Lessons Learned for Reusable Firmware

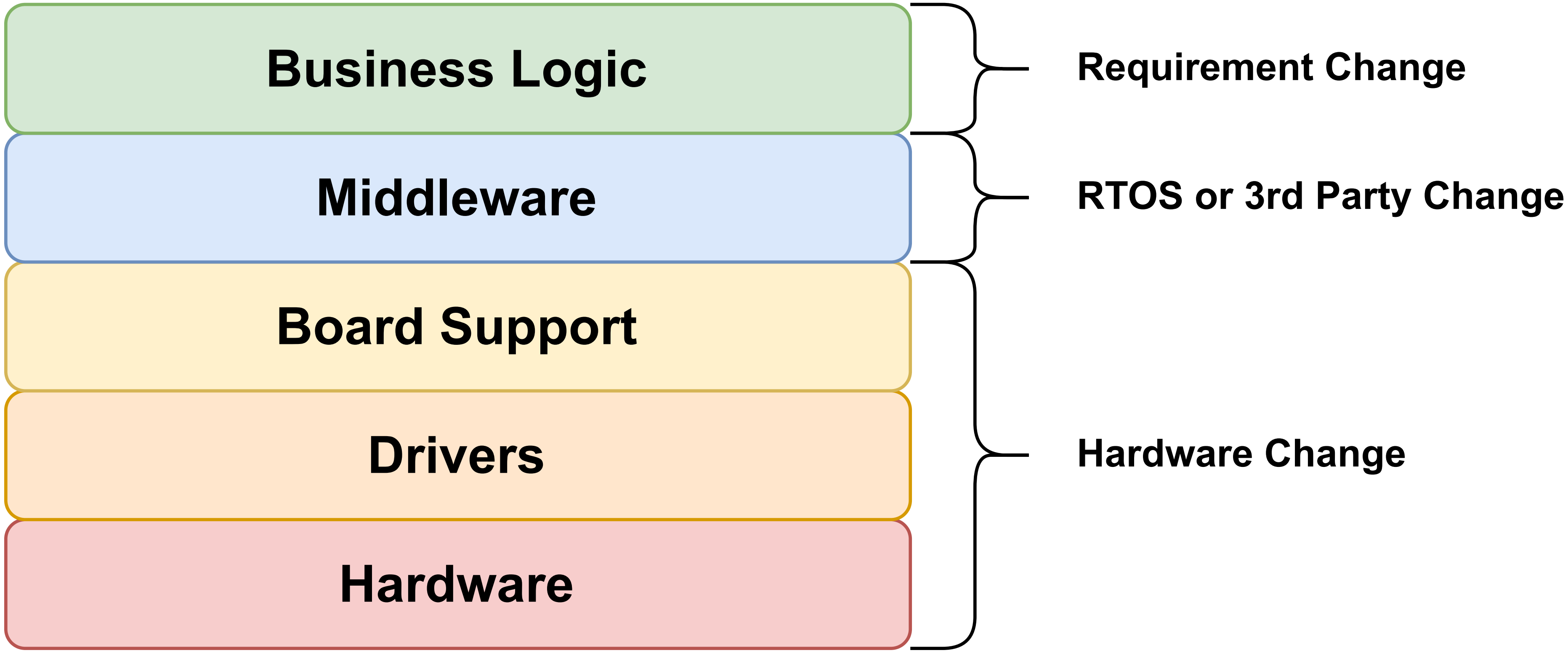
Tobias Zindl


PURE SURVEYING
Leica
mergeTEC

Complexity

- Complexity increases with each generation
- Collaboration necessary
- Integration necessary

Reasons for Changes



Reasons for Changes

Reuse vs. Mutation

original

```
/*...*/
sensor::sensor(spi_driver& spi, /*...*/)
    : /*...*/, m_reset(gpio::port_e, 1, gpio::digital_output) {}
/*...*
```

mutation

```
/*...*/
sensor::sensor(spi_driver& spi, /*...*/)
    : /*...*/, m_reset(gpio::port_e, 2, gpio::digital_output) {}
/*...*
```

