accu 2022

C++ IN THE WORLD OF EMBEDDED SYSTEMS

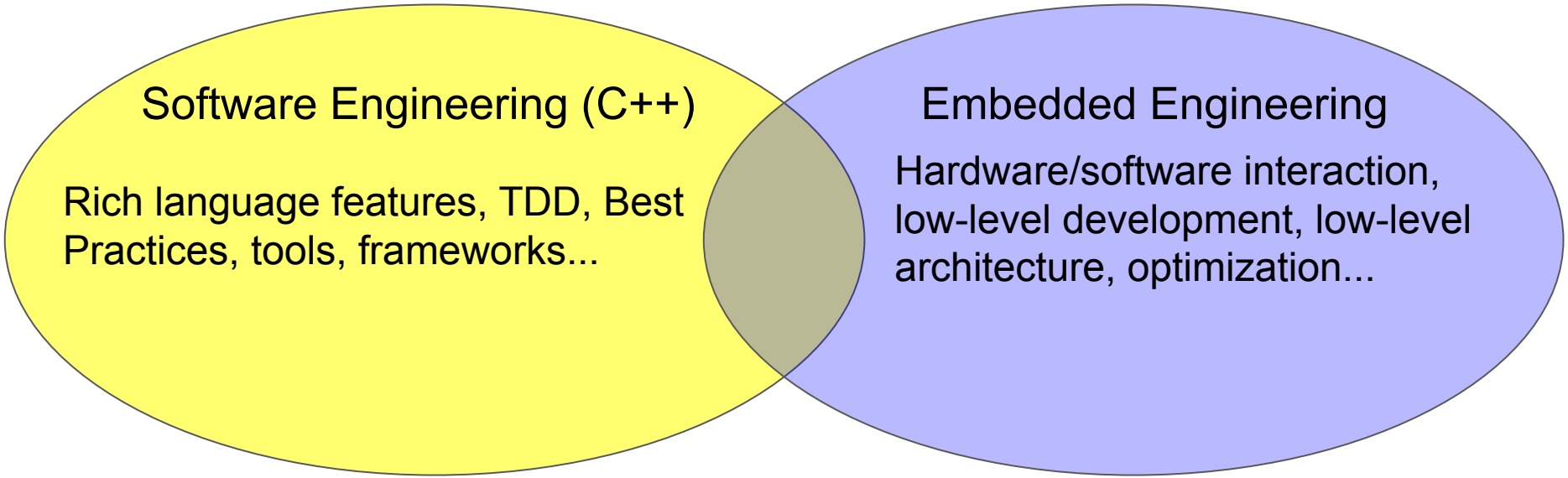
VLADIMIR VISHNEVSKII

Agenda

1. Overview of embedded systems
2. Typical architectures in embedded systems
3. Review of the C++ availability for embedded environments
4. Examples of C++ application for embedded code development

Motivation

Encourage knowledge and experience exchange between software engineering and embedded development.



Embedded systems

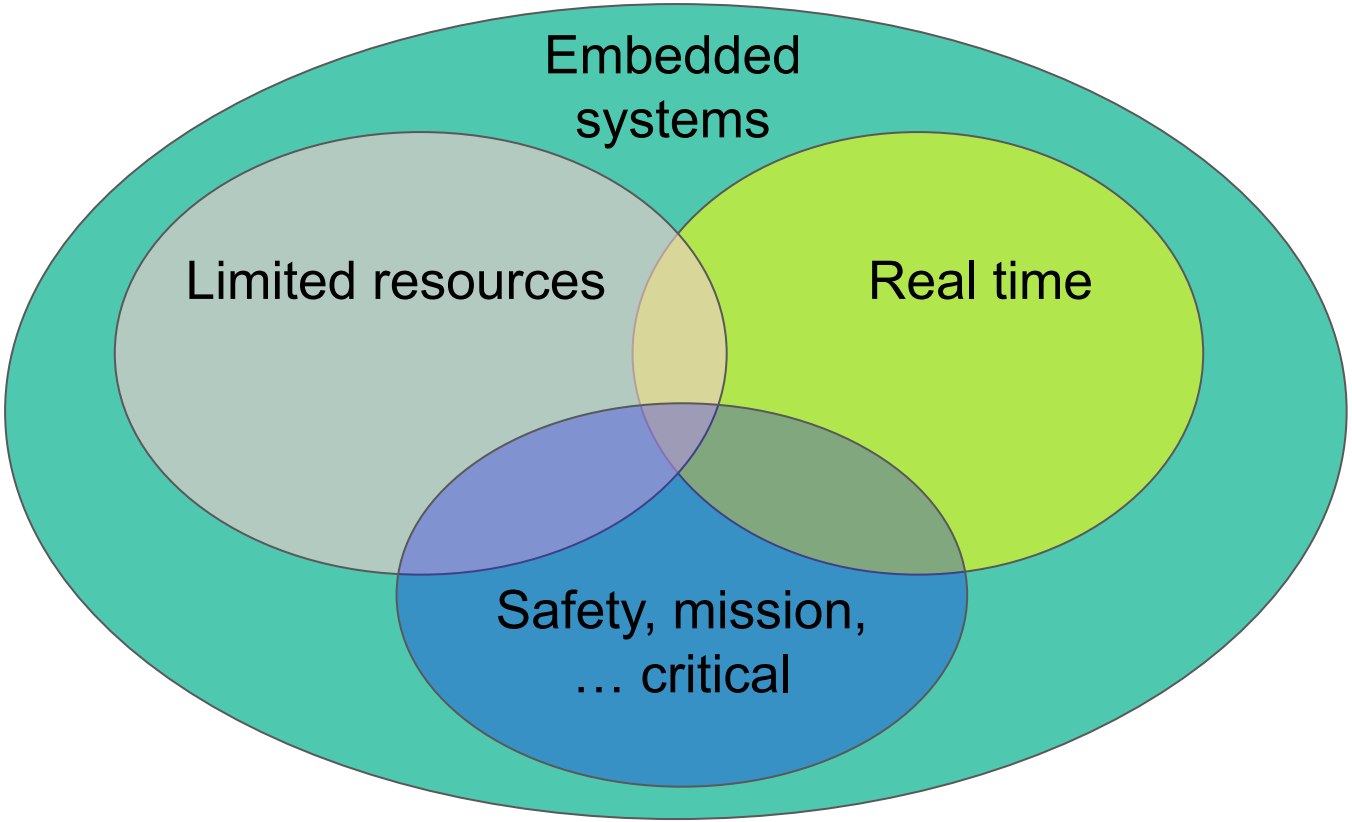
Non-formal definition:

An embedded (computer) system is a **specialized** computer system **designed as a part of another system**.

Examples:

Electronic control units (ECU) in automotive, control elements in “smart” devices.

Embedded systems categories



Embedded development characteristic

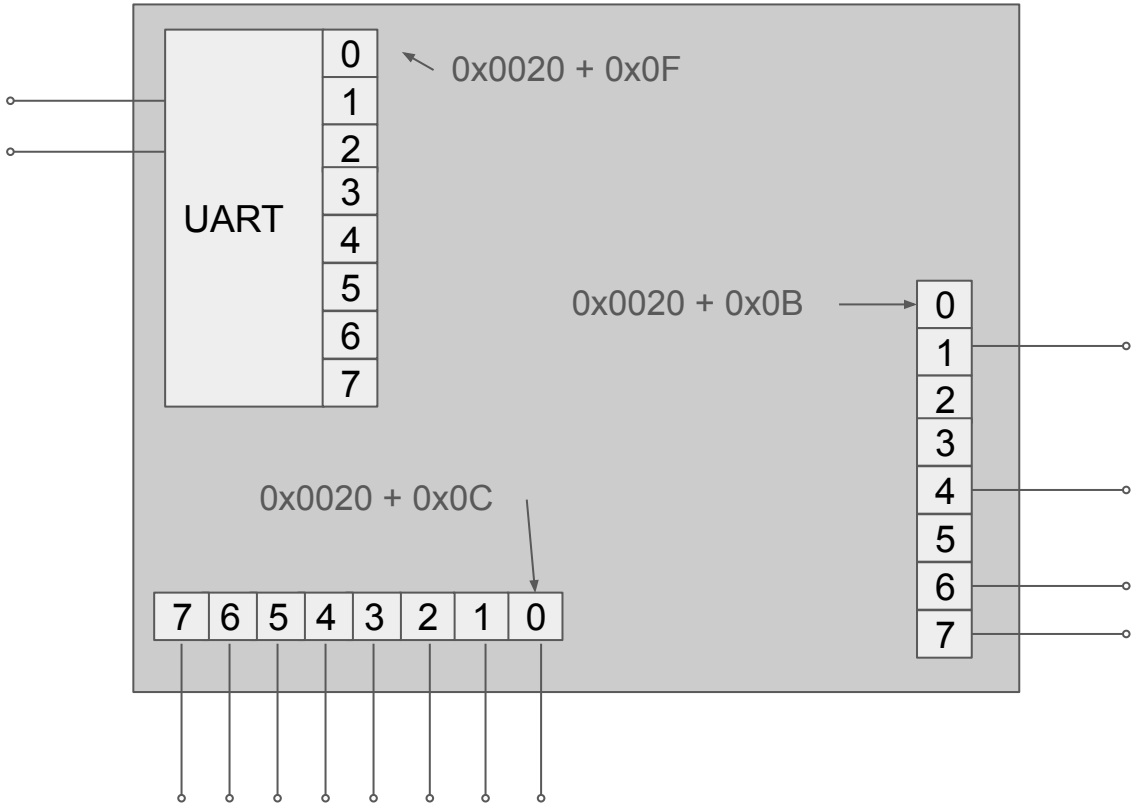
1. Software is deployed on target platform using specialized tooling
2. Diagnostics and debugging can be limited (software/hardware debuggers, disabled debugging, limited physical accessibility)
3. **Hardware can be unstable**
4. **Compilers might have defects**

Embedded development characteristic

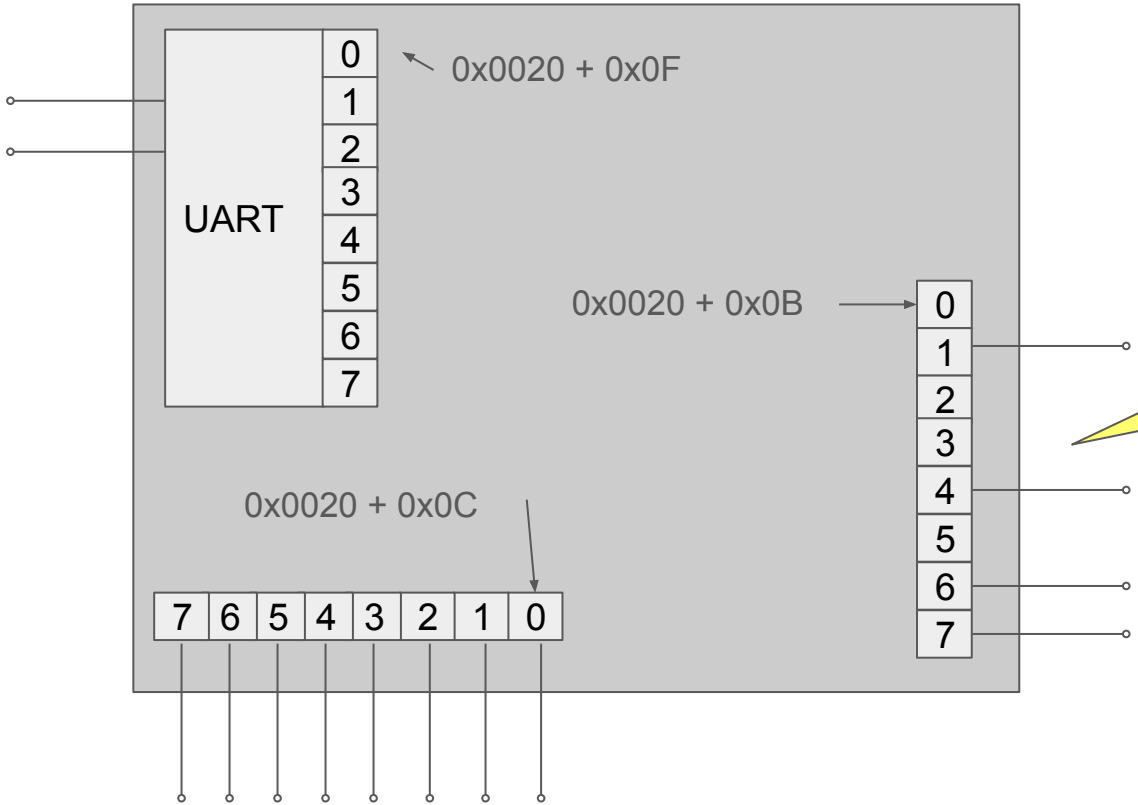
Code quality must be maximized prior to deployment:

- best practices (TDD, review, CI) and Core Guidelines;
- primary testing on developer’s machine (requires code portability).

