C++ in the World of Embedded Systems

VLADIMIR VISHNEVSKII

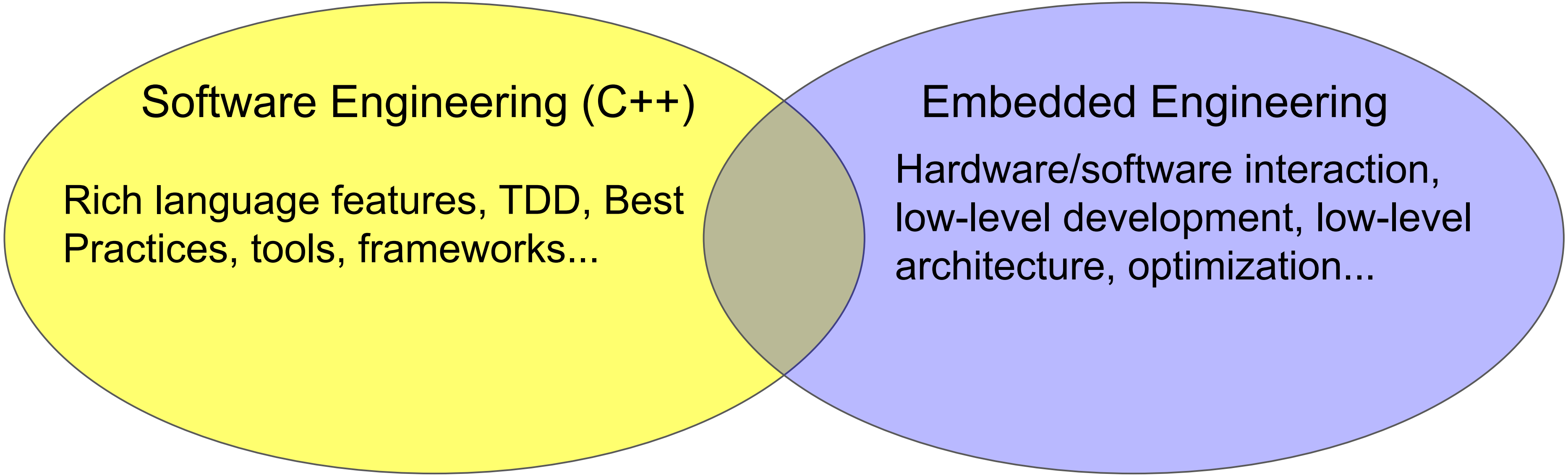
Cppcon
The C++ Conference
2022
September 12th-16th

Agenda

1. Overview of embedded systems
2. Typical architectures in embedded systems
3. Review of the C++ availability, applicability and potential 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

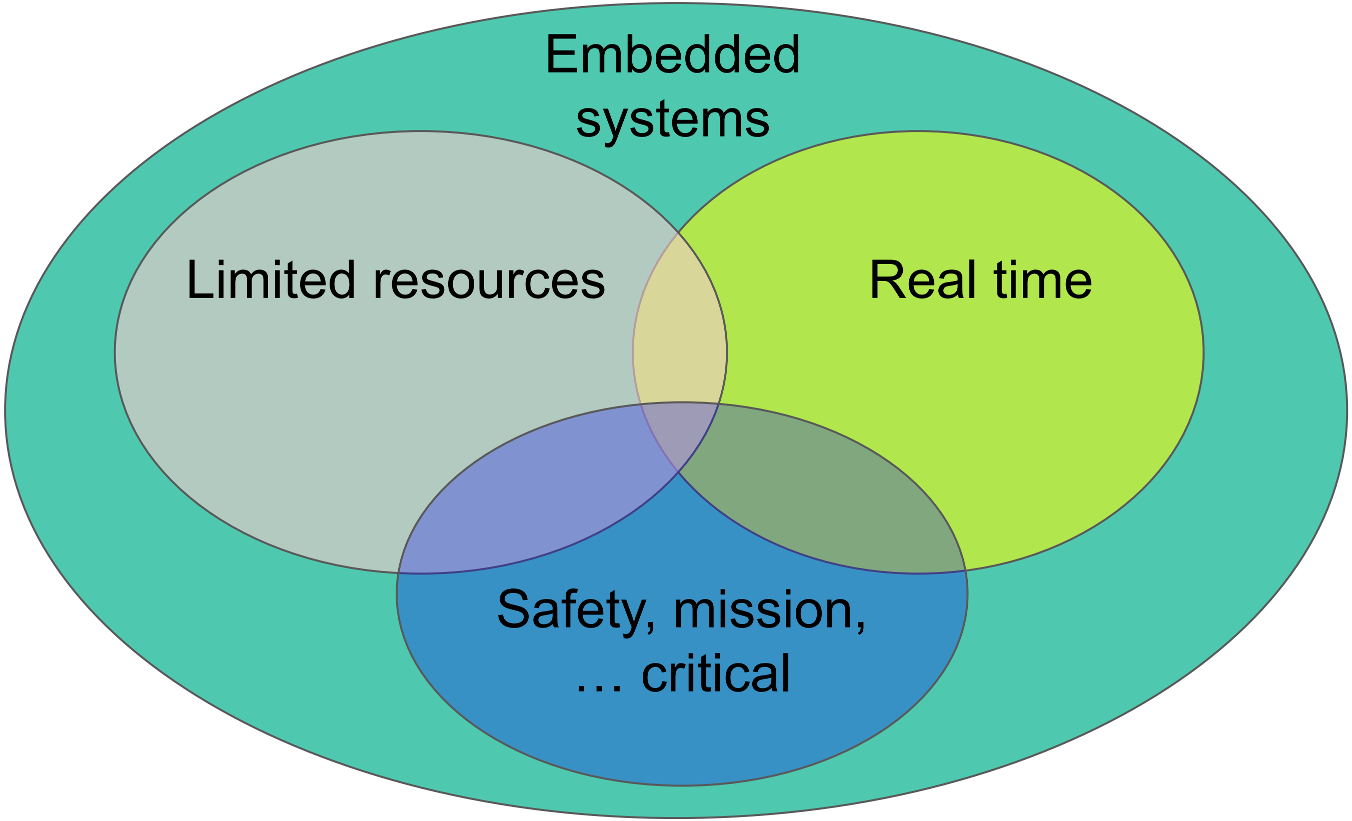
Non-formal definition:

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

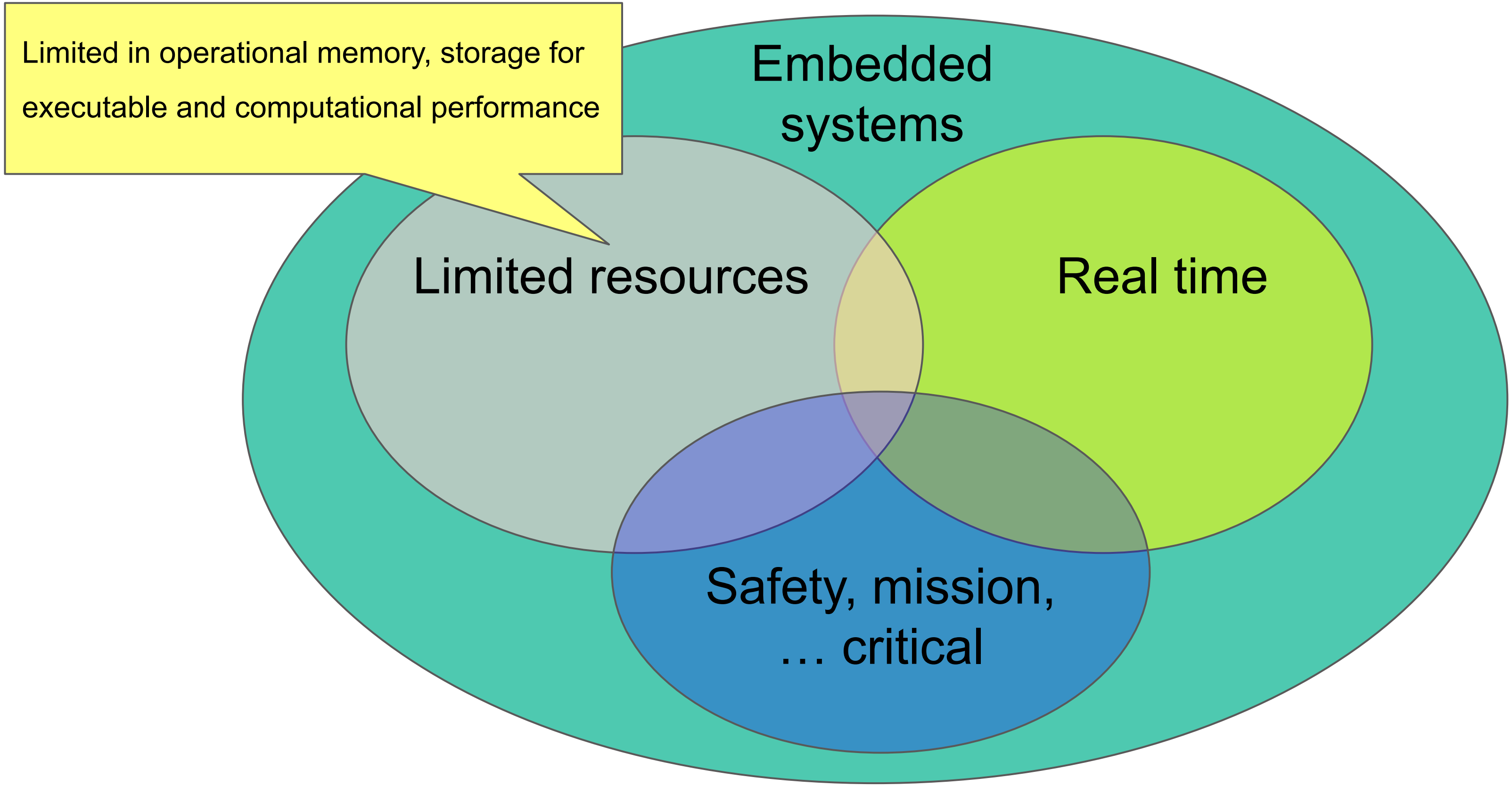
Examples:

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

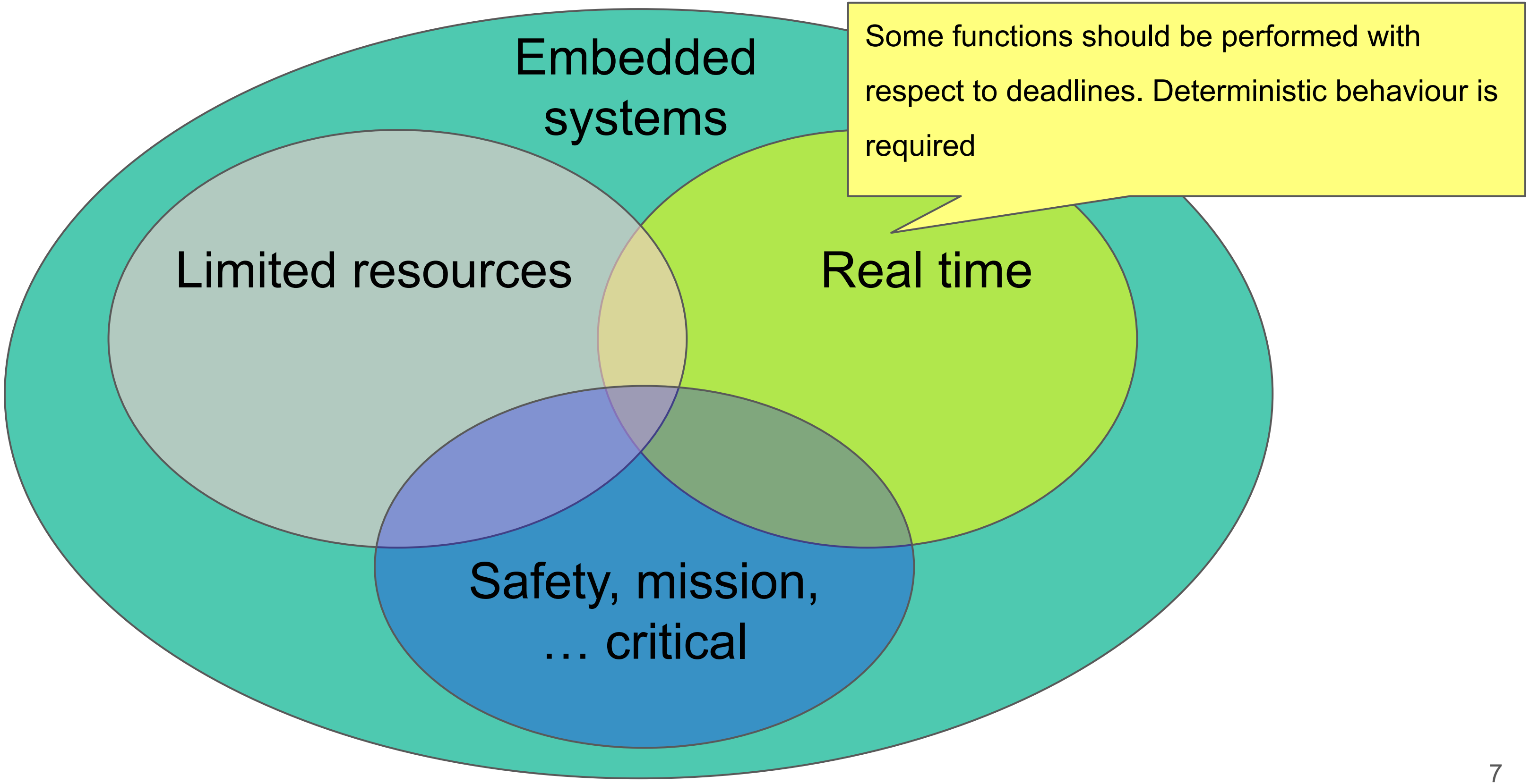
Embedded systems categories



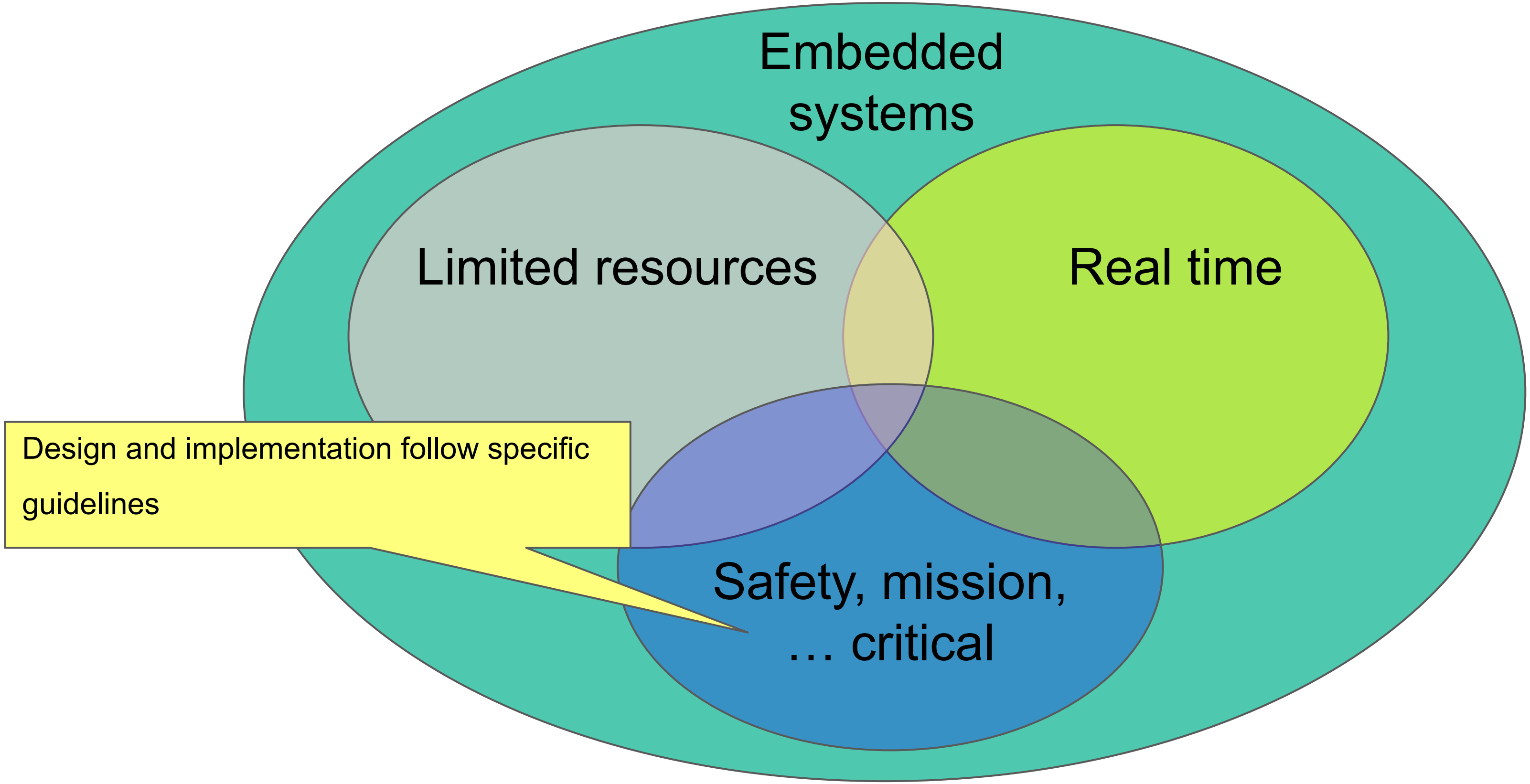
Embedded systems categories



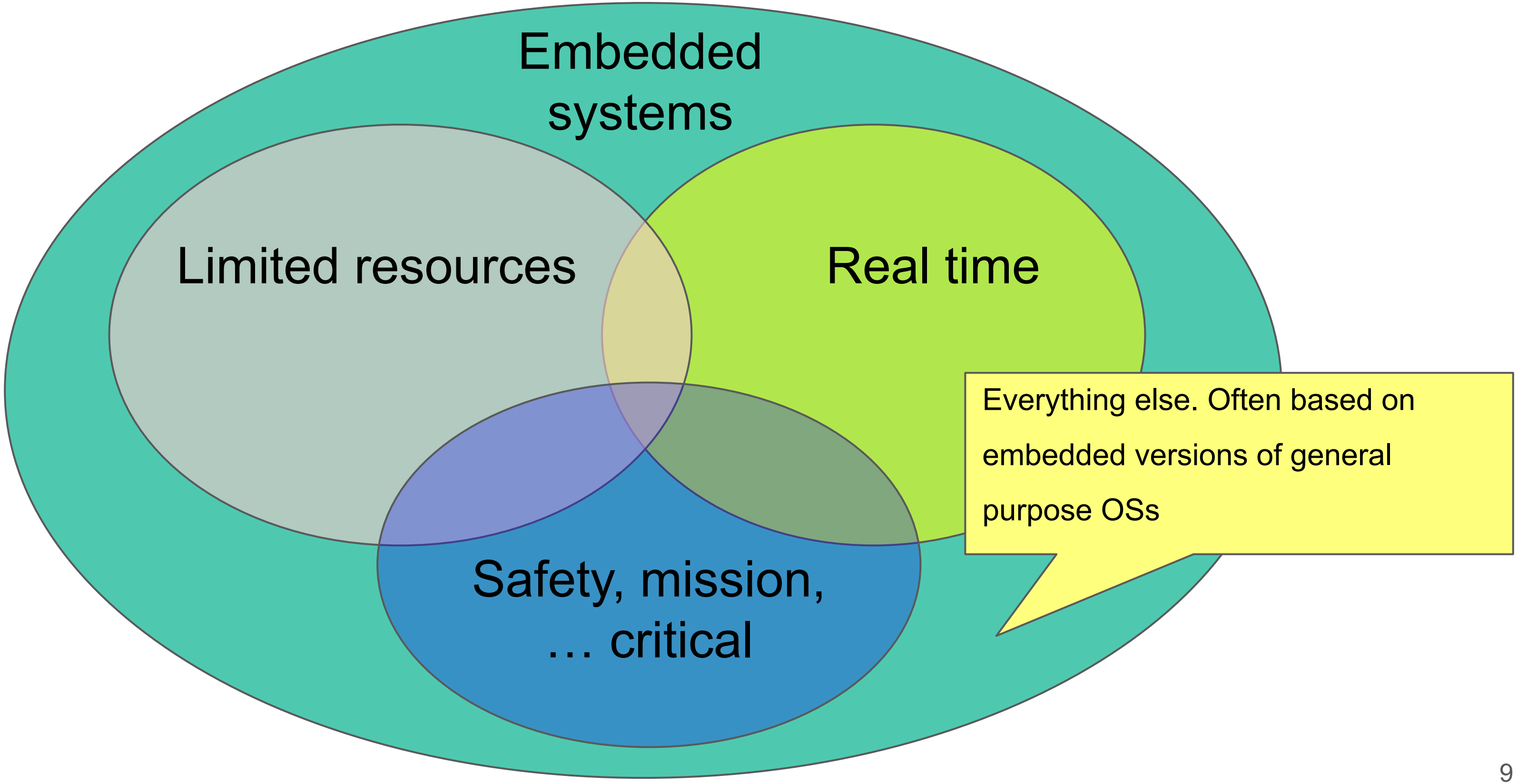
Embedded systems categories



Embedded systems categories



Embedded systems categories



Embedded development characteristic

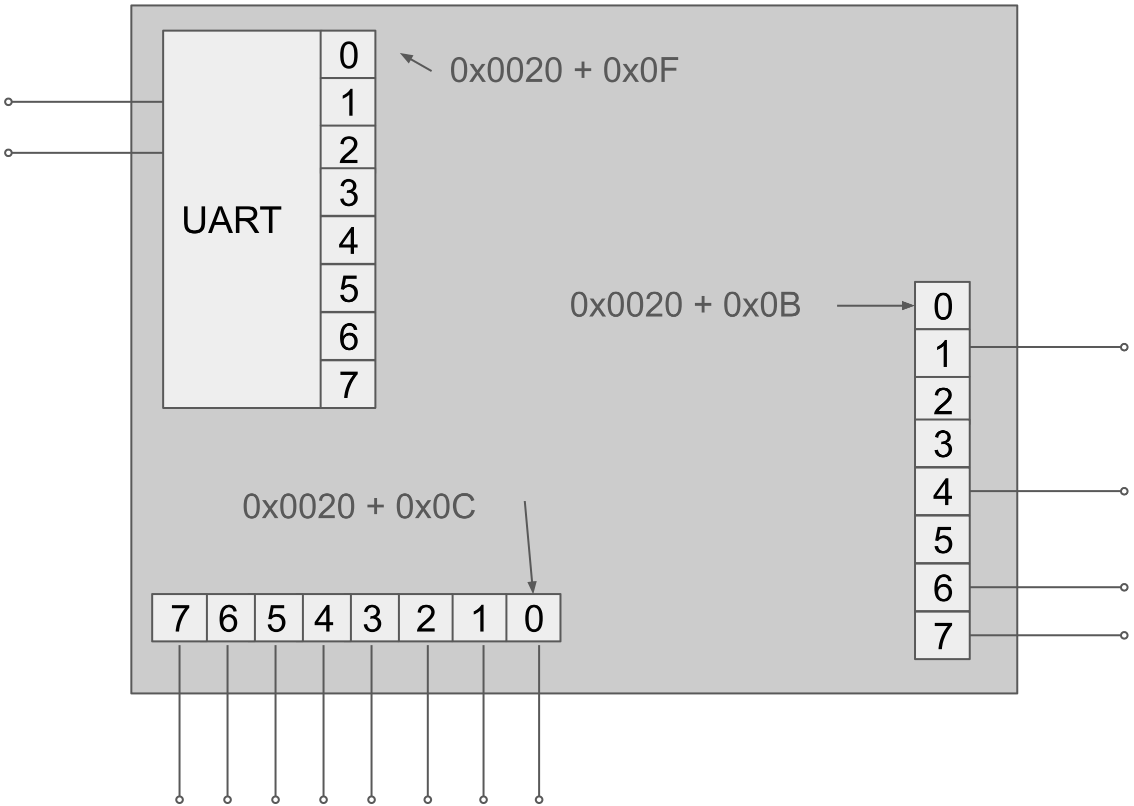
1. Software is deployed on target platform using specialized tooling
2. Diagnostics and debugging can be limited (software/hardware debuggers, disabled debugging, limited physical accessibility)
3. **Hardware can be unstable**
4. **Compilers might have defects**