Reasons for Changes

Reuse vs. Mutation

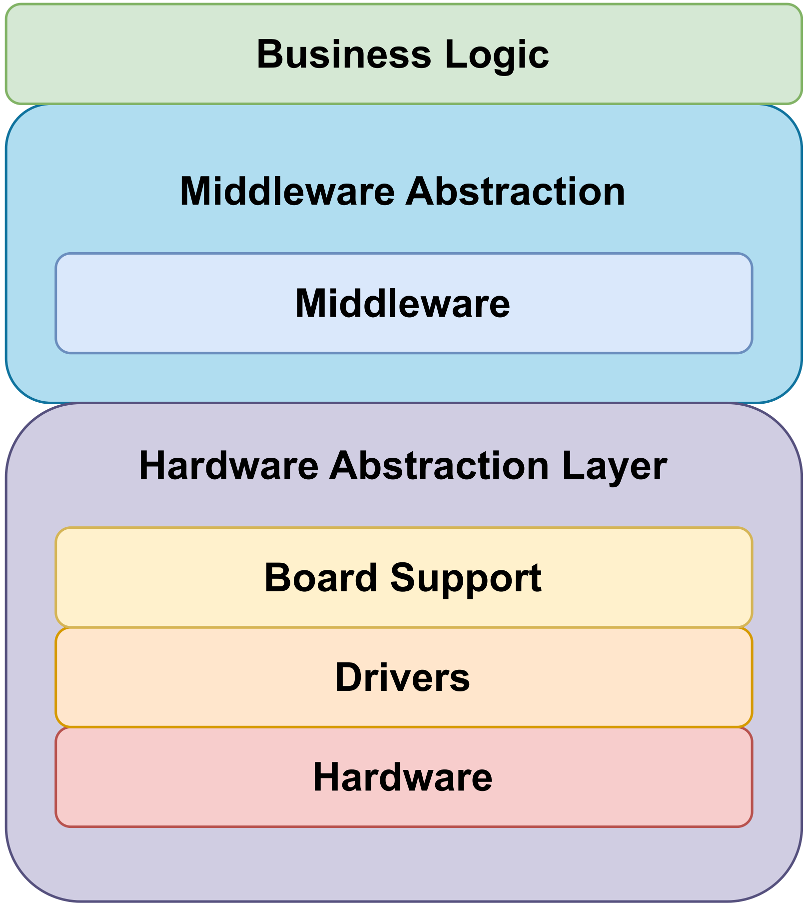
```cpp
// bsp is defined for each board configuration
namespace bsp {
    static constexpr gpio_cfg sensor_reset_pin =
            {gpio::port_e, 2, gpio::digital_output};
}

/*...*/
#include <bsp/pin_cfg.hpp>

sensor::sensor(spi_driver& spi, /*...*/)
    : /*...*/, m_reset(bsp::sensor_reset_pin) {}
/*...*/
```

Reasons for Changes

taking back control



Why reusable Software?

Benefits

- Reuse of functionalities between products
- Management of system complexity
- Reduced development time
- Shorter time to the market

Drawbacks

- Higher initial development time
- Needs to consider "all" projects
  - All development restrictions apply
  - All expectations need to be handled
- Regressions would be a disaster

What needs to be done?

- Create hardware abstraction
- Create middleware abstraction
- Separate business logic components
- Enable separate integration step
- Consider testing for all layers

Hardware Abstraction Layer

Hardware Abstraction Layer

What should it do?

- provide abstraction for all higher layers
- high-level abstraction for hardware access
- provide all needed functions, **should not restrict**
- maintainable and testable
- allow mocking and simulation

Hardware Abstraction Layer

Current Situation

- Existent HALs create dependencies
  - Vendor or hardware specific
  - RTOS specific

- ***To be independent you need your own***

Abstraction & Injection

Runtime abstraction

- Interface abstraction
- Callback injection

Compile time abstraction

- Template abstraction (policies, traits, concepts)
- Header/Linker injection

Hardware Abstraction Layer

base_gpio.hpp

```cpp
template <typename SOC_TYPE>
struct base_gpio {
    using soc_type = SOC_TYPE;
    using device_type = trait::soc_gpio_device_t<SOC_TYPE>;
    using config_type = typename device_type::config_type;

    constexpr base_gpio(config_type cfg) noexcept;
    constexpr const device_type& device() const noexcept;

    inline error_code init(pin_state state) noexcept;
    inline error_code write(pin_state state) noexcept;
    /* ... */
};
```