Trivial hardware platform

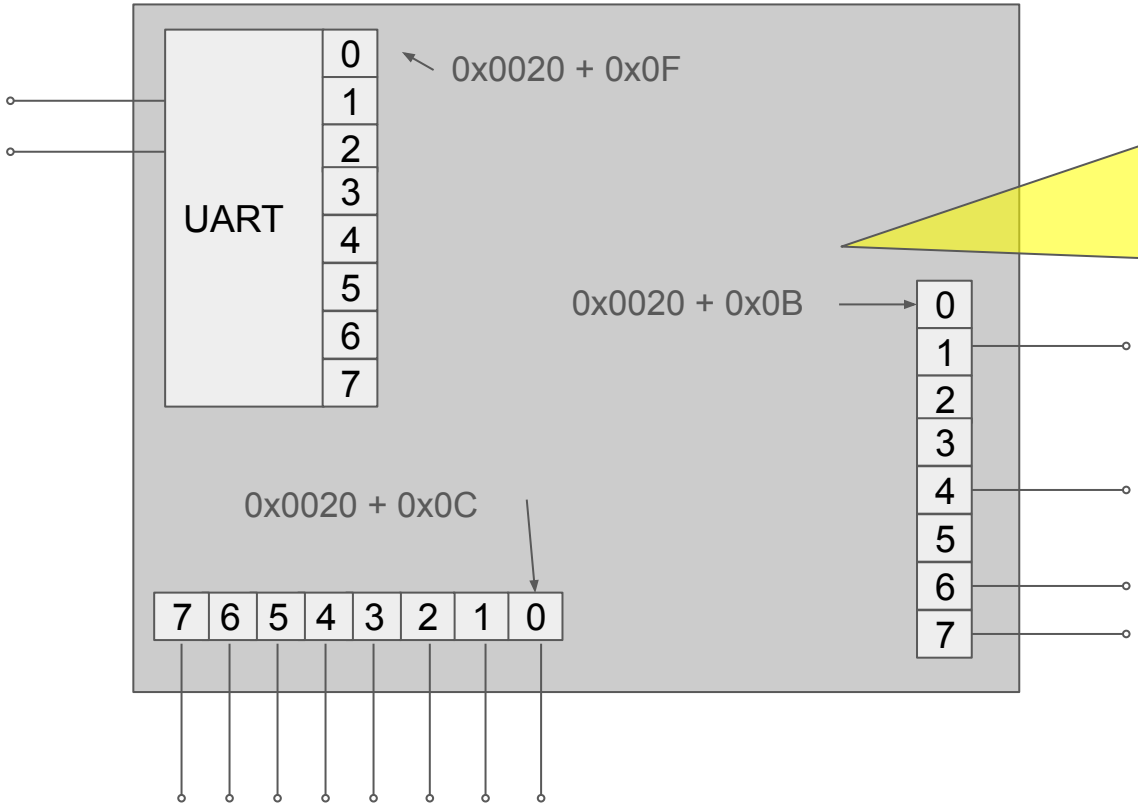


Trivial hardware platform



Input/output lines (**GPIO**-General-Purpose Input/Output) grouped into port

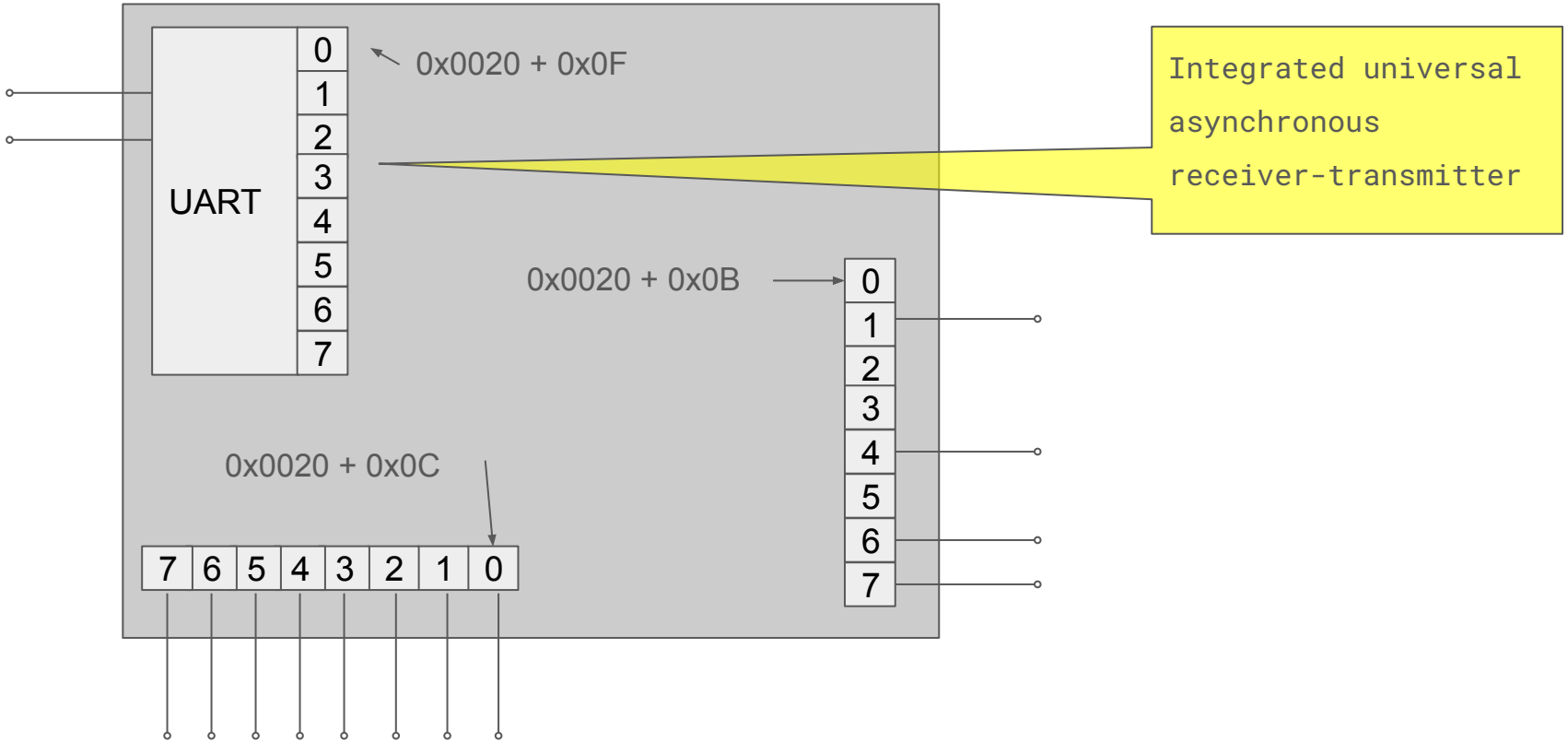
Trivial hardware platform



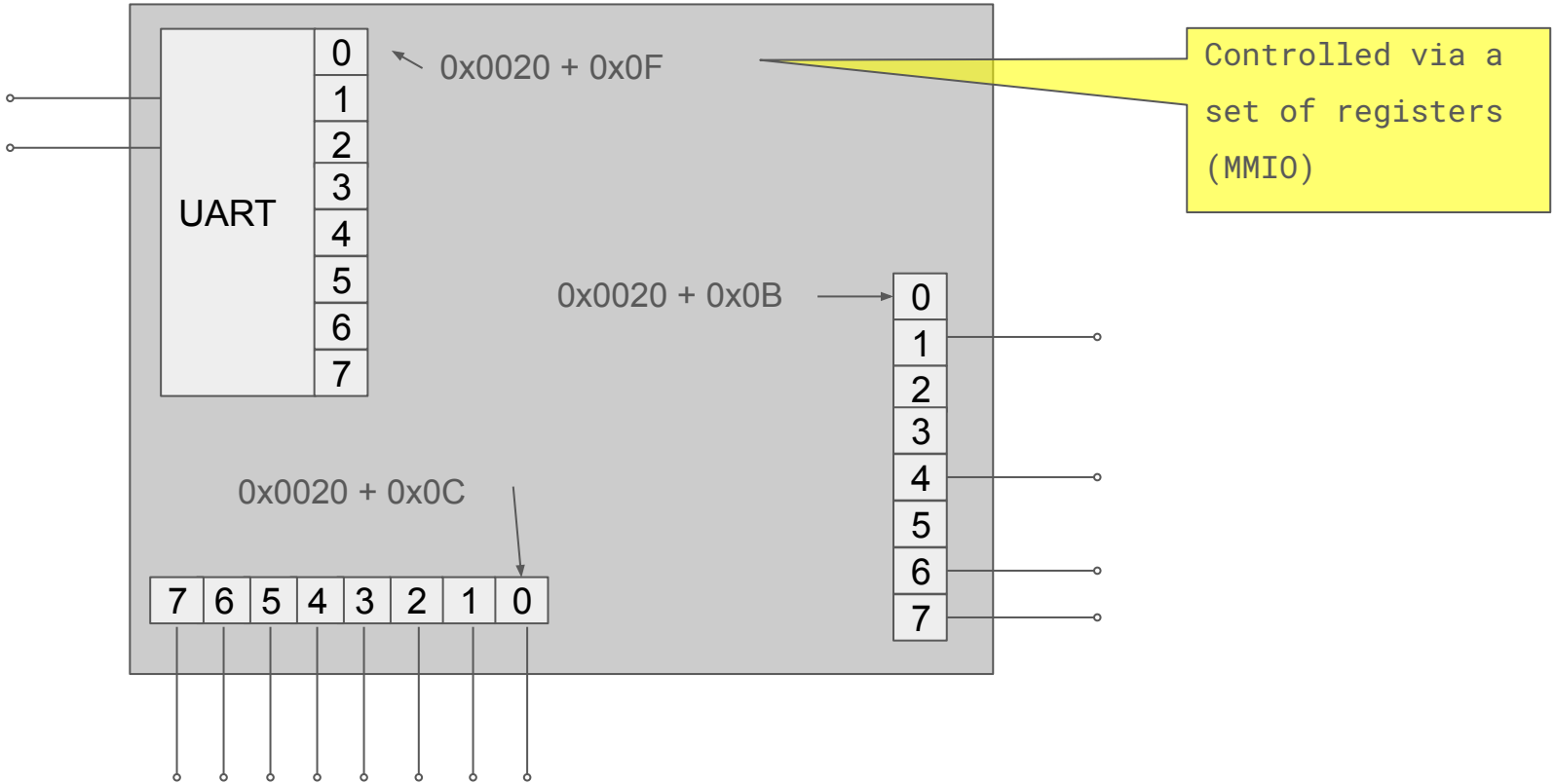
MMIO-Memory Mapped Input/Output

Line state is controlled via bit in a memory cell (**SFR**-Special Function Register)

