependant on	Can not be directly asserted as the interaction is hidden

- Mock objects are used to satisfy unit dependencies and verify indirect communication
- Mocks are test double objects that exhibit predefined behaviour according to test scenario, verify and track interaction with unit under test

Advantages of unit testing

- Isolation from the dependencies
- Localisation of the error
- Logic flows for infrequent/sporadic real-life events can be tested
- Short feedback time, simplicity of development and target independence enhance developer productivity
- Independent progress of component development

Unit testing enablement

- To enable and benefit from unit testing some principles of *design for testability* should be applied
- Most of those principles belong to “best design practices”
- Testability is well connected with reusability and portability as it requires decoupling

Design for testability

- Mock testing is fundamentally based on decoupling and dependency injection principles:
    - Unit should be dependent on abstractions instead of concrete types
    - Real or mock implementations of dependencies are injected
- Composition prevents injection, decomposition is crucial
- Separation of concerns allows to define proper unit boundaries and manage dependencies

Model project. Transport stack decomposition

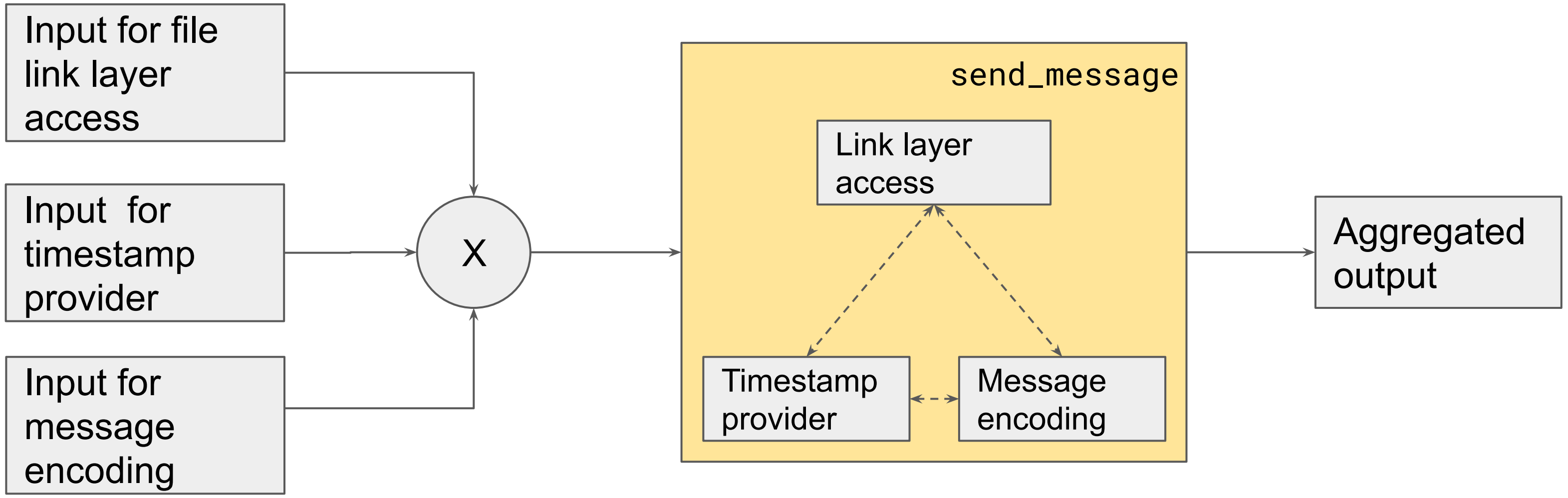
- Primary functional blocks that constitute transport stack:
  - Message encoding
  - Protocol specific timestamp evaluation
  - Link layer access
  - Transmission logic

Project plan. Solution breakdown

- ☐ Transport stack
    - ☐ Message encoding
    - ☐ Timestamp evaluation
    - ☐ Link layer access
    - ☐ Transmission functionality
- ☐ Peripheral controls
- ☐ Application logic

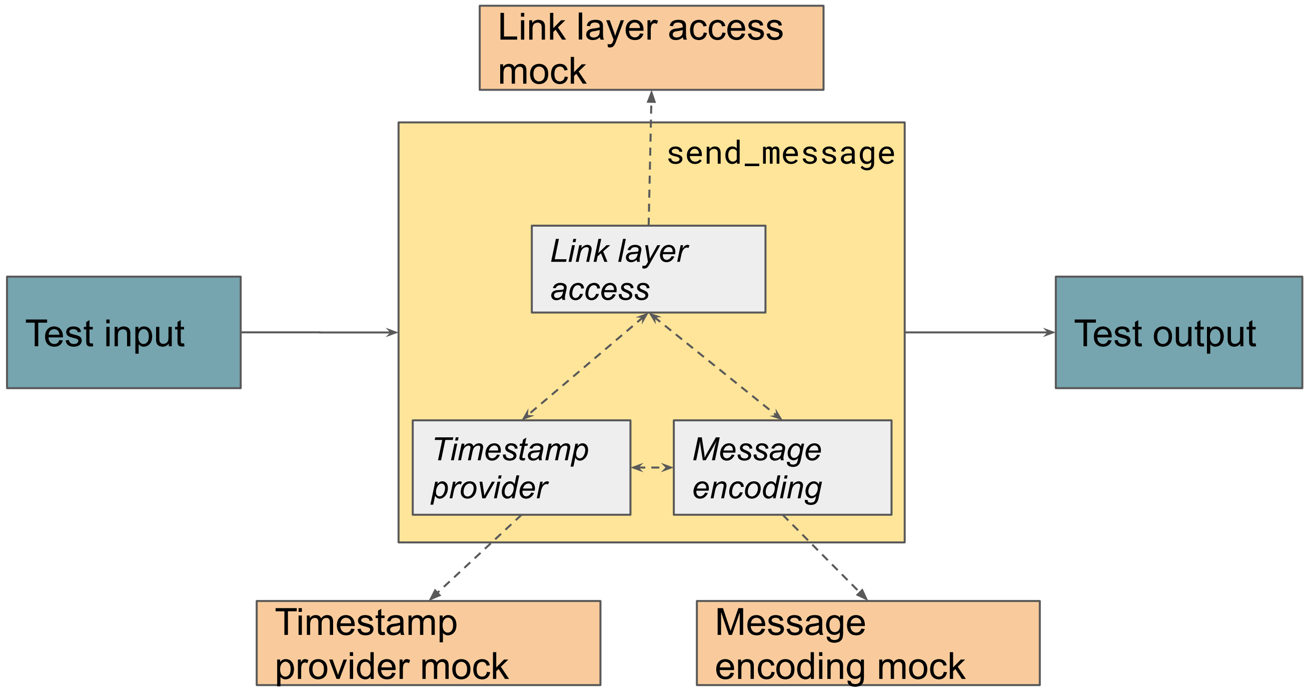
Decomposition

- Test for complex classes or functions requires complex synthesized input
- Output might be not indicative of the issues in the particular function


Input for file link layer access
Input for timestamp provider
Input for message encoding
X
send_message
Link layer access
Timestamp provider
Message encoding
Aggregated output

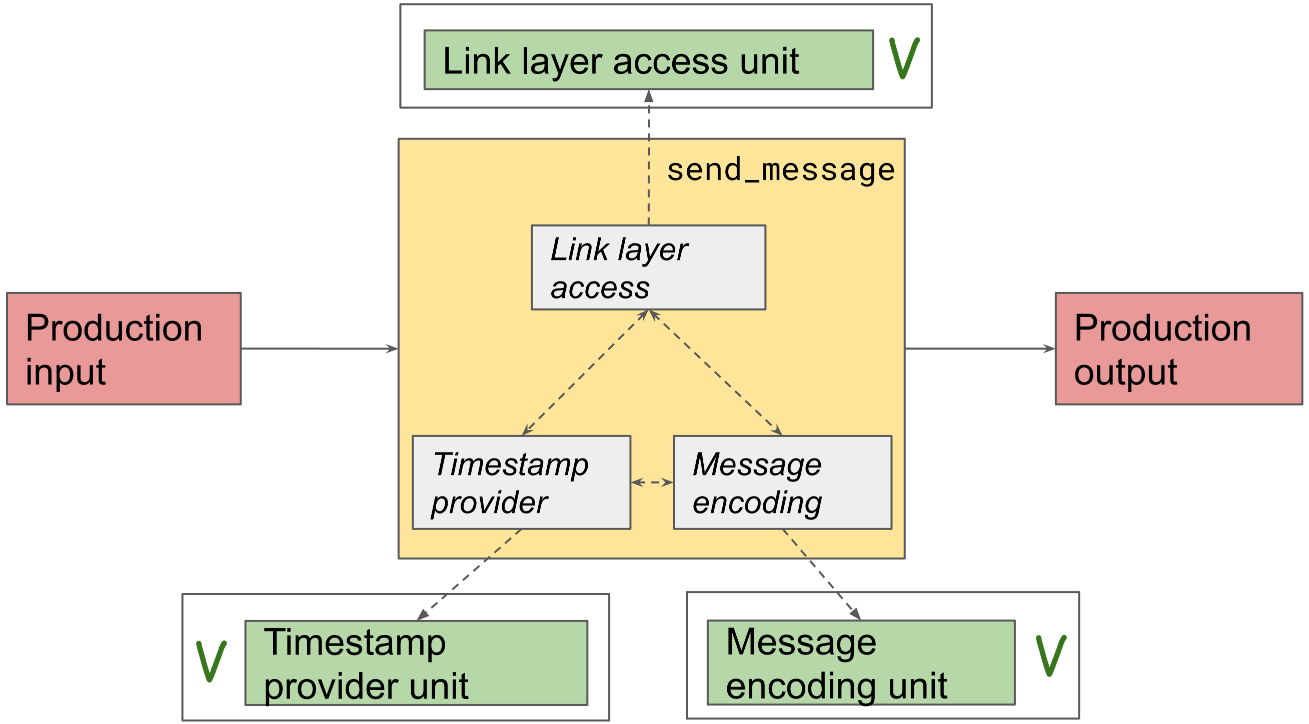
Decomposition

- The unit logic should be dependent on abstractions instead of concrete types
- The mock objects can be injected for testing



Decomposition

- In production, real independently-verified dependencies should be injected



Decomposition

- The following questions can be used to identify potential elements that can be isolated as part of the decomposition process:
    - What is the unit primary responsibility?
    - Are there any hardware/platform interactions?
    - Is there any data transformation involved?
    - Are there any unreproducible internal operations (timestamps, random number generation, e.t.c)
    - Are there any timeouts involved?

Decomposition

- Recurring pattern: switching from composition to aggregation and ownership to sharing
- Should be applied iteratively if needed

Transport stack. Message encoding

- Prepares message according to the protocol specification:

Field	Size (bits)	Description
Magic	4	Fixed value 1010 (binary)
Type	4	1 for data messages
Timestamp	32	Timestamp in network order
Payload size	8	Size of attached payload
Payload		Arbitrary data with length specified in “Payload size” field

- Doesn’t have any dependencies on platform

Code examples

- Use C++20 standard but can be downgraded
- nonstd::expected is used for result types
- Unit tests are based on Google Test/Mock framework

Transport stack. Message encoding

- Function to calculate size of a message to send data of some size:

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

inline auto calculate_message_size(
    size_t const data_size
) noexcept -> nonstd::expected<size_t, error_t> {...}
```

Transport stack. Message encoding

- Function to calculate size of a message to send data of some size:

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

inline auto calculate_message_size(
    size_t const data_size
) noexcept -> nonstd::expected<size_t, error_t> {...}
```

Size of data to send

Transport stack. Message encoding

- Test for error condition (data is too large)

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

inline auto calculate_message_size(
    size_t const data_size
) noexcept -> nonstd::expected<size_t, error_t> {...}
```

```
TEST(test_message_layout, calculate_message_size_data_is_too_large)
{
    constexpr size_t max_payload_size = 255;

    auto const result = calculate_message_size(max_payload_size + 1);
    ASSERT_FALSE(result);
    EXPECT_EQ(result.error(), error_t::data_too_large);
}
```

Transport stack. Message encoding

- Test for error condition (data is too large)

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

inline auto calculate_message_size(
    size_t const data_size
) noexcept -> nonstd::expected<size_t, error_t> {...}
```

```
TEST(test_message_layout, calculate_message_size_data_is_too_large)
{
    constexpr size_t max_payload_size = 255;

    auto const result = calculate_message_size(max_payload_size + 1);
    ASSERT_FALSE(result);
    EXPECT_EQ(result.er    (), error_t::data_too_large);
}
```

Values are specified explicitly according to protocol specification. No constants from implementation reused (to avoid error repetition)

Transport stack. Message encoding

- Test for error condition (data is too large)

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

inline auto calculate_message_size(
    size_t const data_size
) noexcept -> nonstd::expected<size_t, error_t> {...}
```

```
TEST(test_message_layout, calculate_message_size_data_is_too_large)
{
    constexpr size_t max_payload_size = 255;

    auto const result = calculate_message_size(max_payload_size + 1);
    ASSERT_FALSE(result);
    EXPECT_EQ(result.error(), error_t::data_too_large);
}
```

Error code is expected

Transport stack. Message encoding

- Function to build a message:

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

Buffer to place message

Transport stack. Message encoding

- Function to build a message:

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

Data to be sent in a message

Transport stack. Message encoding

- Function to build a message:

```
enum class error_t {
    data_too_large,
    not_enough_space_in_buffer,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

Message timestamp

Transport stack. Message encoding

- Testing of a positive case (buffer has maximum size):

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};
```

Transport stack. Message encoding

- Testing of a positive case (buffer has maximum size):

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};
```

Test values and buffer

Transport stack. Message encoding

- Testing of a positive case (missing interface issue detected):

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<void, error_t> {...}
```

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};

    auto const result
        = assemble_message(buffer, data, test_timestamp);
    ASSERT_TRUE(result);
    EXPECT_THAT(
        <?>,
        ElementsAre(
              0b1010'0001, 0xFA, 0xBC, 0xDE, 0xFF, 0x20, 0x30
        )
    );
}
```

The size of the constructed message is not known

Transport stack. Message encoding

- Testing of a positive case (interface amended):