Hardware Abstraction Layer

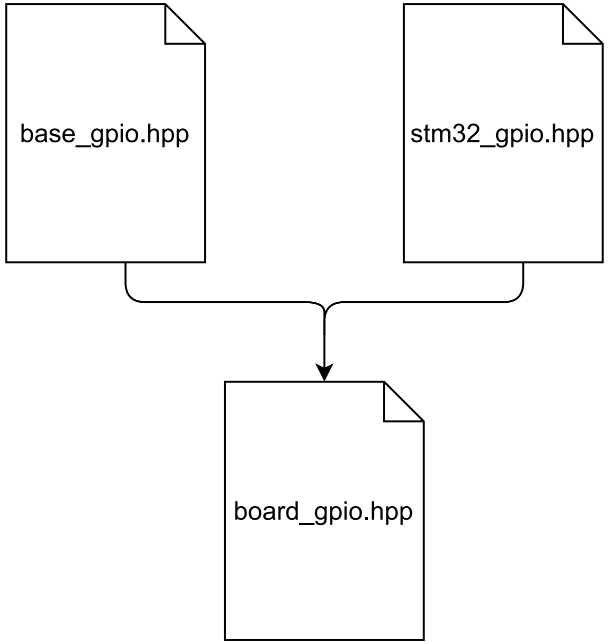
gtest_gpio.hpp

```cpp
template <>
struct base_gpio<tests::gtest> : tests::gtest_base_mock {
    using soc_type = tests::gtest;
    using device_type = trait::soc_gpio_device_t<tests::gtest>;
    using config_type = typename device_type::config_type;

    constexpr base_gpio(config_type cfg) noexcept;
    constexpr const device_type& device() const noexcept;
    /* ... */
};
```

Hardware Abstraction Layer

Board Support



board_gpio.hpp

```cpp
#include <base/base_gpio.hpp>
#include <soc/st/stm32g4.hpp>

/*...*/
using gpio = base_gpio<soc::st::stm32g4>;

// using gpio = base_gpio<soc::nordic::nrf52>;
// using gpio = base_gpio<rtos::zephyr>;
// using gpio = base_gpio<tests::gtest>;
```

Hardware Abstraction Layer

Device Tree

```
pinctrl: pin-controller@48000000 {
    gpioa: gpio@48000000 {
        reg = < 0x48000000 0x400 >;
        # ...
    };
};

leds {
    green_led: led_0 {
        gpios = < &gpioa 0x5 GPIO_ACTIVE_HIGH>;
    };
};
```

```
using gpioa = gpio_ctrl<register<0x48000000 0x400>, /*...*/>;
using green_led_cfg = gpio_cfg<gpioa, 0x05, gpio::active_high>
```

Hardware Abstraction Layer

Summary

- Consider effort
- Choose flexible abstractions
- Hide complexity for the user
- Consider tests and simulations
- Think about code generation

Middleware Abstraction

HEXAGON | Leica Geosystems

Middleware Abstraction

What should it do?

- Enable control for integration
- Needs to be interchangeable
- Needs to enable tests
- Unify APIs and error handling
- Easy to use, hard to misuse
  - ***Lead business logic design***

Middleware Abstraction

Current Situation

- Standards exist
  - Not a good fit for MCUs
- RTOS systems
  - Different APIs & Features
  - Incompatible with Host System

Middleware Abstraction

STL

- Not 3rd Party
- Can be included
  - Define what to use
  - Backporting features

```
namespace mwa {
    template<class R, class P = ::mwa::ratio<1>>
    using duration = std::chrono::duration<R, P>;
}
```

Middleware Abstraction

Thread

Zephyr

Zephyr

```c
k_tid_t k_thread_create(struct k_thread *new_thread,
                        k_thread_stack_t *stack,
                        size_t stack_size,
                        k_thread_entry_t entry,
                        void *p1, void *p2, void *p3,
                        int prio,
                        uint32_t options,
                        k_timeout_t delay);
```