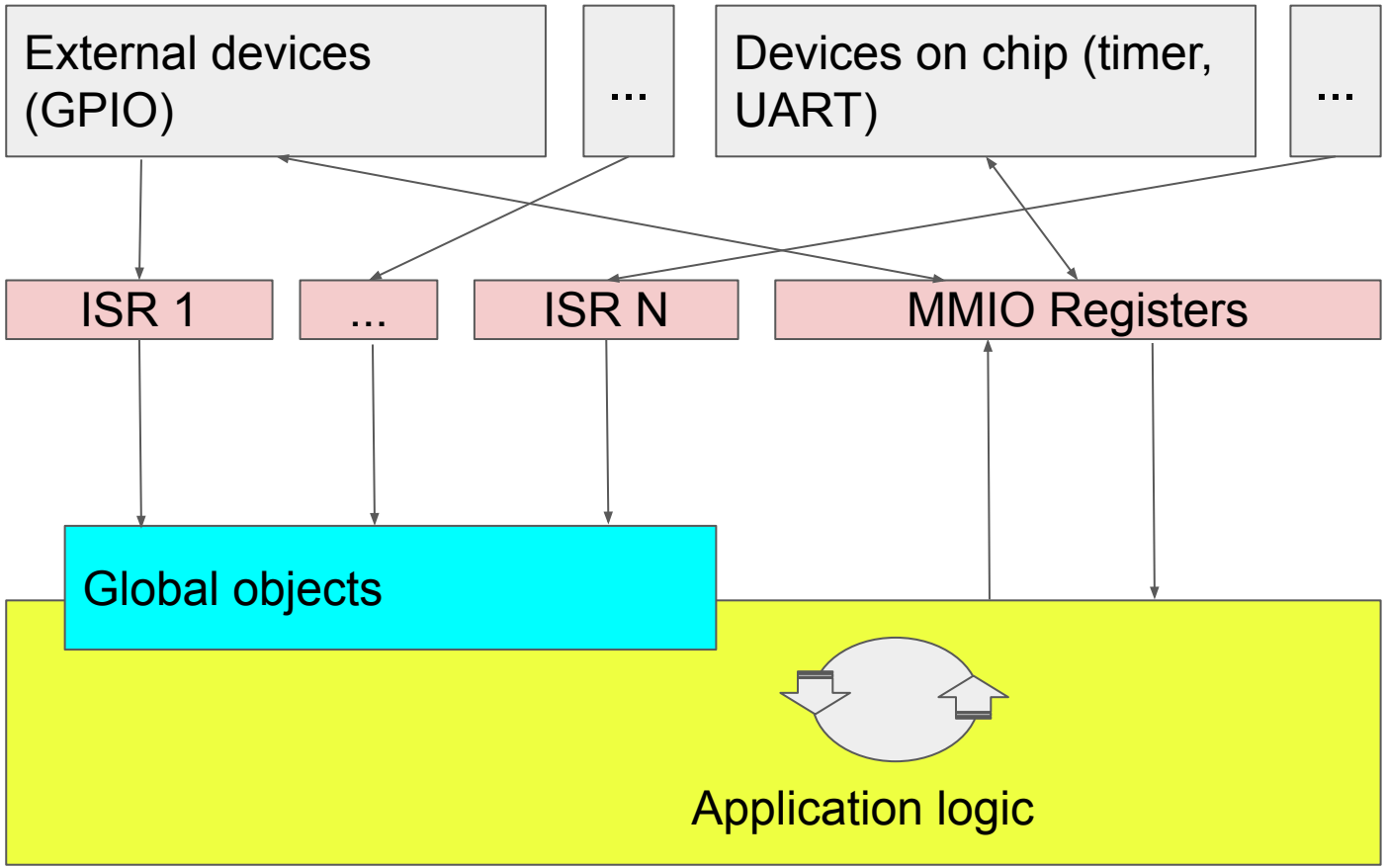
Trivial hardware platform



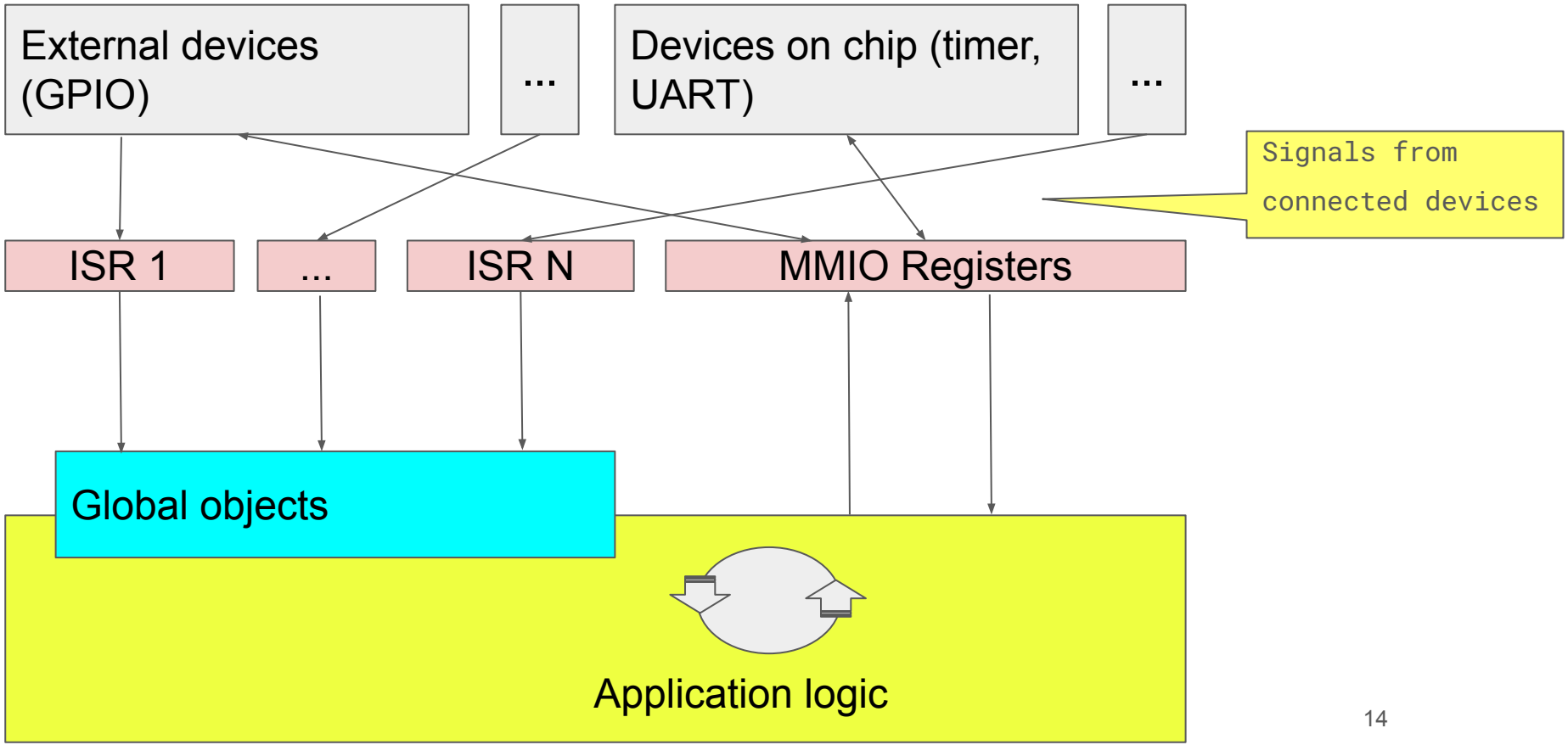
Trivial hardware platform



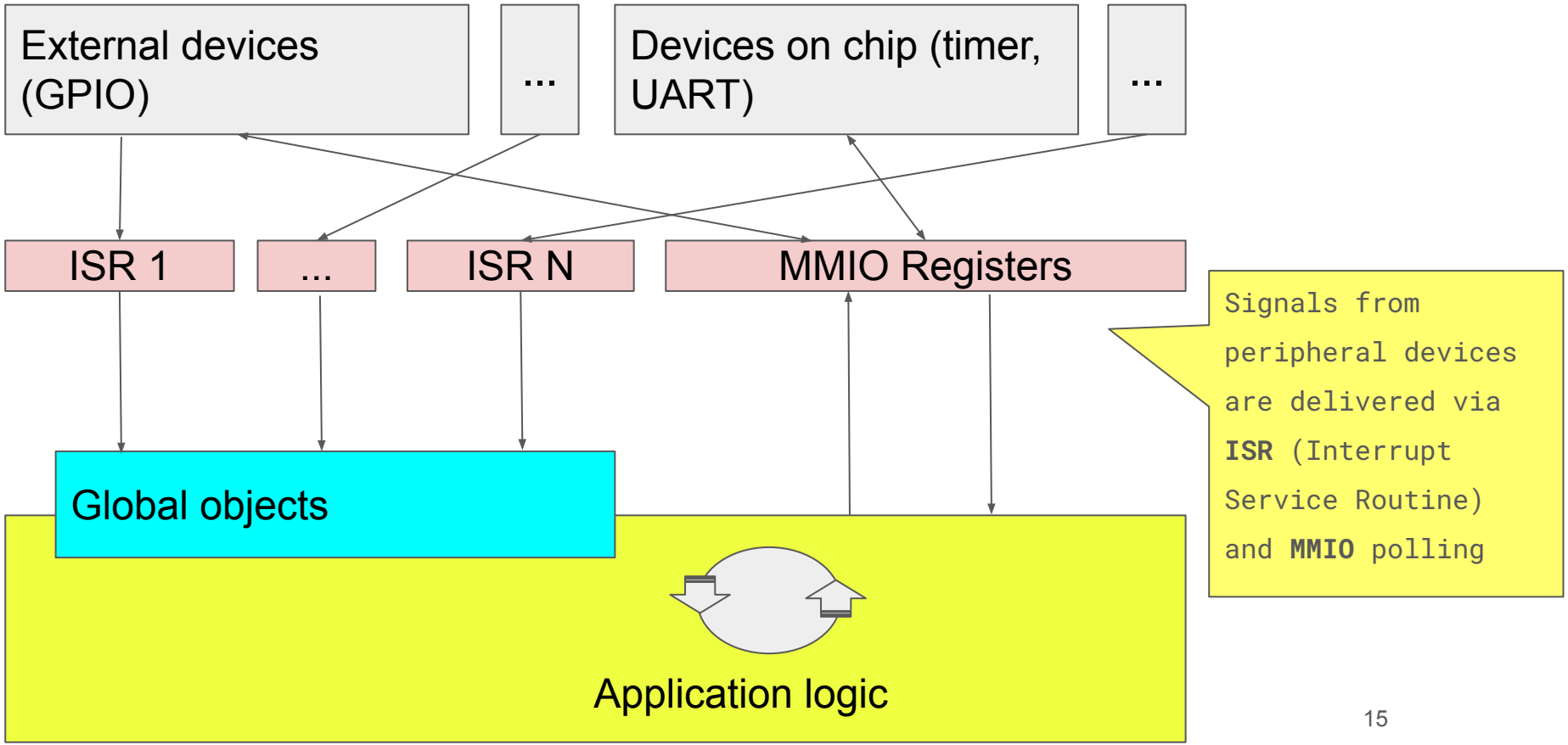
Trivial software architecture



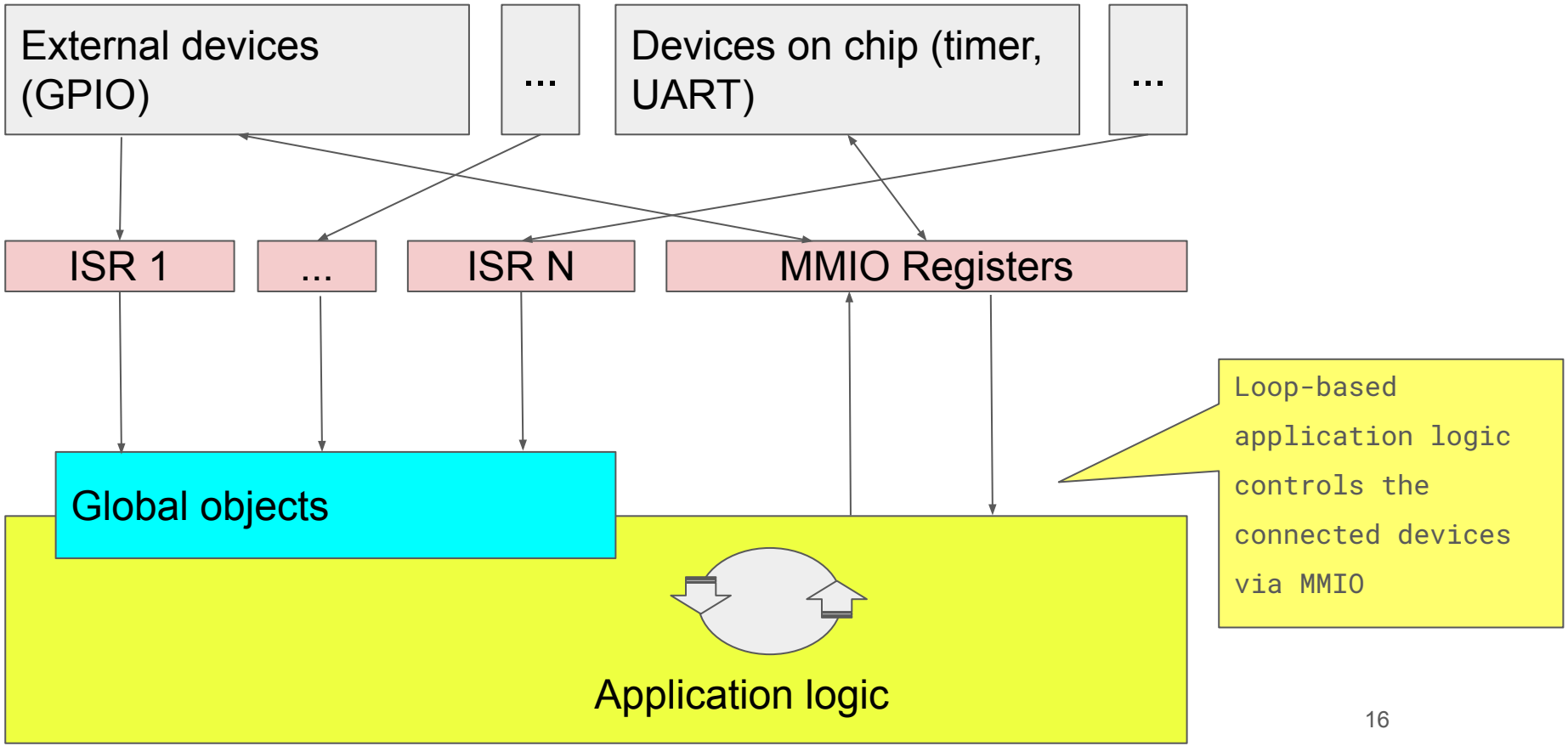
Trivial software architecture



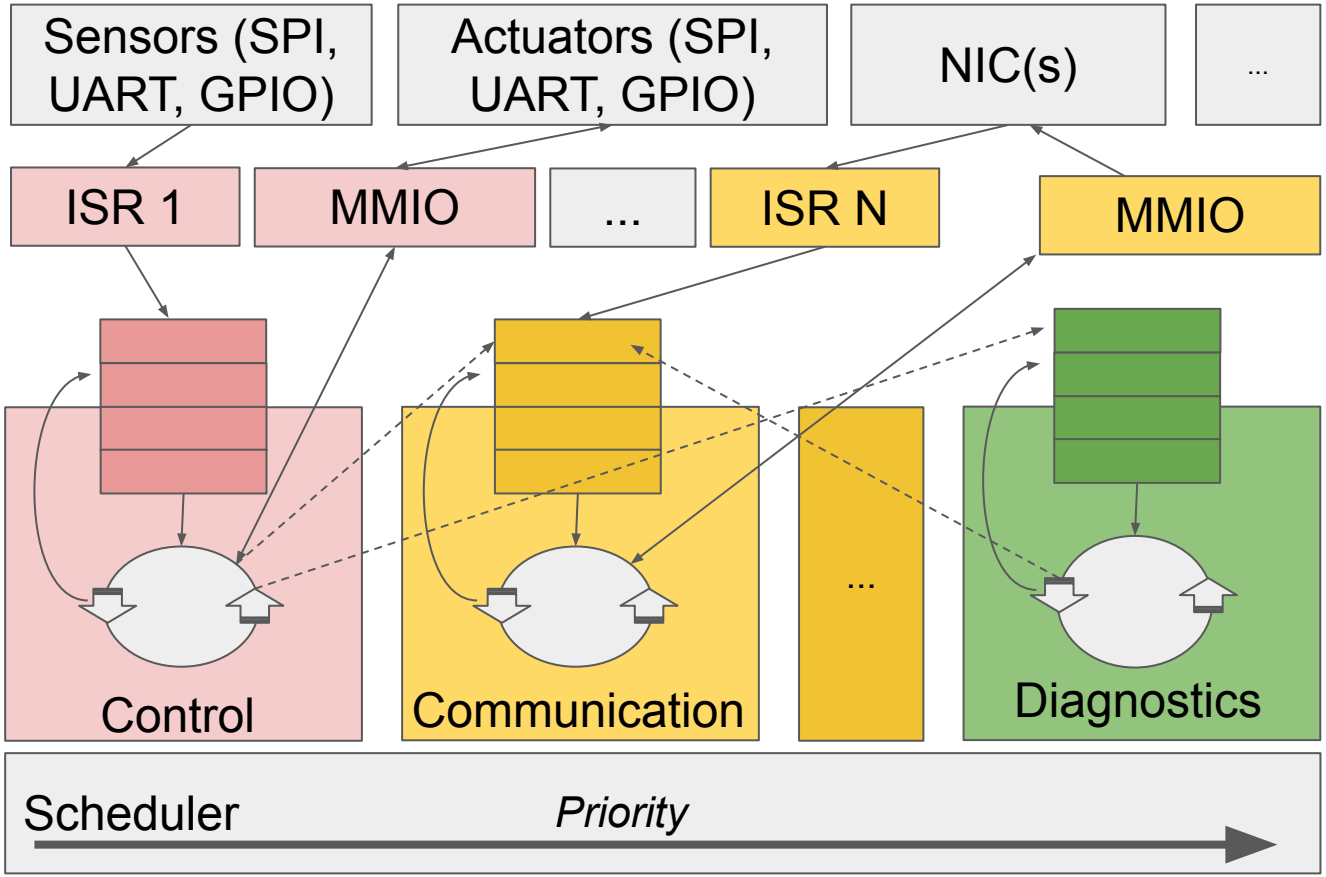
Trivial software architecture



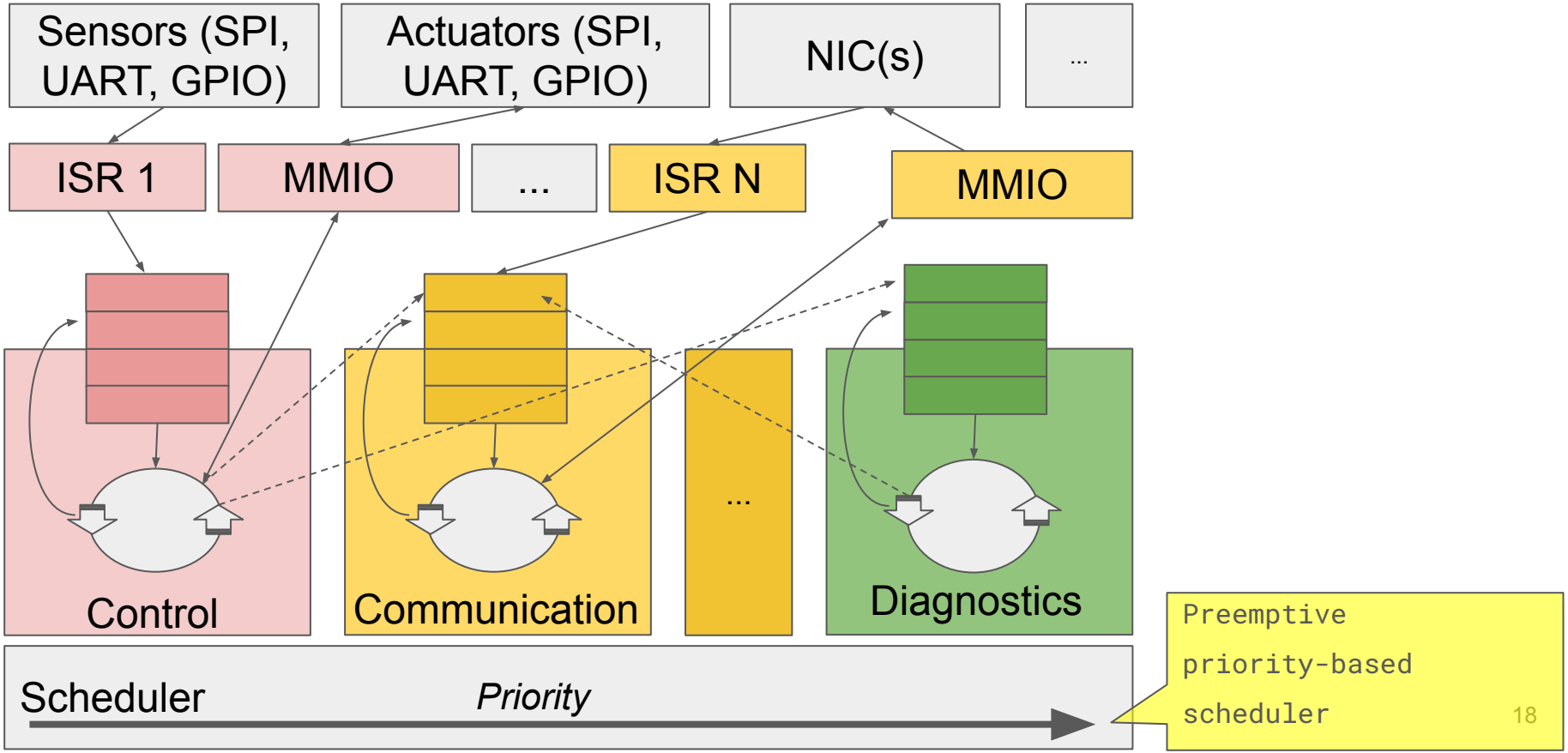
Trivial software architecture



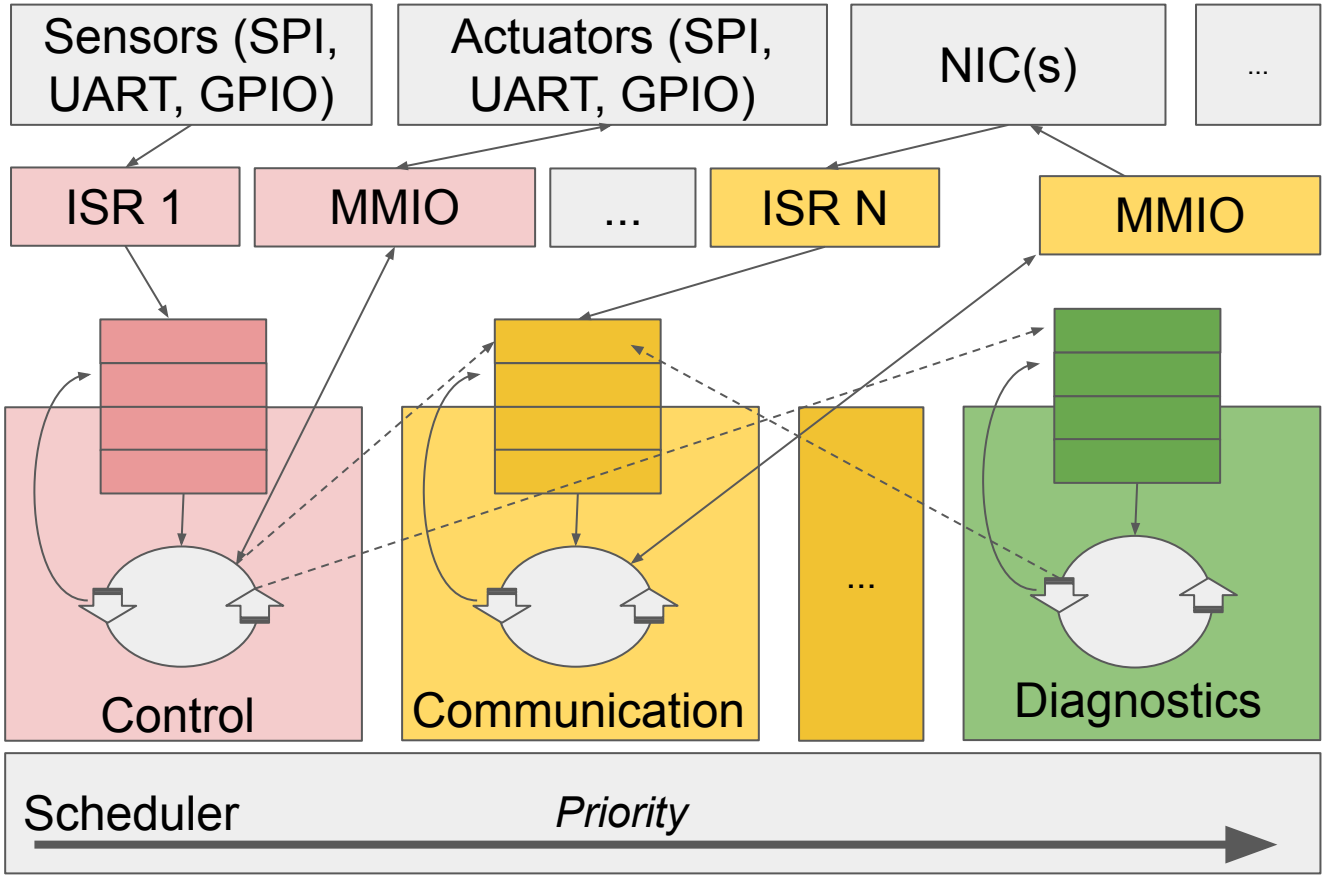
Connected real time system architecture



Connected real time system architecture

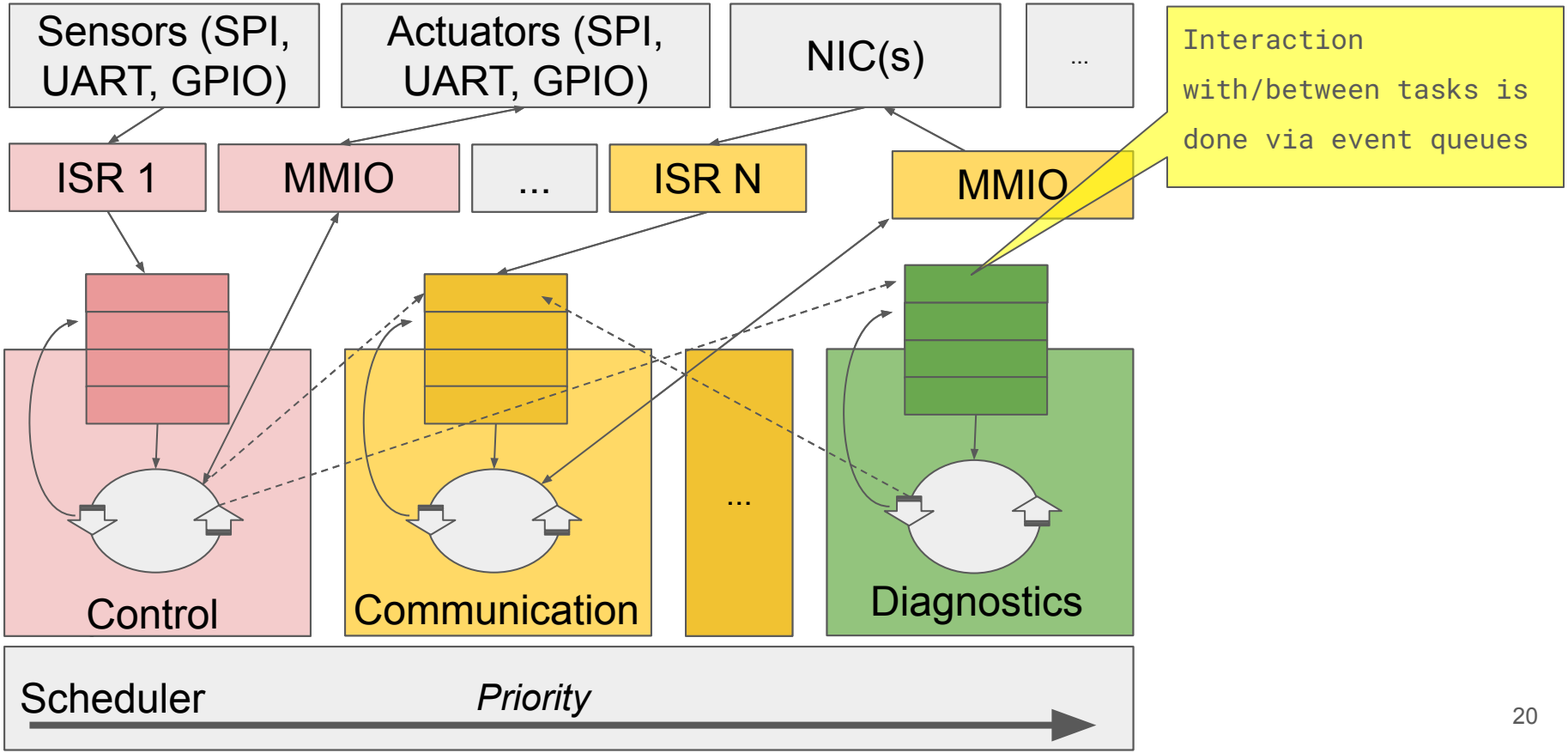


Connected real time system architecture



Tasks have static priorities and work exclusively with some set of resources (including peripheral devices)

Connected real time system architecture



High performance non-embedded applications

- Task-based parallelism with dedicated narrow specialized tasks