Embedded development characteristic

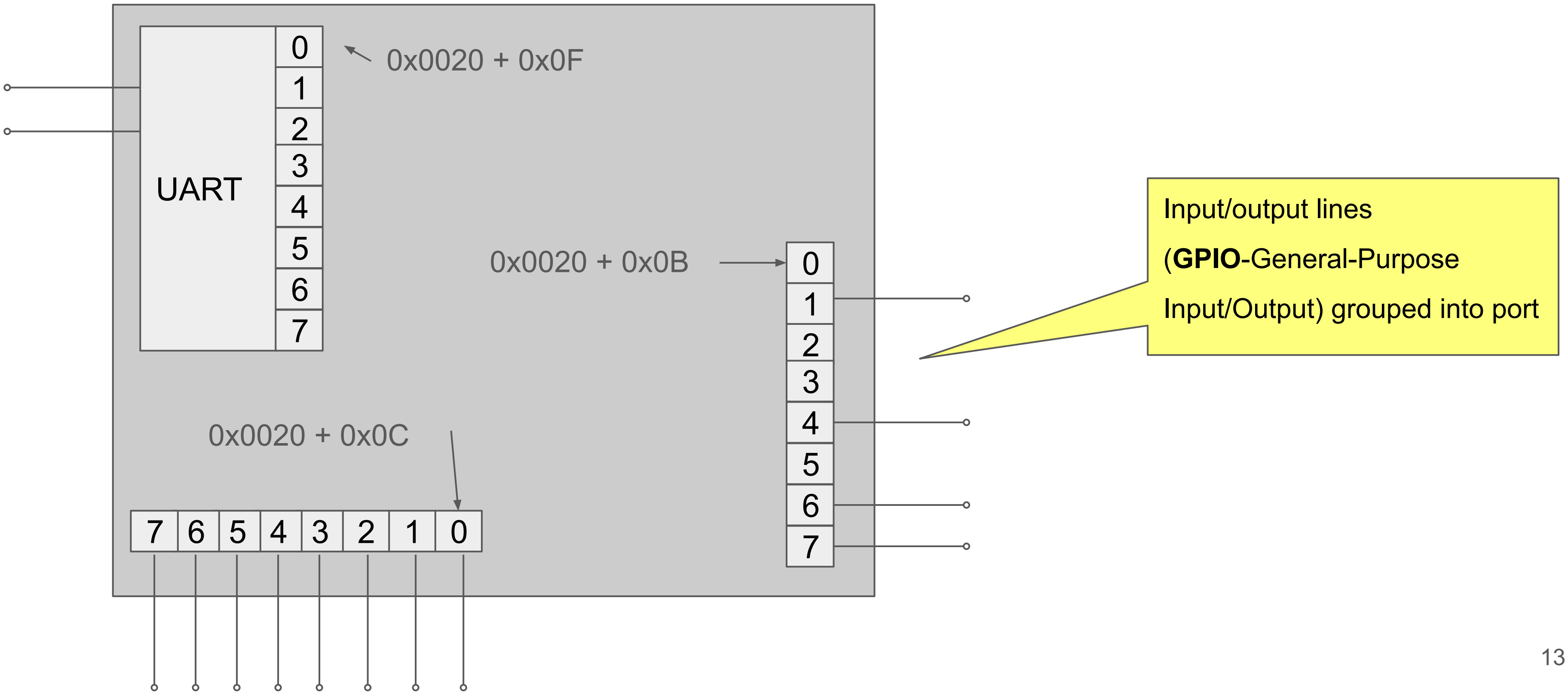
Code quality must be maximized prior to deployment:

- best practices (TDD, review, CI)
- primary testing on developer's machine (requires code portability)

