vollmann engineering gmbh
>

Standard C++ as RTOS API

Interesting Developments

emBO++

March 2019

Detlef Vollmann

vollmann engineering gmbh

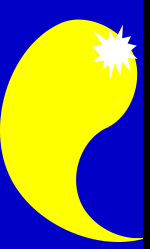
vollmann engineering gmbh
>

Standard C++ as RTOS API

Interesting Developments

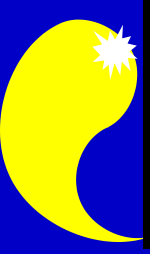
Detlef Vollmann
vollmann engineering gmbh
Luzern, Switzerland

dv@vollmann.ch
http://www.vollmann.ch/



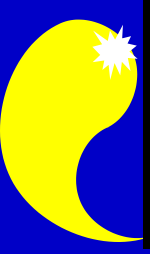
Overview

What?
Why?
How?
Examples
More



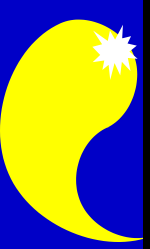
What?

- C++ outlook
  - Abstractions that (probably) will come
- Interface
  - not implementation
  - C++ compiler is not an OS
- Ideas
  - no details
    - lack of time
- References
  - some information is not well known



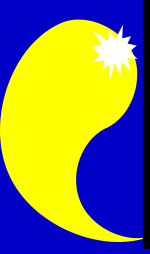
Overview

What?
Why?
How?
Examples
More



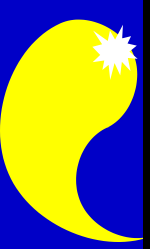
Why?

- Portability
- Testing / Simulation
- Maintenance



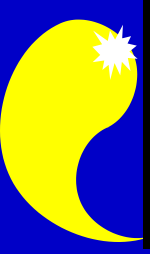
Portability

- Single API to program against
- Porting still not a simple task
  - port underlying implementation
  - testing
  - known limitations / bugs / workarounds
  - certification
- But much more feasible
  - same abstractions
  - existing test cases



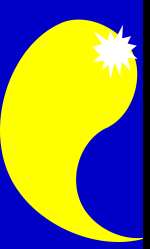
Testing

- Running target software on the host
  - development
  - build / test server
- Testing
  - unit tests
  - SW integration tests
  - complex test scenarios
  - test helpers
    - sanitizers



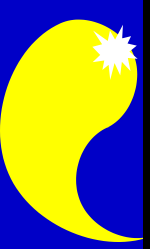
Simulation

- Common interface to simulate
- Different levels of reality
  - timing
  - errors
    - dropped packets
    - flipped flash bits
- Re-usable



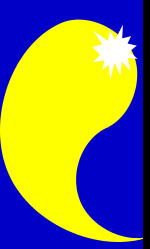
Maintenance

- Well known API
  - new developers on the team
- Re-using components from other projects



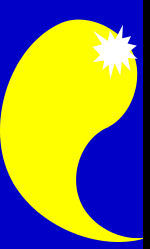
Overview

What?
Why?
How?
Examples
More



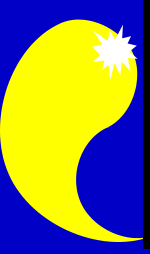
Availability on Host

- Proposed C++ APIs generally available
- Mainly library
- Mostly platform independent
- Some integration may be needed
  - more work for language features
  - coroutines
  - concepts
- Host implementation generally not certification relevant
  - take whatever you can find



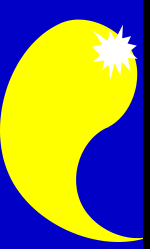
Target Implementation

- You probably have to do it yourselves
  - based on your underlying RTOS
  - or even lower level
- Generally zero overhead possible
- Future open source or commercial implementations probable