FreeRTOS

FreeRTOS

```c
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask);
```

Middleware Abstraction

Thread

thread_abstraction

```cpp
// definition
thread::thread(const thread_cfg& cfg,
               entry_type entry);
// use
struct component  {
    component(const thread_cfg& cfg) noexcept
        : m_thread(cfg, /*...*/) {};
    /*...*/
    thread m_thread;
};
/*...*/
class system_setup  {
    /*...*/
    thread_cfg m_thread_cfg{stack_cfg(addr, size), thread_prio::low};
    component  m_component{m_thread_cfg};
};
```

Middleware Abstraction

Summary

- Lead business logic design
- Consider Host OS from the beginning
- Take care about behavior relevant parts
  - Allow resources to be handled outside
  - Separate configuration and creation
- Provide only what is necessary

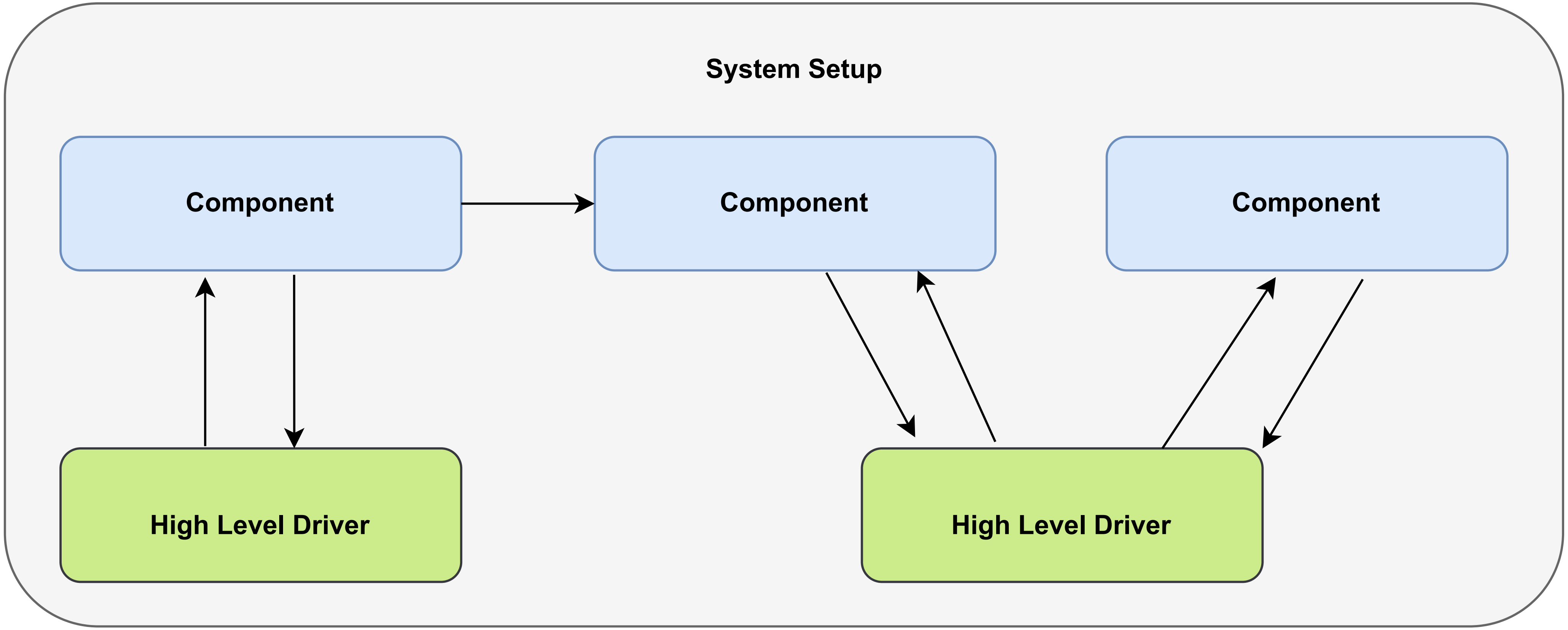
Business Logic

Business Logic

What should it do?