- Async interaction via event queues
- Minimize resource contention
- Optimize memory usage

Embedded architectures and high-performance server/desktop architectures share characteristics.

Programming language requirements

- Minimal overhead
- Portability
- Test-friendliness (some polymorphism)
- Availability of reusable frameworks for typical tasks

C++ satisfies the requirements

Limitations of C++ applicability

- Compiler support for a particular platform
- Availability of standard library components in a platform SDK
- Applicability of programming elements/constructs under platform or project constraints
- Industrial standards limitations

Compiler support of C++ **syntax** standards

GCC	C++20*, C++23*
Clang	C++20*, C++23*
SEGGER ARM Compiler	C++17*
IAR C++ Compiler	C++17*
Texas Instruments ARM Compiler	C++14*
Wind River Diab	C++14*

* - with limitations

Standard library: Hosted & Freestanding

- **Hosted**: contains components dependent on OS (filesystem, thread, ...).

  In practice *libc* for platform satisfies the dependencies on OS. “Real” OS is not required.

- **Freestanding**: doesn’t contain OS specific components.

  Hardly usable - missing core elements (<utility> with *std::move* and *std::forward).*

- Some proprietary SDKs offer subset of std lib.

Limitations of C++ applicability

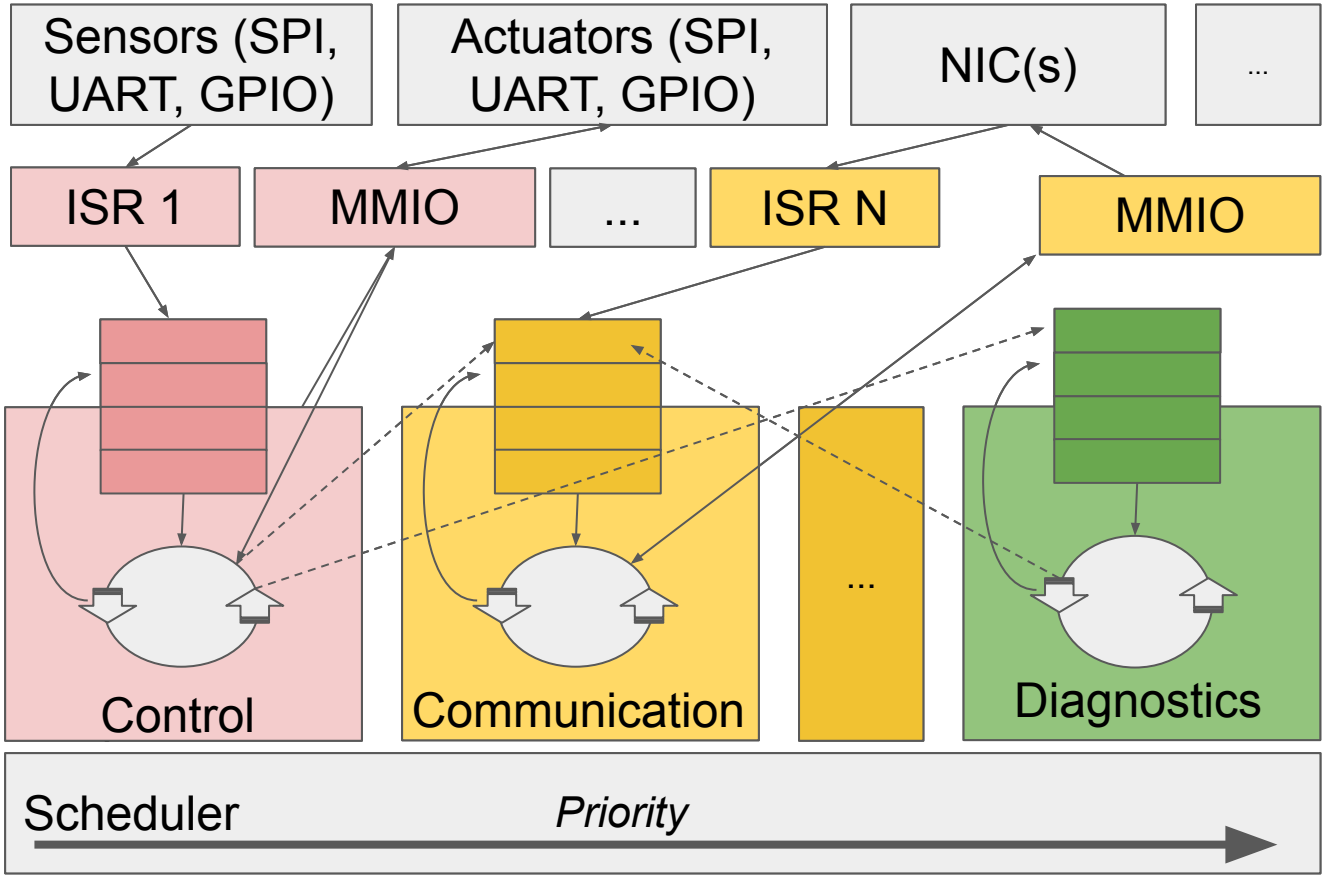
	Limited resources	Real time
Heap allocation	Maintenance overhead + fragmentation	Fragmentation + access synchronization => loss of determinizm
Exceptions	Increase of executable size	Analysis complexity, potential loss of determinism and low performance

Limitations of C++ applicability

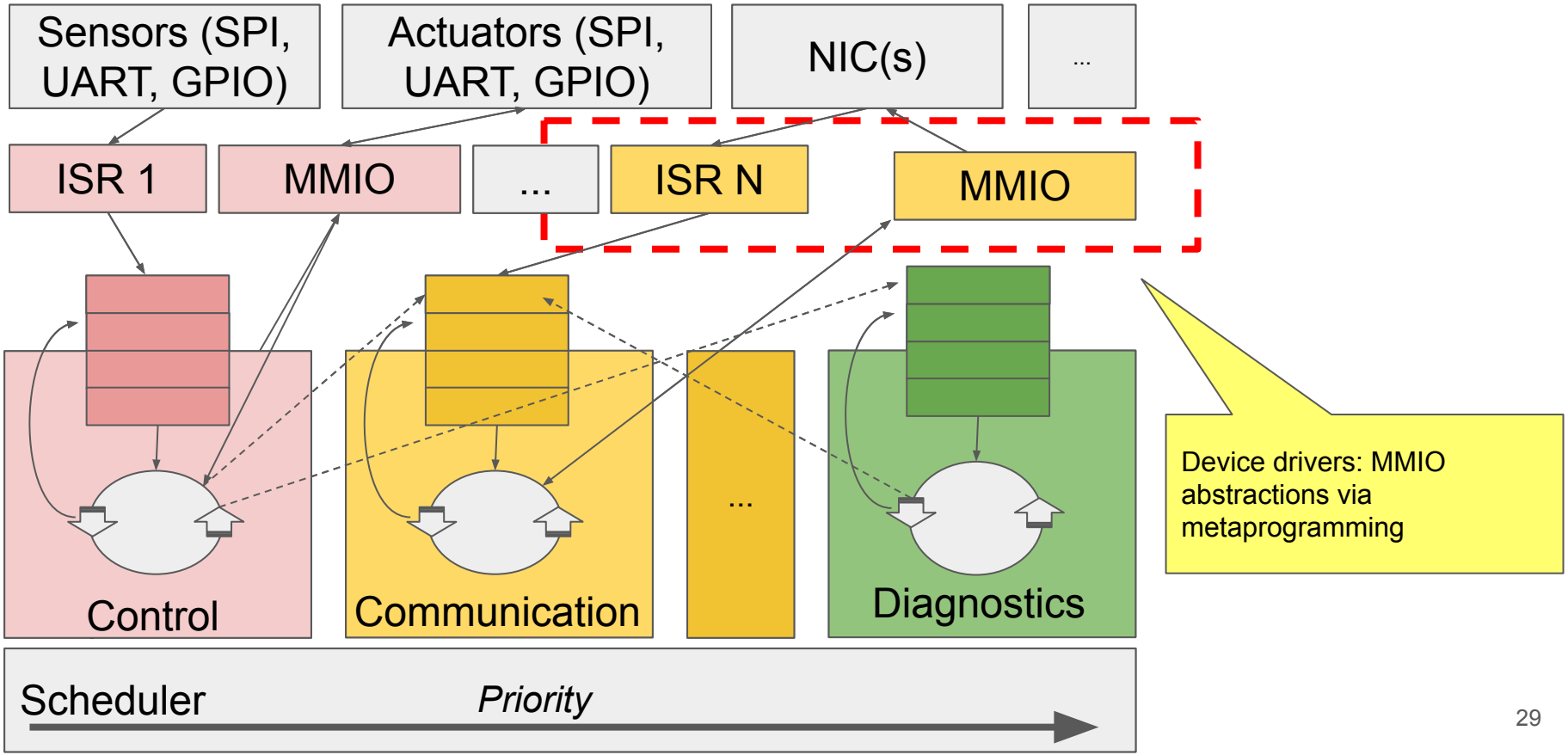
Industrial standards	JSF C++	MISRA C++	AUTOSAR C++ 14
Heap allocation	-+	-	-+
Exceptions	-	+	+

