Practical Guide to Implementing Communication Protocols in C++ (for Embedded Systems)

Table of Contents

Guide to Implementing Communication Protocols in C++ (for Embedded Systems)

Almost every electronic device/component nowadays has to be able to communicate to other devices, components, or outside world over some I/O link. Such communication is implemented using various communication protocols.

At first glance the implementation of communication protocols seems to be quite an easy and straightforward process. Every message has predefined fields, that need to be serialised and deserialised according to the protocol specification. Every serialised message is wrapped in a transport data to ensure a safe delivery to the other end over some I/O link. However, there are multiple pitfalls and wrong design choices that can lead to a cumbersome, bloated, and difficult to maintain source code. It becomes especially noticable when the development of the product progresses, and initially developed small communication protocol grows to contain many more messages than initially planned. Adding a new message in such state can become a tedious, time consuming and error-prone process.

This book suggests flexible, generic and easily extendable design architecture, which allows creation of a generic C++(11) library. This library may be used later on to implement many binary communication protocols using simple declarative statements of class and type definitions.

As stated in the book's title, the main focus of this book is a development for embedded systems (including bare-metal ones). There is no use of RTTI and/or exceptions. I also make a significant effort to minimise usage of dynamic memory allocation and provide means to exclude it altogether if needed. All the presented techniques and design choices are also applicable to non-embedded systems which don't have limitations of the latter.

Code Generation vs C++ Library

The implementation of the binary communication protocols can be a tedious, time consuming and error-prone process. Therefore, there is a growing tendency among developers to use third party code generators for data (de)serialisation. Usually such tools receive description of the protocol data layout in separate source file(s) with a custom grammar, and generate appropriate (de)serialisation code and necessary abstractions to access the data.

There are so many of them: ProtoBuf, Cap'n Proto, MessagePack, Thrift, Kaitai Struct, Protlr, you-name-it... All of these tools are capable of generating **C++** code. However, the generated code quite often is not good enough to be used in embedded systems, especially bare-metal ones. Either the produced **C++** code or the tool itself has **at least** one of the following limitations:

- Inability to specify binary data layout. Many of the tools use their own serialisation format without an ability to provide custom one. It makes them impossible to use to implement already defined and used binary communication protocol.
- Inability to customise underlying types. Most (or all) of the mentioned code generating tools, which do allow customisation of binary data layout, choose to use **std::string** for string fields and/or **std::vector** for lists, as well as (de)serialisation code is generated to use standard streams (**std::istream** and **std::ostream**). Even if such ability is provided, it is usually "global" one and do not allow substitution of types only for specific messages / fields.
- Small number of supported data fields or limited number of their serialisation options. For example, strings can be serialised by being prefixed with their size (which in turn can have different lengths), or being terminated with '\0', or having fixed size with '\0' padding if the string is too short. There are protocols that use all three variants of strings.
- Poor or weak description grammar without an ability to support conditional (de)serialisation. For example, having a value (such as single bit in some bitmask field) which determines whether some other optional field exists or not.
- Lack of polymorphic interface to allow implementation of the common code

for all the defined messages.
- When polymorphic interface with virtual functions is provided, there is no way to exclude generation of unnecessary virtual functions for a particular embedded application. All the provided virtual functions will probably remain in the final image even if they are not used.
- Lack of efficient built-in way of dispatching the deserialised message object into its appropriate handling function. There is a need to provide a separate dispatch table or map from message ID to some callback function or object.
- Lack of ability to override or complement the generated serialisation code with the manually written one where extra logic is required.

The generalisation is hard. Especially when the main focus of the tools' developers is on supporting as many target programming languages as possible, rather than allowing multiple configuration variants of a single specific language. Currently there is no universal **fit all needs** code generation solution that can handle all the existing and being used binary communication protocols. As the result many embedded C++ developers still have to manually implement them rather than relying on the existing tools for code generation.

There is still a way to help them in such endeavour by developing a C++ library which will provide highly configurable classes, usage of which will allow to implement required functionality using simple declarative statements of types and classes definitions (instead of implementing everything from scratch). That's what this book is all about.

Thanks to new language features introduced in **C++11** standard and multiple meta-programming techniques, it becomes possible to write simple, clear, but highly configurable code, which can be used in multiple applications: embedded bare-metal with limited resources, Linux based platform, even the GUI analysis tools. They all can use the same single implementation of the protocol, but each generate the code suitable for the developed platform / application. The C++ compiler itself serves as code generation tool.

Main Challenges

There are multiple challenges that need to be considered prior to starting implementation of any communication protocol. It will guide us into the right direction when designing an overall architecture.