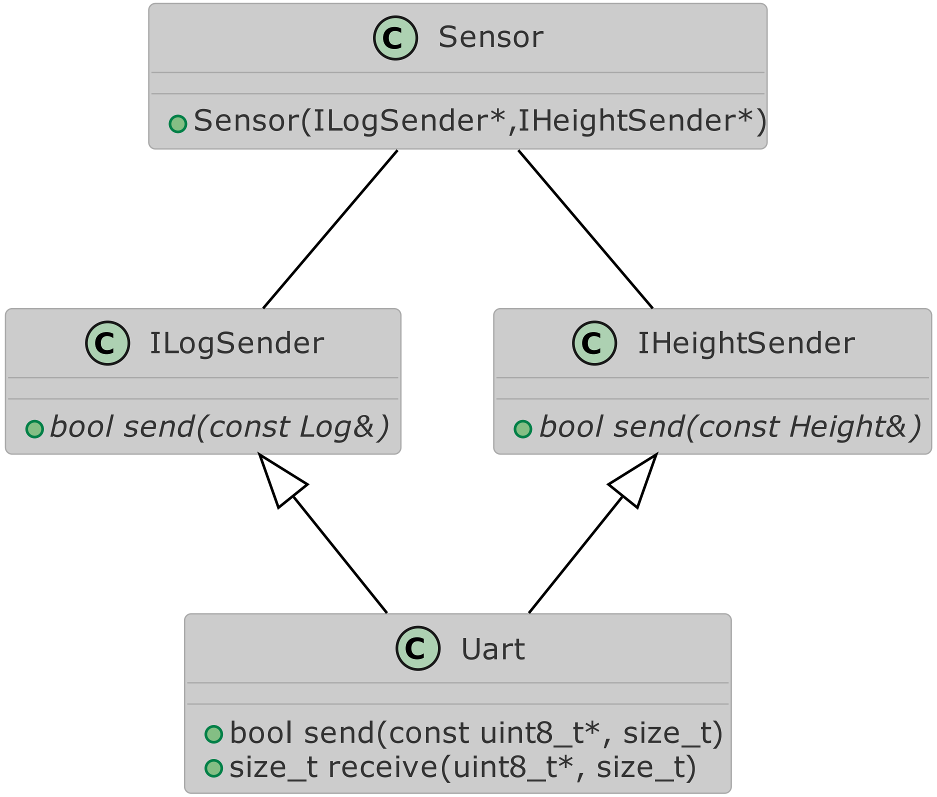
- Setup the system
  - Create and connect resources and components
- System startup and shutdown
- Allow to interchange components