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<data_t, error_t>
{...}
```

Function will return std::span referring to assembled message location in the buffer

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};

    auto const result
        = assemble_message(buffer, data, test_timestamp);
    ASSERT_TRUE(result);
    EXPECT_THAT(
        result.value(),
        ElementsAre(
              0b1010'0001, 0xFA, 0xBC, 0xDE, 0xFF, 0x20, 0x30
        )
    );
}
```

Transport stack. Message encoding

- Testing of a positive case (interface amended):

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::
```

Message bytes can be easily accessed

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};

    auto const result
        = assemble_message(buffer, data, test_timestamp);
    ASSERT_TRUE(result);
    EXPECT_THAT(
        result.value(),
        ElementsAre(
              0b1010'0001, 0xFA, 0xBC, 0xDE, 0xFF, 0x20, 0x30
        )
    );
}
```

Transport stack. Message encoding

- Testing of a positive case:

```
enum class error_t {
    data_too_large,
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<data_t, error_t> {...}
```

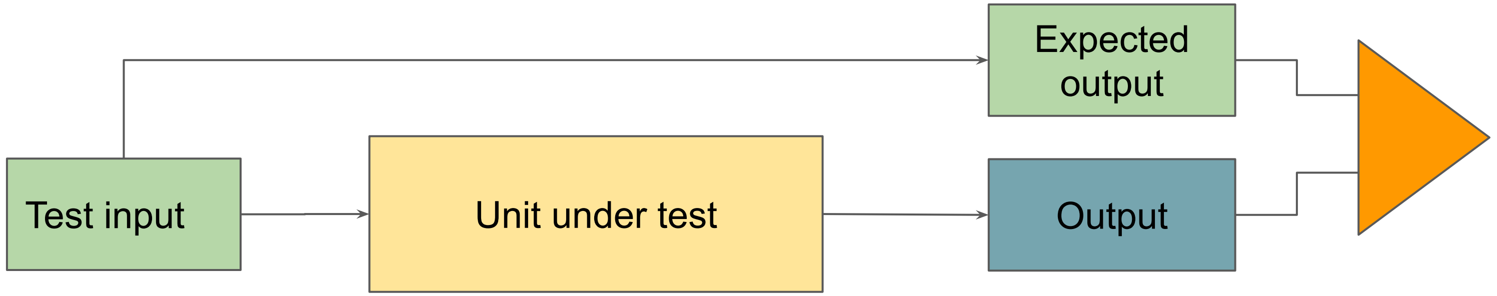
```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};
    std::array<uint8_t, max_message_size> buffer{};

    auto const result
        = assemble_message(buffer, data, test_timestamp);
    ASSERT_TRUE(result);
    EXPECT_THAT(
        result.value(),
        ElementsAre(
              0b1010'0001, 0xFA, 0xBC, 0xDE, 0xFF, 0x20, 0x30
        )
    );
}
```

Values are explicitly specified

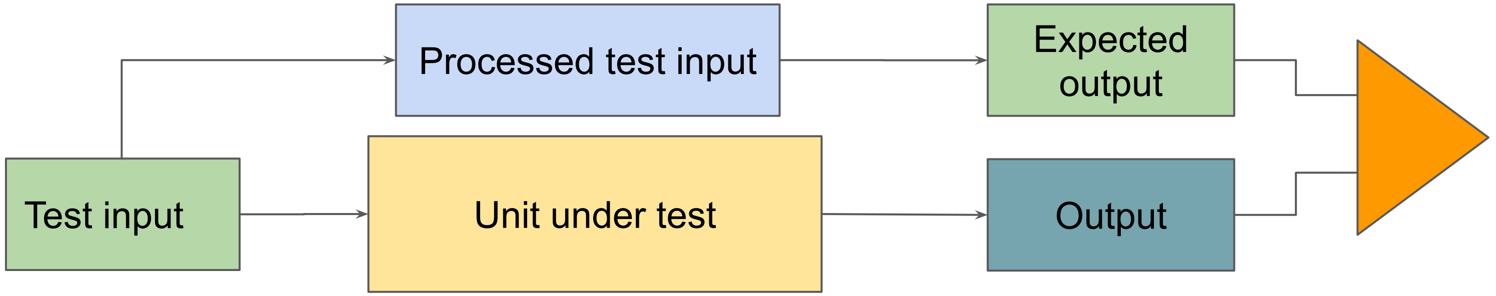
Test simplicity. Decomposition

- Minimalist test logic reduces test defect risks and increases readability
- Explicit test data documents code and test better



Test simplicity. Decomposition

- If it is required to duplicate parts of the logic to prepare expected output, this logic, likely, should be extracted into separate module and mocked in the test
- Decomposition should reach level when test input and expected results can be provided explicitly and be visually comparable



Transport stack. Message encoding

- Message should use network order for timestamp:

```
enum class error_t {
    not_enough_space,
};

using buffer_t = std::span<uint8_t>;
using data_t = std::span<uint8_t const>;

inline auto assemble_message(
    buffer_t const &buffer,
    data_t const &data,
    timestamp_t const timestamp
) noexcept -> nonstd::expected<data_t, error_t> {...}
```

```
TEST(test_message_layout, assemble_message_buffer_is_larger)
{
    constexpr size_t test_data_size{2};
    constexpr timestamp_t test_timestamp{0xFABCDEFF};
    std::array<uint8_t, header_size + test_data_size + 1> buffer{};
    std::array<uint8_t, test_data_size> const data{0x20, 0x30};

    auto const result
        = assemble_message(buffer, data, test_timestamp);
    ASSERT_TRUE(result);
    EXPECT_THAT(
        result.value(),
        ElementsAre(
              0b1010'0001, 0xFA, 0xBC, 0xDE, 0xFF, 0x20, 0x30
        )
    );
}
```

Values converted into network order

Transport stack. Message encoding

- Definition of the function:

```
inline auto assemble_message(buffer_t const &buffer, data_t const &data, timestamp_t const timestamp
) noexcept -> nonstd::expected<data_t, error_t>
{
    if (data.size() > max_payload_size)
        return nonstd::make_unexpected(error_t::data_too_large);

    auto const message_size = data.size() + header_size;
    if (buffer.size() < message_size)
        return nonstd::make_unexpected(error_t::not_enough_space);

    message_header *const header = reinterpret_cast<message_header *>(buffer.data());
    header->type_and_magic = utilities::to_underlying(message_type_t::data) | header_magic;
    header->timestamp = host_to_network(timestamp);
    header->payload_size = static_cast<uint8_t>(data.size());
    std::copy(data.begin(), data.end(), header->payload);

    return data_t{buffer.data(), message_size};
}
```

Transport stack. Message encoding

- Definition of the function:

```
inline auto assemble_message(buffer_t const &buffer, data_t const &data, timestamp_t const timestamp
) noexcept -> nonstd::expected<data_t, error_t>
{
    if (data.size() > max_payload_size)
        return nonstd::make_unexpected(error_t::data_too_large);

    auto const message_size = data.size() + header_size;
    if (buffer.size() < message_size)
        return nonstd::make_unexpected(error_t::not_enough_space);

    message_header *const header = reinterpret_cast<message_header *>(buffer.data());
    header->type_and_magic = utilities::to_underlying(message_type_t::data) | header_magic;
    header->timestamp = host_to_network(timestamp);
    header->payload_size = static_cast<uint8_t>(data.size());
    std::copy(data.begin(), data.end(), header->payload);

    return data_t{buffer.data(), message_size};
}
```

Verified utility function to convert byte order. Doesn't affect readability of the result - no need for further extraction

Byte order conversion function

- Implementation (C++20):

```
constexpr auto host_to_network(std::integral auto const value)
{
    if constexpr (std::endian::native != std::endian::big)
        return utilities::byteswap(value);
    else
        return value;
}
```

- Independent verification for multiple use cases:

```
TEST(test_host_to_network, conversion)
{
    EXPECT_EQ(host_to_network(uint8_t{0xAB}), 0xAB);
    EXPECT_EQ(host_to_network(uint16_t{0xABCD}), 0xCDAB);
    EXPECT_EQ(host_to_network(uint32_t{0xFABCDEFF}), 0xFFDEBCFA);
    //...
}
```

Byte order conversion function

- Implementation (C++20):

```
constexpr auto host_to_network(std::integral auto const value)
{
    if constexpr (std::endian::native != std::endian::big)
        return utilities::byteswap(value);
    else
        return value;
}
```

- Independent verification for multiple use cases:

```
TEST(test_host_to_network, conversion)
{
    EXPECT_EQ(host_to_network(uint8_t{0xAB}), 0xAB);
    EXPECT_EQ(host_to_network(uint16_t{0xABCD}), 0xCDAB);
    EXPECT_EQ(host_to_network(uint32_t{0xFABCDEFF}), 0xFFDEBCFA);
    //...
}
```

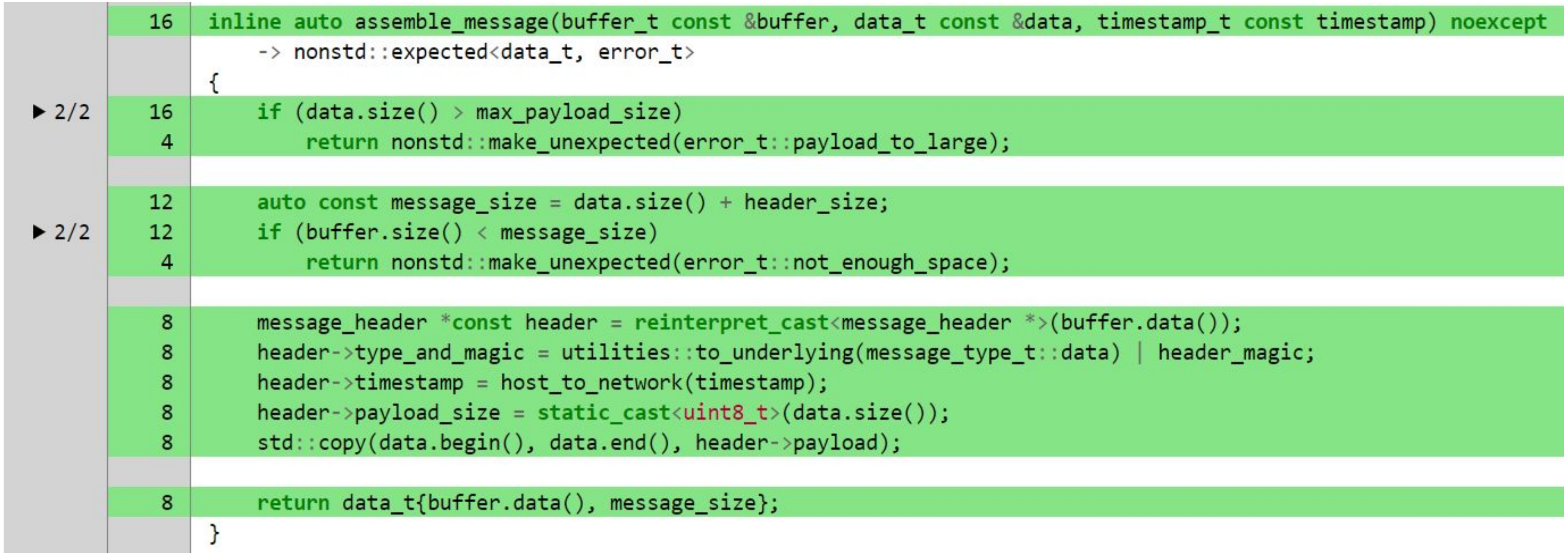
Developed using TDD,

separately from the client code.

Unlikely to contribute issues

Test completeness. Coverage

- Example of full (100%) test coverage:



```
        16  inline auto assemble_message(buffer_t const &buffer, data_t const &data, timestamp_t const timestamp) noexcept
                -> nonstd::expected<data_t, error_t>
            {
▶ 2/2   16      if (data.size() > max_payload_size)
         4          return nonstd::make_unexpected(error_t::payload_to_large);

        12      auto const message_size = data.size() + header_size;
▶ 2/2   12      if (buffer.size() < message_size)
         4          return nonstd::make_unexpected(error_t::not_enough_space);

         8      message_header *const header = reinterpret_cast<message_header *>(buffer.data());
         8      header->type_and_magic = utilities::to_underlying(message_type_t::data) | header_magic;
         8      header->timestamp = host_to_network(timestamp);
         8      header->payload_size = static_cast<uint8_t>(data.size());
         8      std::copy(data.begin(), data.end(), header->payload);

         8      return data_t{buffer.data(), message_size};
            }
```

Project plan. Solution breakdown

- ☐ Transport stack
    - ☑️ Message encoding
    - ☐ Timestamp evaluation
    - ☐ Link layer access
    - ☐ Transmission functionality
- ☐ Peripheral controls
- ☐ Application logic

Transport stack. Message timestamp

- Protocol requires that timestamp is included in the header that is evaluated using the following algorithm:
    - Synchronized time provided by main time source is used if available
    - If enabled by static configuration, local time is used as a fallback
    - Default timestamp (0) is returned if sources are not available

Transport stack. Message timestamp

- Function to evaluate timestamp:

```
auto get_timestamp(bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

- Can use platform API functions as time sources (platform/time.h)

```
typedef uint32_t timestamp_t;

#ifdef __cplusplus
extern "C" {
#endif

timestamp_t get_sync_time();
timestamp_t get_local_time();

#ifdef __cplusplus
}
#endif
```

Transport stack. Message timestamp

- Function to evaluate timestamp:

```
auto get_timestamp(bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

Function declaration

- Can use platform API functions as time sources (platform/time.h)

```
typedef uint32_t timestamp_t;

#ifdef __cplusplus
extern "C" {
#endif

timestamp_t get_sync_time();
timestamp_t get_local_time();

#ifdef __cplusplus
}
#endif
```

Transport stack. Message timestamp

- Function to evaluate timestamp:

```
auto get_timestamp(bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

- Can use platform API functions as time sources (platform/time.h)

```
typedef uint32_t timestamp_t;

#ifdef __cplusplus
extern "C" {
#endif

timestamp_t get_sync_time();
timestamp_t get_local_time();

#ifdef __cplusplus
}
#endif
```

Platform functions that can be used as time sources

Transport stack. Message timestamp

- Platform functions can be used as time sources
- Logic belongs to protocol implementation that can be ported to other targets (host, test doubles, other embedded platform) with other time sources (e.g. chrono framework)
- For testing purposes the dependencies should be substituted by mocks tracking and verifying accesses

Abstraction layer design. Runtime polymorphism

- Dependency on an abstract class that defines interface:

```
struct i_time_sources {
    virtual auto get_sync_time() const -> timestamp_t = 0;
    virtual auto get_local_time() const -> timestamp_t = 0;
protected:
    ~i_time_sources() = default;
};

auto get_timestamp(i_time_sources const &time_sources, bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

Abstraction layer design. Runtime polymorphism

- Dependency on an abstract class that defines interface:

```
struct i_time_sources {
    virtual auto get_sync_time() const -> timestamp_t = 0;
    virtual auto get_local_time() const -> timestamp_t = 0;
protected:
    ~i_time_sources() = default;
};

auto get_timestamp(i_time_sources const &time_sources, bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

Pure virtual class as interface

Abstraction layer design. Runtime polymorphism

- Dependency on an abstract class that defines interface:

```
struct i_time_sources {
    virtual auto get_sync_time() const -> timestamp_t = 0;
    virtual auto get_local_time() const -> timestamp_t = 0;
protected:
    ~i_time_sources() = default;
};

auto get_timestamp(i_time_sources const &time_sources, bool fallback_to_local_time) noexcept -> types::timestamp_t;
```

Injected by reference

Runtime polymorphism. Testing

- For testing purposes, mock implementation can be injected:

```
namespace mocks {
struct time_sources final : i_time_sources {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const, override));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const, override));
};
} // namespace mocks
```

Runtime polymorphism. Testing

- For testing purposes, mock implementation can be injected:

```
namespace mocks {
struct time_sources final : i_time_sources {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const, override));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const, override));
};
} // namespace mocks
```

Mock class inherits interface

Abstraction implementation

- Implementation “wraps” platform access:

```
struct platfrom_time_sources final : i_time_sources {
    auto get_sync_time() const -> timestamp_t override;
    auto get_local_time() const -> timestamp_t override;
};

auto platfrom_time_sources::get_sync_time() const -> timestamp_t
{
    return ::get_sync_time();
}
auto platfrom_time_sources::get_local_time() const -> timestamp_t
{
    return ::get_local_time();
}
```

Implementation uses platform functions

Abstraction implementation. Generic issue

- Parameters and results forwarding:

```
class platfrom_abstraction : public i_platfrom {
public:
    auto do_something(param_t const param, other_param_t const other_param, <...>) const -> result_type_t
    {
        return ::platfrom_do_something(some_fixed_param, param, other_param, some_fixed_param, <...>);
    }
};
```

Forwarding and handling of parameters and results should be tested - can contain errors. Moreso for complex logic

Runtime polymorphism

- Forwarding parameters:

```
class posix_socket_factory : public i_socket_factory {
public:
    auto make(socket_domain const domain, socket_type const type, protocol_type const protocol) override -> ...
    {
        auto const descriptor = ::socket(
              utilities::to_underlying(domain), utilities::to_underlying(type),
              utilities::to_underlying(protocol)
             );
        //...
    }
//...
};
```

- `socket` function declaration (`sys/socket.h`):

```
extern int socket (int __domain, int __type, int __protocol) __THROW;
```

Runtime polymorphism

- Forwarding parameters:

```
class posix_socket_factory : public i_socket_factory {
public:
    auto make(socket_domain const domain, socket_type const type, protocol_type const protocol) override -> ...
    {
        auto const descriptor = ::socket(
               utilities::to_underlying(domain), utilities::to_underlying(type),
               utilities::to_underlying(protocol)
             );
        //...
    }
//...
};
```

Strong types as parameters

- `socket` function declaration (`sys/socket.h`) :

```
extern int socket (int __domain, int __type, int __protocol) __THROW;
```

Runtime polymorphism

- Forwarding parameters:

```
class posix_socket_factory : public i_socket_factory {
public:
    auto make(socket_domain const domain, socket_type const type, protocol_type const protocol) override -> ...
    {
        auto const descriptor = ::socket(
               utilities::to_underlying(domain), utilities::to_underlying(type),
               utilities::to_underlying(protocol)
             );
        //...
    }
//...
};
```

Casted to integers to be passed. Can be mispositioned

- `socket` function declaration (`sys/socket.h`) :

```
extern int socket (int __domain, int __type, int __protocol) __THROW;
```

Runtime polymorphism

- Separation of compilation units
- Isolation of implementation and “sealed” dependencies
- Runtime injection suitable for dynamic configuration and plugins

Runtime polymorphism. Limitations

- Requires corresponding class hierarchy (adapters should be implemented)
- Dynamic dispatch implementation can introduce extra executable code and indirect calls, increasing latency

Testing platform dependent code

- Decomposition can reach the point when code directly dependent on platform facilities should be tested
- Tests should verify access to platform API

Link time substitution

- Implementation of the dependencies is substituted by mock implementation during link time
- Functions are redefined to provide test behaviour (forward calls to mocks)

Link time substitution

- Header file for the platform mock:

```cpp
#include "platform/time.h"

#include "gmock/gmock.h"

namespace test {

struct platform_mock {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const));
};

void register_platfrom_mock(platform_mock &mock);
void unregister_platform_mock();

}  // namespace test
```

Link time substitution

- Header file for the platform mock:

```
#include "platform/time.h"

#include "gmock/gmock.h"

namespace test {

struct platform_mock {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const));
};

void register_platfrom_mock(platform_mock &mock);
void unregister_platform_mock();

}  // namespace test
```

Platform header is included

Link time substitution

- Header file for the platform mock:

```
#include "platform/time.h"

#include "gmock/gmock.h"

namespace test {

struct platform_mock {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const));
};

void register_platfrom_mock(platform_mock &mock);
void unregister_platform_mock();

}  // namespace test
```

Platform mock definition

Link time substitution

- Header file for the platform mock:

```
#include "platform/time.h"

#include "gmock/gmock.h"

namespace test {

struct platform_mock {
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const));
};

void register_platfrom_mock(platform_mock &mock);
void unregister_platform_mock();

}  // namespace test
```

Utilities to register/unregister a mock object

Link time substitution

- Source file for the platform mock:

```
#include "test/platform.hpp"
namespace test {
namespace {
platform_mock *global_mock{};
}  // namespace

void register_platfrom_mock(platform_mock &mock) { global_mock = &mock; }
void unregister_platform_mock() { global_mock = nullptr; }
}  // namespace test

extern "C" {
timestamp_t get_sync_time() { return global_mock->get_sync_time(); }
timestamp_t get_local_time() { return global_mock->get_local_time(); }
}
```

Platform mock singleton

Link time substitution

- Source file for the platform mock:

```
#include "test/platform.hpp"
namespace test {
namespace {
platform_mock *global_mock{};
}  // namespace

void register_platfrom_mock(platform_mock &mock) { global_mock = &mock; }
void unregister_platform_mock() { global_mock = nullptr; }
}  // namespace test

extern "C" {
timestamp_t get_sync_time() { return global_mock->get_sync_time(); }
timestamp_t get_local_time() { return global_mock->get_local_time(); }
}
```

Mock registration/unregistration

Link time substitution

- Source file for the platform mock:

```
#include "test/platform.hpp"
namespace test {
namespace {
platform_mock *global_mock{};
}  // namespace

void register_platfrom_mock(platform_mock &mock) { global_mock = &mock; }
void unregister_platform_mock() { global_mock = nullptr; }
}  // namespace test

extern "C" {
timestamp_t get_sync_time() { return global_mock->get_sync_time(); }
timestamp_t get_local_time() { return global_mock->get_local_time(); }
}
```

Test implementation that forwards calls to the mock

Link time substitution

- Test code:

```
using namespace ::testing;

class test_static_timestamp_provider : public Test {
    auto SetUp() override { test::register_platfrom_mock(mock); }
    auto TearDown() override { test::unregister_platform_mock(); }

protected:
    StrictMock<test::platform_mock> mock{};
};

TEST_F(test_static_timestamp_provider, synced_time_available)
{
    constexpr timestamp_t result_timestamp{100};
    EXPECT_CALL(mock, get_sync_time).WillOnce(Return(result_timestamp));
    EXPECT_EQ(transport::get_timestamp(true), result_timestamp);
}
```

Fixture owns and manages the mock

Link time substitution

- Test code:

```
using namespace ::testing;

class test_static_timestamp_provider : public Test {
    auto SetUp() override { test::register_platfrom_mock(mock); }
    auto TearDown() override { test::unregister_platform_mock(); }

protected:
    StrictMock<test::platform_mock> mock{};
};

TEST_F(test_static_timestamp_provider, synced_time_available)
{
    constexpr timestamp_t result_timestamp{100};
    EXPECT_CALL(mock, get_sync_time).WillOnce(Return(result_timestamp));
    EXPECT_EQ(transport::get_timestamp(true), result_timestamp);
}
```

Expectation setup

Link-time substitution

- Can be used for testing C code
- Well suitable for dependencies on C-APIs
- Doesn’t require dependency injection based design
- Doesn’t require any code redesign, can be used with legacy code
- Doesn’t introduce any overhead and do not rely on compiler optimization

Link-time substitution. Limitations

- Requires testing environment (that can be generated from headers)
- Do not encourage and prototype decoupling: dependencies remain in the code

Timestamp provider portability

- Timestamp provider portability was listed as a requirement
- Other dependency injection technique should be considered

Function pointers

- Pointers to functions are passed to dependent code

Function pointers

- Well known C practice, used widely in embedded and low-level

```
struct time_sources {
    timestamp_t (*get_sync_time)();
    timestamp_t (*get_local_time)();
};

timestamp_t get_timestamp(struct time_sources const *time_sources, BOOL fallback_to_local_time);
```

- Can be slightly modernized in C++:

```
struct time_sources {
    std::add_pointer_t<timestamp_t()> get_sync_time;
    std::add_pointer_t<timestamp_t()> get_local_time;
};

auto get_timestamp(time_sources const &time_sources, bool fallback_to_local_time) noexcept -> timestamp_t;
```

Function pointers

- Well known C practice, used widely in embedded and low-level

```
struct time_sources {
    timestamp_t (*get_sync_time)();
    timestamp_t (*get_local_time)();
};

timestamp_t get_timestamp(struct time_sources const *time_sources, BOOL fallback_to_local_time);
```

Pointers to functions with specified signatures

- Can be slightly modernized in C++:

```
struct time_sources {
    std::add_pointer_t<timestamp_t()> get_sync_time;
    std::add_pointer_t<timestamp_t()> get_local_time;
};

auto get_timestamp(time_sources const &time_sources, bool fallback_to_local_time) noexcept -> timestamp_t;
```

Function pointers

- Well-known C practice, used widely in embedded and low-level

```
struct time_sources {
    timestamp_t (*get_sync_time)();
    timestamp_t (*get_local_time)();
};

timestamp_t get_timestamp(struct time_sources const *time_sources, BOOL fallback_to_local_time);
```

- Can be slightly modernized in C++:

```
struct time_sources {
    std::add_pointer_t<timestamp_t()> get_sync_time;
    std::add_pointer_t<timestamp_t()> get_local_time;
};

auto get_timestamp(time_sources const &time_sources, bool fallback_to_local_time) noexcept -> timestamp_t;
```

Left-to-right member declaration

Function pointers

- Sample code:

```
inline auto get_timestamp(time_sources const &time_sources, bool fallback_to_local_time) noexcept -> timestamp_t
{
    if (auto const sync_timestamp = time_sources.get_sync_time(); sync_timestamp > 0) {
            return sync_timestamp;
    }
    return fallback_to_local_time ? time_sources.get_local_time() : timestamp_t{};
}
```

Function pointers

- Sample code:

```
inline auto get_timestamp(time_sources const &time_sources, bool fallback_to_local_time) noexcept -> timestamp_t
{
    if (auto const sync_timestamp = time_sources.get_sync_time(); sync_timestamp > 0) {
            return sync_timestamp;
    }
    return fallback_to_local_time ? time_sources.get_local_time() : timestamp_t{};
}
```

Function indirect invocation

Function pointers

- Sample client code:

```
transport::time_sources time_sources{
    .get_sync_time = ::get_sync_time,
    .get_local_time = ::get_local_time
};


auto const timestamp = transport::get_timestamp(time_sources, true);
```

Designated initializers (starting with C++20)

Function pointers

- Sample client code:

```
transport::time_sources time_sources{
    .get_sync_time = ::get_sync_time,
    .get_local_time = ::get_local_time
};

auto const timestamp = transport::get_timestamp(time_sources, true);
```

Can be used with standalone functions, static member functions and captureless lambda-s

Function pointers. Limitations

- Introduces explicit indirect call that can prevent compiler optimization and typically increases code latency (the problem mostly manifests itself if the compiler optimizes for space when inlining is reduced)
- Function pointer is not associated with an object that can hold some “context”: testing will need to use singleton mocks (similar to link-time substitution)

Polymorphic function wrappers

- Based on type erasure
- Standard supporting facility is `std::function`
- Non-allocating alternatives: `eastl::fixed_function`, `sg14/inplace_function`

Polymorphic function wrappers

- Function wrappers store callables:

```
using timestamp_source = eastl::fixed_function<config::capture_list_size, timestamp_t()>;
class timestamp_provider final {
public:
    timestamp_provider(timestamp_source sync_time, timestamp_source local_time, <...>) noexcept
        : _sync_time{std::move(sync_time)} , _local_time{std::move(local_time)}, <...>
    {}
// ...
```

Polymorphic function wrappers

- Function wrappers store callables:

```
using timestamp_source = eastl::fixed_function<config::capture_list_size, timestamp_t()>;
class timestamp_provider final {
public:
    timestamp_provider(timestamp_source sync_time, tim     sourc
        : _sync_time{std::move(sync_time)} , _local_time{std::
    {}
// ...
```

Alternative for `std::function` with static storage for capture list (no dynamic memory allocation performed)

Polymorphic function wrappers

- Can be used with all types of callable objects:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_sync_time(); }, [&mock](){ return mock.get_local_time(); }, true
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{std::ref(get_sync_time), std::ref(get_local_time), true};
```

Polymorphic function wrappers

- Can be used with all types of callable objects:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_sync_time(); }, [&mock](){ return mock.get_local_ti
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{std::ref(get_sync_time), std::ref(get_local_time), true};
```

Can accept lambda-s and standalone functions

Polymorphic function wrappers

- Can be used with all types of callable objects:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_sync_time(); }, [&mock](){ return mock.get_loca
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{std::ref(get_sync_time), std::ref(get_local_time), true};
```