Code Boilerplating

The communication protocols are notorious for creating a boilerplate code. As a whole, most of them are very similar, they define various messages with their internal fields, define serialisation rules for all the fields and wrap them in some kind of transport information to ensure safe delivery of the message over the I/O link.

When serialising any message, all its fields must be serialised in predefined order. There is also very limited number of field types that is usually used:

- **numeric values** - may differ in sizes, being signed or unsigned, have a different ranges of valid values, etc...
- **enumeration values** - similar to numeric values, but have a very limited range of valid values, and `enum` type is usually used to operate the values, just for convenience.
- **bitmask values** - similar to numeric values, but each bit has a different meaning.
- **strings** - may differ in the way they are serialised (zero-suffixed or size-prefixed).
- **lists** of raw bytes or other fields - may have fixed (predefined) or variable size.
- **bundles of multiple fields** - may be used as a single element of a **list**.
- **bitfields** - similar to bundles, but internal member fields have a length of several bits (instead of bytes).

The number of field types is quite small, but the number of different nuances when serialising or using a single field is much bigger. It is very difficult to generalise such use and most developers don't even bother to come up with something generic. As the result they experience a *deja-vu* feeling every time they have to implement a new message or add a new field into an existing message. There is a strong feeling that the code is being duplicated, but there is no obvious and/or easy way to minimise it.

Runtime Efficiency

In most cases the messages are differentiated by some numeric ID value. When a new message is received over some I/O link, it needs to be identified and dispatched to appropriate handling function. Many developers implement this logic using simple `switch` statement. However, after about 7 - 10 `case`s such dispatch mechanism becomes quite inefficient, and its inefficiency grows with number of new messages being introduced. When not having a limitation of inability to use dynamic memory allocation and/or exception, some developers resort to standard collections (`std::map` for example) of pointer to functions or `std::function` objects. Bare-metal developers usually stick to the `switch` statement option incurring certain performance penalties when the implemented communication protocol grows.

Protocol Extension Effort

Also keep in mind the development effort that will be required to introduce a new message to the protocol being implemented. The number of different places in the existing code base, that need to be modified/updated, must obviously be kept at a minimum. Ideally no more than 2 or 3, but most implementations I've seen significantly bypass these numbers. In many cases developers forget to introduce compile time checks, such as `static_assert` statements to verify that all the required places in the code have been updated after new message was introduced. Failure to do so results in unexpected bugs and extended development effort to find and fix them.

What about extending an existing message by adding an extra field at the end or even in the middle? How easy is it going to be and how much development time needs to be spent? How error-prone is it going to be?

Inter-System Reuse

Quite often the implementation of the same protocol needs to be reused between different systems. For example, some embedded sensor device needs to communicate its data to a management server (both implemented in C++) and it would be wise to share the same implementation of the communication protocol on both ends. However, managing the I/O link and usage of various data structures may be different for both of them. Making the implementation of the communication protocol system dependent may make such reuse impossible.

Sometimes different teams are responsible for implementation of different systems, that use the same communication protocol but that reside on different ends of the communication link. Usually such teams make an upfront decision not to share the implementation of the communication protocol they use. Even in this case, making the implementation system dependent is a bad idea. It may be necessary to develop some additional protocol testing tools because the other team has not completed the development of their product in time.

Intra-System Reuse

It is not uncommon for various embedded systems to add extra I/O interfaces in the next generations of the device hardware, which can be used to communicate with other devices using the same protocol. For example, the first generation of some embedded sensor communicates its data over TCP/IP network link to some data management server. The second generation adds a Bluetooth interface that allows to communicate the same data to a tablet of the person working nearby. The application level messages, used to communicate the data, are the same for the server and the tablet. However, the transport wrapping information for TCP/IP and Bluetooth will obviously differ. If initial implementation of the communication protocol hasn't properly separated the application level messages and wrapping transport data, it's going to be difficult, time consuming and error-prone to introduce a new communication channel via Bluetooth I/O link.

Goal

Our primary goal is to come up with an architecture that:

- does NOT depend or make any assumptions on the system it is running on.
- does NOT make any hard-coded assumptions on the resources available to the system, such as dynamic memory allocation, exceptions, RTTI, etc...
- has an **efficient** way to parse the incoming message and dispatch it to an appropriate handler. The runtime complexity shouldn't exceed `O(log(n))` , where `n` is a total number of messages in the protocol.
- provides quick, easy and straightforward way of adding new messages to the protocol.
- has as little connection as possible between the application level messages and wrapping transport data, which allows easy substitution of the latter if need arises.