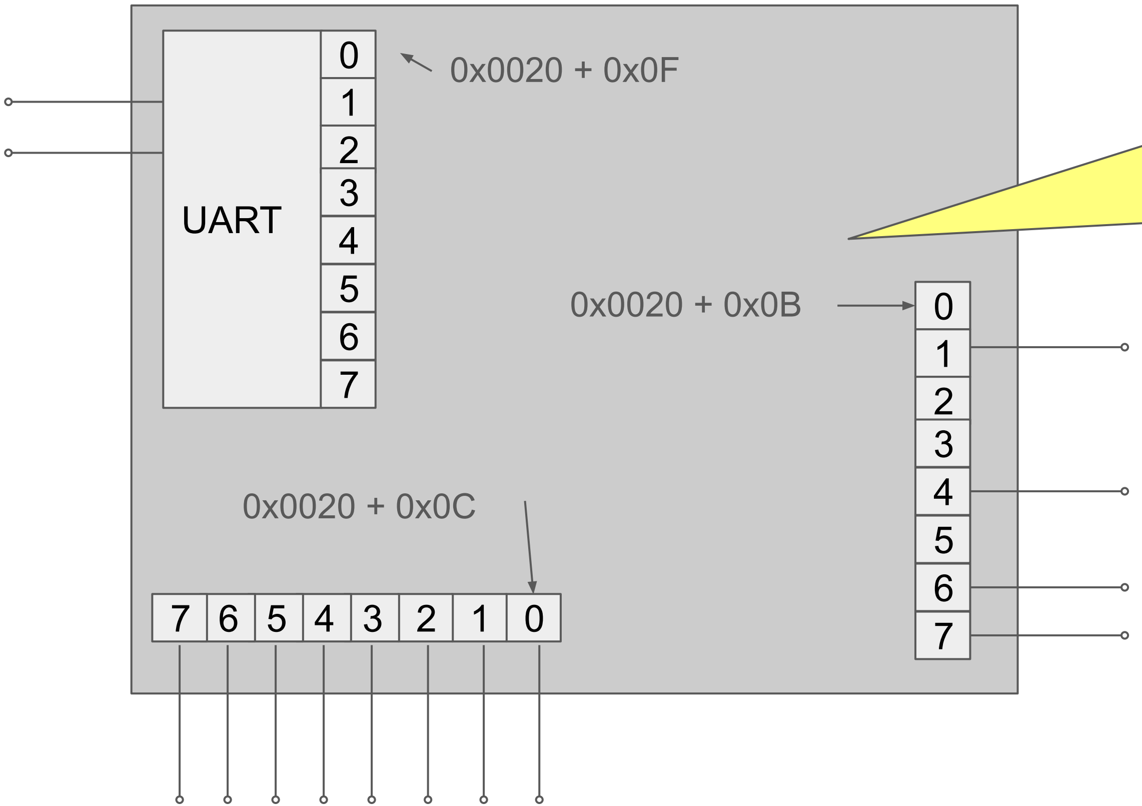
Trivial hardware platform



Trivial hardware platform



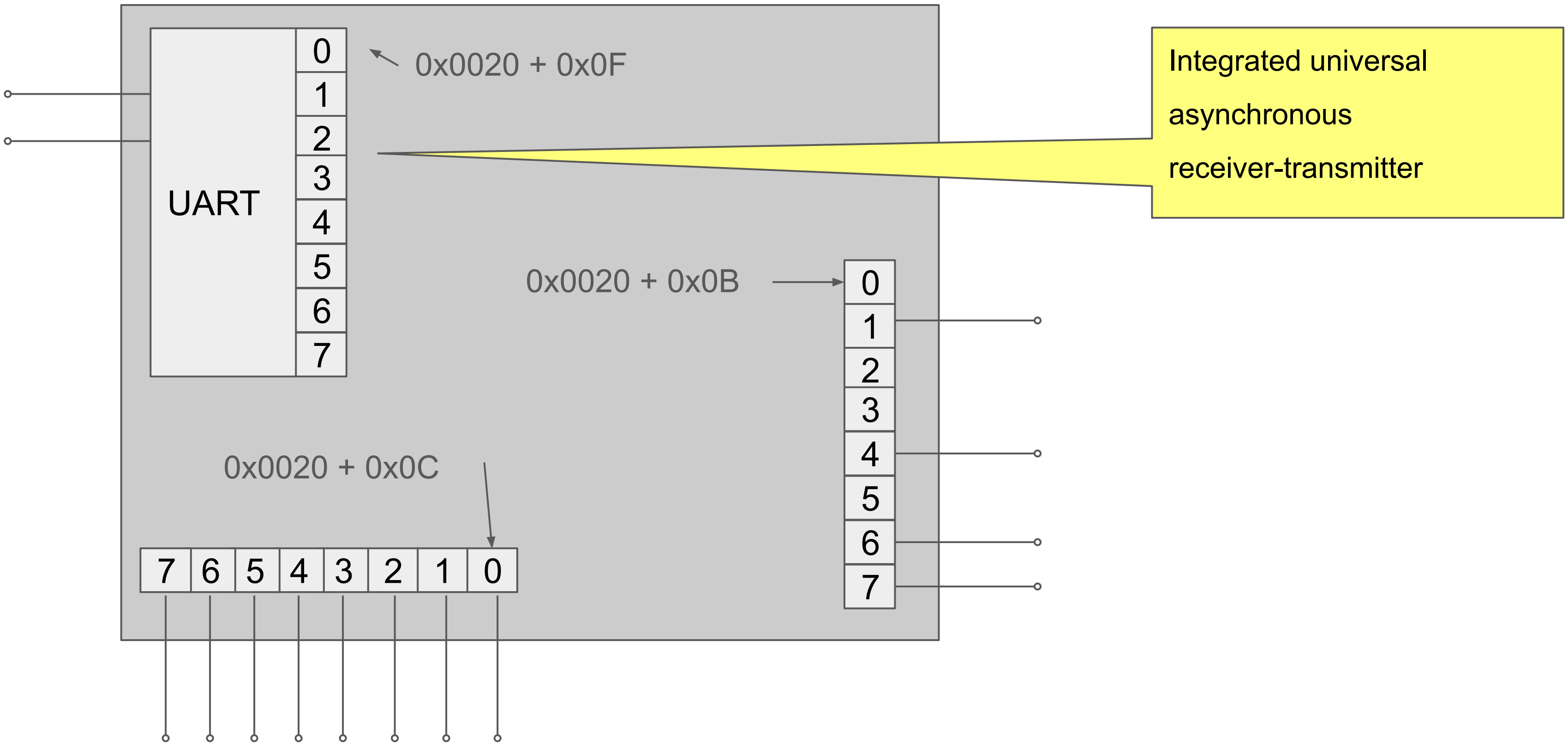
Trivial hardware platform



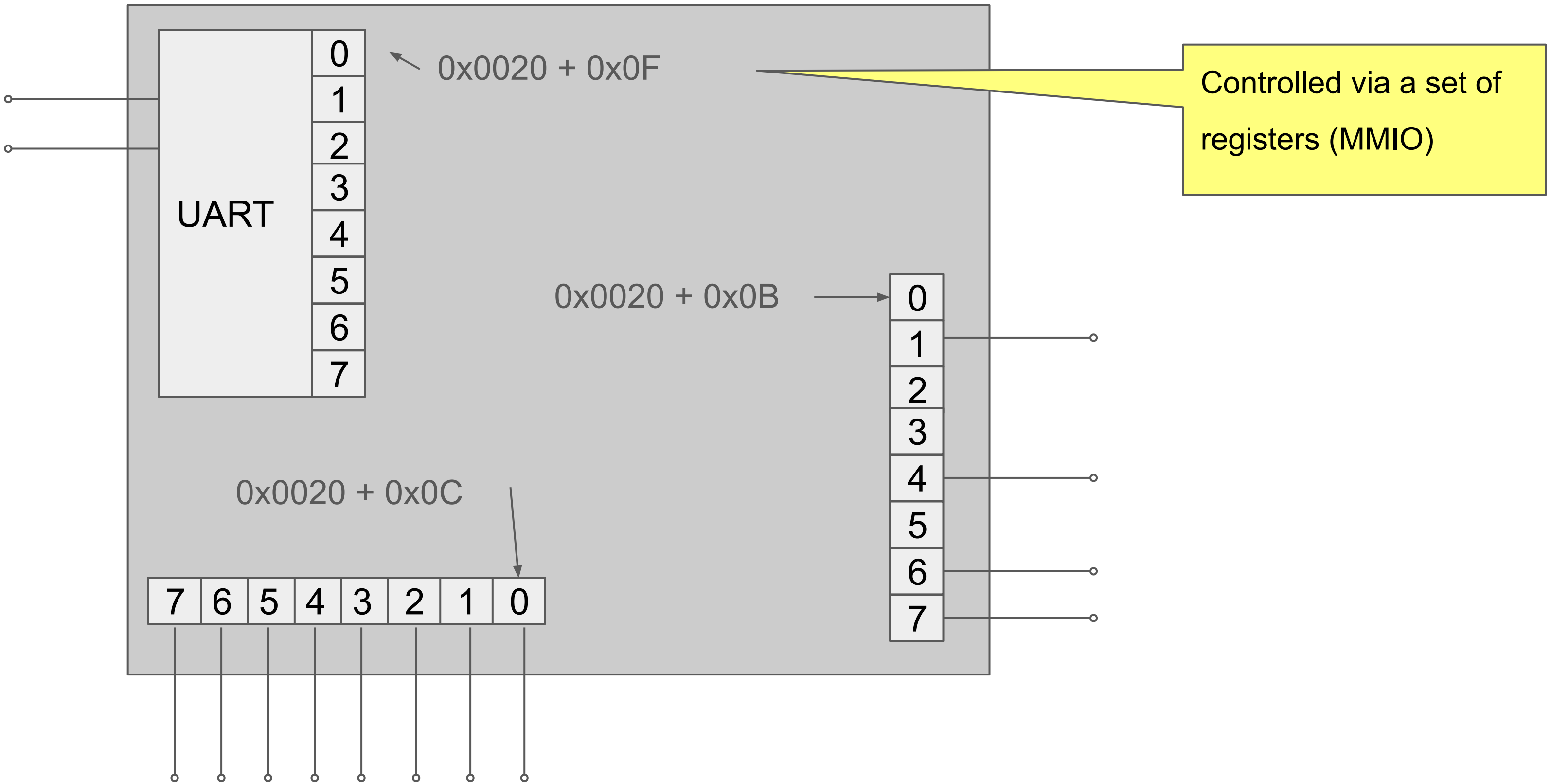
MMIO-Memory Mapped Input/Output

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

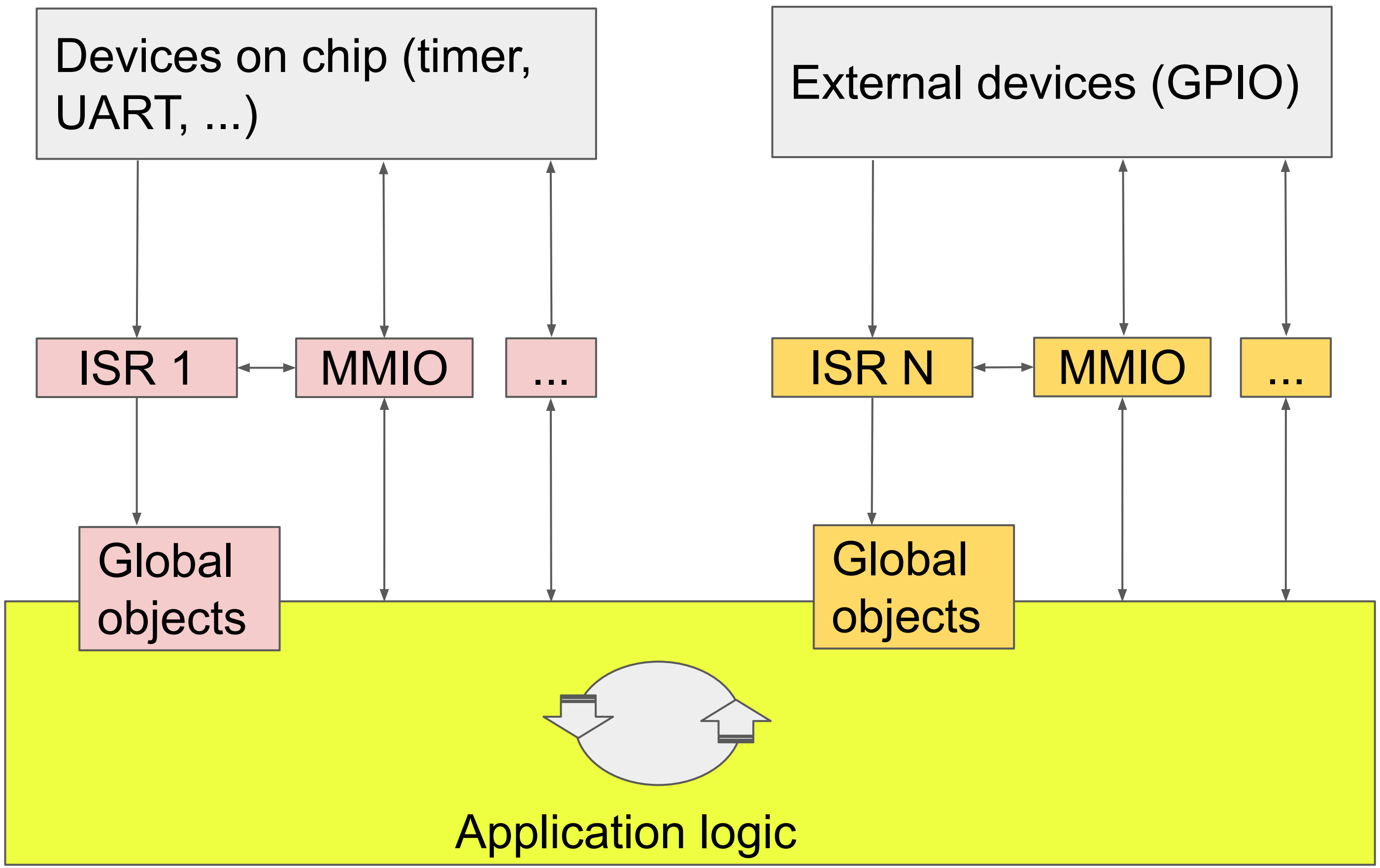
Trivial hardware platform