Can be used to forward calls to captured object

Polymorphic function wrappers

- Can be used with all types of callable objects:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
      [&mock](){ return mock.get_sync_time(); }, [&mock](){ return mock.get_local_time(); }, true
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{std::ref(get_sync_time), std::ref(get_local_time), true};
```

Including mocks in tests

Polymorphic function wrappers

- Can be used with all types of callable objects:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_sync_time(); }, [&mock](){ return mock.get_local_time(); }, true
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{std::ref(get_sync_time), std::ref(get_local_time), true};
```

Passing mock with call operator by reference

Polymorphic function wrappers

- Timestamp evaluation:

```
class timestamp_provider final {
public:
    // ...
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const sync_timestamp = _sync_time(); sync_timestamp > 0) {
            return sync_timestamp;
        }
        return _fallback_to_local_time ? _local_time() : timestamp_t{};
    }
private:
    timestamp_source _sync_time;
    timestamp_source _local_time;
    bool const _fallback_to_local_time;
};
```

Polymorphic function wrappers

- Timestamp evaluation:

```
class timestamp_provider final {
public:
    // ...
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const sync_timestamp = _sync_time(); sync_timestamp > 0) {
            return sync_timestamp;
        }
        return _fallback_to_local_time ? _local_time() : timestamp_t{};
    }
private:
    timestamp_source _sync_time;
    timestamp_source _local_time;
    bool const _fallback_to_local_time;
};
```

Invocation

Polymorphic function wrappers. Limitations

- Introduce runtime overhead, including indirect calls, most compilers can not eliminate (with some exceptions)
- One wrapper object is needed per interface method

Compile time polymorphism

- Dependent code has to be generic
- Dependency type is specified by template argument
- Dependency and dependent code need to be compatible (substitution should not fail)

Compile time polymorphism. Object construction

- Forwarding constructor and template argument deduction guide:

```
template <typename GetSyncTime, typename GetLocalTime>
class timestamp_provider final {
public:
    template <typename Sync, typename Local>
    timestamp_provider(Sync &&get_sync_time, Local &&get_local_time, bool <...>) noexcept
        : _get_sync_time{std::forward<Sync>(get_sync_time)}
        , _get_local_time{std::forward<Local>(get_local_time)}
        , <...>
    {}
private:
    GetSyncTime _get_sync_time;
    GetLocalTime _get_local_time;
// …
};

template <typename Sync, typename Local>
timestamp_provider(Sync &&, Local &&, bool) -> timestamp_provider<Sync, Local>;
```

Constructor with perfect forwarding

Compile time polymorphism. Object construction

- Forwarding constructor and template argument deduction guide:

```
template <typename GetSyncTime, typename GetLocalTime>
class timestamp_provider final {
public:
    template <typename Sync, typename Local>
    timestamp_provider(Sync &&get_sync_time, Local &&get_local_time, bool <...>) noexcept
        : _get_sync_time{std::forward<Sync>(get_sync_time)}
        , _get_local_time{std::forward<Local>(get_local_time)}
        , <...>
    {}
private:
    GetSyncTime _get_sync_time;
    GetLocalTime _get_local_time;
// …
};

template <typename Sync, typename Local>
timestamp_provider(Sync &&, Local &&, bool) -> timestamp_provider<Sync, Local>;
```

Template argument deduction guide. Links constructor parameters to class template parameters. Factory function with perfect forwarding can be used instead

Compile time polymorphism. Object construction

- Template can be used with left and right values:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_synched_time(); }, [&mock](){ return mock.get_local_time(); }, true
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{get_sync_time, get_local_time, true};
```

Compile time polymorphism. Object construction

- Template can be used with left and right values:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_synched_time(); }, [&mock](){ return mock.g [illegible]_time()
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{get_sync_time, get_local_time, true};
```

Visually similar to type erase but the templates are instantiated with deduced arguments (distinct types are produced)

Compile time polymorphism. Object construction

- Template can be used with left and right values:

```
timestamp_provider const time_provider{[](){ return timestamp_t{}; }, ::get_local_time, true};
// or
timestamp_provider const time_provider{::get_sync_time, [&clock](){ clock.get_time(); }, true};
// or
StrictMock<mocks::time_sources> mock;
timestamp_provider const time_provider{
       [&mock](){ return mock.get_synched_time(); }, [&mock](){ return mock.get_local_time(); }, true
};
// or
namespace mocks {
struct time_source_function {
    MOCK_METHOD(timestamp_t, call, (), (const));
    auto operator()() const { return call(); }
};
}  // namespace mocks
StrictMock<mocks::time_source_function> get_sync_time;
StrictMock<mocks::time_source_function> get_local_time;
timestamp_provider const timestamp_provider{get_sync_time, get_local_time, true};
```

Deduced to lvalue reference

Compile time polymorphism

- Invocation:

```
template <typename GetSyncTime, typename GetLocalTime>
class timestamp_provider final {
public:
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const synced_timestamp = _get_sync_time(); sync_timestamp > 0) {
            return synced_timestamp;
        }
        return _fallback_to_local_time ? _get_local_time() : timestamp_t{};
    }
private:
    GetSyncTime _get_sync_time;
    GetLocalTime _get_local_time;
    bool const _fallback_to_local_time;
};
```

Compile time polymorphism

- Invocation:

```
template <typename GetSyncTime, typename GetLocalTime>
class timestamp_provider final {
public:
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const synced_timestamp = _get_sync_time(); sync_timestamp > 0) {
            return synced_timestamp;
        }
        return _fallback_to_local_time ? _get_local_time() : timestamp_t{};
    }
private:
    GetSyncTime _get_sync_time;
    GetLocalTime _get_local_time;
    bool const _fallback_to_local_time;
};
```

Invocations of the callables

Compile time polymorphism. Explicit indirection

- Function pointers are explicitly injected:

```
inline auto get_scaled_time() noexcept {
    return ::get_local_time() * 100;
}

transport::timestamp_provider timestamp_provider{
       ::get_sync_time, get_scaled_time, true};

auto const timestamp = tp.timestamp();
```

Compile time polymorphism. Explicit indirection

- Function pointers are explicitly injected:

```
inline auto get_scaled_time() noexcept {
    return ::get_local_time() * 100;
}

transport::timestamp_provider timestamp_provider{
       ::get_sync_time, get_scaled_time, true};

auto const timestamp = tp.timestamp();
```

Explicit injection of function pointers. Indirection elimination and inlining depends on code structure, build parameters and compiler abilities

Compile time polymorphism. Explicit indirection

- Injection of a closure object instead of a function:

```
inline auto get_scaled_time() noexcept {
    return ::get_local_time() * 100;
}

transport::timestamp_provider timestamp_provider{
       [](){ return ::get_sync_time(); },
       [](){ return get_scaled_time(); },
       true};

auto const timestamp = tp.timestamp();
```

Compile time polymorphism. Explicit indirection

- Injection of a closure object instead of a function:

```
inline auto get_scaled_time() noexcept {
    return ::get_local_time() * 100;
}

transport::timestamp_provider timestamp_provider{
       [](){ return ::get_sync_time(); },
       [](){ return get_scaled_time(); },
       true};

auto const timestamp = tp.timestamp();
```

Function calls wrapped in lambda-s. No indirect call will be emitted. `get_scaled_time` is likely to be inlined

Compile time polymorphism. Further improvement

- Remaining issue:

```
template <typename GetSyncTime, typename GetLocalTime>
class timestamp_provider final {
public:
    template <typename Sync, typename Local>
    timestamp_provider(Sync &&get_sync_time, Local &&get_local_time, bool <...>) noexcept
        : _get_sync_time{std::forward<Sync>(get_sync_time)}
        , _get_local_time{std::forward<Local>(get_local_time)}
        , <...>
    {}
private:
    GetSyncTime _get_sync_time;
    GetLocalTime _get_local_time;
    bool const _fallback_to_local_time;
};
```

Individual parameter for each dependency. Approach can be unsustainable

Compile time polymorphism

- Injection of a single object:

```
template <typename Config>
class timestamp_provider final {
public:
//…
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const synced_timestamp = _config.get_sync_time(); synced_timestamp > 0) {
            return synced_timestamp;
        }
        return _fallback_to_local_time ? _config.get_local_time() : timestamp_t{};
    }
private:
    Config _config;
    bool const _fallback_to_local_time;
};
```

Compile time polymorphism

- Injection of a single object:

```
template <typename Config>
class timestamp_provider final {
public:
//…
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const synced_timestamp = _config.get_sync_time(); synced_timestamp > 0) {
            return synced_timestamp;
        }
        return _fallback_to_local_time ? _config.get_local_time() : timestamp_t{};
    }
private:
    Config _config;
    bool const _fallback_to_local_time;
};
```

Config type should have required methods

Compile time polymorphism

- Injection of a single object:

```
template <typename Config>
class timestamp_provider final {
public:
    template <typename ConfigP>
    timestamp_provider(ConfigP && config, bool const fallback_to_local_time) noexcept
        : _config{std::forward<ConfigP>(config)}, _fallback_to_local_time{fallback_to_local_time}
    {}
//...
private:
    Config _config;
    bool const _fallback_to_local_time;
};

template <typename Config>
timestamp_provider(Config &&, bool) -> timestamp_provider<Config>;
```

Forwarding constructor

Compile time polymorphism

- Injection of a single object:

```
template <typename Config>
class timestamp_provider final {
public:
    template <typename ConfigP>
    timestamp_provider(ConfigP && config, bool const fallback_to_local_time) noexcept
        : _config{std::forward<ConfigP>(config)}, _fallback_to_local_time{fallback_to_local_time}
    {}
//...
private:
    Config _config;
    bool const _fallback_to_local_time;
};

template <typename Config>
timestamp_provider(Config &&, bool) -> timestamp_provider<Config>;
```

Deduction guide

Compile time polymorphism

- Injection of a single object:

```
template <typename Config>
class timestamp_provider final {
public:
    template <typename ConfigP>
    timestamp_provider(ConfigP && config, bool const fallback_to_local_time) noexcept
        : _config{std::forward<ConfigP>(config)}, _fallback_to_local_time{fallback_to_local_time}
    {}
//...
private:
    Config _config;
    bool const _fallback_to_local_time;
};

template <typename Config>
timestamp_provider(Config &&, bool) -> timestamp_provider<Config>;
```

Can be deduced to `Config` or `Config &`. Done for convenience. For testability passing by reference is important.

Compile time polymorphism

- Alternative to lambda-s is a class with methods wrapping function calls:

```
struct timestamp_provider_config {
    auto get_sync_time() const { return ::get_sync_time(); }
    auto get_local_time() const { return get_scaled_time(); }
};

timestamp_provider_config config{};
transport::timestamp_provider timestamp_provider{config, true};
```

Compile time polymorphism

- Alternative to lambda-s is a class with methods wrapping function calls:

```
struct timestamp_provider_config {
    auto get_sync_time() const { return ::get_sync_time(); }
    auto get_local_time() const { return get_scaled_time(); }
};

timestamp_provider_config config{};
transport::timestamp_provider timestamp_provider{config, true};
```

No indirect calls

Compile time polymorphism

- Alternative to lambda-s is a class with methods wrapping function calls:

```
struct timestamp_provider_config {
    auto get_sync_time() const { return ::get_sync_time(); }
    auto get_local_time() const { return get_scaled_time(); }
};

timestamp_provider_config config{};
transport::timestamp_provider timestamp_provider{config, true};
```

Can have errors in forwarding of parameters and return value.
Similar to runtime polymorphism example

Compile time polymorphism

- Perfect forwarding between member and standalone functions:

```
struct some_depencency{
    template <typename... T>
    decltype(auto) some_method(T &&...params) const
    {
        return some_function(std::forward<T>(params)...);
    }
    //...
};
```

- Can be generalized using macro:

```
#define FORWARD_CALL(method_name, target_function)           \
    template <typename... T>                                 \
    decltype(auto) method_name(T &&...params) const          \
    {                                                        \
        return target_function(std::forward<T>(params)...);  \
    }
```

Compile time polymorphism

- Perfect forwarding between member and standalone functions:

```
struct some_depencency{
    template <typename... T>
    decltype(auto) some_method(T &&...params) const
    {
        return some_function(std::forward<T>(params)...);
    }
    //...
};
```

Solution with perfect forwarding reduces risk of incorrect forwarding

- Can be generalized using macro:

```
#define FORWARD_CALL(method_name, target_function)          \
    template <typename... T>                                \
    decltype(auto) method_name(T &&...params) const         \
    {                                                       \
        return target_function(std::forward<T>(params)...);   \
    }
```

Compile time polymorphism

- Perfect forwarding between member and standalone functions:

```
struct some_depencency{
    template <typename... T>
    decltype(auto) some_method(T &&...params) const
    {
        return some_function(std::forward<T>(params)...);
    }
    //...
};
```

- Can be generalized using macro:

```
#define FORWARD_CALL(method_name, target_function)          \
    template <typename... T>                                 \
    decltype(auto) method_name(T &&...params) const          \
    {                                                        \
        return target_function(std::forward<T>(params)...);  \
    }
```

Macro reduces boilerplate and error possibility

Compile time polymorphism

- Standalone functions “transformed” into class methods:

```
struct timestamp_provider_config {
    FORWARD_CALL(get_sync_time, ::get_sync_time);
    FORWARD_CALL(get_local_time, ::get_local_time);
    // FORWARD_CALL(get_local_time, get_scaled_time);
};

timestamp_provider_config config{};
transport::timestamp_provider timestamp_provider{config, true};
```

Compile time polymorphism

- Can be a regular class:

```
class timestamp_provider_config {
public:
    auto get_sync_time() const -> timestamp_t;
    auto get_local_time() const -> timestamp_t;
private:
//...
};

timestamp_provider_config config{};
transport::timestamp_provider timestamp_provider{config, true};
```

Can implement complex logic, have dependencies and state. Should be tested (aforementioned abstraction layer issue)

Compile time polymorphism

- Implementation would also accept function pointers (if needed):

```
struct timestamp_provider_config {
    std::add_pointer_t<timestamp_t()> get_sync_time;
    std::add_pointer_t<timestamp_t()> get_local_time;
};

timestamp_provider_config config{...};
transport::timestamp_provider timestamp_provider{config, true};
```