Our ultimate goal would be creation of a generic C++(11) library, that can assist in implementation of many binary communication protocols. Such library will provide all the necessary types and classes, usage of which will make the implementation of the required communication protocol easy, quick and straightforward process of using simple declarative statements. It will significantly reduce the amount of boilerplate code and boost the development process.

Audience

The primary intended audience of this book is **intermediate** to **professional** C++ developers who feel comfortable with templates and are not afraid of template meta-programming.

In order to achieve all our goals of platform and runtime environment independence, there is little other choice but to use templates with significant amount of meta-programming techniques in order to allow compiler to generate the best code suitable for the system being developed.

Code Examples

This book contains multiple C++ code examples and snippets. Their main purpose is to demonstrate ideas expressed in the book as well as guide developers into the right direction. There are no huge code listings (nobody reads them anyway) and no detailed explanations for every line of code. I expect the readers to understand the demonstrated idea and take it to the next level themselves.

In order to demonstrate the idea I rarely use **production** level code, at least not up front. I will start with something simple and non-generic and gradually increase the complexity and/or genericity.

I'm also a huge fan of Non-Virtual Interface (NVI) Idiom and often my examples will look like this:

```
class SomeInterface
{
public:
    void someFunction()
    {
        someFunctionImpl();
    }

protected:
    virtual void someFunctionImpl() = 0;
};

class Derived : public SomeInterface
{
protected:
    virtual void someFunctionImpl() override {...}
};
```

The non virtual interface function is supposed to check pre- and post-conditions of the polymorphic invocation if such exist as well as execute some common code if such is required. I tend to write the code similar to above even when there are no pre- and post-conditions to check and no common code to execute. Please don't be surprised when seeing such constructs throughout the book.

Final Outcome

The ideas summarised in this book are not just *theoretical* ones. There is ready to use implementation called COMMS Library. It provides all the necessary types and classes to make the definition of the custom messages as well as wrapping transport data fields to be simple declarative statements of type and class definitions, which specify **WHAT** needs to be implemented. The library internals handle the **HOW** part.

NOTE, that the ideas expressed in this book are very basic ones, the mentioned COMMS Library is much more advanced than the examples provided in this book. It is recommended to read the library's tutorial and, as an excersice, to think about ways how the provided examples can be extended to support the described features.

The COMMS Library is a part of a bigger project called CommsChampion. It contains generic plug-in based tools for visualisation and analysis of the communication protocols, which have been implemented using the provided library.

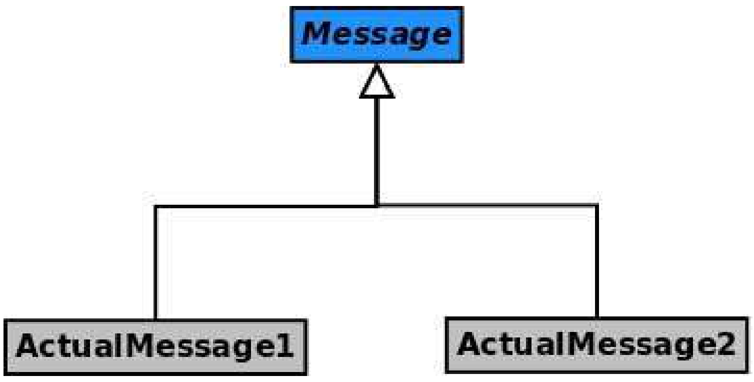
There is also a bundling project. It lists all the open protocols that have been implemented using the COMMS Library.

Contribution

If you have any suggestions, requests, bug fixes, spelling mistakes fixes, or maybe you feel that some things are not explained properly, please feel free to e-mail me to **arobenko@gmail.com**.

Message

Most C++ developers intuitively choose to express every independent message as a separate class, which inherit from a common interface.



This is a step to the **right** direction. It becomes easy and convenient to write a common code that suites all possible messages:

```
class Message
{
public:
    void write(...) const {
        writeImpl(...);
    }
    ...
protected:
    // Implements writing to a buffer functionality
    virtual void writeImpl(...) const = 0;
};

class ActualMessage1 : public Message
{
    ...
protected:
    virtual void writeImpl(...) const override {...};
};

class ActualMessage2 : public Message
{
    ...
protected:
    virtual void writeImpl(...) const override {...};
};

// Send any message
void sendMessage(const Message& msg)
{
    ...
    msg.write(...); // write message to a buffer
    ...// send buffer contents over I/O link;
}
```