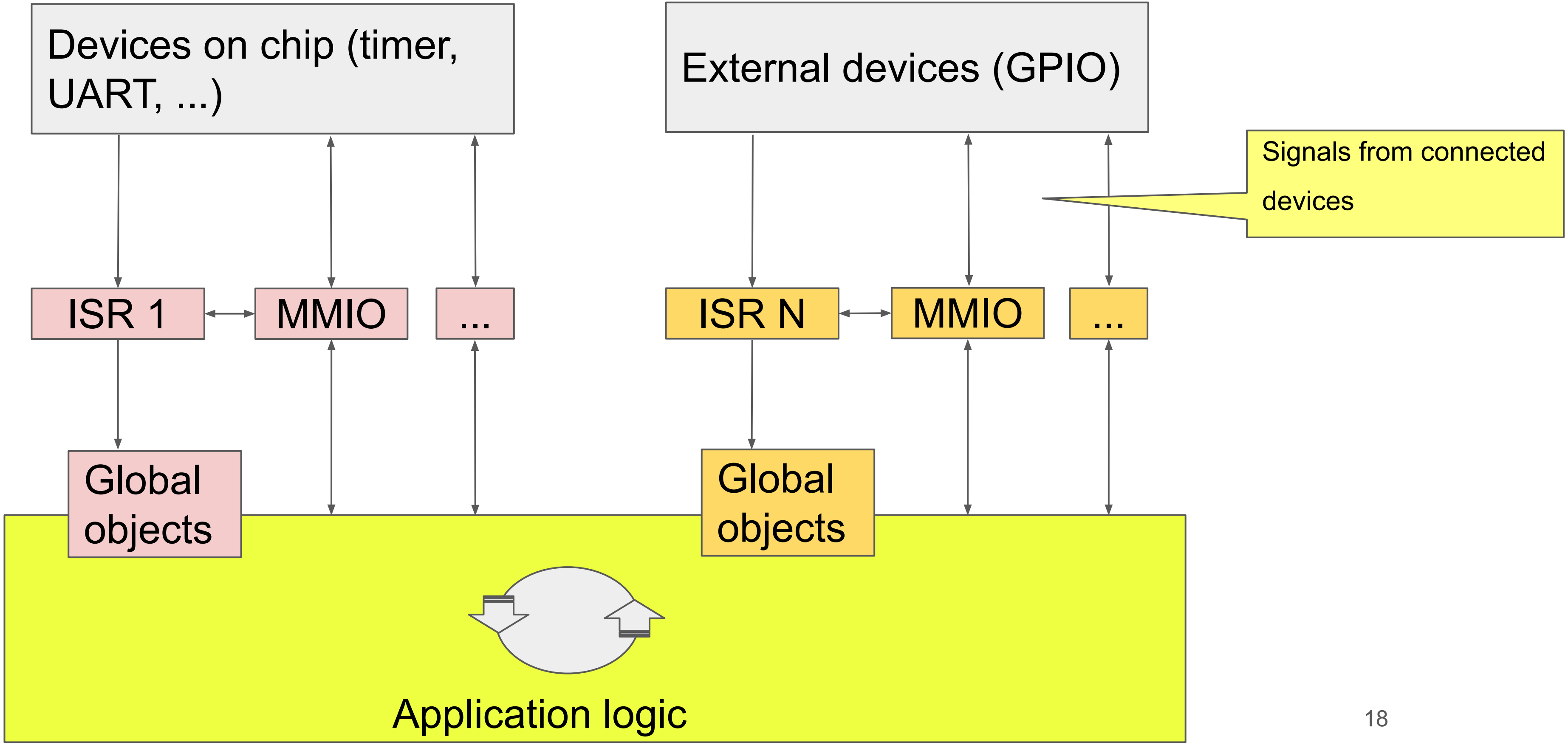
Trivial hardware platform



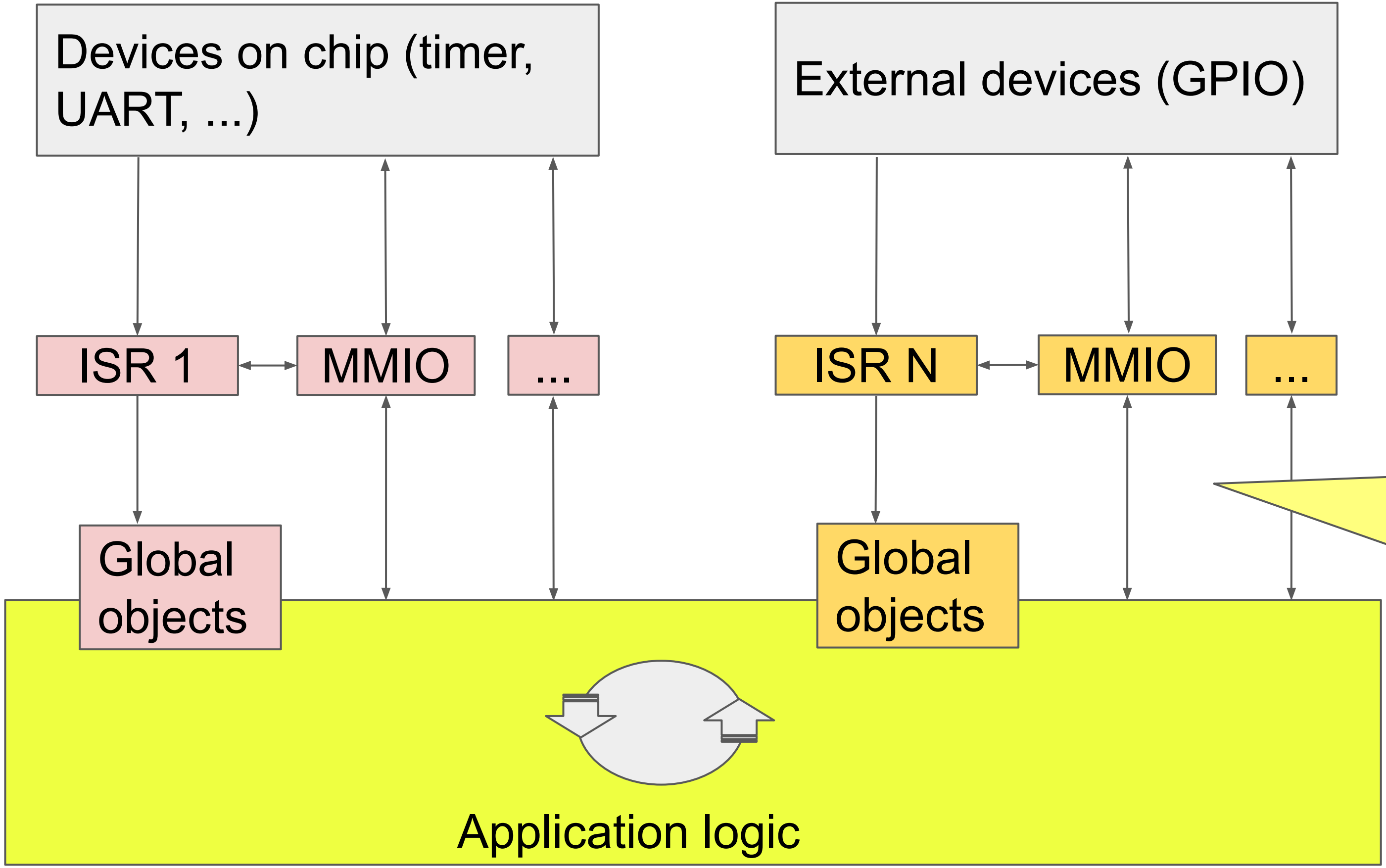
Trivial software architecture



Trivial software architecture

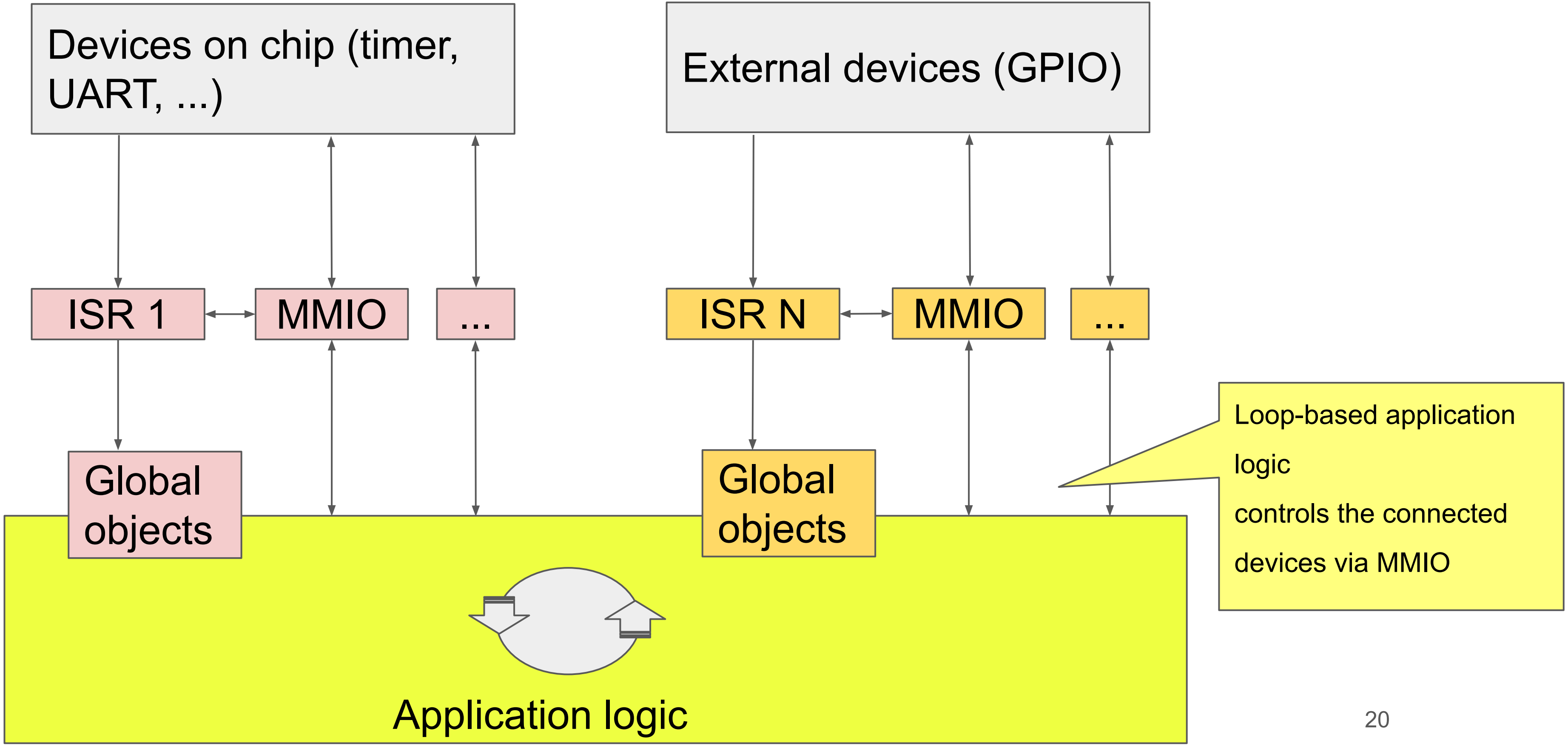


Trivial software architecture

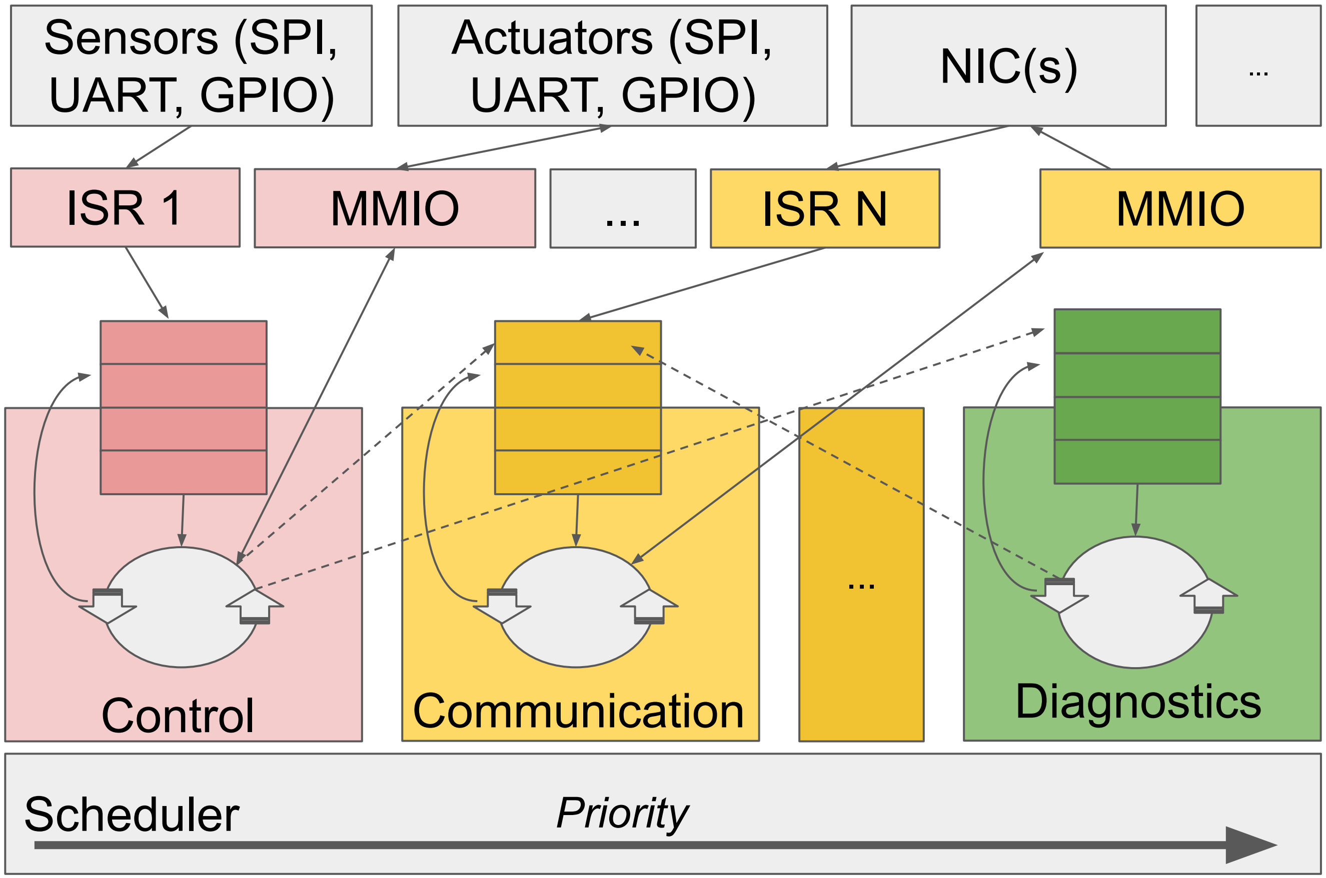


Signals from peripheral devices are delivered via **ISR** (Interrupt Service Routine) and **MMIO** polling