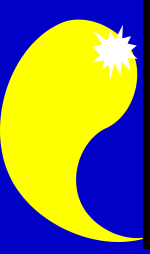
Overview

- What?
- Why?
- How?
- Examples
  - Main Loop
  - FreeRTOS
  - ARM Mbed
- More



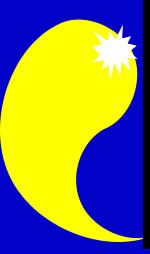
Examples

- Main loop
  - no OS
  - still benefits from standard interface
- FreeRTOS
  - typical mechanisms for small (minimal) RTOS
- ARM Mbed
  - pretty rich environment



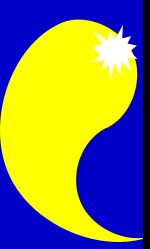
Overview

- What?
- Why?
- How?
- Examples
  - Main Loop
  - FreeRTOS
  - ARM Mbed
- More



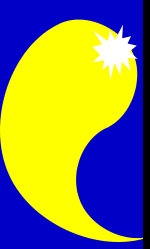
I/O

- Stream interface
- Interrupts
  - `[[gnu::interrupt]]`
- Queue interface
  - single ended lock-free
- ISR simulated as threads
  - memory model fits deliberately



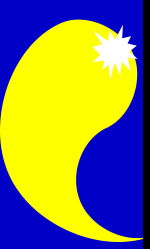
Timer

- Not event based
  - main loop
- `<chrono>`
  - provide your own HW based `clock`



FreeRTOS

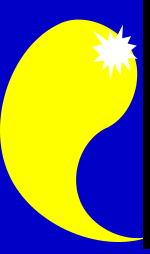
- Typical RTOS mechanisms
- Scheduler / Task
- Queue
- Semaphore
- Mutex



Scheduler / Tasks

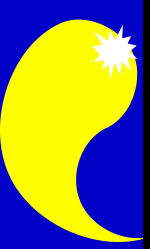
- Core element of (RT)OS
- *It allows applications to be organized as a collection of independent threads of execution.*
- FreeRTOS creation function

```
BaseType_t
xTaskCreate(TaskFunction_t taskCode,
            char const *name,
            unsigned short stackDepth,
            void *parameters,
            UBaseType_t priority,
            TaskHandle_t *createdTask);
```

Executors

- `std::thread` provides specific implementation
  - not suitable as generic interface
- Executors intended to provide generic interface for "threads of execution"