Reading and Writing

When new raw data bytes are received over some I/O link, they need to be deserialised into the custom message object, then dispatched to an appropriate handling function. When continuing **message as an object** concept, expressed in previous chapter, it becomes convenient to make a reading/writing functionality a responsibility of the message object itself.

```
class Message
{
public:
    ErrorStatus read(...) {
        return readImpl(...);
    }

    ErrorStatus write(...) const {
        return writeImpl(...);
    }
    ...

protected:
    // Implements reading from the buffer functionality
    virtual ErrorStatus readImpl(...) = 0;

    // Implements writing to a buffer functionality
    virtual ErrorStatus writeImpl(...) const = 0;
};

class ActualMessage1 : public Message
{
    ...
protected:
    virtual ErrorStatus readImpl(...) override {...};
    virtual ErrorStatus writeImpl(...) const override {...};
};

class ActualMessage2 : public Message
{
    ...
protected:
    virtual ErrorStatus readImpl(...) override {...};
    virtual ErrorStatus writeImpl(...) const override {...};
};
```

There is obviously a need to know success/failure status of the read/write operation. The `ErrorStatus` return value may be defined for example like this:

```
enum class ErrorStatus
{
    Success,
    NotEnoughData,
    BufferOverflow,
    InvalidMsgData,
    ...
};
```

Let's assume, that at the stage of parsing transport wrapping information, the ID of the message was retrieved and appropriate actual message object was created in an **efficient** way. This whole process will be described later in the Transport chapter.

Once the appropriate message object was created and returned in some kind of smart pointer, just call the `read(...)` member function of the message object:

```
using MsgPtr = std::unique_ptr<Message>;

MsgPtr msg = processTransportData(...)
auto es = msg.read(...); // read message payload
if (es != ErrorStatus::Success) {
    ... // handle error
}
```

Data Structures Independence

One of our goals was to make the implementation of the communication protocol to be system independent. In order to make the code as generic as possible we have to eliminate any dependency on specific data structures, where the incoming raw bytes are stored before being processed, as well as outgoing data before being sent out.

The best way to achieve such independence is to use iterators instead of specific data structures and make it a responsibility of the caller to maintain appropriate buffers:

```
template <typename TReadIter, typename TWriteIter>
class Message
{
public:
    using ReadIterator = TReadIter;
    using WriteIterator = TWriteIter;

    ErrorStatus read(ReadIterator& iter, std::size_t len) {
        return readImpl(iter, len);
    }

    ErrorStatus write(WriteIterator& iter, std::size_t len) const
 {
        return writeImpl(iter, len);
    }
    ...

protected:
    // Implements reading from the buffer functionality
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) = 0;

    // Implements writing to a buffer functionality
    virtual ErrorStatus writeImpl(WriteIterator& iter, std::size
_t len) const = 0;
};

template <typename TReadIter, typename TWriteIter>
class ActualMessage1 : public Message<TReadIter, TWriteIter>
{
    using Base = Message<TReadIter, TWriteIter>;
public:
    using Base::ReadIterator;
    using Base::WriteIterator;
    ...

protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) override {...};
    virtual ErrorStatus writeImpl(WriteIterator& iter, std::size
_t len) const override {...};
};
```

Please note, that iterators are passed by reference, which allows the increment and assignment operations required to implement serialisation/deserialisation functionality.

Also note, that the same implementation of the read/write operations can be used in any system with any restrictions. For example, the bare-metal embedded system cannot use dynamic memory allocation and must serialise the outgoing messages into a static array, which forces the definition of the write iterator to be `std::uint8_t*` .

```
using EmbReadIter = const std::uint8_t*;
using EmbWriteIter = std::uint8_t*;
using EmbMessage = Message<EmbReadIter, EmbWriteIter>
using EmbActualMessage1 = ActualMessage1<EmbReadIter, EmbWriteIt
er>
using EmbActualMessage2 = ActualMessage2<EmbReadIter, EmbWriteIt
er>

std::array<std::uint8_t, 1024> outBuf;
EmbWriteIter iter = &outBuf[0];

EmbActualMessage1 msg;
msg.write(iter, outBuf.size());
auto writtenCount = std::distance(&outBuf[0], iter); // iter was
 incremented
```

The Linux server system which resides on the other end of the I/O link doesn't have such limitation and uses `std::vector<std::uint8_t>` to store outgoing serialised messages. The generic and data structures independent implementation above makes it possible to be reused:

```
using LinReadIter = const std::uint8_t*;
using LinWriteIter = std::back_insert_iterator<std::vector<std::
uint8_t> >;
using LinMessage = Message<LinReadIter, LinWriteIter>
using LinActualMessage1 = ActualMessage1<LinReadIter, LinWriteIt
er>
using LinActualMessage2 = ActualMessage2<LinReadIter, LinWriteIt
er>

std::vector<std::uint8_t> outBuf;
LinWriteIter iter = std::back_inserter(outBuf);

LinActualMessage1 msg;
msg.write(iter, outBuf.max_size());
auto writtenCount = outBuf.size();
```