Compile time polymorphism

- Interfaces can also be injected:

```
struct i_time_sources {
    virtual auto get_sync_time() const -> timestamp_t = 0;
    virtual auto get_local_time() const -> timestamp_t = 0;
protected:
    ~i_time_sources() = default;
};

i_time_sources const &time_sources = ...;
transport::timestamp_provider timestamp_provider{time_sources, true};
```

Compile time polymorphism. Platform decoupling

- Implementation is still dependent on platform:

```
template <typename Config>
class timestamp_provider final {
public:
//...
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const synced_timestamp = _config.get_sync_time(); synced_timestamp > 0) {
            return synced_timestamp;
        }
        return _fallback_to_local_time ? _config.get_local_time() : timestamp_t{};
    }
private:
    Config _config;
    bool const _fallback_to_local_time;
};
```

Using contracts and types of platform API

Compile time polymorphism. Platform decoupling

- Generalization:

```
template <typename Config>
class timestamp_provider final {
public:
    using timestamp_t = typename std::remove_reference_t<Config>::timestamp_t;
    //…
    auto timestamp() const noexcept -> timestamp_t
    {
        if (auto const sync_timestamp = _config.get_sync_time(); _config.is_valid_timestamp(sync_timestamp)) {
            return sync_timestamp;
        }
        return _fallback_to_local_time ? _config.get_local_time()
                                       : _config.default_timestamp();
    }
private:
    Config _config;
    bool const _fallback_to_local_time;
};
```

Specifics moved to the configuration. Should be independently tested

Compile time polymorphism. Platform decoupling

- Configuration for the target platform:

```
struct timestamp_provider_config {
    using timestamp_t = ::timestamp_t;

    FORWARD_CALL(get_sync_time, ::get_sync_time);
    FORWARD_CALL(get_local_time, ::get_local_time);

    auto is_valid_timestamp(timestamp_t const timestamp) const noexcept -> bool
    {
        return timestamp > 0;
    }

    auto default_timestamp() const noexcept -> timestamp_t
    {
        return timestamp_t{};
    }
};
```

Compile time polymorphism. Template constraints

- Concept to constrain configuration type and document required API:

```
template <
    typename Config,
    typename Timestamp = typename std::remove_reference_t<Config>::timestamp_t>
concept timestamp_provider_config = requires(Config c) {
    { c.get_sync_time() } -> std::same_as<Timestamp>;
    { c.get_local_time() } -> std::same_as<Timestamp>;
    { c.is_valid_timestamp(std::declval<Timestamp>()) } -> std::same_as<bool>;
    { c.default_timestamp() } -> std::same_as<Timestamp>;
};
```

Compile time polymorphism. Template constraints

- Concept to constrain configuration type and document required API:

```
template <
    typename Config,
    typename Timestamp = typename std::remove_reference_t<Config>::timestamp_t>
concept timestamp_provider_config = requires(Config c) {
    { c.get_sync_time() } -> std::same_as<Timestamp>;
    { c.get_local_time() } -> std::same_as<Timestamp>;
    { c.is_valid_timestamp(std::declval<Timestamp>()) } -> std::same_as<bool>;
    { c.default_timestamp() } -> std::same_as<Timestamp>;
};
```

Requires functions and specifies relations between return types and parameters

Compile time polymorphism. Platform decoupling

- Configuration for the target platform:

```
struct timestamp_provider_config {
    using timestamp_t = ::timestamp_t;

    FORWARD_CALL(get_sync_time, ::get_sync_time);
    FORWARD_CALL(get_local_time, ::get_local_time);

    auto is_valid_timestamp(timestamp_t const timestamp) const noexcept -> bool
    {
        return timestamp > 0;
    }

    auto default_timestamp() const noexcept -> timestamp_t
    {
        return timestamp_t{};
    }
};

static_assert(timestamp_provider_config_like<timestamp_provider_config>);
```

Concept can be used to verify compatibility without instantiation

Compile time polymorphism. Unit testing

- Config mock class:

```
namespace mocks {
struct timestamp_provider_config {
    using timestamp_t = types::timestamp_t;
    MOCK_METHOD(timestamp_t, get_sync_time, (), (const));
    MOCK_METHOD(timestamp_t, get_local_time, (), (const));
    MOCK_METHOD(bool, is_valid_timestamp, (timestamp_t), (const));
    MOCK_METHOD(timestamp_t, default_timestamp, (), (const));
};
}  // namespace mocks
```

Compile time polymorphism

- Test case example:

```
TEST(test_generic_timestamp_provider, synced_time_not_available_fallback_enabled)
{
    StrictMock<mocks::timestamp_provider_config> const time_sources;

    constexpr timestamp_t invalid_timestamp{100};
    constexpr timestamp_t result_timestamp{200};

    EXPECT_CALL(time_sources, get_sync_time)
        .WillOnce(Return(invalid_timestamp));
    EXPECT_CALL(time_sources, is_valid_timestamp(invalid_timestamp))
        .WillOnce(Return(false));
    EXPECT_CALL(time_sources, get_local_time)
        .WillOnce(Return(result_timestamp));

    timestamp_provider const timestamp_provider{time_sources, true};
    EXPECT_EQ(timestamp_provider.timestamp(), result_timestamp);
}
```

Project plan. Solution breakdown

- ☐ Transport stack
  - ☑ Message encoding
  - ☑ Timestamp evaluation
  - ☐ Link layer access
  - ☐ Transmission functionality
- ☐ Peripheral controls
- ☐ Application logic

Compile time polymorphism. Advantages

- Indirect calls can be avoided, good potential for compiler optimization
- Templatized code can be further optimized by a compiler (elimination of unreachable code e.t.c)
- Can be used to substitute dependencies on concrete types

Compile time polymorphism. Limitations

- Dependent code has to be generic
- Static dependency injection (compile time)
- The dependent code and dependencies have to be in a single translation unit (increases compilation time but supports compiler optimization)

Compile time polymorphism. Applications

- Can be used with other platform dependent types:

```cpp
template <typename T>
concept ofstream_like = requires(T t) {
    static_cast<bool>(t);
    t.open(std::declval<char const *>());
    t << std::declval<std::string>();
    //...
};

void try_to_store_record(ofstream_like auto& stream, std::string const& file_path, std::string const& record)
{
    stream.open(file_path.c_str());
    if (!stream)
        return;

    stream << record;
}
```

Compile time polymorphism. Applications

- Can be used with other platform dependent types:

```
template <typename T>
concept ofstream_like = requires(T t) {
    static_cast<bool>(t);
    t.open(std::declval<char const *>());
    t << std::declval<std::string>();
    //...
};

void try_to_store_record(ofstream_like auto& stream, std::string const& file_path, std::string const& record)
{
    stream.open(file_path.c_str());
    if (!stream)
        return;

    stream << record;
}
```

Concept for `ofstream`-like object

Compile time polymorphism. Applications

- Can be used with other platform dependent types:

```
template <typename T>
concept ofstream_like = requires(T t) {
    static_cast<bool>(t);
    t.open(std::declval<char const *>());
    t << std::declval<std::string>();
    //...
};

void try_to_store_record(ofstream_like auto& stream, std::string const& file_path, std::string const& record)
{
    stream.open(file_path.c_str());
    if (!stream)
        return;

    stream << record;
}
```

Injected object of `ofstream_like` type

Compile time polymorphism. Applications

- Pre C++20 version:

```
template <typename OfStream>
void try_to_store_record(OfStream& stream, std::string const& file_path, std::string const& record)
{
    stream.open(file_path.c_str());
    if (!stream)
        return;

    stream << record;
}
```

Regular function template

Compile time polymorphism. Applications

- Mock type to verify stream operations:

```
namespace mocks {
struct stream {
    MOCK_METHOD(void, open, (char const *));
    MOCK_METHOD(bool, to_bool_op, (), (const));
    operator bool() const { return to_bool_op(); }
    MOCK_METHOD(stream &, insert_into_stream_op, (std::string const &));
    stream &operator<<(std::string const &s) { return insert_into_stream_op(s); }
};
}
```

Compile time polymorphism. Applications

- Test for the function:

```
TEST(test_file_output, test_open_and_write)
{
    StrictMock<mocks::stream> stream;
    constexpr char const *file_name{"file_name"};
    constexpr char const *record{"record"};
    {
        InSequence seq;
        EXPECT_CALL(stream, open(StrEq(file_name)));
        EXPECT_CALL(stream, to_bool_op).WillOnce(Return(true));
        EXPECT_CALL(stream, insert_into_stream_op(StrEq(record))).WillOnce(ReturnRef(stream));
    }
    try_to_store_record(stream, file_name, record);
}
```

Compile time polymorphism. Applications

- Test for the function:

```
TEST(test_file_output, test_open_and_write)
{
    StrictMock<mocks::stream> stream;
    constexpr char const *file_name{"file_name"};
    constexpr char const *record{"record"};
    {
        InSequence seq;
        EXPECT_CALL(stream, open(StrEq(file_name)));
        EXPECT_CALL(stream, to_bool_op).WillOnce(Return(true));
        EXPECT_CALL(stream, insert_into_stream_op(StrEq(record))).WillOnce(ReturnRef(stream));
    }
    try_to_store_record(stream, file_name, record);
}
```

Mock is injected

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<MutexP>(mutex)}
    {}
    synchronized_stack() = default;
    //...
private:
    Container _container{};
    Mutex _mutex{};
};

template <stack_like_accessible Container, lockable Mutex>
synchronized_stack(Container&&, Mutex&&)
       -> synchronized_stack<typename std::remove_reference_t<Container>::value_type, Container, Mutex>;
```

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<
    {}
    synchronized_stack() = default;
    //...
private:
    Container _container{};
    Mutex _mutex{};
};

template <stack_like_accessible Container, lockable Mutex>
synchronized_stack(Container&&, Mutex&&)
       -> synchronized_stack<typename std::remove_reference_t<Container>::value_type, Container, Mutex>;
```

Container and Mutex types should be specified by template arguments

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<MutexP>(mutex)}
    {}
    synchronized_stack() = default;
    //...
private:
    Container _container{};
    Mutex _mutex{};
};

template <stack_like_accessible Container, lockable Mutex>
synchronized_stack(Container&&, Mutex&&)
      -> synchronized_stack<typename std::remove_reference_t<Container>::value_type, Container, Mutex>;
```

Forwarding constructor and deduction guides are used

Compile time polymorphism. Applications

- Concepts for required types:

```
template <typename T>
concept lockable = requires(T t) {
    t.lock();
    t.unlock();
};

template <typename T>
concept stack_like_accessible = requires(T t) {
    typename std::remove_reference_t<T>::value_type;
    t.push_back(std::declval<typename std::remove_reference_t<T>::value_type>());
    //...
};
```

Compile time polymorphism. Applications

- Concepts for required types:

```cpp
template <typename T>
concept lockable = requires(T t) {
    t.lock();
    t.unlock();
};

template <typename T>
concept stack_like_accessible = requires(T t) {
    typename std::remove_reference_t<T>::value_type;
    t.push_back(std::declval<typename std::remove_reference_t<T>::value_type>());
    //...
};
```

Corresponds to `locable` named requirement

Compile time polymorphism. Applications

- Concepts for required types:

```
template <typename T>
concept lockable = requires(T t) {
    t.lock();
    t.unlock();
};

template <typename T>
concept stack_like_accessible = requires(T t) {
    typename std::remove_reference_t<T>::value_type;
    t.push_back(std::declval<typename std::remove_reference_t<T>::value_type>());
    //...
};
```

Requires that container has functions needed for stack implementation

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<MutexP>(mutex)}
    {}
    synchronized_stack() = default;
    //...
    void push(T const& v)
    {
        std::scoped_lock lock{_mutex};
        _container.push_back(v);
    }
private:
    Container _container{};
    Mutex _mutex{};
};
```

locable object is compatible with other standard facilities

Compile time polymorphism. Applications

- Mock definitions:

```
namespace mocks {

template <typename T>
struct vector {
    using value_type = T;
    MOCK_METHOD(void, push_back, (T const &));
};

struct mutex {
    MOCK_METHOD(void, lock, ());
    MOCK_METHOD(void, unlock, ());
};
}  // namespace mocks
```

Compile time polymorphism

- Test code:

```
TEST(test_synced_queue, test_push_back)
{
    StrictMock<mocks::vector<int>> vector;
    StrictMock<mocks::mutex> mutex;

    synchronized_stack stack{vector, mutex};

    constexpr int test_int{100};
    {
        InSequence seq;
        EXPECT_CALL(mutex, lock);
        EXPECT_CALL(vector, push_back(test_int));
        EXPECT_CALL(mutex, unlock);
    }
    stack.push(test_int);
}
```

Compile time polymorphism

- Test code:

```
TEST(test_synced_queue, test_push_back)
{
    StrictMock<mocks::vector<int>> vector;
    StrictMock<mocks::mutex> mutex;

    synchronized_stack stack{vector, mutex};

    constexpr int test_int{100};
    {
        InSequence seq;
        EXPECT_CALL(mutex, lock);
        EXPECT_CALL(vector, push_back(test_int));
        EXPECT_CALL(mutex, unlock);
    }
    stack.push(test_int);
}
```

Mocks are used for tests

Compile time polymorphism

- Test code:

```
TEST(test_synced_queue, test_push_back)
{
    StrictMock<mocks::vector<int>> vector;
    StrictMock<mocks::mutex> mutex;

    synchronized_stack stack{vector, mutex};

     constexpr int test_int{100};
     {
         InSequence seq;
         EXPECT_CALL(mutex, lock);
         EXPECT_CALL(vector, push_back(test_int));
         EXPECT_CALL(mutex, unlock);
     }
    stack.push(test_int);
}
```

Correct synchronization sequence is verified

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<MutexP>(mutex)}
    {}
    synchronized_stack() = default;
    //...
    void push(T const& v)
    {
        std::scoped_lock lock{_mutex};
        _container.push_back(v);
    }
private:
    Container _container{};
    Mutex _mutex{};
};
```

Lock should have a name otherwise will be “immediately” destroyed, unlocking the mutex

Compile time polymorphism. Applications

- Synchronized stack:

```
template <typename T, stack_like_accessible Container = std::vector<T>, lockable Mutex = std::mutex>
class synchronized_stack {
public:
    template <stack_like_accessible ContainerP, lockable MutexP>
    synchronized_stack(ContainerP&& container, MutexP&& mutex)
        : _container{std::forward<ContainerP>(container)}, _mutex{std::forward<MutexP>(mutex)}
    {}
    synchronized_stack() = default;
    //...
    void push(T const& v)
    {
        std::scoped_lock lock{_mutex};
        _container.push_back(v);
    }
private:
    Container _container{};
    Mutex _mutex{};
};
```

Default constructor without dependencies. Default types are used

Compile time polymorphism. Applications

- Production code example:

```
synchronized_stack<int> stack{};
stack.push(100);
// or
std::vector<int> vector;
std::mutex mutex;
synchronized_stack stack_adapter{vector, mutex};
```

Compile time polymorphism. Applications

- Production code:

```
synchronized_stack<int> stack{};
stack.push(100);
// or
std::vector<int> vector;
std::mutex mutex;
synchronized_stack stack_adapter{vector, mutex};
```

Only value type is specified.

Uses (owns) dependencies of default types

(`std::vector` and `std::mutex`)

Compile time polymorphism. Applications

- Production code example:

```
synchronized_stack<int> stack{};
stack.push(100);
// or
std::vector<int> vector;
std::mutex mutex;
synchronized_stack stack_adapter{vector, mutex};
```

Or accesses `vector` and `mutex` by reference

Dependency injection. Summary

- Behaviour of tooling differs: tooling used in the project should be considered
- Optimization for space is expected to reduce inlining
- Function optimization is likely to be inhibited by indirection
- Compile time solutions do not introduce indirection and are unlikely to block compiler optimization effort
- Compile-time injection is preferred for decomposition of the logic on “hot path”

Low-level communication. Link layer

- LLIO interface links board to the HMI
- Link layer access API belongs to core components and is offered by the Platform

Low-level communication. Link layer

- Functionality is provided by the Platform API:

```
typedef struct {
    uint8_t *ptr;
    size_t size;
} frame_descriptor;

typedef frame_descriptor *frame_descriptor_ptr;

enum link_layer_error {
    NO_ERROR = 0,
    DEVICE_BUSY = -5,
    FRAME_CORRUPTED = -10,
};

int allocate_frame(size_t size, frame_descriptor_ptr *mem_dscr);
int commit_frame(frame_descriptor_ptr dscr);
void release_frame(frame_descriptor_ptr dscr);
```

Low-level communication. Link layer

- Functionality is provided by the Platform API:

```
typedef struct {
    uint8_t *ptr;
    size_t size;
} frame_descriptor;

typedef frame_descriptor *frame_descriptor_ptr;

enum link_layer_error {
    NO_ERROR = 0,
    DEVICE_BUSY = -5,
    FRAME_CORRUPTED = -10,
};

int allocate_frame(size_t size, frame_descriptor_ptr *mem_dscr);
int commit_frame(frame_descriptor_ptr dscr);
void release_frame(frame_descriptor_ptr dscr);
```

Frame for data transmission

Low-level communication. Link layer

- Functionality is provided by the Platform API:

```c
typedef struct {
    uint8_t *ptr;
    size_t size;
} frame_descriptor;

typedef frame_descriptor *frame_descriptor_ptr;

enum link_layer_error {
    NO_ERROR = 0,
    DEVICE_BUSY = -5,
    FRAME_CORRUPTED = -10,
};

int allocate_frame(size_t size, frame_descriptor_ptr *mem_dscr);
int commit_frame(frame_descriptor_ptr dscr);
void release_frame(frame_descriptor_ptr dscr);
```

Frame should be acquired

Low-level communication. Link layer

- Functionality is provided by the Platform API:

```
typedef struct {
    uint8_t *ptr;
    size_t size;
} frame_descriptor;

typedef frame_descriptor *frame_descriptor_ptr;

enum link_layer_error {
    NO_ERROR = 0,
    DEVICE_BUSY = -5,
    FRAME_CORRUPTED = -10,
};

int allocate_frame(size_t size, frame_descriptor_ptr *mem_dscr);
int commit_frame(frame_descriptor_ptr dscr);
void release_frame(frame_descriptor_ptr dscr);
```

And then either committed or released

Low-level communication. Link layer

- To provide safe resource management RAII wrapper can be introduced
- Code will be based on specific API contract
- Several test strategies are available: link time substitution, compile time injection

Link layer wrapper. Test cases

- Test for release on scope exit:

```
TEST(test_link_layer, acquire_frame_no_commit)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, release_frame(&descriptor));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);
    EXPECT_THAT(frame->data(), SpanEq(std::span{address, size}));
    EXPECT_THAT(frame->as_buffer(), SpanEq(std::span{address, size}));
}
```

Link layer wrapper. Test cases

- Test for release on scope exit:

```
TEST(test_link_layer, acquire_frame_no_commit)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, release_frame(&descriptor));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);
    EXPECT_THAT(frame->data(), SpanEq(std::span{address, size}));
    EXPECT_THAT(frame->as_buffer(), SpanEq(std::span{address, size}));
}
```

Link layer wrapper with injected mock configuration. Returns frame wrapper

Link layer wrapper. Test cases

- Test for release on scope exit:

```
TEST(test_link_layer, acquire_frame_no_commit)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, release_frame(&descriptor));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);
    EXPECT_THAT(frame->data(), SpanEq(std::span{address, size}));
    EXPECT_THAT(frame->as_buffer(), SpanEq(std::span{address, size}));
}
```

Returned frame span matches the parameters returned by system API mock

Link layer wrapper. Test cases

- Test for release on scope exit:

```
TEST(test_link_layer, acquire_frame_no_commit)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, release_frame(&descriptor));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);
    EXPECT_THAT(frame->data(), SpanEq(std::span{address, size}));
    EXPECT_THAT(frame->as_buffer(), SpanEq(std::span{address, size}));
}
```

No commit, so release should be called

Link layer wrapper. Test cases

- Test for commit:

```
TEST(test_link_layer, commit_no_error)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, commit_frame(&descriptor)).WillOnce(Return(NO_ERROR));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);

    auto const commit_result = layer.commit_frame(std::move(*frame));
    ASSERT_TRUE(commit_result);
}
```

Link layer wrapper. Test cases

- Test for release on scope exit:

```
TEST(test_link_layer, commit_no_error)
{
    constexpr size_t size{100};
    uint8_t *const address{reinterpret_cast<uint8_t *>(200)};
    frame_descriptor descriptor{address, size};

    StrictMock<mocks::config> config;

    EXPECT_CALL(config, allocate_frame(size, _))
        .WillOnce(DoAll(SetArgPointee<1>(&descriptor), Return(NO_ERROR)));
    EXPECT_CALL(config, commit_frame(&descriptor)).WillOnce(Return(NO_ERROR));

    link_layer layer{config};
    auto frame = layer.acquire_frame(size);
    ASSERT_TRUE(frame);

    auto const commit_result = layer.commit_frame(std::move(*frame));
    ASSERT_TRUE(commit_result);
}
```

Commit has expected corresponding platform call. No `release_frame call` expected

Link layer wrapper. Implementation

- Implementation for the `acquire_frame` method:

```
auto acquire_frame(size_t const size) noexcept -> nonstd::expected<frame_holder_t, error_t>
{
    frame_descriptor_ptr descriptor{};
    if (auto const result = _config.allocate_message(size, &descriptor); result != NO_ERROR) {
        return nonstd::make_unexpected(error_t{result});
    }
    return frame_holder_t{_config, descriptor};
}
```

Link layer wrapper. Generalization

- Possible generalization for the method (if required):

```
auto acquire_frame(size_t const size) noexcept -> nonstd::expected<frame_holder_t, error_t>
{
    typename config_t::frame_descriptor_ptr_t descriptor{};
    if (auto const result = _config.allocate_message(size, &descriptor); !_config.is_success(result)) {
        return nonstd::make_unexpected(_config.to_error(result));
    }
    return frame_holder_t{_config, descriptor};
}
```

Types are “injected”

Link layer wrapper. Generalization

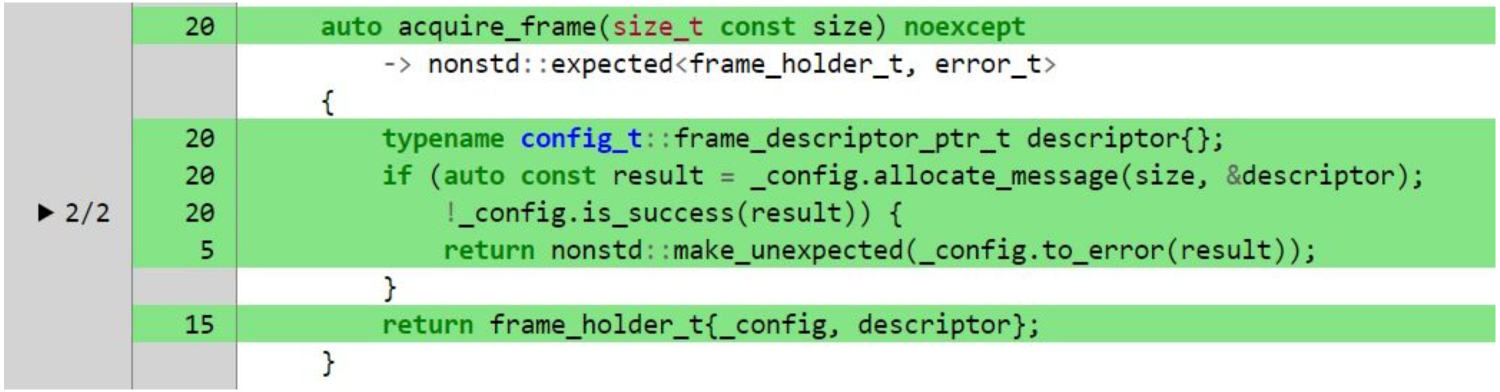
- Possible generalization for the method (if required):

```
auto acquire_frame(size_t const size) noexcept -> nonstd::expected<frame_holder_t, error_t>
{
    typename config_t::frame_descriptor_ptr_t descriptor{};
    if (auto const result = _config.allocate_message(size, &descriptor); !_config.is_success(result)) {
        return nonstd::make_unexpected(_config.to_error(result));
    }
    return frame_holder_t{_config, descriptor};
}
```

Error handling is defined by injected configuration. Should be mocked in tests and separately verified for production code

Link layer wrapper

- Method coverage:



```
          20   auto acquire_frame(size_t const size) noexcept
                   -> nonstd::expected<frame_holder_t, error_t>
               {
          20       typename config_t::frame_descriptor_ptr_t descriptor{};
          20       if (auto const result = _config.allocate_message(size, &descriptor);
▶ 2/2     20           !_config.is_success(result)) {
           5           return nonstd::make_unexpected(_config.to_error(result));
                   }
          15       return frame_holder_t{_config, descriptor};
               }
```

Project plan. Solution breakdown

- ☐ Transport stack
    - ☑ Message encoding
    - ☑ Timestamp evaluation
    - ☑ Link layer access
    - ☐ Transmission functionality
- ☐ Peripheral controls
- ☐ Application logic

Transport stack. Primary logic

- Transport stack is responsible of orchestrating message preparation and transmission

Transport stack implementation

- Switch from initial function:

```
using data_t = std::span<uint8_t const>;
enum class result_t {
    ok,
};

auto send_message(data_t const &data) -> result_t;
```

- To the class that holds dependencies

```
template <
    typename LinkLayer,
    typename MessageBuilder,
    typename TimestampProvider>
class message_sender final {
public:
    auto send(data_t const &data) noexcept -> nonstd::expected<void, error_t> {...}
//...
};
```

Transport stack implementation

- Constraints for class dependencies:

```
template <
    link_layer_like LinkLayer,
    message_builder_like MessageBuilder,
    timestamp_provider_like TimestampProvider>
class message_sender final {
public:
    auto send(data_t const &data) noexcept -> nonstd::expected<void, error_t> {...}
//...
};
```

Dependencies can also be constrained using concepts

Transport stack implementation

- Fixture containing test data and mocks

```
class test_message_sender : public test_send_message {
protected:
    std::array<uint8_t, 2> const data{0x10, 0x20};
    size_t const data_size{2};
    size_t const message_size{4};
    timestamp_t const timestamp{1000};

    StrictMock<mocks::link_layer> link_layer;
    StrictMock<mocks::timestamp_provider> timestamp_provider;
    StrictMock<mocks::message_builder> message_builder;
    StrictMock<mocks::frame_holder> frame{};
};
```

Transport stack implementation

- Fixture containing test data and mocks

```
class test_message_sender : public test_send_message {
protected:
    std::array<uint8_t, 2> const data{0x10, 0x20};
    size_t const data_size{2};
    size_t const message_size{4};
    timestamp_t const timestamp{1000};

    StrictMock<mocks::link_layer> link_layer;
    StrictMock<mocks::timestamp_provider> timestamp_provider;
    StrictMock<mocks::message_builder> message_builder;
    StrictMock<mocks::frame_holder> frame{};
};
```

Mocks for dependencies

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Times(2)
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_t::some_error)))
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t>{}));

    transport::message_sender message_sender{link_layer, message_builder, timestamp_provider};
    EXPECT_TRUE(message_sender.send(data));
}
```

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Times(2)
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_t::s
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t>{})

    transport::message_sender message_sender{link_layer, message_builder, time
    EXPECT_TRUE(message_sender.send(data));
}
```

Message size calculated for data size is correctly forwarded to the frame request

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Times(2)
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_t::some_
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t>{}));

    transport::message_sender message_sender{link_layer, message_builder, timestamp_provider};
    EXPECT_TRUE(message_sender.send(data));
}
```

Buffer associated with mocked frame is used to place message

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Ti
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t

    transport::message_sender message_sender{link_layer, message_builder, timestamp_provider};
    EXPECT_TRUE(message_sender.send(data));
}
```

Timestamp retrieved from timestamp provider is forwarded to the message builder

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Times(2)
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_t::some_error)))
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t>{}));

    transport::message_sender message_sender{link_layer, message_builder, timestamp_p
    EXPECT_TRUE(message_sender.send(data));
}
```