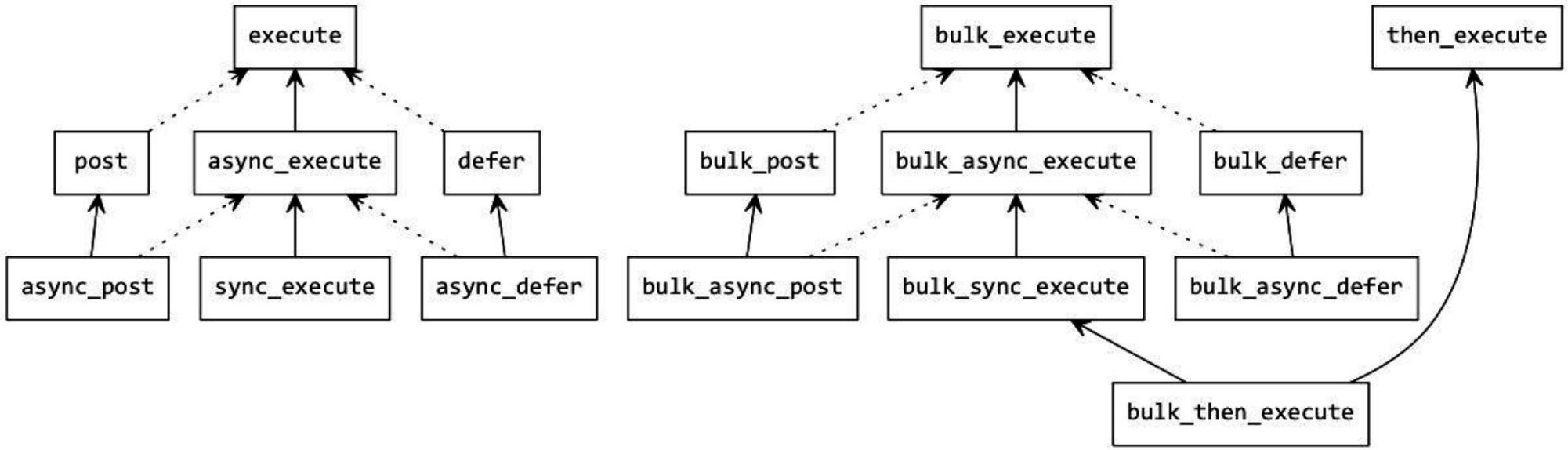


Executors

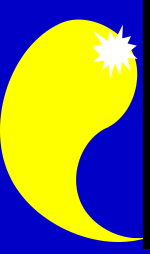
- Execution contexts
  - OS, thread pool, ...
- Executors as light-weight handle / wrapper to execution contexts
- Executors create tasks
  - all tasks in one execution context handled by same scheduler
- Executors can hold additional information
  - stack size, priority, ...

New Executor Interface

- Execution functions from P0443 R1 (Feb 2017)



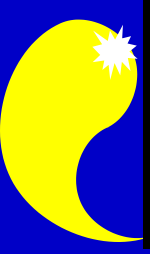
- Too many for SG1



New Executor Interface

- CopyConstructible (noexcept)
- EqualityComparable (noexcept)
- One of the execution functions
  - from P0443 R5 (Mar 2018)

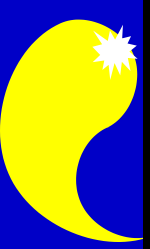
```
void execute(F) const;
Future twoway_execute(F) const;
Future then_execute(F, Future) const;
void bulk_execute(F, size_t, PF) const;
Future bulk_twoway_execute(F, size_t, RF, PF) con
Future bulk_then_execute(F, size_t, Future,
                         RF, PF) const;
```

bulk_twoway_execute

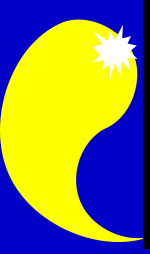
```
template<class Function, class Factory1, class Factory2>
executor_future_t<Executor,std::invoke_result_t<Factory1>>
bulk_twoway_execute(Function f,
                    executor_shape_t<Executor> shape,
                    Factory1 result_factory,
                    Factory2 shared_parameter_factory) const;
```

- Return type dependent on executor and `result_factory`
- `shape`: currently just a `size_t` and number of agents
- `result_factory`: creates a result object `r` that's passed by reference to the function
- `shared_parameter_factory`: may create a (e.g. synchronization) object that's passed to the function
- Creates `shape` execution agents that all call `f`
- `f` is called as `f(i, r, s)`
- Data to be worked on can be passed via lmbda capture or via `shared_parameter_factory`



Executor Properties

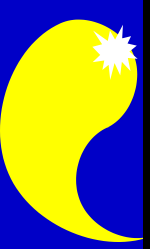
- Executors have properties
  - depending on the available syntactic interface
  - depending on the semantics of the interface
  - to provide extra information
- Direction
  - `oneway`, `twoway`, `then`
- Cardinality
  - `single`, `bulk`
- Blocking
  - `never_blocking`, `possibly_blocking`, `always_blocking`
- Continuation, more work, progress, new thread, allocator
- User defined properties are possible



require/prefer

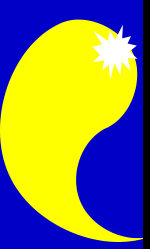
- require() and prefer() to get specific properties

```
auto newExec1 = require(oldExec,
                        oneway,
                        single,
                        never_blocking);

auto newExec2 = prefer(newExec1,
                       outstanding_work);
```

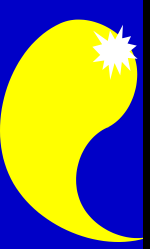
Executors Usage

- Target implements execution context and executor wrappers on top of RTOS
- Host implements whatever it wants
  - just executor wrappers on top of `std::thread_pool`
  - own execution context on top of pthread
    - full priority and stack size simulation
    - real-time timings
- Application takes global prototype executor
  - defined by separate host / target part