Business Logic



Business Logic

Communication



Direct Interface

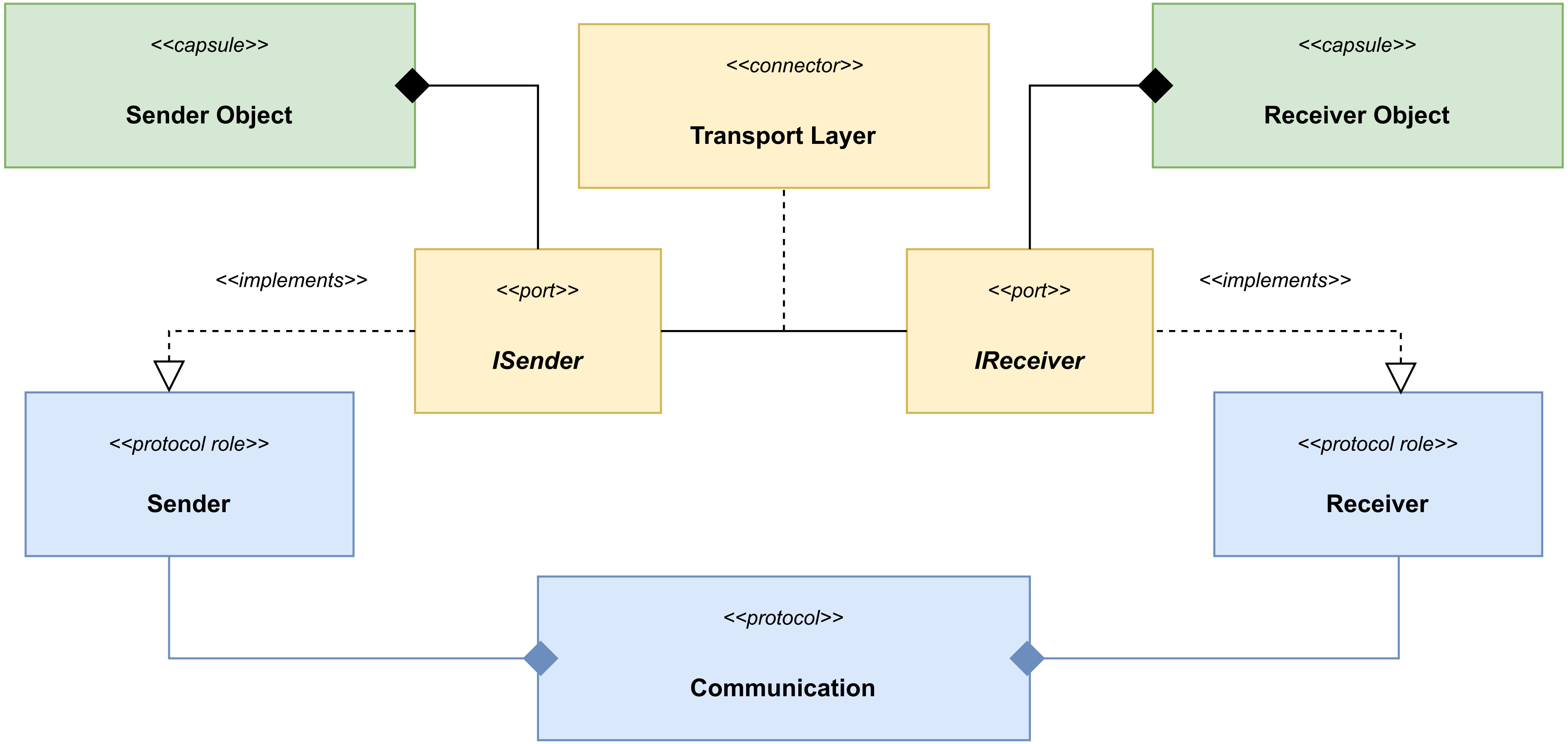
- Creates dependencies between components
- Can inject unwanted behavior

Protocol Abstraction

- Layer over the communication
- Only protocol knows messages
  - Handles decoding and dispatching
  - Handles encoding and sending
- Thread transition can be enforced