Frame is acquired and committed

Transport stack. Unit tests

- Error recovery test. Only primary logic is verified:

```
TEST_F(test_message_sender, recoverd_from_error_mesage_sent)
{
    buffer_t const buffer{reinterpret_cast<uint8_t *>(0x100), 300};

    EXPECT_CALL(frame, as_buffer).WillOnce(Return(buffer));
    EXPECT_CALL(timestamp_provider, timestamp).WillOnce(Return(timestamp));
    EXPECT_CALL(message_builder, calculate_message_size(data_size)).WillOnce(Return(message_size));
    EXPECT_CALL(link_layer, acquire_frame(message_size)).WillOnce(Return(mocks::frame_holder_proxy{frame}));
    EXPECT_CALL(message_builder, assemble_message(SpanEq(buffer), SpanEq(std::span{data}), timestamp))
        .WillOnce(Return(buffer));
    EXPECT_CALL(link_layer, commit_frame(FrameHolderAddressEq(&frame))).Times(2)
        .WillOnce(Return(nonstd::make_unexpected(mocks::link_layer::error_t::some_error)))
        .WillOnce(Return(nonstd::expected<void, mocks::link_layer::error_t>{}));

    transport::message_sender message_sender{link_layer, message_builder, timestamp
    EXPECT_TRUE(message_sender.send(data));
}
```

First commit fails but second is successful after retry

Project plan. Solution breakdown

- ☑ Transport stack
    - ☑ Message encoding
    - ☑ Timestamp evaluation
    - ☑ Link layer access
    - ☑ Transmission functionality
- ☐ Peripheral controls
- ☐ Application logic

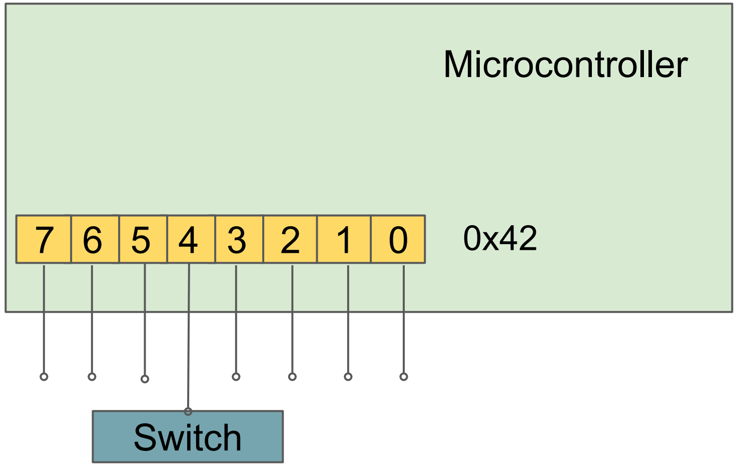
TDD and compile time injection

- Enables top-down approach for decomposition with dependency inversion (concepts define “interfaces”)
- Enables bottom-up approach: no need to rewrite existing classes to be injected
- Decomposition without runtime overhead can be achieved

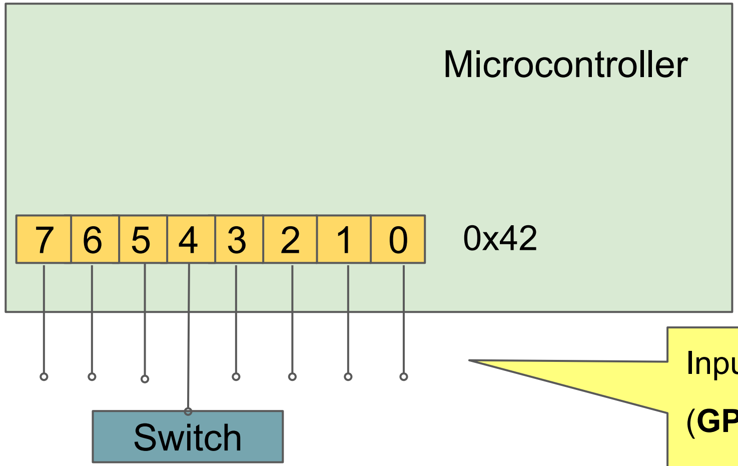
TDD of hardware dependent code

- Typically, on embedded platforms, interaction with hardware is performed using MMIO (Memory Mapped Input/Output)
- Device drivers are software modules controlling hardware using MMIO and Interrupt mechanism

GPIO/MMIO layout



GPIO/MMIO layout

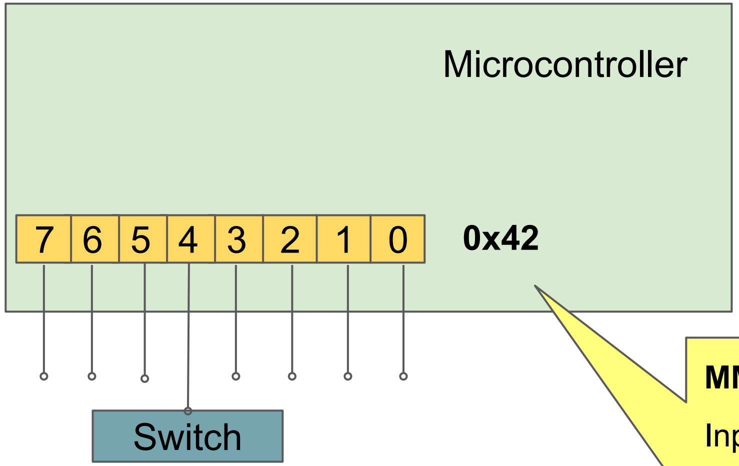


Input/output lines

(**GPIO**-General-Purpose

Input/Output)

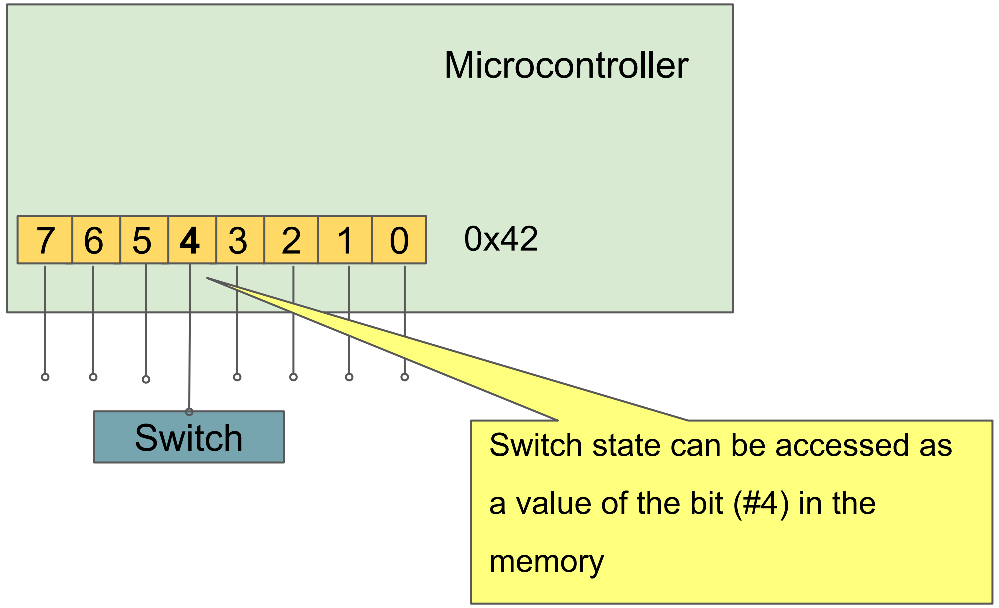
GPIO/MMIO layout



MMIO-Memory Mapped Input/Output

Line state is controlled via a bit in a memory cell with specified address (0x42)

GPIO/MMIO layout


Microcontroller
7
6
5
4
3
2
1
0
0x42
Switch
Switch state can be accessed as a value of the bit (#4) in the memory

Hardware dependent code unit testing strategy

- Develop (or use existing) verified abstractions encapsulating MMIO operations
- Generate hardware layout using abstractions from headers/architectural documents
- Use compile time dependency injection for TDD of a production code

MMIO wrappers

- Concept for the memory location:

```cpp
template <typename T>
concept location_like = std::unsigned_integral<typename T::type> && requires {
    { T::pointer() } -> std::same_as<typename T::type volatile *>;
};
```

- Implementation of the MMIO read operation:

```cpp
template <location_like Location, uint8_t Offset>
struct ro_bit {
    using type = typename Location::type;
    static_assert(Offset < std::numeric_limits<type>::digits);

    auto is_set() const noexcept -> bool
    {
        constexpr type value_mask = type{1} << Offset;
        return (*Location::pointer() & value_mask) == value_mask;
    }
};
```

MMIO wrappers

- Concept for the memory location:

```
template <typename T>
concept location_like = std::unsigned_integral<typename T::type> && requires {
    { T::pointer() } -> std::same_as<typename T::type volatile *>;
};
```

Conformant type provides pointer to an unsigned integral object of type T

- Implementation of the MMIO read operation:

```
template <location_like Location, uint8_t Offset>
struct ro_bit {
    using type = typename Location::type;
    static_assert(Offset < std::numeric_limits<type>::digits);

    auto is_set() const noexcept -> bool
    {
        constexpr type value_mask = type{1} << Offset;
        return (*Location::pointer() & value_mask) == value_mask;
    }
};
```

MMIO wrappers

- Concept for the memory location:

```
template <typename T>
concept location_like = std::unsigned_integral<typename T::type> && requires {
    { T::pointer() } -> std::same_as<typename T::type volatile *>;
};
```

- Implementation of the MMIO read operation:

```
template <location_like Location, uint8_t Offset>
struct ro_bit {
    using type = typename Location::type;
    static_assert(Offset < std::numeric_limits<type>::digits);

    auto is_set() const noexcept -> bool
    {
        constexpr type value_mask = type{1} << Offset;
        return (*Location::pointer() & value_mask) == value_mask;
    }
};
```

Compile time check of the bit offset bound

MMIO wrappers

- Concept for the memory location:

```
template <typename T>
concept location_like = std::unsigned_integral<typename T::type> && requires {
    { T::pointer() } -> std::same_as<typename T::type volatile *>;
};
```

- Implementation of the MMIO read operation:

```
template <location_like Location, uint8_t Offset>
struct ro_bit {
    using type = typename Location::type;
    static_assert(Offset < std::numeric_limits<type>::digits);

    auto is_set() const noexcept -> bool
    {
        constexpr type value_mask = type{1} << Offset;
        return (*Location::pointer() & value_mask) == value_mask;
    }
};
```

Bit operation with dereferenced memory block provided by location_like object

MMIO wrappers. Verification

- For testing of the facilities test location can be used:

```
template <auto* Address>
struct location {
    using type = std::remove_pointer_t<decltype(Address)>;
    static_assert(std::unsigned_integral<type>);

    static auto pointer() noexcept -> type* { return Address; }
};
```

- Test verifies memory access operation correctness:

```
TEST(test_mmio_bit, is_set_bit_is_set)
{
    static uint8_t const volatile object = 0b0000'1000;

    using test_bit = mmio::ro_bit<mocks::location<&object>, 3>;
    test_bit const bit{};
    EXPECT_TRUE(bit.is_set());
}
```

MMIO wrappers. Verification

- For testing of the facilities test location can be used:

```
template <auto* Address>
struct location {
    using type = std::remove_pointer_t<decltype(Address)>;
    static_assert(std::unsigned_integral<type>);

    static auto pointer() noexcept -> type* { return Address; }
};
```

Address of an object of integral unsigned type as non-type template parameter (`auto` is C++20)

- Test verifies memory access operation correctness:

```
TEST(test_mmio_bit, is_set_bit_is_set)
{
    static uint8_t const volatile object = 0b0000'1000;

    using test_bit = mmio::ro_bit<mocks::location<&object>, 3>;
    test_bit const bit{};
    EXPECT_TRUE(bit.is_set());
}
```

MMIO wrappers. Verification

- For testing of the facilities test location can be used:

```
template <auto* Address>
struct location {
    using type = std::remove_pointer_t<decltype(Address)>;
    static_assert(std::unsigned_integral<type>);

    static auto pointer() noexcept -> type* { return Address; }
};
```

- Test verifies memory access operation correctness:

```
TEST(test_mmio_bit, is_set_bit_is_set)
{
    static uint8_t const volatile object = 0b0000'1000;

    using test_bit = mmio::ro_bit<mocks::location<&object>, 3>;
    test_bit const bit{};
    EXPECT_TRUE(bit.is_set());
}
```

Can be used with objects with static storage duration

MMIO wrappers. Production application

- For production, location object pointing to MMIO registers is used:

```
template <std::unsigned_integral T, uintptr_t Address>
struct location {
    using type = T;
    static T volatile *pointer() noexcept
    {
        return reinterpret_cast<T volatile *>(Address);
    }
};
```

- MMIO wrappers can represent various peripheral devices:

```
using switch_gpio_line = mmio::ro_bit<mmio::location<uint8_t, 0x42>, 4>;
```

MMIO wrappers. Production application

- For production, location object pointing to MMIO registers is used:

```
template <std::unsigned_integral T, uintptr_t Address>
struct location {
    using type = T;
    static T volatile *pointer() noexcept
    {
        return reinterpret_cast<T volatile *>(Address);
    }
};
```

Address can be provided as literal

- MMIO wrappers can represent various peripheral devices:

```
using switch_gpio_line = mmio::ro_bit<mmio::location<uint8_t, 0x42>, 4>;
```

MMIO wrappers. Production application

- For production, location object pointing to MMIO registers is used:

```
template <std::unsigned_integral T, uintptr_t Address>
struct location {
    using type = T;
    static T volatile *pointer() noexcept
    {
        return reinterpret_cast<T volatile *>(Address);
    }
};
```

Reinterprets address as a pointer to T

- MMIO wrappers can represent various peripheral devices:

```
using switch_gpio_line = mmio::ro_bit<mmio::location<uint8_t, 0x42>, 4>;
```

MMIO wrappers. Production application

- For production, location object pointing to MMIO registers is used:

```
template <std::unsigned_integral T, uintptr_t Address>
struct location {
    using type = T;
    static T volatile *pointer() noexcept
    {
        return reinterpret_cast<T volatile *>(Address);
    }
};
```

- MMIO wrappers can represent various peripheral devices:

```
using switch_gpio_line = mmio::ro_bit<mmio::location<uint8_t, 0x42>, 4>;
```

Alias, preferably generated, from documentation or SDK headers. Address, as literal, according to MMIO specification.