JSF C++ - Joint Strike Fighter Air Vehicle C++ (2005)
MISRA C++ - Motor Industry Software Reliability Association C++ (2008)
AUTOSAR C++14 - AUTomotive Open System ARchitecture C++14 (2017)

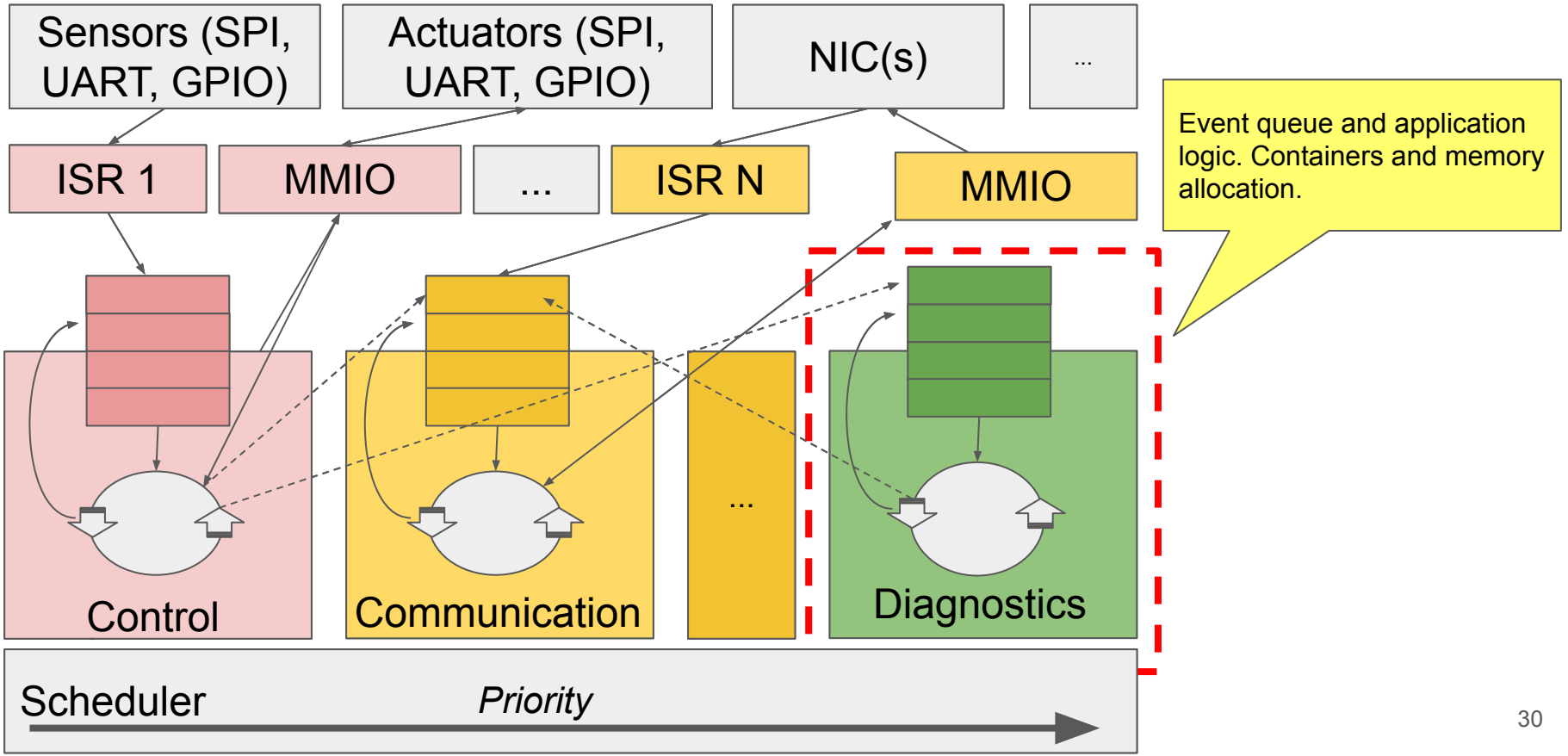
C++ application. Examples



C++ application. Examples

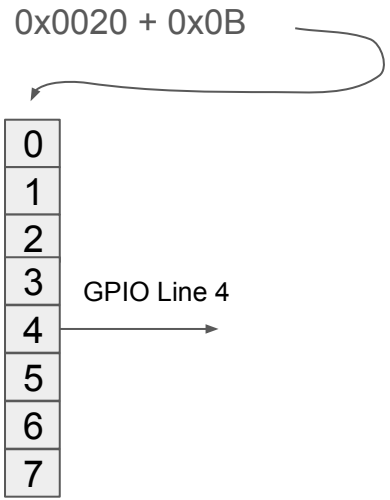


C++ application. Examples



C++ for device driver development

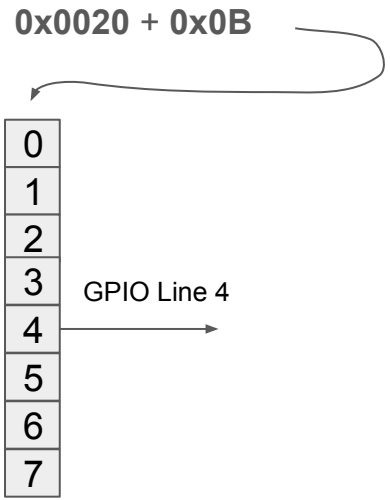
MMIO register




```
#define __SFR_OFFSET 0x20
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTD _SFR_IO8(0x0B)

#define _BV(bit) (1 << (bit))
#define PORTD4 4
#define PD4 PORTD4

PORTD |= _BV(PD4);
```

C++ for device driver development

MMIO register

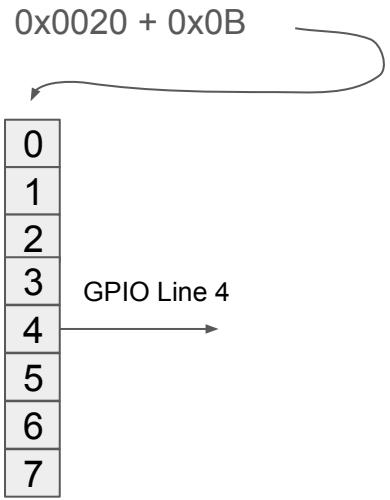



```
#define __SFR_OFFSET 0x20
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTD _SFR_IO8(0x0B)


#define _BV(bit) (1 << (bit))
#define PORTD4 4
#define PD4 PORTD4


PORTD |= _BV(PD4);
```

Address calculation

C++ for device driver development

MMIO register

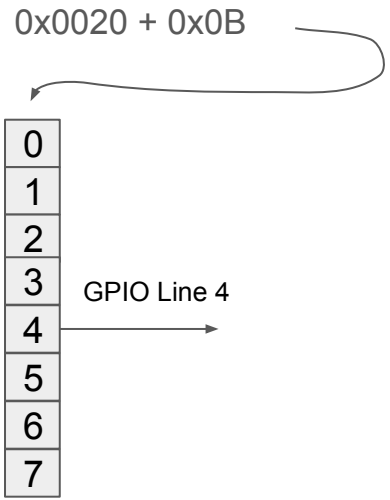



```
#define __SFR_OFFSET 0x20
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTD _SFR_IO8(0x0B)

#define _BV(bit) (1 << (bit))
#define PORTD4 4
#define PD4 PORTD4

PORTD |= _BV(PD4);
```

Mask calculation

C++ for device driver development

MMIO register

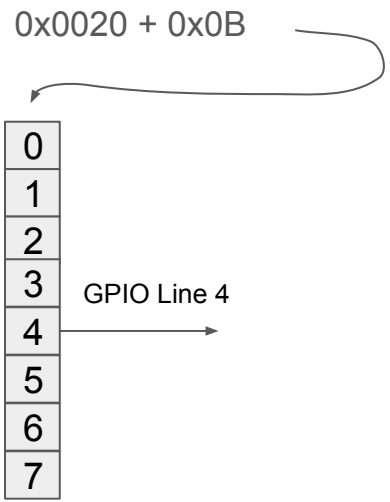



```
#define __SFR_OFFSET 0x20
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTD _SFR_IO8(0x0B)


#define _BV(bit) (1 << (bit))
#define PORTD4 4
#define PD4 PORTD4


PORTD |= _BV(PD4);
```

Setting the bit

C++ for device driver development

MMIO register



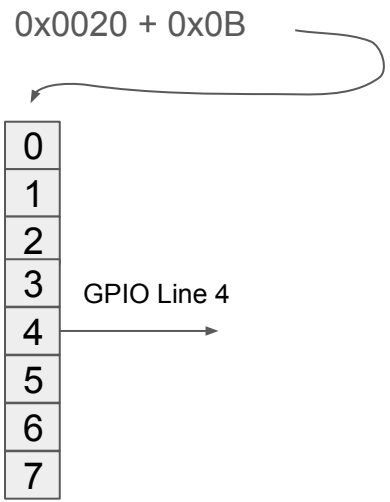
C-based implementation problems:

1. API code and code dependent on API can be executed only on target platform

2. No compile-time checks despite the fact that all register sizes and masks are known

C++ for device driver development

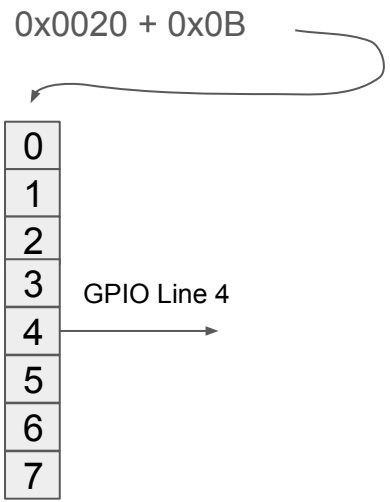
MMIO register




```
template <typename T, uintptr_t Address,
                  uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                    (Base + Address);
    }
};
```

C++ for device driver development

MMIO register

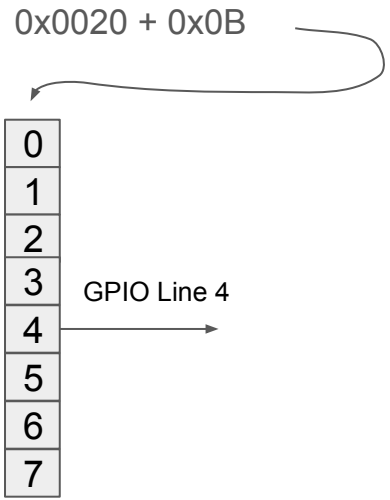



```
template <typename T, uintptr_t Address,
                      uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                        (Base + Address);
    }
};
```

