Core Middleware

Davide Rizzi

Version 1.0, 2016-04-13

Middleware reference

Overview

The Core Middleware is a topic-based publish/subscribe middleware (see Wikipedia) that makes it possible to speed up the software development by leveraging on code reuse.

Applications are built connecting together logical blocks (nodes) by means of named communication channels (topics). Each topic has its own specific message type. Each node can transmit (publish) on and can receive (subscribe) from any channel (topic).

In this document a more in-depth description of the following objects will be given.

1. Messages
2. Publishers
3. Subscribers
4. Nodes

Messages

Messages represent the data type exchanged between publishers and subscribers.

From a Core user perspective messages are defined using a JSON file. The file must be valid according to an Apache Avro [1: Apache Avro LINK] schema.

Example: Led.json

```
{
  "name": "Led", ①
  "description": "Status LED", ②
  "namespace": "@", ③
  "fields": [ ④
    {
      "name": "led", ⑤
      "description": "Which led?", ⑥
      "type": "UINT32", ⑦
      "size": 1 ⑧
    },
    {
      "name": "value",
      "description": "Value of the led",
      "type": "UINT8",
      "size": 1
    }
  ]
}
```

① the `name` of the message field must match the name of the JSON file; `name` must be at most 15 characters long

② a brief description

③ by default ('@') the namespace will be the name of the package the configuration resides in; `namespace` can be used to override it

④ messges cannot have nested data structures

⑤ field `name` must be at most 15 characters long

⑥ a brief description

⑦ the supported field types are `CHAR`, `INT8`, `UINT8`, `INT16`, `UINT16`, `INT32`, `UINT32`, `INT64`, `UINT64`, `FLOAT32`, `FLOAT64`, `TIMESTAMP`

⑧ array type fields are specified using the `size` attribute, scalar types require to specify 1 as the `size` of the field

Messages are converted to source code using the command line tools. The code generator relies on the C preprocessor to keep the generated code clean and readable.

Example: Source code generated from Led.json

```
#pragma once

#include <Core/MW/CoreMessage.hpp>

namespace common_msgs {

CORE_MESSAGE_BEGIN(Led) // Status LED
    CORE_MESSAGE_FIELD(led, UINT32, 1) // Which led?
    CORE_MESSAGE_FIELD(value, UINT8, 1) // Value of the led
CORE_MESSAGE_END

}
```

Array type fields will be implemented using `Core::MW::Array<typename T>` class template (that is a stripped down std::array).

From the source code point of view, messages are classes that inherit from `Core::MW::Message` class.

Publishers and Subscribers

Publisher

A publisher is defined as an instance of `Core::MW::Publisher<typename MessageType>` class.

The class template must be specialized using the message type of the topic it will publish to.

How to publish data

```
common_msgs::Led* msg;

Core::MW::Publisher<common_msgs::Led> publisher;

if (publisher.alloc(msg)) { ①

    msgp->led   = 4;    ②
    msgp->value = 0;

    if (!publisher.publish(*msg)) { ③
        ...
    }
}
```

① ask the publisher to allocate a message

② valorize the message

③ publish it

The `Core::MW::Publisher::alloc()` method will return false where there are no subscribers to the topic.
The `Core::MW::Publisher::publish()` method will return false when TODO.

It is up to the node to tell the system that the publisher exists. This will be discussed in the Nodes section.

Subscriber

A subscriber is defined as an instance of `Core::MW::Subscriber<typename MessageType, unsigned QUEUE_LENGTH>` class.

The class template must be specialized using the message type of the topic it will subscribe to and the length of the message queue. The message queue length (`QUEUE_LENGTH`) *must be* at least 2.

There are 2 ways to consume the data received:

- registering a callback
- using `Core::MW::Subscriber::fetch()`

It is up to the node to tell the system that the subscriber exists. This will be discussed in the Nodes section.