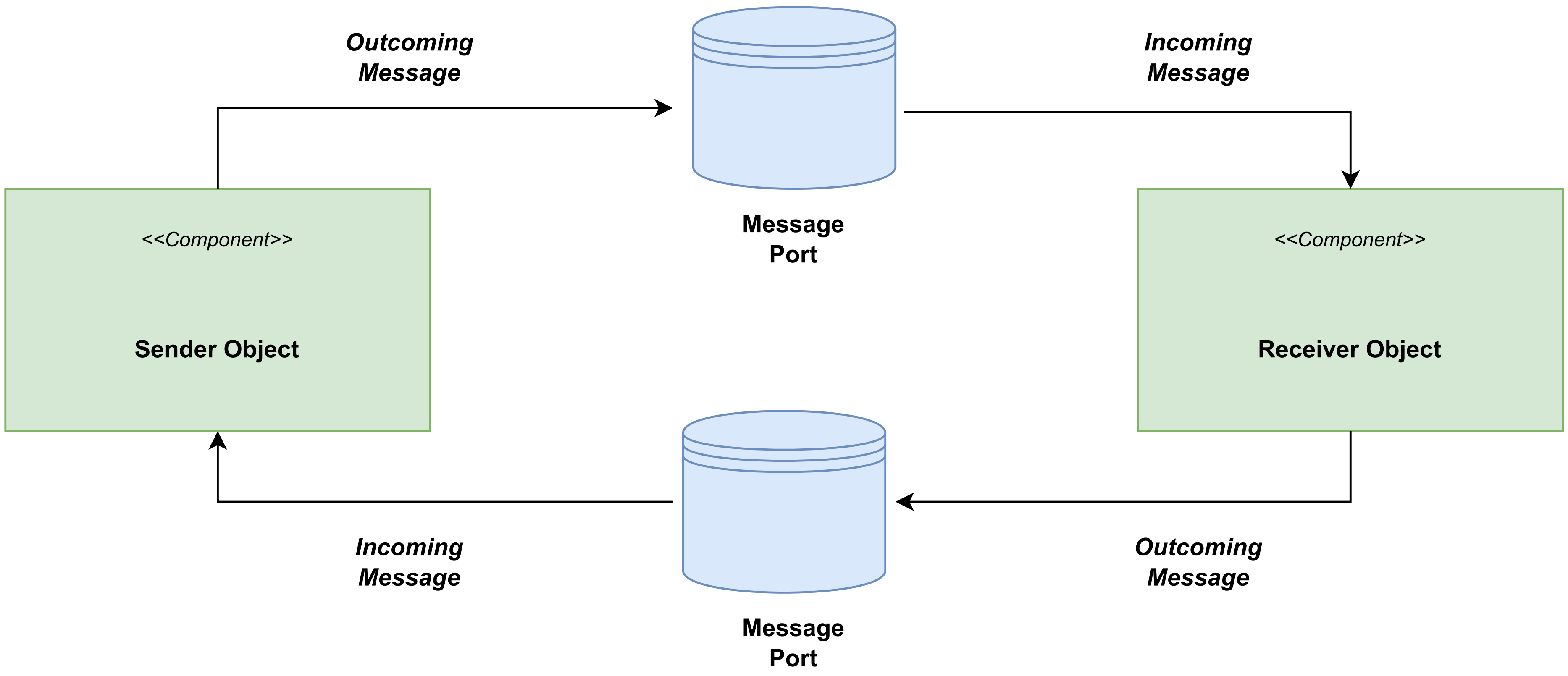
Communication

ROOM-Pattern



Communication

Microservice



Communication

Message Port

write_port.hpp

```cpp
template<class RECEIVER_T, typename PROTOCOL_T>
struct write_port {
    using protocol_type = PROTOCOL_T;
    using receiver_type = RECEIVER_T;
    using message_types = trait::incoming_messages_t<receiver_type>

    write_port(receiver_type& receiver) noexcept;

    template <typename MSG_T>
    requires concepts::contains<trait::convert_t<MSG_T, protocol_type>,
                                message_types>
    bool write(const MSG_T& msg) noexcept {
        const auto msg_out = convert<protocol_type>(msg);
        return receive_message(m_receiver, msg_out);
        // could also be dispatched over message_queue, ipc etc.
    }
    /*...*/
};
```

Communication

Receiving Messages

component.hpp

```cpp
template <typename>
struct incoming_messages {
    using type = ::util::sequence<>;
};

template <>
struct incoming_messages<component> {
    using type = ::util::sequence<log_msg, ...>;
};

bool receive_message(component& obj, const log_msg& msg) {/*...*/}
bool receive_message(component& obj, const auto& msg) {/*...*/}
```

Communication

Sending Messages

i_component.hpp

```cpp
struct i_component_write {
    virtual bool write(const component_msg&) const noexcept = 0;
};

template<typename PROTOCOL_T>
struct component_write_port : i_component_write {
    bool write(const component_msg& msg) const noexcept override;
    /*...*/
};

struct component {
    component(i_component_write& port) noexcept;
};
```

Communication

Sending Messages

component.hpp

```cpp
template<typename ... Ts>
class write_port_ref<util::sequence<Ts...>> {
public:
    using operation_types =
        typename detail::write_port_operations<Ts...>::type;

    template<typename T>
    write_port_ref(const T& obj);

    template<typename T>
    void write(const T& msg) const noexcept;
    /*...*/
};

struct component {
    using write_port_type =
        write_port_ref<outcoming_messages_t<component>>;
    component(write_port_type port) : m_port(port) {}
};
```

Integration

System Setup

system_setup.hpp

```
class system_setup {
    /*...*/

    system_setup() noexcept
        : m_uart(uart_cfg)
        , m_component(component::write_port_type{m_port}, thread_cfg)
    {/*...*/}

    bool init() noexcept;
    bool startup() noexcept;
    bool shutdown() noexcept;
    /*...*/

    uart_driver m_uart;
    component   m_component;

    using component_port = write_port<uart_driver, protocol::inhouse>
    component_port m_port{m_uart};
};
```

Summary

Keep in Mind

- Consider all possibilities for changes
- Isolate the hardware and take control
- Isolate the middleware and allow host build
- Design components independent and interchangeable
- Consider testing over all abstractions of your design

What's next?