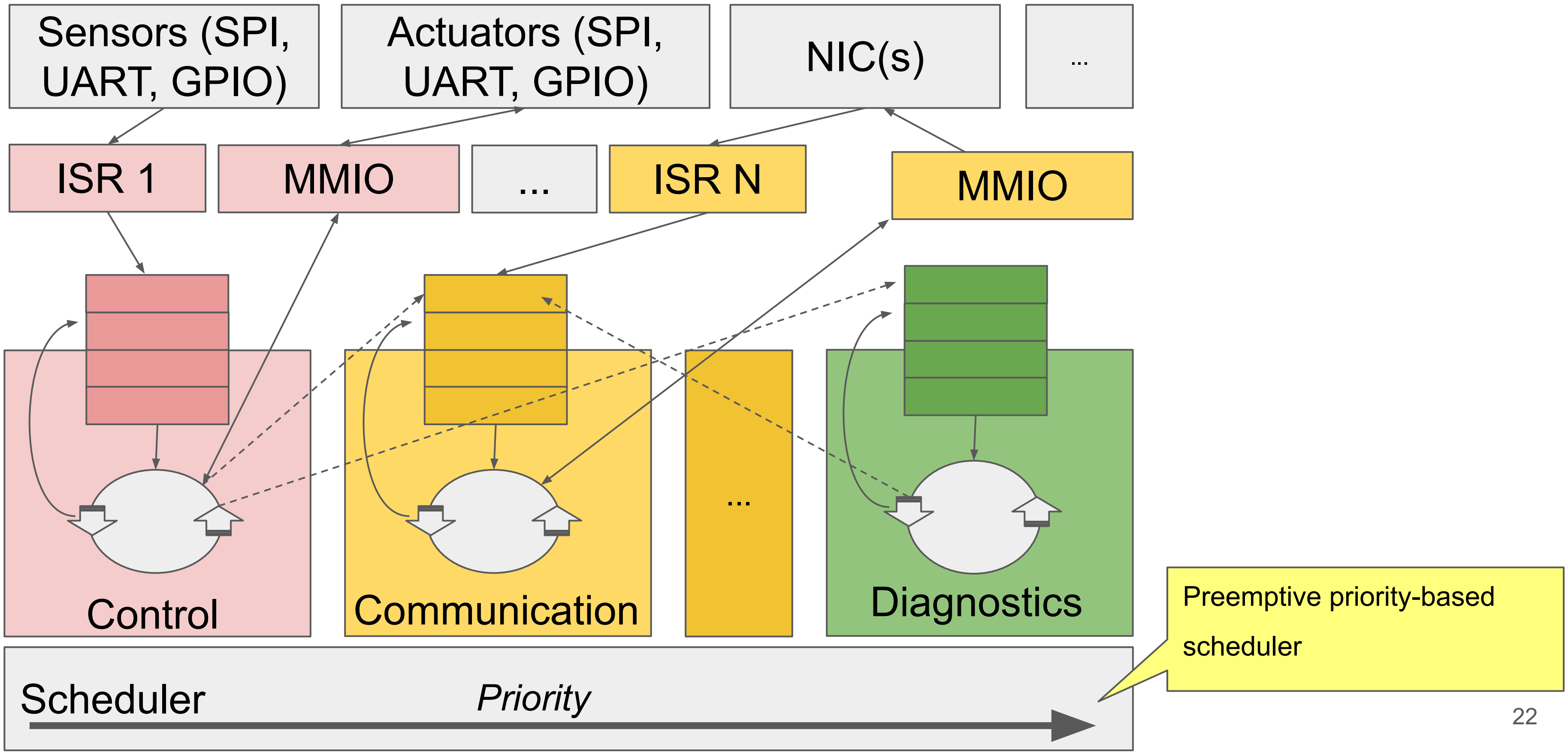
Trivial software architecture



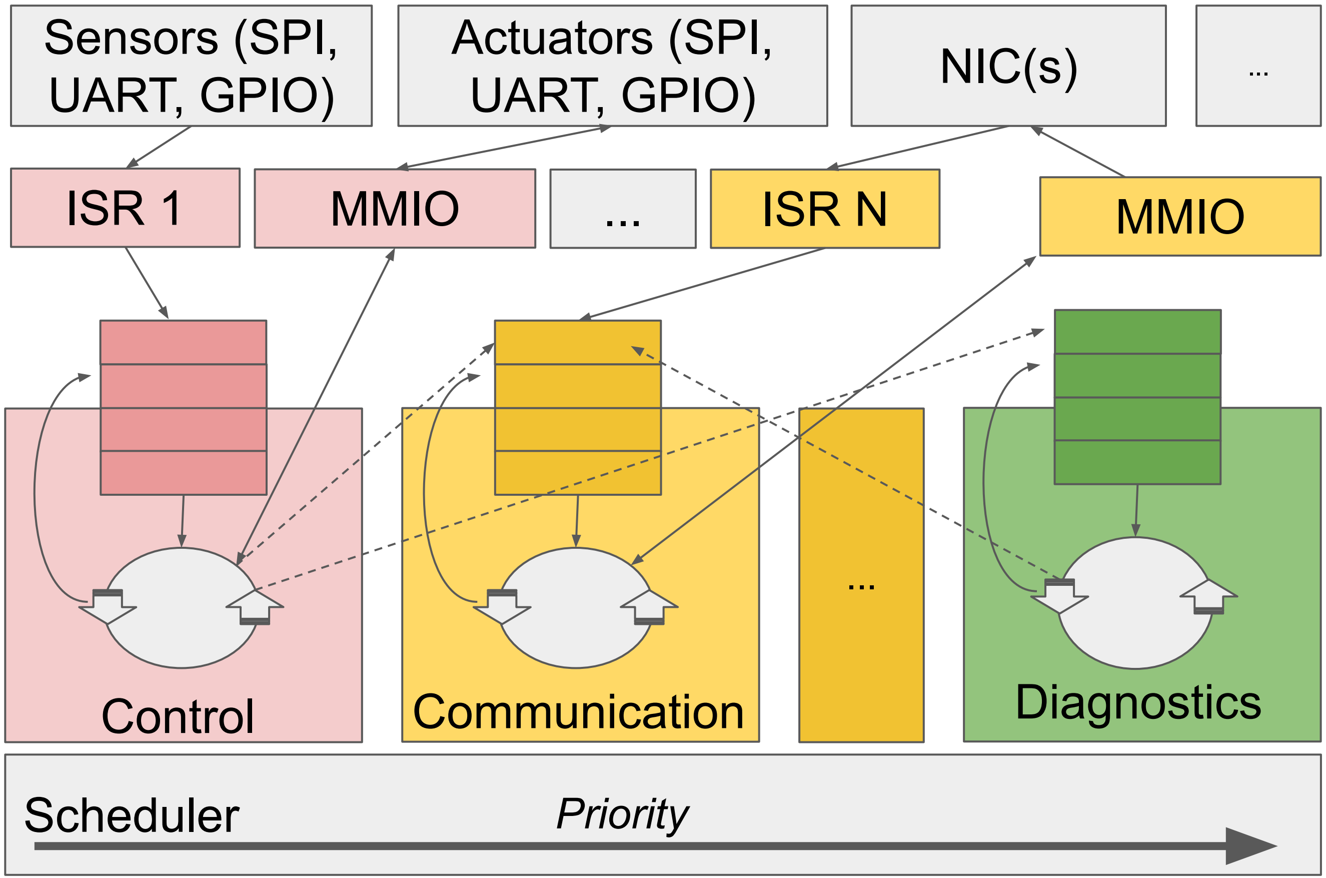
Connected real time system architecture



Connected real time system architecture

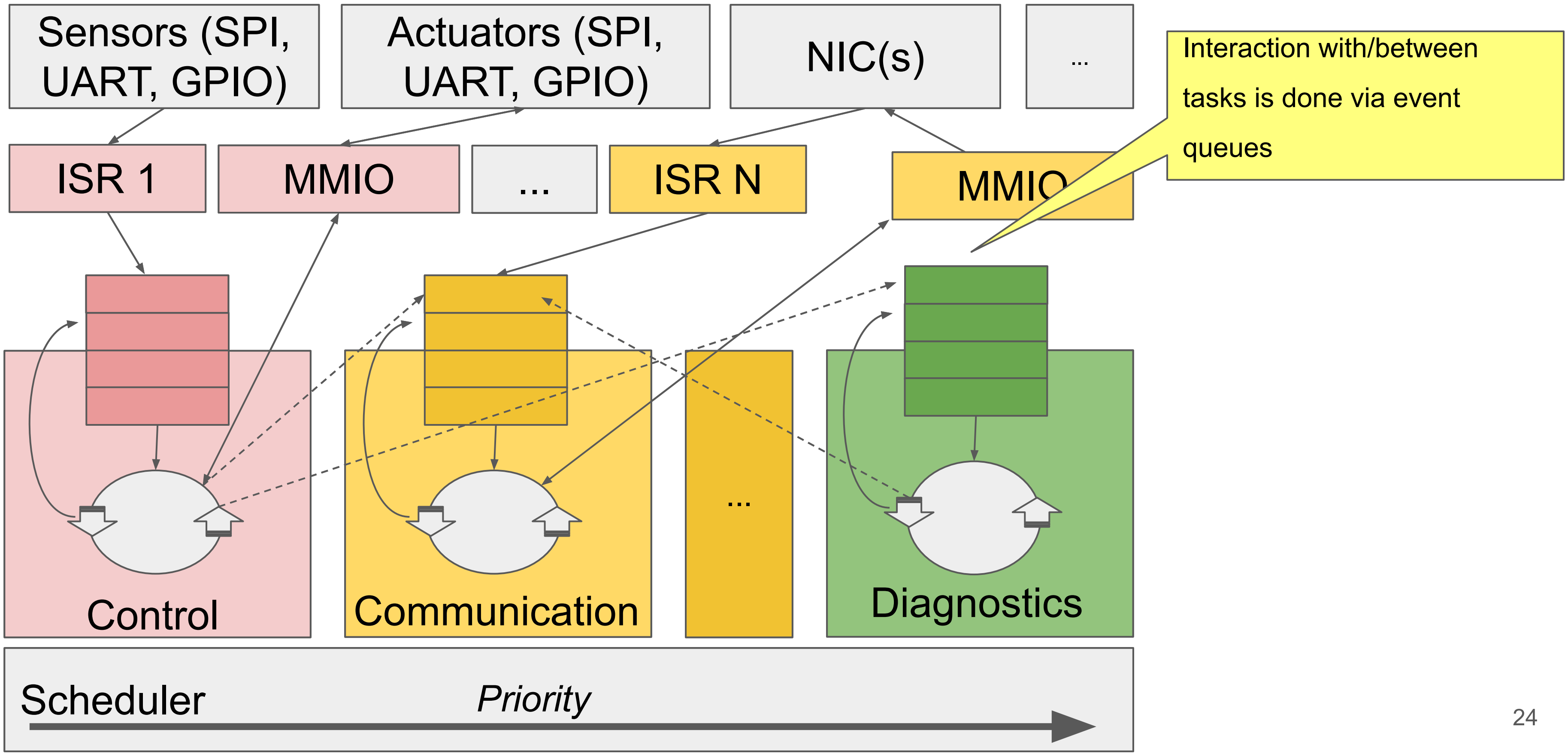


Connected real time system architecture



Tasks have static priorities and work exclusively with some set of resources (including memory and 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
- 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 standard library
- For some platforms standard library is not available

Limitations of C++ applicability

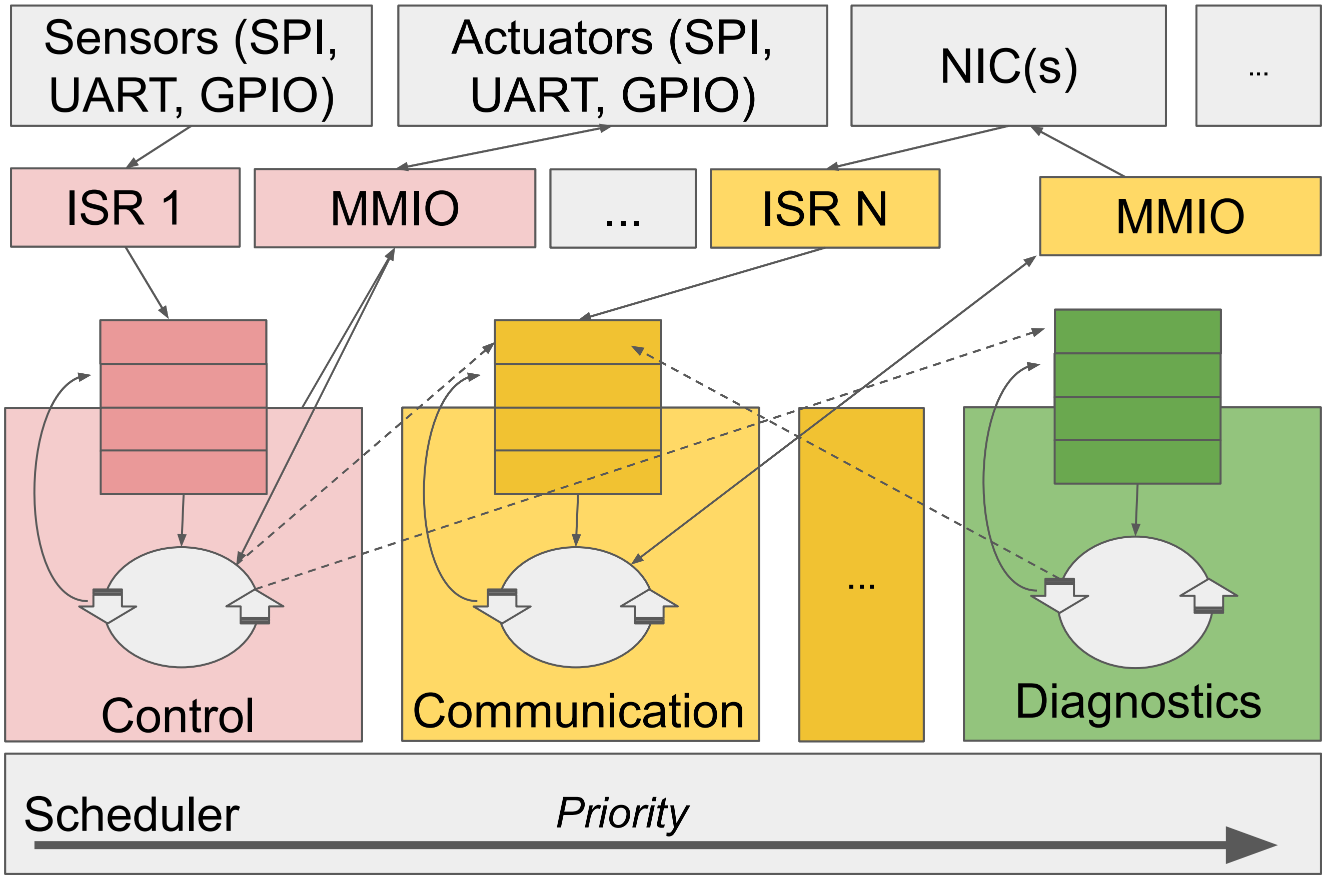
	Limited resources	Real time
Heap allocation	Maintenance overhead + fragmentation	Fragmentation + access synchronization => loss of determinizm
Exceptions	Increase of executable size	Analysis complexity, potential loss of determinism and low performance

Limitations of C++ applicability

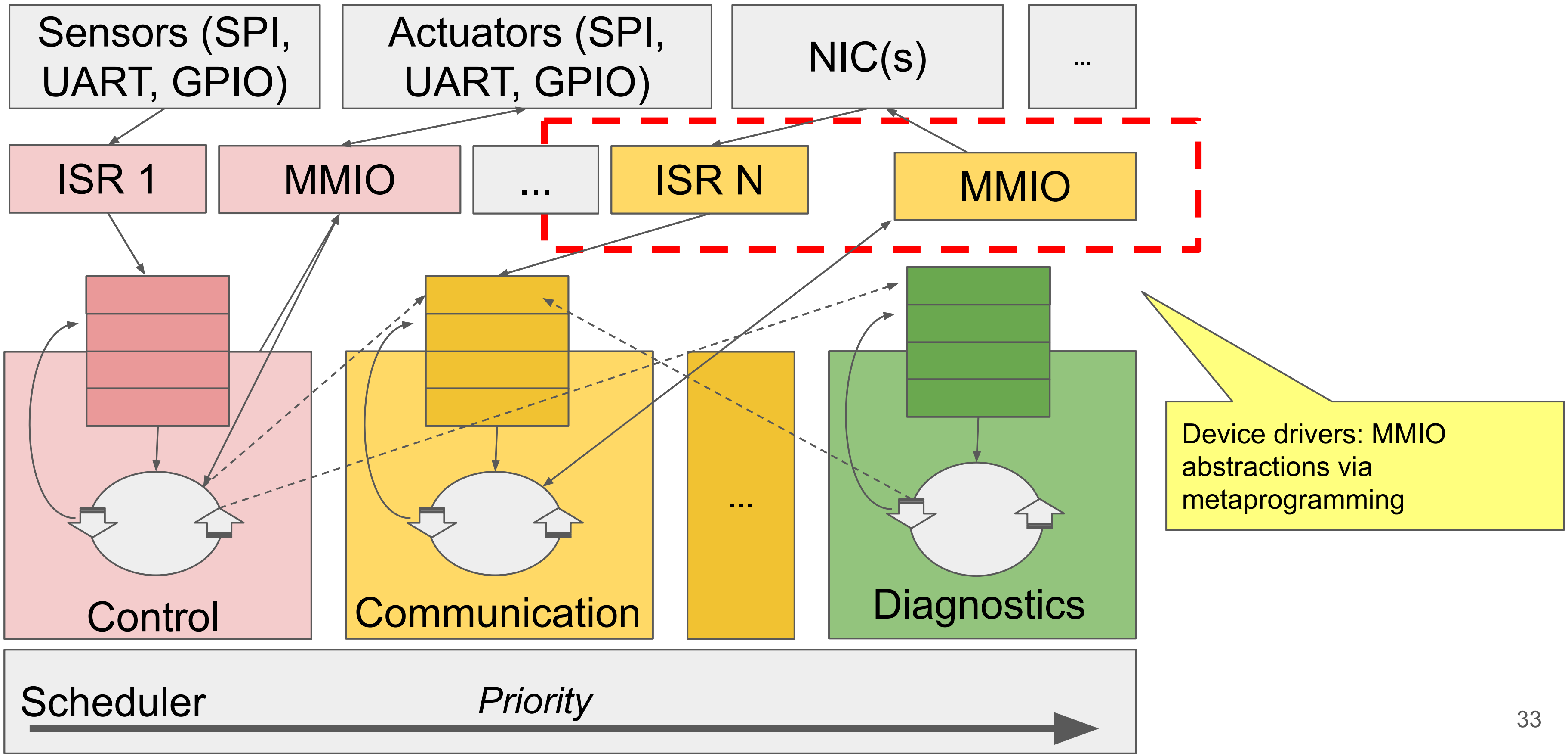
Industrial standards	JSF C++	MISRA C++	AUTOSAR C++ 14
Heap allocation	-+	-	-+
Exceptions	-	+	+