Callback

It is possible to register a subscriber callback the Middleware will call on its spin() method (more on this later).

The callback function signature for a `Core::MW::Subscriber<typename MessageType, unsigned QUEUE_LENGTH>` subscriber is:

Callback data type

```
typedef bool (* Callback)(
    const MessageType& msg,
    Node*              node
);
```

The callback function can be registered using `Core::MW::Subscriber::set_callback()` method.

The node parameter is required to easily allow multiple instance of the subscriber (as the callback is a C function-pointer).

Callback function example

```
...

subscriber.set_callback(LedSubscriber::callback); ①

...

bool
LedSubscriber::callback(
        const common_msgs::Led& msg,
        Core::MW::Node*         node
)
{
    LedSubscriber* _this = static_cast<LedSubscriber*>(node); ②

    _this->setLedValue(msg.value);
}
```

① register the callback

② pointer to instance of the node

If the callback is a class member function, it *MUST* be made `static` (to exclude the parameter `this` from its signature).

fetch()

```
common_msgs::Led* msg;

if (subscriber.fetch(msg)) {
    setLedValue(msg.value);
}
```

The `Core::MW::Subscriber::fetch()` method will return false where there are no messages in the queue.

Nodes

A node is nothing but a process that runs on a module and that communicate with other nodes using a topic based publish/subscribe pattern (see Wikipedia).

From the source code point of view a node can be defined in two different ways:

- as subclass of `Core::MW::CoreNode`
- as an OS thread

The first approach permits to control the execution flow and is the preferred way to implement a node.
The latter approach, whilst apparently simpler, does not offer any control on the Node lifecycle.

Nodes defined as subclasses of `Core::MW::CoreNode` share other important features:

- they can be described using a JSON file, allowing automatic code generation of the main.cpp file by a tool
- they can be easily configured by means of a configuration system

CoreNode

ICoreNode & the node lifecycle

The `Core::MW::ICoreNode` interface describes the lifecycle of a node.

ICoreNode interface (header file)

```
class ICoreNode
{
public:
    enum class State {
        NONE, // Node has not been set up
        SET_UP, // wait
        INITIALIZING, // -> onInitialize
        INITIALIZED, // wait
        CONFIGURING, // -> onConfigure
        CONFIGURED, // wait
        PREPARING_HW, // -> onPrepareHW
        HW_READY, // wait
        PREPARING_MW, // -> onPrepareMW
        MW_READY, // wait
        IDLE, // wait
        STARTING, // -> onStart
        LOOPING, // -> onLoop
        STOPPING, // -> onStop
        FINALIZING, // -> onFinalize
```

```
        ERROR, // wait forever
        TEARING_DOWN // killing
    };

    enum class Action {
        INITIALIZE, CONFIGURE, PREPARE_HW, PREPARE_MW, START, STOP, FINALIZE
    };

    virtual bool
    setup() = 0;

    virtual bool
    teardown() = 0;

    virtual bool
    execute(
            Action what
    ) = 0;

    virtual State
    state() = 0;


    virtual ~ICoreNode() {}

protected:
    virtual bool
    onInitialize() = 0;

    virtual bool
    onConfigure() = 0;

    virtual bool
    onPrepareHW() = 0;

    virtual bool
    onPrepareMW() = 0;

    virtual bool
    onStart() = 0;

    virtual bool
    onLoop() = 0;

    virtual bool
    onStop() = 0;

    virtual bool
    onError() = 0;

    virtual bool
```

```
    onFinalize() = 0;


protected:
    ICoreNode() :
        _currentState(State::NONE)
    {}

    State _currentState;
};
```

When are firstly created, `Core::MW::ICoreNode` must be in a `ICoreNode::State::NONE` state. Calling `ICoreNode::setup()` will set up the node execution. State will change to `ICoreNode::State::SET_UP` once the node is ready.