Near future

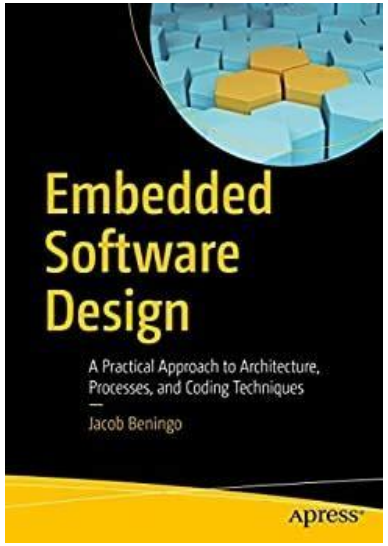
- https://github.com/zie87/talks
    - Slides & Notes
    - Code Samples

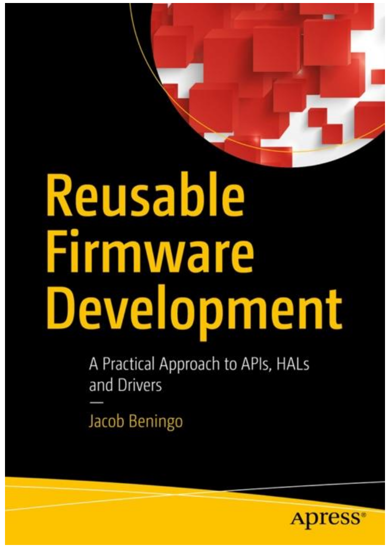
Distant future

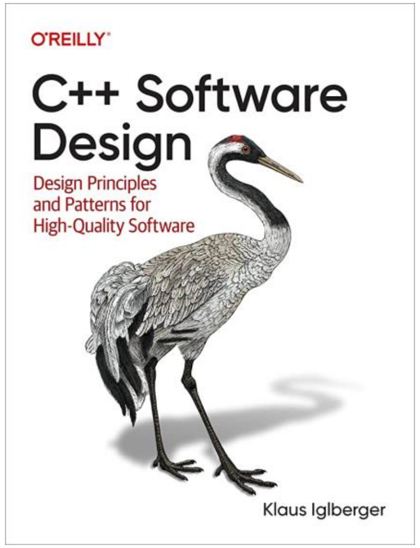
- Hardware Abstraction Layer
- Device Tree Parser
- Generic System Setup

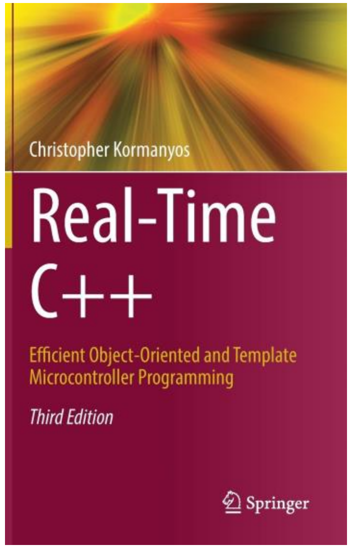
Book References

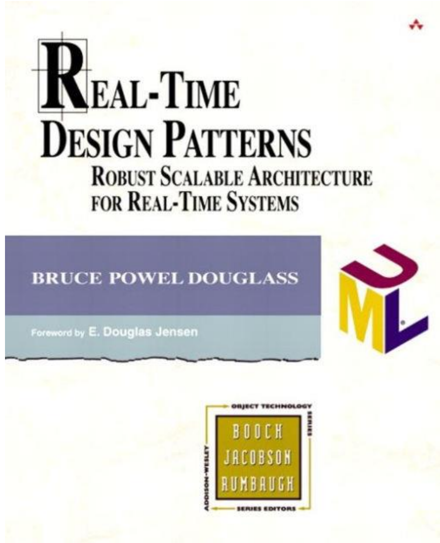
Learn more











Learn more about us

Follow us

Thank You!

Questions?