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)

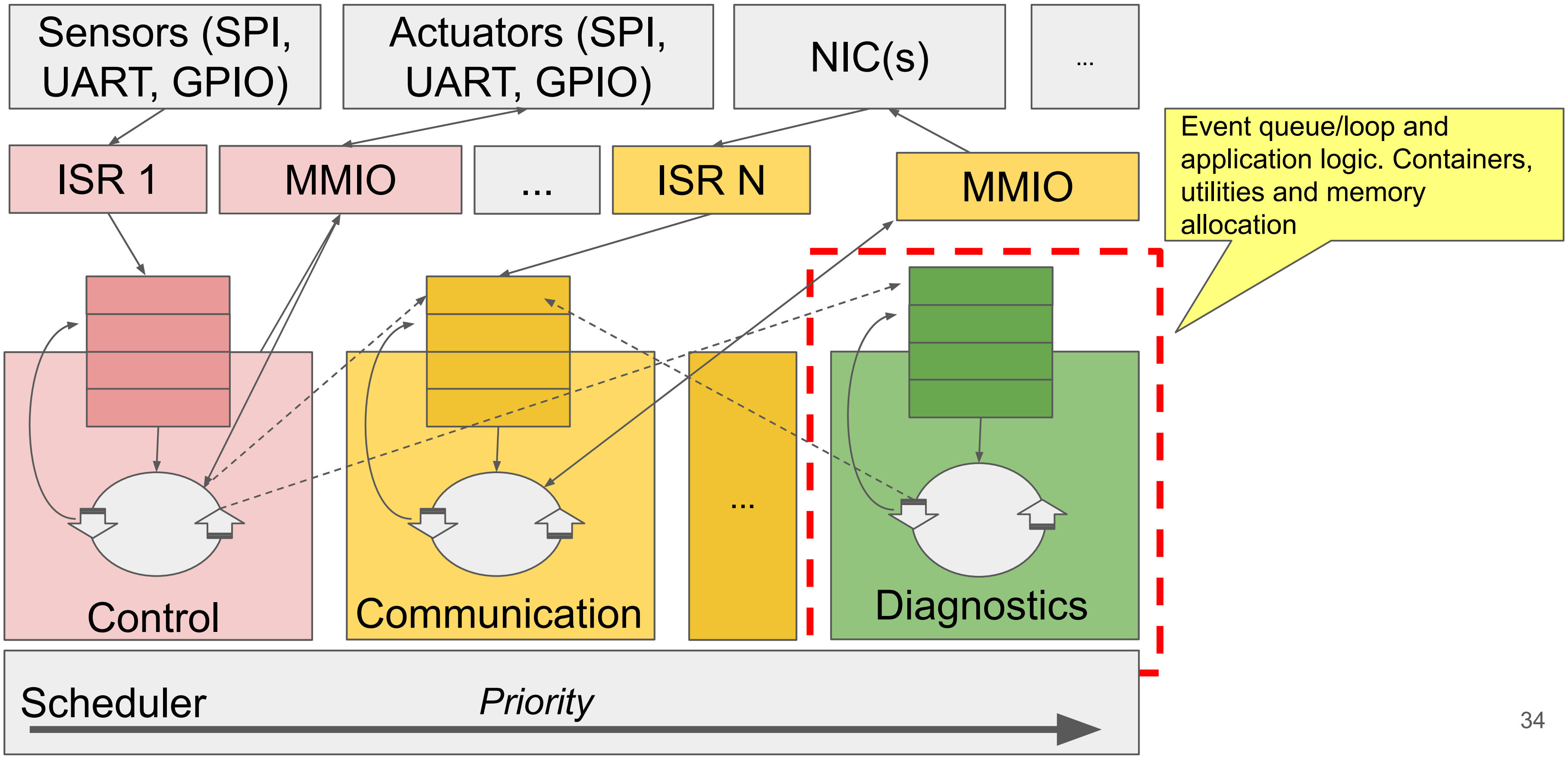
C++ application. Examples



C++ application. Examples

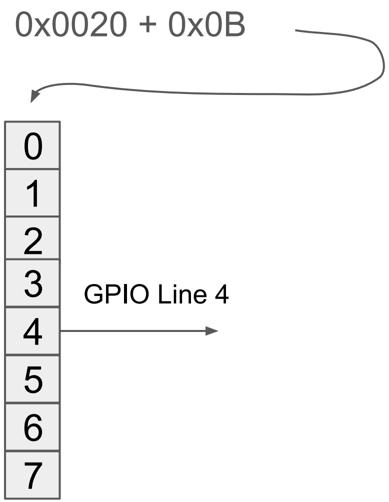


C++ application. Examples



C++ for device driver development

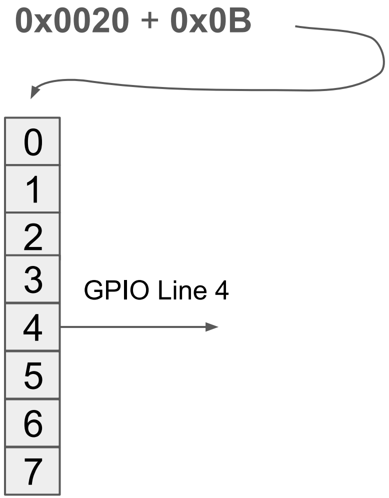
MMIO register




```
#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

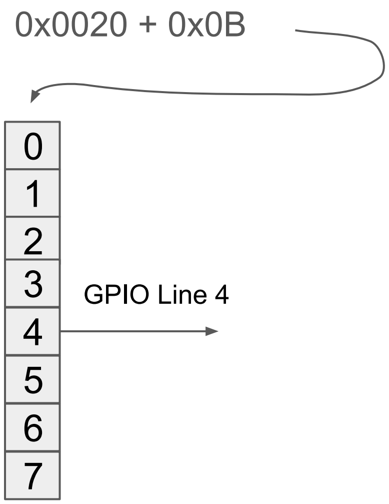



```
#define __SFR_OFFSET 0x20

#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

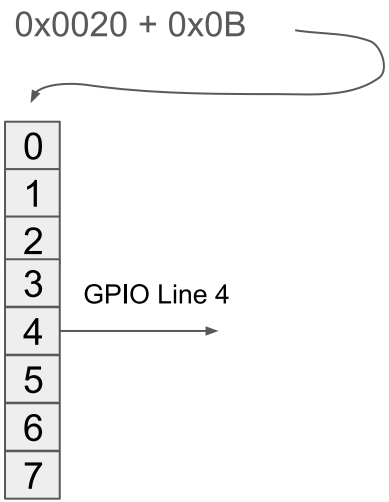



```
#define __SFR_OFFSET 0x20
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define PORTD _SFR_IO8(0x0B)


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


PORTD |= _BV(PD4);
```

Mask calculation

C++ for device driver development

MMIO register

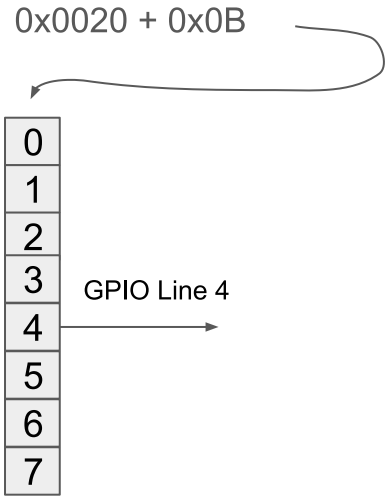



```
#define __SFR_OFFSET 0x20

#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



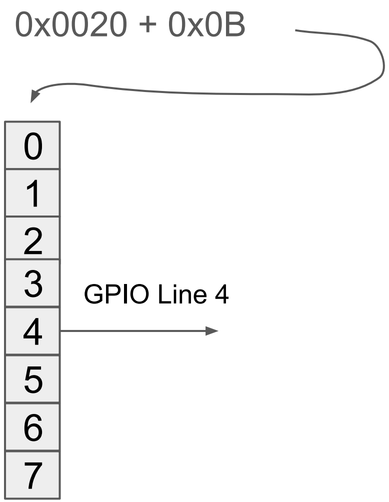
C-based implementation issues:

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

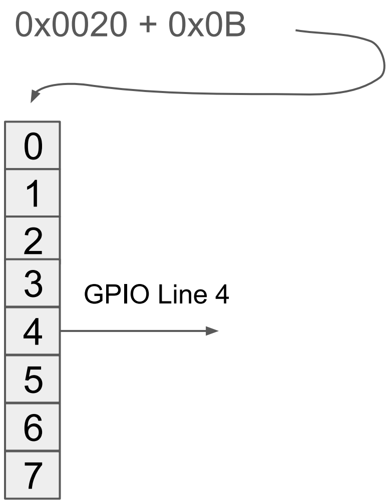
MMIO register




```cpp
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

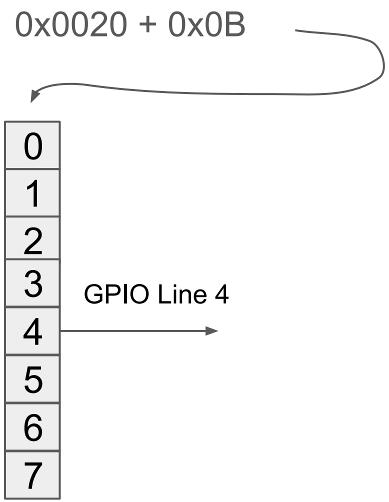



```cpp
template <typename T, uintptr_t Address,
                    uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                    (Base + Address);
    }
};
```

Referenced object type, base and offset

C++ for device driver development

MMIO register

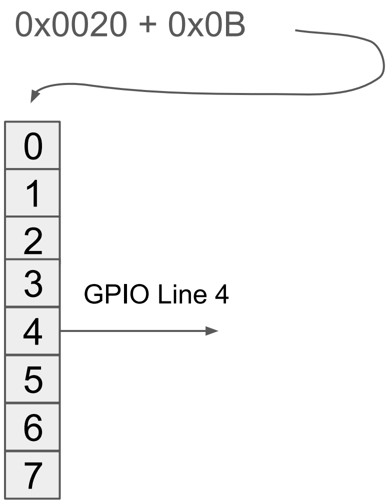



```cpp
template <typename T, uintptr_t Address,
                  uintptr_t Base>
struct address_with_base {
    using type = T;
    static T volatile *get_address() noexcept
    {
        return reinterpret_cast<T volatile*>
                        (Base + Address);
    }
};
```

Address calculation

C++ for device driver development

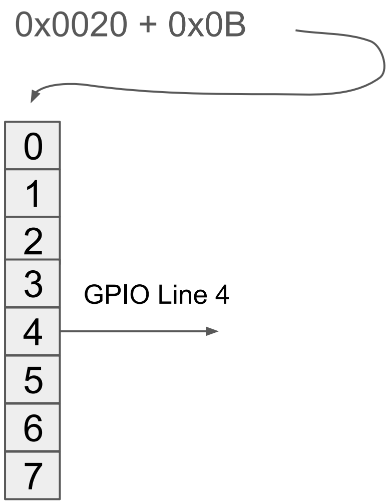
MMIO register




```
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

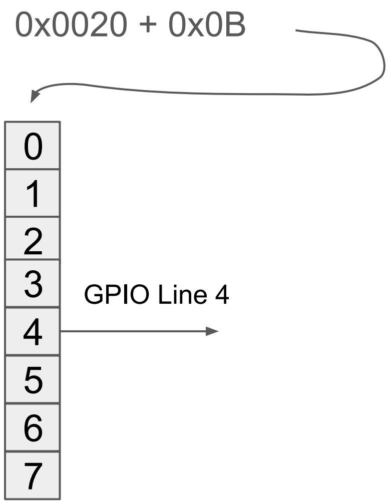