Referenced object type, base and offset

C++ for device driver development

MMIO register

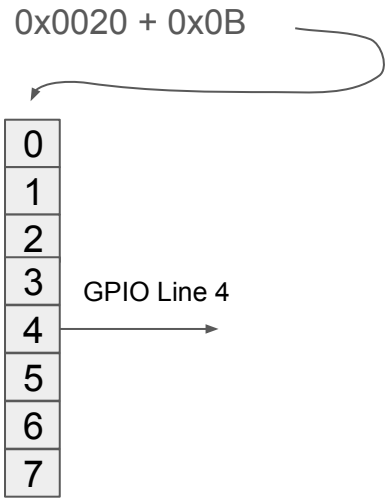



```
template <typename T, uintptr_t Address,
                      uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                          (Base + Address);
    }
};
```

Address calculation

C++ for device driver development

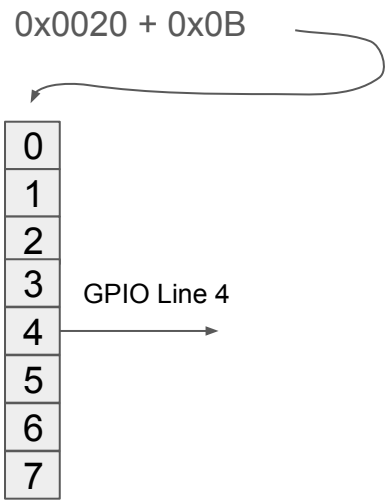
MMIO register




```cpp
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

namespace ports::d {
using out
    = mmio::write_object<special_func_reg<uint8_t, 0x0B>, 0, 8>;
//...
}


ports::d::out::bits<4> pin;
pin.set();
```

C++ for device driver development

MMIO register

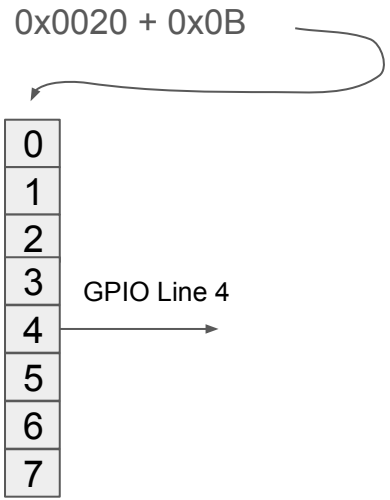



```
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

namespace ports::d {
using out
    = mmio::write_object<special_func_reg<uint8_t, 0x0B>, 0, 8>;
//...
}


ports::d::out::bits<4> pin;
pin.set();
```

Alias for concrete architecture

C++ for device driver development

MMIO register

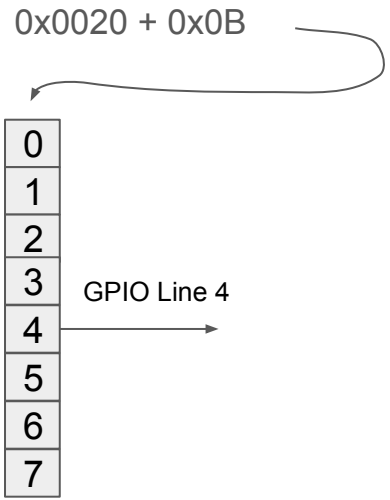



```
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

namespace ports::d {
using out
    = mmio::write_object<special_func_reg<uint8_t, 0x0B>, 0, 8>;
//...
}


ports::d::out::bits<4> pin;
pin.set();
```

Concrete register description

C++ for device driver development

MMIO register




```
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

namespace ports::d {
using out
    = mmio::write_object<special_func_reg<uint8_t, 0x0B>, 0, 8>;
//...
}


ports::d::out::bits<4> pin;
pin.set();
```

Mask instantiation (4th bit can be accessed)

C++ MMIO abstractions. Checks

```
template <typename Location, uint8_t Offset, uint8_t Length = 1>
struct write_object {
    using Type = typename Location::type;
    template <uint8_t... bit_offsets>
    using bits = bits_write_object<
        std::enable_if_t<detail::check_bounds<Type, bit_offsets...>(
            Offset, Length), Location>,
        Offset,
        Length,
        bit_offsets...>;
```

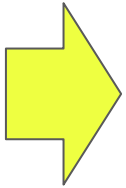
Compile-time check of the register boundaries

C++ MMIO abstractions. Optimizations

```
void set()
{
    if constexpr (detail::does_cover_the_whole_object<Type>(Offset, Length)) {
        *Location::get_address() = mask;
    } else {
        constexpr typename Location::type valueMask
            = detail::generate_mask<typename Location::type>(Offset, Length);
        *Location::get_address() &= (valueMask | (mask << Offset));
    }
}
```

C++ MMIO abstractions. Address abstraction

```
template <typename T, uintptr_t Address,
                      uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                      (Base + Address);
    }
};
```



```
template <typename T,
                      T *Address>
struct mock_address
{
    using type = T;
    static T *get_address() noexcept
    {
        return Address;
    }
};
```

C++ MMIO abstractions. Testing

```
static uint8_t object = 0;

TEST(mmio, WrapperCoversTheWholeObjectAndObjectIsZero_SetValue_TheObjectHasNewValue)
{
    constexpr uint8_t newValue = 0b10101011;
    mmio::write_object<mock_address<uint8_t, &object>, 0, 8> reg;
    reg.set(newValue);
    EXPECT_EQ(object, newValue);
}
```

Mock,handling the address of a static object

Device drivers. Dependency injection

Run-time polymorphism	1. Extra code and indirection might be unacceptable for performance critical areas. 2. Dependency on compiler ability to devirtualize the call.
Compile-time polymorphism	1. No virtual call. 2. No need to introduce hierarchies where they are not needed.

Device driver. Testing.

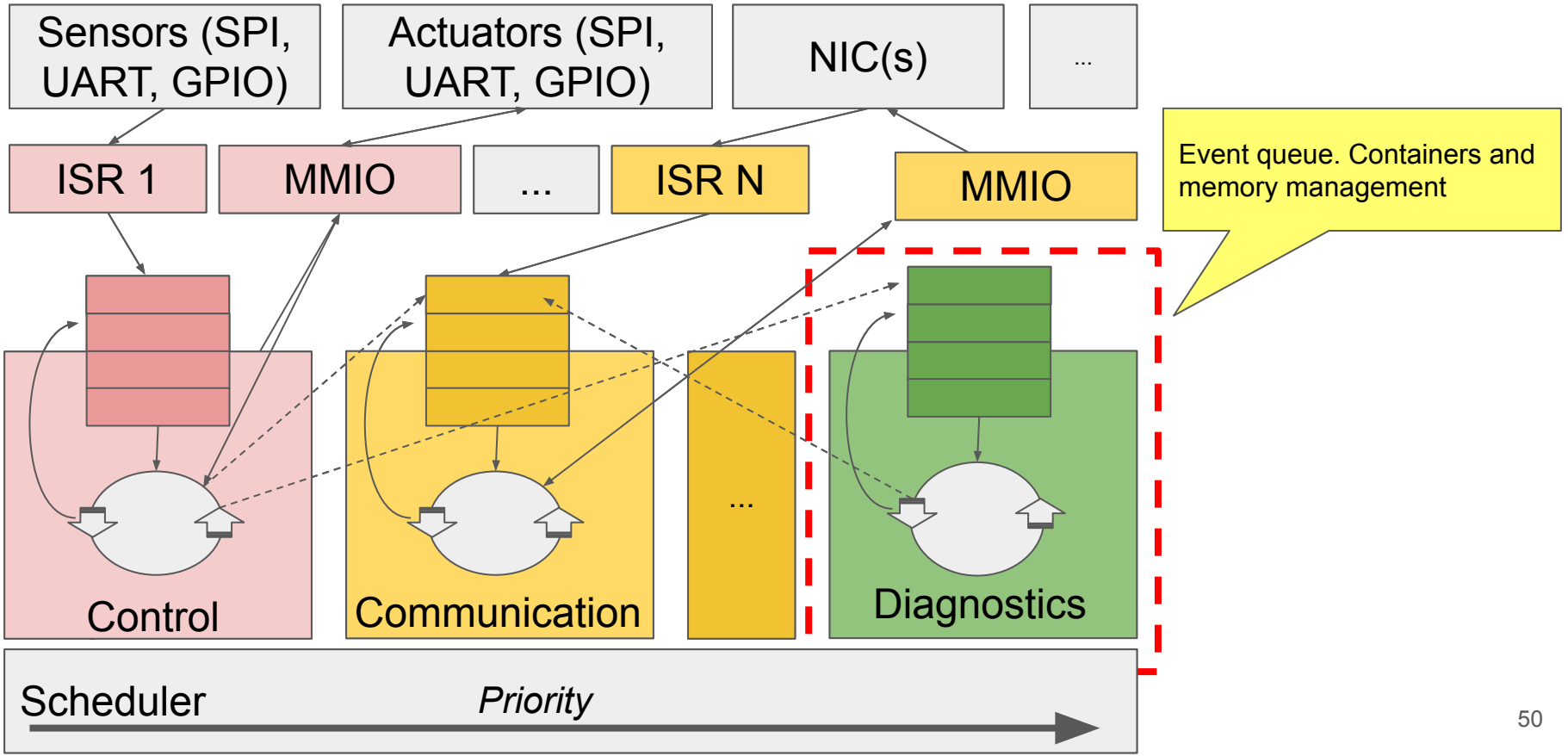
```
template <typename TxInterruptControllBit, typename RxTxRegisters>
struct uart_control {...};

TEST(uart_control, enable_tx_interrupt_bit_is_set)
{
    StrictMock<MockBit> tx_intr_control_bit;
    //...
    EXPECT_CALL(tx_intr_control_bit, set());
     //...
    drivers::uart::uart_control uart_control(tx_intr_control_bit, rxtx);

    uart_control.enable_tx_interrupt();
}
```

```
struct MockBit {
    MOCK_METHOD0(set, void());
    MOCK_METHOD0(unset, void());
};
```

C++ for device drivers development. Summary

1. Testable and portable code based on template metaprogramming and compile-time polymorphism.
2. MMIO abstraction idea is not novel (Kvasir). But no “standard/default” framework.
3. Considering new language features, the topic can be interesting for metaprogramming “gurus”.

C++ application. Examples



Memory allocation

Option without heap-allocation:

- objects with automatic or static storage duration;
- objects (containers) with embedded storage;
- custom allocation (custom allocator, new/delete overloading);
- pool allocation;
- intrusive containers.

Containers and algorithms

Framework	Exception handling	Heap allocation	Dependency on std
Boost*	Configurable	Not used*	Hosted
EASTL*	Configurable	Not used*	Freestanding
STL	Default by design	By design	Part of Hosted

1. Boost (*-container, intrusive, pool).
2. EASTL - Electronic Arts Standard Template Library (*-fixed_list, fixed_set, fixed_*...).
3. STL - Standard Template Library

Event type

```
class Event {
public:
     Timestamp getTimestamp() const;
//...
private:
    eastl::fixed_function<config::max_function_capture_buffer> _function;
//...
#ifdef EVENT_DEBUG_ENABLED
    eastl::fixed_string<config::max_event_info_string> _event_info;
#endif
};
```

Types with fixed size storage

```
class Event {
public:
     Timestamp getTimestamp() const;
//...
private:
    eastl::fixed_function<config::max_function_capture_buffer> _function;
//...
#ifdef EVENT_DEBUG_ENABLED
    eastl::fixed_string<config::max_event_info_string> _event_info;
#endif
};
```

Alternatives for std types but without heap allocation

Containers. Custom allocator

```
std::vector<Event
     , allocator_once_returns_pointer_to_static_buffer_of_size<N>> v(...);

v.reserve(N);

if (v.size() < v.capacity())
    v.emplace_back(...);
```

Containers. Custom allocator

```
std::vector<Event
     , allocator_once_returns_pointer_to_static_buffer_of_size<N>> v(...);

v.reserve(N);

if (v.size() < v.capacity())
    v.emplace_back(...);
```