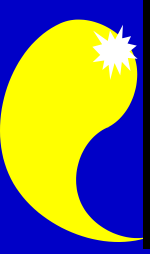
Queues

- Important communication mechanism
- Between different tasks
- Between ISRs and tasks
  - needs to be lock-free on ISR side



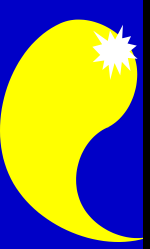
Queues

- Fixed sized
- Locking and lock-free queues
- `push`, `pop` and `close` interface
- Single ended wrappers
- Stream iterator interface



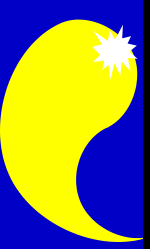
Queues

```
enum class queue_op_status {success, empty, full,
                            closed, busy };

template <typename Value>
class buffer_queue {
public:
  explicit buffer_queue(size_t max_elems);
  template <typename Iter>
  buffer_queue(size_t max_elems, Iter first, Iter last);
  ~buffer_queue();

  void close();

  Value value_pop();

  void push(const Value& x);
  void push(Value&& x);
};
```

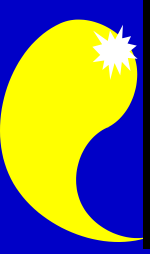
More Queue Functions

```
template <typename Value>
class buffer_queue
{
public:
  bool is_closed();
  bool is_empty();
  bool is_full();
  bool is_lock_free();

  queue_op_status wait_pop(Value&);
  queue_op_status try_pop(Value&);
  queue_op_status nonblocking_pop(Value&);

  queue_op_status wait_push(const Value& x);
  queue_op_status try_push(const Value& x);
  queue_op_status nonblocking_push(const Value& x);
};
```

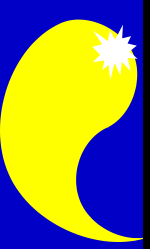
Semaphore

- Mutexes not intended as signalling mechanism between different tasks
- Semaphores fill the gap
  - incremented on one side
  - decremented on other side
- Can be modelled as valueless queue



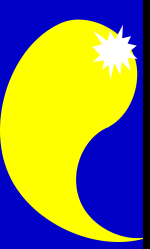
Semaphore

```
template<ptrdiff_t least_max_value>
class counting_semaphore {
public:
    explicit counting_semaphore(ptrdiff_t);
    ~counting_semaphore();

    void release(ptrdiff_t update = 1);
    void acquire() noexcept;
    bool try_acquire() noexcept;

private:
    ptrdiff_t counter; // exposition only
};
```

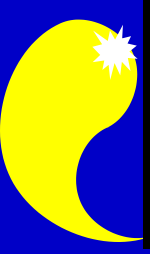
Mutex

- Special case of semaphore
- Mainly a concept in C++11
- May be templated on executor later



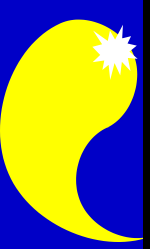
ARM Mbed

- RTX core: like FreeRTOS
- Stream / File interface: filesystemlibrary in C++17
- Events, timers, connectivity: Networking TS (ASIO)
  - ASIO provides framework for asynchronous event handling
  - plays together with coroutines



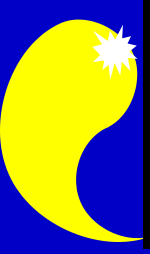
Coroutines

- Write state machines as normal functions
- Stackful coroutines
- Stackless coroutines



References

- These slides:
  `http://www.vollmann.ch/`
- Standard proposals, TSes:
  `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/`
- Executors design:
  `2018/p0761r2.pdf`
- Networking TS:
  `2018/n4771.pdf`
- Queues:
  `2019/p0260r3.html`
- Semaphores:
  `2019/p1135r3.html`



Questions

- ??????????????????????????????????