Then is it possible to cycle all the states using `ICoreNode::execute()`.
The order of the actions to execute must be exactly:

1. `ICoreNode::Action::INITIALIZE`
2. `ICoreNode::Action::CONFIGURE`
3. `ICoreNode::Action::PREPARE_HW`
4. `ICoreNode::Action::PREPARE_MW`
5. `ICoreNode::Action::START`
6. `ICoreNode::Action::STOP`
7. `ICoreNode::Action::FINALIZE`

`ICoreNode::execute()` must return `false` if an out of order action is requested.

To every action corresponds an event: `ICoreNode::onInitialize()`, `ICoreNode::onConfigure()`, `ICoreNode::onPrepareHW()`, `ICoreNode::onPrepareMW()`, `ICoreNode::onStart()`, `ICoreNode::onLoop()`, `ICoreNode::onStop()`, `ICoreNode::onFinalize()`.

The meaning of the events can be summarized as follows:

- `ICoreNode::onInitialize()`: memory allocation, default memeber value assignement, ...
- `ICoreNode::onConfigure()`: configuration, ...
- `ICoreNode::onPrepareHW()`: hardware setup, device probing, ...
- `ICoreNode::onPrepareMW()`: publishers advertising and subscribers subscribing, callback settings, ...
- `ICoreNode::onStart()`: hardware starting, ...
- `ICoreNode::onLoop()`: the main loop
- `ICoreNode::onStop()`: hardware stopping, ...
- `ICoreNode::onFinalize()`: memory deallocation, ...

Whenever an unrecoverable error occurs (i.e.: when an event member function returns `false`), the state of the node must become `ICoreNode::State::ERROR` and `ICoreNode::onError()` event must be

called. The only way to recover must then be to tear down the node, calling `ICoreNode::teardown()`. `ICoreNode::teardown()` must also be called when a CoreNode object is destroyed.

See ICoreNode state machine for a description of the state transitions.