```cpp
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

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


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

Alias for concrete architecture

C++ for device driver development

MMIO register

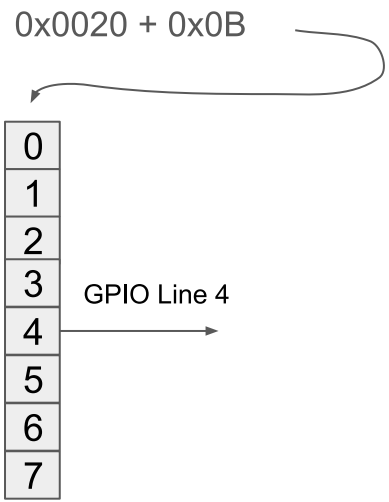



```
template<typename T, uintptr_t Address>
using special_func_reg
    = mmio::address_with_base<T, Address, __SFR_OFFSET>;

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));
    }
}
```

Load/Store optimization

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)

{

    using test_register = mmio::write_object<mock_address<uint8_t, &object>, 0, 4>;

    test_register::bits<0, 1, 3> register_bits;

    register_bits.set();

    EXPECT_EQ(object, 0b1011);

}
```

Mock, providing the address of a static object

C++ MMIO abstractions

1. The layout for the platform, based on developed abstractions, can be generated from SDK C-code or specification files
2. Classes representing MMIO can be used to implement more complex pieces, e.g. drivers

Device drivers. Dependency injection

<table>
<tr><td>Callbacks (C-style function pointers or functional objects)</td><td rowspan="2">1. Extra code and indirection might be unacceptable for performance critical areas
2. Dependency on compiler ability to optimize a call (devirtualization, LTO)</td></tr>
<tr><td>Run-time polymorphism</td></tr>
<tr><td>Compile-time polymorphism</td><td>1. No virtual call (no indirection)
2. No need to introduce hierarchies where they are not needed</td></tr>
</table>

Device driver. Testing

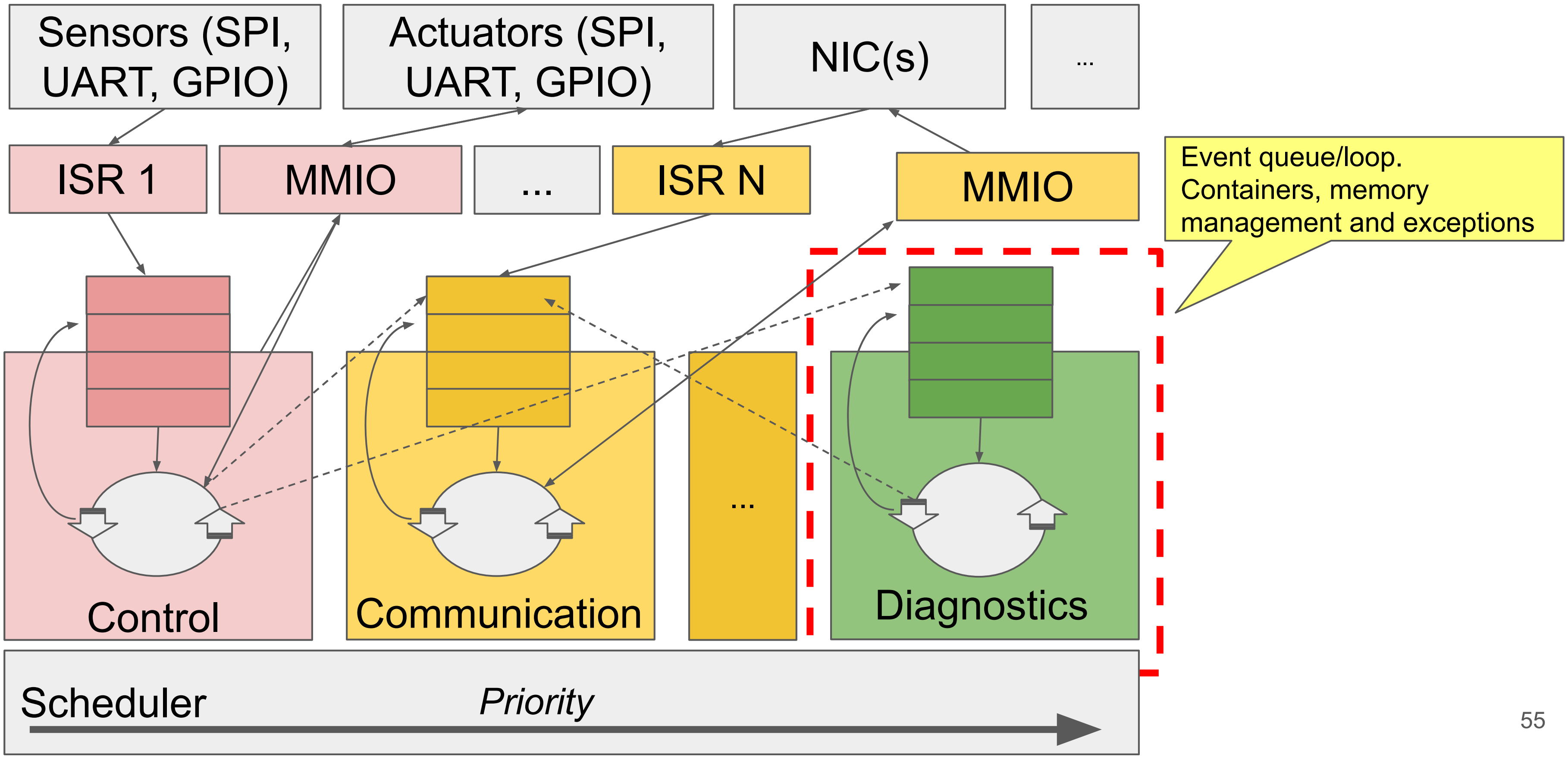
```
template <typename TxInterruptControllBit, typename RxTxRegisters>
struct uart_control {...};

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 can be developed 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



Reusable event queue/loop

```
using task_event_loop = event_loop<...>;
task_event_loop loop;

void task_worker()
{
    loop.run();
}

void some_function(task_event_loop &loop)
{
    loop.schedule(some_timeout, [](){ gpio_led.set(); });
}
```

Reusable event queue/loop

```
using task_event_loop = event_loop<...>;
task_event_loop loop;

void task_worker()
{
    loop.run();
}

void some_function(task_event_loop &loop)
{
    loop.schedule(some_timeout, [](){ gpio_led.set(); });
}
```

Configurable generic event loop class

Reusable event queue/loop

```
using task_event_loop = event_loop<...>;
task_event_loop loop;

void task_worker()
{
    loop.run();
}

void some_function(task_event_loop &loop)
{
    loop.schedule(some_timeout, [](){ gpio_led.set(); });
}
```

Loop is executed in the task main function

Reusable event queue/loop

```
using task_event_loop = event_loop<...>;
task_event_loop loop;

void task_worker()
{
    loop.run();
}

void some_function(task_event_loop &loop)
{
    loop.schedule(some_timeout, [](){ gpio_led.set(); });
}
```

Clients can schedule arbitrary work

Memory allocation

Option without heap-allocation:

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

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*	Configurable*	Hosted*
EASTL*	Not used*	Configurable	None
ETL	Not used	Configurable	None

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Ex	ency on std
STL, Utility lib	**By standard***	By	Hosted
Boost*	Not used*	Co	
EASTL*	Not used*	Configurable	None
ETL	Not used	Configurable	None

For STL containers custom allocators can be provided.

For some components in Utility lib that is not possible (`std::function`)

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	td
STL, Utility lib	By standard*	**By standard***	
Boost*	Not used*	Configurable*	
EASTL*	Not used*	Configurable	None
ETL	Not used	Configurable	None

No standard customization for handling exceptional situations. Typically `std::abort` is called instead of throwing

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	[illegible]	[illegible] of Hosted
Boost*	**Not used***	[illegible]	[illegible]ted*
EASTL*	Not used*	[illegible]	[illegible]e
ETL	Not used	Configurable	None

Components of `container` and `intrusive` libs can be used without heap.

Components of `pool` can be used to facilitate custom allocation

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	
Boost*	Not used*	**Configurable***	
EASTL*	Not used*	Configurable	
ETL	Not used	Configurable	None

Components of `container` framework can invoke user-defined handlers instead of exception throwing.
Facilities from `intrusive` and `pool` can be used without exceptions

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*		**Hosted***
EASTL*	Not used*		None
ETL	Not used	Configurable	None

Explicit dependencies on standard headers in code. E.g. `<cstddef>` (freestanding) in `container` or `<functional>` (hosted) in `pool`

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*	Configurable*	Hosted*
EASTL*	**Not used***	C[illegible]	[illegible]ne
ETL	Not used	Configurable	None

Components with prefix “`fixed_`” in names can be used without heap allocation

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*	Configurable*	Hosted*
EASTL*	Not used*	Configurable	None
ETL	**Not used**		one

Library is designed to be independent from heap allocation

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*	Configurable*	Hosted*
EASTL*	N		
ETL	Not used		

The frameworks are based on different design decisions and have distinct features but the core functionality is similar. Both contain alternative implementations of standard utilities and algorithms

1. STL - Standard Templa
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not used*	Configurable*	Hosted*
EASTL*	Not used*	**Configurable**	[illegible]
ETL	Not used	**Configurable**	[illegible]

Exception throwing, custom assertion handlers, disabled checks (reliance on external checks)

1. STL - Standard Template Library
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Containers, algorithms and utilities

Framework	Heap allocation	Exceptions handling	Dependency on std
STL, Utility lib	By standard*	By standard*	Part of Hosted
Boost*	Not		Hosted*
EASTL*	Not		**None**
ETL	Not		**None**

No explicit dependencies on standard headers in the code.
Alternative implementations of standard utilities are used instead. Can be adopted for the target platform by providing required definitions

1. STL - Standard Te
2. Boost (https://github.com/boostorg/boost)
3. EASTL - Electronic Arts Standard Template Library (https://github.com/electronicarts/EASTL)
4. ETL - Embedded Template Library (https://github.com/ETLCPP/etl)

Event type

```
class Event {
public:
     Timestamp getTimestamp() const;
//...
private:
    eastl::fixed_function<config::max_capture_list_buffer_size, void()> _function;
//...
#ifdef EVENT_DEBUG_ENABLED
    eastl::fixed_string<char, config::max_event_info_string_len, false> _event_info;
#endif
};
```

Types with fixed size storage

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

Alternatives for standard 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(...);
```

Redundant external checks to avoid exceptional situation. No standard customizable alternative to exception throwing. Typically abort is called if exceptions are disabled

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

Fixed-capacity vector compatible with stl.

Size specified at compile time.

Alternatives are:

`eastl::fixed_vector`, `etl::vector`

Container compatibility

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

Throwing on overflow can be disabled independently from global exception settings using extra template parameter. Assertion handler (possibly custom defined) will be invoked instead

Container compatibility

```cpp
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{...});
} else { /*...*/ }
auto const& top_item = q.top();
```

Heap data structure using `static_vector` as backend storage.

Alternatives are `boost::priority_queue`, `eastl::priority_queue` and `etl::priority_queue`

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{...});
} else { /*...*/ }
auto const& top_item = q.top();
```

Redundant checks if exceptions are not used.

Alternatively user defined callbacks `throw_bad_alloc`, `throw_out_of_range`, … can be provided

Fixed-capacity containers

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

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

Fixed-capacity containers

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

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

Available in EASTL along with `fixed_map`, `fixed_hash_map`, etc. Alternative is `etl::set`

Fixed-capacity containers

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

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

Maximum capacity

Fixed-capacity containers

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

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

Heap allocation as fallback is disabled

Fixed-capacity containers

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

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

For EASTL and ETL three options are available: exceptions, custom assert handlers or disabled internal checks

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{...});
```

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_time

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{...});
```

Service data block for intrusive data structure

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(

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{...});
```

Intrusive set. No allocation is performed on insertion

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{...});
```

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{...});
```

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

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
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{...});
```

Memory block allocation and object construction. Insertion into `set` will not throw if predicate is not throwing

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);
```

Factory method returns `variant`

No heap allocation guaranteed.

Alternatives: `boost::variant`, `eastl::variant`, `etl::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`, `eastl::optional`, `etl::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();
```

Some imitation of proposed `expected` type.

Guaranteed not to allocate.

`boost::outcome` is an advanced alternative

Containers, algorithms and utilities. Summary

- Although with some limitations standard library can be used in the embedded environment
- Multiple 3rd-party frameworks offer large choice of standard-like and non-standard containers and utilities addressing heap allocation and exception handling limitations
- In case of unavailability of standard library several 3rd-party frameworks can be used as a substitution

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?