Switch driver

- Concept for the MMIO bit representation:

```
template <typename MmioBit>
concept ro_bit = requires(MmioBit bit) {
    { bit.is_set() } -> std::same_as<bool>;
};
```

- Implementation of the trivial driver:

```
template <ro_bit StateBit>
class polling_driver {
public:
    polling_driver(StateBit const& state_bit) : _state_bit{state_bit} {}

    auto pressed() const -> bool { return _state_bit.is_set(); }

private:
    StateBit const& _state_bit;
};
```

Switch driver

- Concept for the MMIO bit representation:

```
template <typename MmioBit>
concept ro_bit = requires(MmioBit bit) {
    { bit.is_set() } -> std::same_as<bool>;
};
```

Needs to have `is_set` method

- Implementation of the trivial driver:

```
template <ro_bit StateBit>
class polling_driver {
public:
    polling_driver(StateBit const& state_bit) : _state_bit{state_bit} {}

    auto pressed() const -> bool { return _state_bit.is_set(); }

private:
    StateBit const& _state_bit;
};
```

Switch driver

- Concept for the MMIO bit representation:

```
template <typename MmioBit>
concept ro_bit = requires(MmioBit bit) {
    { bit.is_set() } -> std::same_as<bool>;
};
```

- Implementation of the trivial driver:

```
template <ro_bit StateBit>
class polling_driver {
public:
    polling_driver(StateBit const& state_bit) : _state_bit{state_bit} {}

    auto pressed() const -> bool { return _state_bit.is_set(); }

private:
    StateBit const& _state_bit;
};
```

Driver encapsulates access to MMIO object

Switch driver. Unit test

- Mock MMIO object injected for tests:

```
namespace mocks {
struct mmio_bit {
    MOCK_METHOD(bool, is_set, (), (const));
};
static_assert(ro_bit<mmio_bit>);
}  // namespace mocks

TEST(test_switch_driver, polling_toggle)
{
    StrictMock<mocks::mmio_bit> const state_bit;
    EXPECT_CALL(state_bit, is_set).WillOnce(Return(false)).WillOnce(Return(true));

    polling_driver const test_switch{state_bit};
    EXPECT_FALSE(test_switch.pressed());
    EXPECT_TRUE(test_switch.pressed());
}
```

Mock MMIO object used for testing

Interrupt driven switch driver

- Concept for event handler:

```
template <typename Function>
concept event_handler_like = concepts::invocable_exact_r<Function, void, bool>;
```

- Implementation of the driver:

```
template <ro_bit StateBit, event_handler_like EventHandler>
class isr_driver {
public:
    template <typename Handler>
    isr_driver(StateBit const& state_bit, Handler&& handler)
        : _state_bit{state_bit}, _event_handler{std::forward<Handler>(handler)} {}

    auto isr() -> void { _event_handler(_state_bit.is_set()); }

private:
    StateBit const& _state_bit;
    EventHandler const _event_handler;
};
```

Interrupt driven switch driver

- Concept for event handler:

```
template <typename Function>
concept event_handler_like = concepts::invocable_exact_r<Function, void, bool>;
```

Concept of the user event handler function

- Implementation of the driver:

```
template <ro_bit StateBit, event_handler_like EventHandler>
class isr_driver {
public:
    template <typename Handler>
    isr_driver(StateBit const& state_bit, Handler&& handler)
        : _state_bit{state_bit}, _event_handler{std::forward<Handler>(handler)} {}

    auto isr() -> void { _event_handler(_state_bit.is_set()); }

private:
    StateBit const& _state_bit;
    EventHandler const _event_handler;
};
```

Interrupt driven switch driver

- Concept for event handler:

```
template <typename Function>
concept event_handler_like = concepts::invocable_exact_r<Function, void, bool>;
```

- Implementation of the driver:

```
template <ro_bit StateBit, event_handler_like EventHandler>
class isr_driver {
public:
    template <typename Handler>
    isr_driver(StateBit const& state_bit, Handler&& handler)
        : _state_bit{state_bit}, _event_handler{std::forward<Handler>(handler)} {}

    auto isr() -> void { _event_handler(_state_bit.is_set()); }
private:
    StateBit const& _state_bit;
    EventHandler const _event_handler;
};
```

Callable as a handler forwarded in constructor

Interrupt driven switch driver

- Concept for event handler:

```
template <typename Function>
concept event_handler_like = concepts::invocable_exact_r<Function, void, bool>;
```

- Implementation of the driver:

```
template <ro_bit StateBit, event_handler_like EventHandler>
class isr_driver {
public:
    template <typename Handler>
    isr_driver(StateBit const& state_bit, Handler&& handler)
        : _state_bit{state_bit}, _event_handler{std::forward<Handler>(handle

    auto isr() -> void { _event_handler(_state_bit.is_set()); }

private:
    StateBit const& _state_bit;
    EventHandler const _event_handler;
};
```

Function to be called from interrupt service routine (ISR) registered for interrupt.
Logic to retrieve the state is encapsulated in the driver.
Trivial example that forwards the state to a custom handler.

Interrupt driven switch driver

- Testing of the event propagation:

```
namespace mocks {
struct event_handler {
    MOCK_METHOD(void, call, (bool));
    auto operator()(bool const value) { return call(value); }
};
}  // namespace mocks

TEST(test_switch_driver, isr_switch_off)
{
    StrictMock<mocks::mmio_bit> const state_bit;
    EXPECT_CALL(state_bit, is_set).WillOnce(Return(false));
    StrictMock<mocks::event_handler> event_handler;
    EXPECT_CALL(event_handler, call(false));

    isr_driver test_switch{state_bit, event_handler};
    test_switch.isr();
}
```

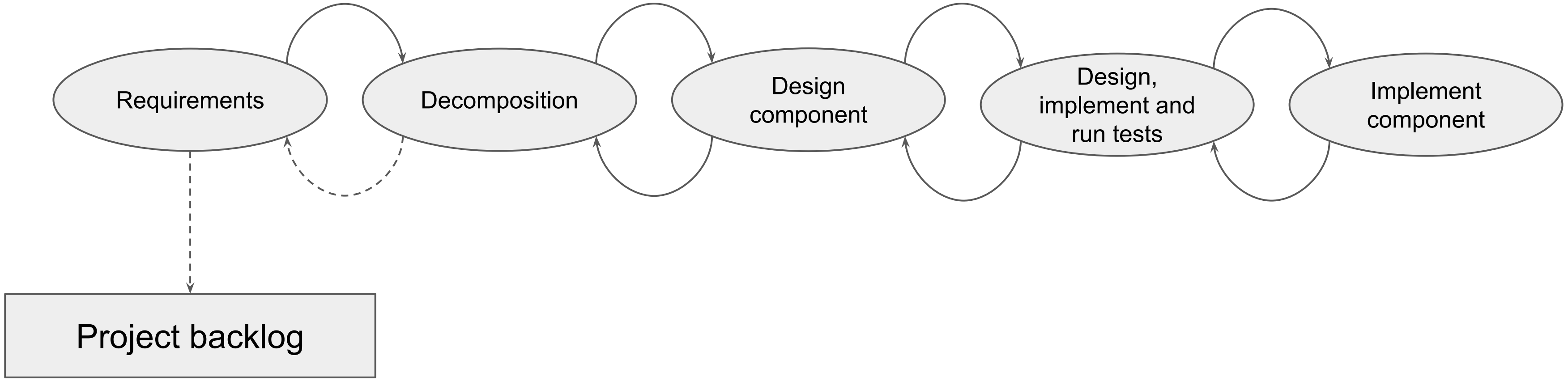
Test triggers the ISR function to verify interrupt handling

Project plan. Solution breakdown

- ✅ Transport stack
    - ✅ Message encoding
    - ✅ Timestamp evaluation
    - ✅ Link layer access
    - ✅ Transmission functionality
- ✅ Peripheral controls
- ☐ Application logic

Unit testing and decomposition. Process overview

- In practice, decomposition becomes prerequisite for testable unit design
- Decomposition can trigger requirements update/refinement
- Requirements for new components can be introduced into backlog



TDD with unit testing outcome

- As the result of the process application:
  - Usability of the proposed interfaces and architecture are verified
  - Unclarities/issues in requirements are clarified/fixed
  - Requirements for dependencies are elaborated and tested
  - Implementation is verified and supplemented with tests
  - Tests become a part of a CI process
  - Tests act as documentation and usage examples

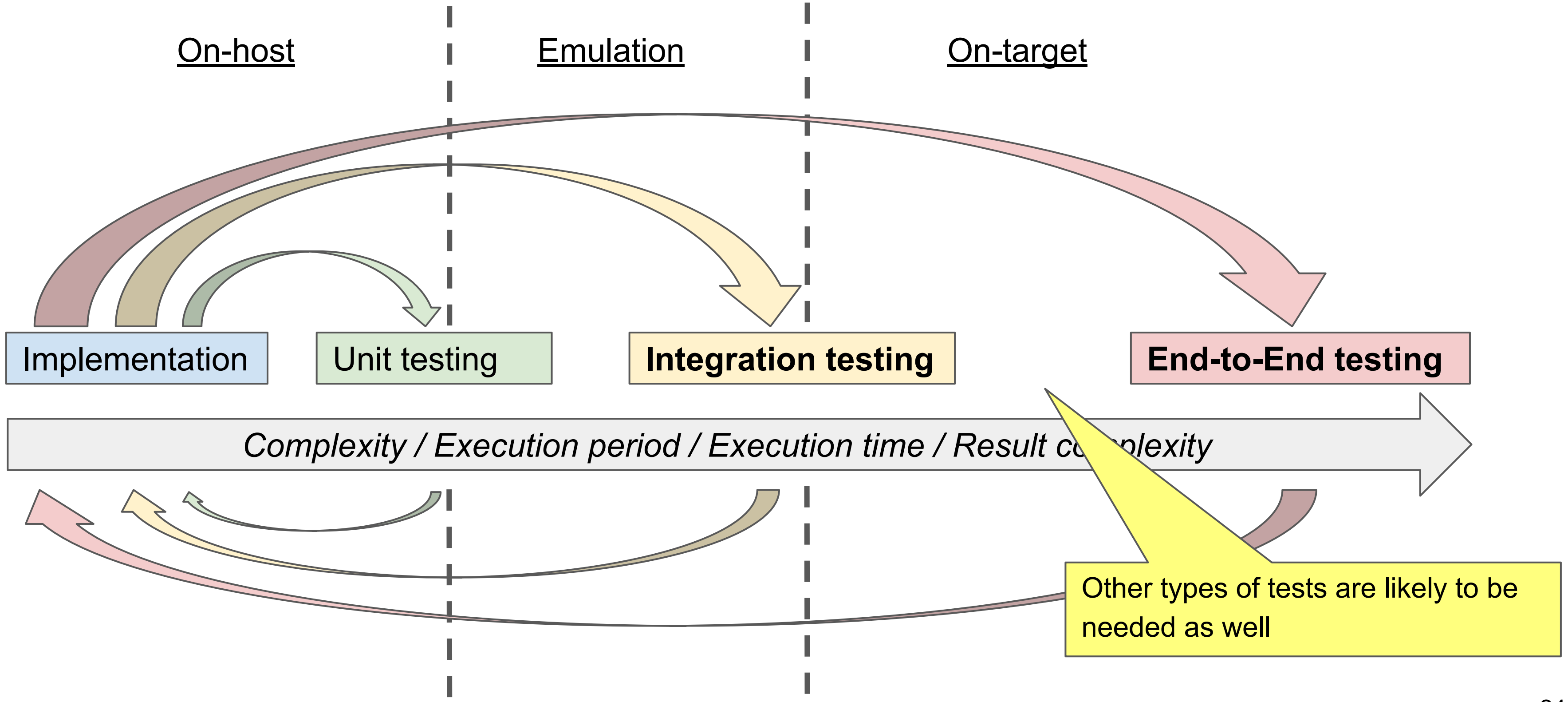
- Further development/modification should follow TDD by extending/modifying existing test-set
- Verified code is change friendly

Unit testing sufficiency

- Unit testing can be affected by the following potential issues:
  - Incorrect assumptions or understanding of specifications (of the module itself or dependencies)
  - Defects in test implementation, incorrect behaviour of test-doubles
  - Portability issues in code and tooling in case of cross platform development

- Solution typically consists of multiple modules and designed to be deployed on specific platform(s). It is required to verify solution accordingly

Typical TDD cycles for cross platform development



Other types of tests are likely to be needed as well

Unit testing sufficiency

- If unit testing can have flaws is it needed?
  - Test diversity increases confidence
  - Potential issues do not invalidate other advantages (short execution time, error localization, portability)
  - Unit testing efficiently drives development of components while integration and end-to-end tests can require complex software and sophisticated test doubles, shifting availability to later phases of development cycle

Production integration

- Class assembles standalone functions:

```
struct message_builder {
    using error_t = transport::message::error_t;
    FORWARD_CALL(assemble_message, transport::message::assemble_message);
    FORWARD_CALL(calculate_message_size, ::transport::message::calculate_message_size);
};
```

Production integration

- Class assembles standalone functions:

```
struct message_builder {
    using error_t = transport::message::error_t;
    FORWARD_CALL(assemble_message, transport::message::assemble_message);
    FORWARD_CALL(calculate_message_size, transport::message::calculate_message_size);
};
```

Functions will be injected into `message_sender`

Production integration

- Assembling components together:

```
auto main(int /*argc*/, char ** /*argv*/) -> int
{
    using switch_gpio_line = mmio::ro_bit<mmio::location<uint8_t, 0x42>, 4>;
    switch_gpio_line const gpio_line{};
    switch_driver::polling_driver const driver{gpio_line};

    llio::link_layer link_layer{link_layer_config_t{}};
    transport::timestamp_provider const timestamp_provider{timestamp_provider_config{}, true};
    transport::message_sender message_sender{link_layer, message_builder{}, timestamp_provider};

    application::switch_monitor monitor{driver, message_sender};

    while (true) {
        monitor.loop_once();
    }

    return 0;
}
```

Project plan. Solution breakdown

- ✅ Transport stack
    - ✅ Message encoding
    - ✅ Timestamp evaluation
    - ✅ Link layer access
    - ✅ Transmission functionality
- ✅ Peripheral controls
- ✅ Application logic

Thank you!

Questions?