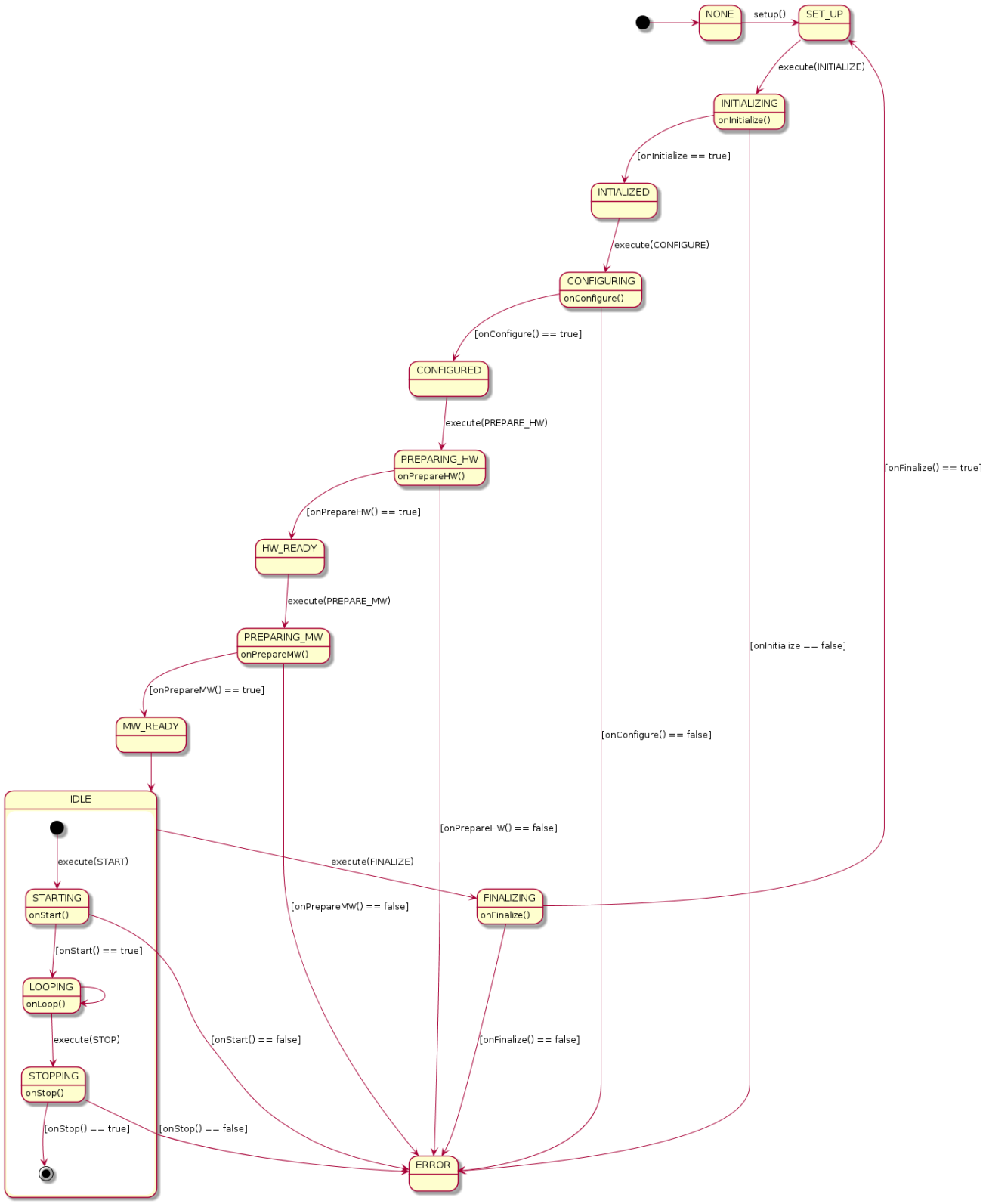

Figure 1. ICoreNode state machine

For clarity in ICoreNode state machine there is no reference to `Core::MW::ICoreNode::teardown()`, but it must always be possible to call it in every moment.+ The node must switch to a `ICoreNode::State::TEARING_DOWN` state and then, to `ICoreNode::State::NONE`.

CoreNode

The `Core::MW::CoreNode` class offers the developer the possibility to cleanly implement all the possible aspects related to a node by implementing the `Core::MW::ICoreNode` interface.

Calling `CoreNode::setup()` will create the node execution thread. State will change to `CoreNode::State::SET_UP` once the thread has been created.

The event member functions by default return `true`, so it is not needed to implement these in `Core::MW::CoreNode` derived classes.

Implementing a new node requires to extend `Core::MW::CoreNode`, and to override the default event member functions as required.
`Core::MW::CoreNode` constructor takes 2 arguments, the node name and the node thread priority.

A `Core::MW::CoreNode` object can be added to a`Core::MW::CoreNodeManager`(such as a`Core::MW::CoreModule`).

In this way all the nodes lifecycles are syncronized. It is also easier and more concise than calling all the execute() while checking for the return value and states.

CoreNode derived node (header file)

```
#pragma once

#include <Core/MW/CoreNode.hpp>
#include <Core/MW/Publisher.hpp>

#include <common_msgs/Led.hpp>
#include <led/PublisherConfiguration.hpp>

#include <array>

namespace led {
    class Publisher:
        public Core::MW::CoreNode
    {
public: // CONSTRUCTOR and DESTRUCTOR
        Publisher(
                const char*                     name,
                Core::MW::Thread::PriorityEnum priority = Core::MW::Thread
::PriorityEnum::NORMAL
        );
        virtual
        ~Publisher();

public: // CONFIGURATION
        PublisherConfiguration configuration;

private: // PUBLISHERS and SUBSCRIBERS
        Core::MW::Publisher<common_msgs::Led> _publisher;

private: // PRIVATE MEMBERS
        uint32_t _toggle;

private: // EVENTS
        bool
        onPrepareMW();

        bool
        onLoop();
    };
}
```

OS thread

In this approach a node is defined directly as a thread.