Allocator returning static buffer big enough to store N objects

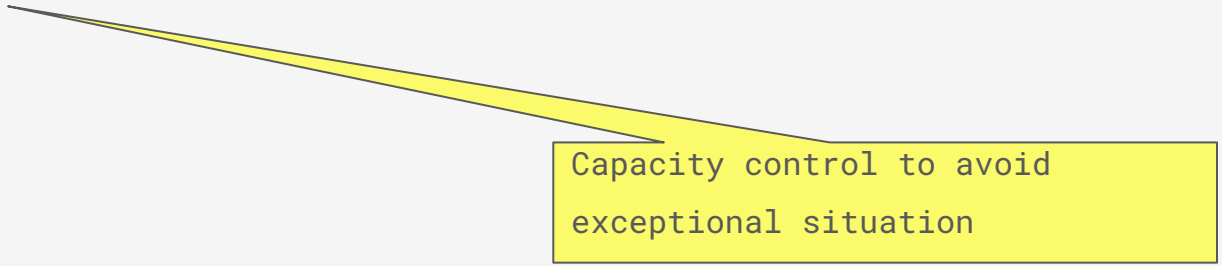
Containers. Custom allocator

```
std::vector<Event
     , allocator_once_returns_pointer_to_static_buffer_of_size<N>> v(...);

v.reserve(N);

if (v.size() < v.capacity())
    v.emplace_back(...);
```

Single memory allocation request

Containers. Custom allocator

```
std::vector<Event
     , allocator_once_returns_pointer_to_static_buffer_of_size<N>> v(...);

v.reserve(N);

if (v.size() < v.capacity())
    v.emplace_back(...);
```



Container compatibility

```
using queue_container = boost::container::static_vector<Event, 3>;
auto cmp_by_timestamp = [](auto const& left, auto const& right) {
    return left.getTimestamp() > right.getTimestamp();
};
using queue = std::priority_queue<Event, queue_container, decltype(cmp_by_timestamp)>;
queue q(cmp_by_timestamp);
if (q.size() < queue_container::static_capacity) {
    q.push(Event{100});
} else { /*...*/ }
auto const& top_item = q.top();
```

Container compatibility

```
using queue_container = boost::container::static_vector<Event, 3>;
auto cmp_by_timestamp = [](auto const& left, auto const& right) {
    return left.getTimestamp() > right.getTimestamp();
};
using queue = std::priority_queue<Event, queue_container, decltype(cmp_by_timestamp)>;
queue q(cmp_by_timestamp);
if (q.size() < queue_container::static_capacity) {
    q.push(Event{100});
} else { /*...*/ }
auto const& top_item = q.top();
```

Fixed-capacity vector compatible with stl

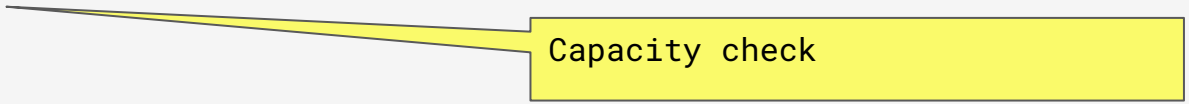
Container compatibility

```
using queue_container = boost::container::static_vector<Event, 3>;
auto cmp_by_timestamp = [](auto const& left, auto const& right) {
    return left.getTimestamp() > right.getTimestamp();
};
using queue = std::priority_queue<Event, queue_container, decltype(cmp_by_timestamp)>;
queue q(cmp_by_timestamp);
if (q.size() < queue_container::static_capacity) {
    q.push(Event{100});
} else { /*...*/ }
auto const& top_item = q.top();
```

Heap data structure using static_vector as backend storage. Alternative is boost::priority_queue. Overhead due to copying in heap operations

Container compatibility

```
using queue_container = boost::container::static_vector<Event, 3>;
auto cmp_by_timestamp = [](auto const& left, auto const& right) {
    return left.getTimestamp() > right.getTimestamp();
};
using queue = std::priority_queue<Event, queue_container, decltype(cmp_by_timestamp)>;
queue q(cmp_by_timestamp);
if (q.size() < queue_container::static_capacity) {
    q.push(Event{100});
} else { /*...*/ }
auto const& top_item = q.top();
```



Fixed-size containers

```
eastl::fixed_set<Event, 100, false
    , decltype(cmp_by_timestamp)> queue(cmp_by_timestamp);


if (queue.size() < queue.max_size()) {
    queue.insert(Event{1000});
}
```

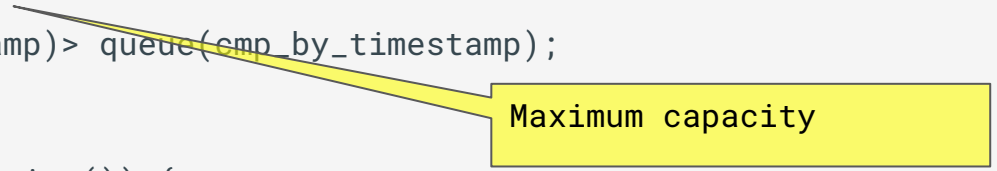
Fixed-size containers

```
eastl::fixed_set<Event, 100, false
    , decltype(cmp_by_timestamp)> queue(cmp_by_timestamp);

if (queue.size() < queue.max_size()) {
    queue.insert(Event{1000});
}
```

Available in **EASTL**
along with fixed_vector,
fixed_string, fixed_map,
fixed_hash_map, ...

Fixed-size containers

```
eastl::fixed_set<Event, 100, false
    , decltype(cmp_by_timestamp)> queue(cmp_by_timestamp);


if (queue.size() < queue.max_size()) {
    queue.insert(Event{1000});
}
```



Fixed-size containers

```
eastl::fixed_set<Event, 100, false
    , decltype(cmp_by_timestamp)> queue(cmp_by_timestamp);


if (queue.size() < queue.max_size()) {
    queue.insert(Event{1000});
}
```

Heap allocation is disabled

Fixed-size containers

```
eastl::fixed_set<Event, 100, false
    , decltype(cmp_by_timestamp)> queue(cmp_by_timestamp);


if (queue.size() < queue.max_size()) {
    queue.insert(Event{1000});
}
```

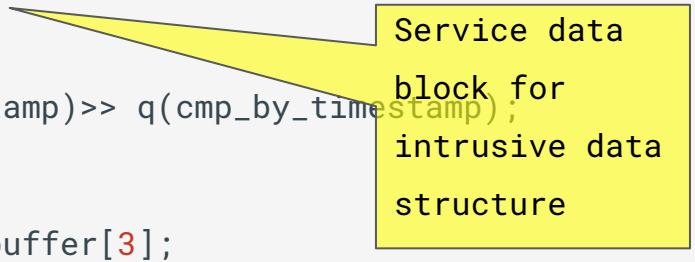
Space availability check

Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};
boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);

std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[3];
boost::simple_segregated_storage<size_t> storage;
storage.add_block(buffer, sizeof(buffer), sizeof(Event));

auto mem = storage.malloc();
if (mem) q.insert(*new (mem) Event({100}));
```

Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};

boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);


std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[3];

boost::simple_segregated_storage<size_t> storage;

storage.add_block(buffer, sizeof(buffer), sizeof(Event));


auto mem = storage.malloc();

if (mem) q.insert(*new (mem) Event({100}));
```



Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};

boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);


std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[3];
boost::simple_segregated_storage<size_t> storage;
storage.add_block(buffer, sizeof(buffer), sizeof(Event));


auto mem = storage.malloc();
if (mem) q.insert(*new (mem) Event({100}));
```

Intrusive set. No allocation is performed

Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};

boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);


std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[N];
boost::simple_segregated_storage<size_t> storage;
storage.add_block(buffer, sizeof(buffer), sizeof(Event));


auto mem = storage.malloc();
if (mem) q.insert(*new (mem) Event({100}));
```

Aligned storage for N Event objects

Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};

boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);


std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[N];
boost::simple_segregated_storage<size_t> storage;
storage.add_block(buffer, sizeof(buffer), sizeof(Event));


auto mem = storage.malloc();
if (mem) q.insert(*new (mem) Event({100}));
```

Free-list pool with size equal to sizeof(Event)

Pool allocation and intrusive containers

```
class Event : public boost::intrusive::set_base_hook<> {/*...*/};


std::aligned_storage_t<sizeof(Event), alignof(Event)> buffer[3];
boost::simple_segregated_storage<size_t> storage;
storage.add_block(buffer, sizeof(buffer), sizeof(Event));


boost::intrusive::set<Event
     , boost::intrusive::compare<decltype(cmp_by_timestamp)>> q(cmp_by_timestamp);
auto mem = storage.malloc();
if (mem) q.insert(*new (mem) Event({100}));
```

Memory block allocation and object construction

Custom allocation

```
struct Event {
    static void* operator new(std::size_t sz);
    static void operator delete(void* p);
};

void* Event::operator new(std::size_t)
{
    return event_allocator::allocate();
}

void Event::operator delete(void* p)
{
    return event_allocator::free(p);
}
```

Custom allocation

```
struct Event {
    static void* operator new(std::size_t sz);
    static void operator delete(void* p);
};

void* Event::operator new(std::size_t)
{
    return event_allocator::allocate();
}

void Event::operator delete(void* p)
{
    return event_allocator::free(p);
}
```

Static methods access shared allocator

Custom allocation

```
template <uint32_t Id>
struct Event {
    static void* operator new(std::size_t sz);
    static void operator delete(void* p);
};

template <uint32_t Id>
void* Event<id>::operator new(std::size_t)
{
    return event_allocator<Id>::allocate();
}

template <uint32_t Id>
void Event<Id>::operator delete(void* p)
{
    return event_allocator<Id>::free(p);
}
```

Different instantiations will access different allocator instances

Custom allocation

```
template <uint32_t Id>
struct Event {
    static void* operator new(std::size_t sz);
    static void operator delete(void* p);
};

template <uint32_t Id>
void* Event<id>::operator new(std::size_t)
{
    return event_allocator<Id>::allocate();
}

template <uint32_t Id>
void Event<Id>::operator delete(void* p)
{
    return event_allocator<Id>::free(p);
}
```

```
template <uint32_t Id>
class event_allocator {
    static void* allocate();
    static buffer static_buffer;
};

template <uint32_t Id>
buffer event_allocator<Id>
     ::static_buffer;
```

Polymorphism based on variant

```
struct Spi { void send(); };

struct Uart { void send(); };

using transports = std::variant<Spi, Uart>;

transports make_transport() { return Uart{}; }


auto transport = make_transport();

std::visit([](auto &tr) { tr.send(); }, transport);
```

Polymorphism based on variant

```
struct Spi { void send(); };

struct Uart { void send(); };

using transports = std::variant<Spi, Uart>;

transports make_transport() { return Uart{}; }


auto transport = make_transport();

std::visit([](auto &tr) { tr.send(); }, transport);
```

No heap allocation guaranteed.

Alternatives: boost::variant, nonstd::variant

Polymorphism based on variant

```
struct Spi { void send(); };

struct Uart { void send(); };

using transports = std::variant<Spi, Uart>;

transports make_transport() { return Uart{}; }


auto transport = make_transport();

std::visit([](auto &tr) { tr.send(); }, transport);
```

Factory method returns variant

Polymorphism based on variant

```
struct Spi { void send(); };

struct Uart { void send(); };

using transports = std::variant<Spi, Uart>;

transports make_transport() { return Uart{}; }


auto transport = make_transport();

std::visit([](auto &tr) { tr.send(); }, transport);
```

Duck-typing inside generic lambda

Error handling without exceptions

```
std::optional<Data> read_data();
//...
if (const auto &data = read_data(); data) {
    process_data(*data);
} else {
    //...
}
```

Error handling without exceptions

```
std::optional<Data> read_data();
//...
if (const auto &data = read_data(); data) {
    process_data(*data);
} else {
    //...
}
```

Guaranteed not to allocate.

Alternatives: boost::optional, nonstd::optional

Error handling without exceptions

```
std::optional<Data> read_data();
//...
if (const auto &data = read_data(); data) {
    process_data(*data);
} else {
    //...
}
// or
template<typename T>
using value_or_error = std::variant<T, std::error_code>;
value_or_error<Data> read_data();
```

Guaranteed not to allocate.
Specialized implementations:
nonstd::expected from
expected-lite or boost::outcome

Conclusion

- Best practices – well-known to the C++ community – are essential in embedded development
- C++ language has mechanisms enabling testability and portability of the embedded code (encapsulation, polymorphism)
- Availability of the 3rd party frameworks applicable for embedded development facilitates development
- There is an overlap between embedded and non-embedded domains (HPT, game-dev). Knowledge exchange is beneficial

Thank you!

Questions?