OS thread: node

```
void
test_subscriber_node(
        void* arg
)
{
    Core::MW::Node node("test_sub");
    Core::MW::Subscriber<common_msgs::String64, 5> sub;
    common_msgs::String64* msgp;

    (void)arg;
    chRegSetThreadName("test_sub");

    node.subscribe(sub, "test");

    for (;;) {
        node.spin(Core::MW::Time::ms(1000));

        if (sub.fetch(msgp)) {
            module.stream.printf("%s\r\n", msgp->data);
            sub.release(*msgp);
        }
    }
}
```

And later on, create a thread with it.

OS thread: thread

```
...
Core::MW::Thread::create_heap(NULL, THD_WORKING_AREA_SIZE(1024), NORMALPRIO,
test_subscriber_node, nullptr);
...
```

Configurations

From a Core user perspective node configurations are defined using a JSON file. The file must be valid according to an Apache Avro [1: Apache Avro LINK] schema.

Example: LedPublisherConfiguration.json

```
{
  "name": "LedPublisherConfiguration", ①
  "description": "LED Publisher node configuration", ②
  "namespace": "@", ③
  "fields": [ ④
    {
      "name": "topic", ⑤
      "description": "Name of the topic to publish to", ⑥
      "type": "CHAR", ⑦
      "size": 16 ⑧
    },
    {
      "name": "led",
      "description": "Which led",
      "type": "UINT32",
      "size": 1
    }
  ]
}
```

① the `name` of the message field must match the name of the JSON file; `name` must be at most 15 characters long

② a brief description

③ by default ('@') the namespace will be the name of the package the configuration resides in; `namespace` can be used to override it

④ configurations cannot have nested data structures

⑤ field `name` must be at most 15 characters long

⑥ a brief description

⑦ the supported field types are `CHAR`, `INT8`, `UINT8`, `INT16`, `UINT16`, `INT32`, `UINT32`, `INT64`, `UINT64`, `FLOAT32`, `FLOAT64`, `TIMESTAMP`

⑧ array type fields are specified using the `size` attribute, scalar types require to specify 1 as the `size` of the field

Configurations are converted to source code using the command line tools. The code generator relies on the C preprocessor to keep the generated code clean and readable.

Example: Source code generated from LedPublisherConfiguration.json

```
#pragma once

#include <Core/MW/CoreConfiguration.hpp>

namespace led {

CORE_CONFIGURATION_BEGIN(LedPublisherConfiguration) //LED Publisher node configuration
    CORE_CONFIGURATION_FIELD(topic, CHAR, 16) // Name of the topic to publish to
    CORE_CONFIGURATION_FIELD(led, UINT32, 1) // Which led
CORE_CONFIGURATION_MAP_BEGIN(2)
    CORE_CONFIGURATION_MAP_ENTRY(LedPublisherConfiguration, topic)
    CORE_CONFIGURATION_MAP_ENTRY(LedPublisherConfiguration, led)
CORE_CONFIGURATION_MAP_END()
CORE_CONFIGURATION_END()

}
```

Array type fields will be implemented using `Core::MW::Array` class template (that is a stripped down std::array).

`Core::MW::CoreConfiguration` class implements a kind-of reflection system, which allows to access derived class members by their name:

Using `Core::MW::CoreConfiguration`

```
led::LedPublisherConfiguration configuration;

configuration.led = 3;

// IS THE SAME AS

configuration["led"] = 3;
```

This is useful for a runtime configuration of the node.

As this system relies on type deduction for template argument and return value types, special care must be used in assignments using the `[]` operator. This is specially true in assignments of numeric literals. It is better to always force the type by using an explicit cast.

define user-defined literals such as _s16, _u8