Data Serialisation

The `readImpl()` and `writeImpl()` member functions of the actual message class are supposed to properly serialise and deserialise message fields. It is a good idea to provide some common serialisation functions accessible by the actual message classes.

```
template <typename TReadIter, typename TWriteIter>
class Message
{
protected:
    template <typename T>
    static T readData(ReadIterator& iter) {...}

    template <typename T>
    static void writeData(T value, WriteIterator& iter) {...}
};
```

The `readData()` and `writeData()` static member functions above are responsible to implement the serialisation and deserialisation of the values using the right endian.

Depending on a communication protocol there may be a need to serialise only part of the value. For example, some field of communication protocol is defined to have only 3 bytes. In this case the value will probably be stored in a variable of `std::uint32_t` type. There must be similar set of functions, but with additional template parameter that specifies how many bytes to read/write:

```
template <typename TReadIter, typename TWriteIter>
class Message
{
protected:
    template <std::size_t TSize, typename T>
    static T readData(ReadIterator& iter) {...}

    template <std::size_t TSize, typename T>
    static void writeData(T value, WriteIterator& iter) {...}
};
```

CAUTION: The interface described above is very easy and convenient to use and quite easy to implement using straightforward approach. However, any variation of template parameters create an instantiation of new binary code, which may create significant code bloat if not used carefully. Consider the following:

- Read/write of signed vs unsigned integer values. The serialisation/deserialisation code is identical for both cases, but won't be considered as such when instantiating the functions. To optimise this case, there is a need to implement read/write operations only for unsigned value, while the “signed” functions become wrappers around the former. Don't forget a sign extension operation when retrieving partial signed value.
- The read/write operations are more or less the same for any length of the values, i.e of any types: (unsigned) char, (unsigned) short, (unsigned) int, etc... To optimise this case, there is a need for internal function that receives length of serialised value as a run time parameter, while the functions described above are mere wrappers around it.
- Usage of the iterators also require caution. For example reading values may be performed using regular `iterator` as well as `const_iterator`, i.e. iterator pointing to `const` values. These are two different iterator types that will duplicate the “read” functionality if both of them are used.

All the consideration points stated above require quite complex implementation of the serialisation/deserialisation functionality with multiple levels of abstraction which is beyond the scope of this book. It would be a nice exercise to try and implement them yourself. You may take a look at util/access.h file in the COMMS Library of the comms_champion project for reference.

Dispatching and Handling

When a new message arrives, its appropriate object is created, and the contents are deserialised using `read()` member function, described in previous chapter. It is time to dispatch it to an appropriate handling function. Many developers use the `switch` statement or even a sequence of `dynamic_cast`s to identify the real type of the message object and call appropriate handling function.

As you may have guessed, this is pretty inefficient, especially when there are more than 7-10 messages to handle. There is a much better way of doing a dispatch operation by using a C++ ability to differentiate between functions with the same name but with different parameter types. It is called Double Dispatch Idiom.

Let's assume we have a handling class `Handler` that is capable of handling all possible messages:

```
class Handler
{
public:
    void handle(ActualMessage1& msg);
    void handle(ActualMessage2& msg);
    ...
}
```

Then the definition of the messages may look like this:

```
class Message
{
public:
    void dispatch(Handler& handler)
    {
        dispatchImpl(handler);
    }
    ...

protected:
    virtual void dispatchImpl(Handler& handler) = 0;
};

class ActualMessage1 : public Message
{
    ...
protected:
    virtual void dispatchImpl(Handler& handler) override
    {
        handler.handle(*this); // invokes handle(ActualMessage1&
);
    }
};

class ActualMessage2 : public Message
{
    ...
protected:
    virtual void dispatchImpl(Handler& handler) override
    {
        handler.handle(*this); // invokes handle(ActualMessage2&
);
    }
};
```

Then the following code will invoke appropriate handling function in the `Handler` object:

```
using MsgPtr = std::unique_ptr<Message>;

MsgPtr msg = ... // any custom message object;
Handler handler;

msg->dispatch(handler); // will invoke right handling function.
```

Please note, that the `Message` interface class doesn't require the definition of the `Handler` class, the forward declaration of the latter is enough. The `Handler` also doesn't require the definitions of all the actual messages being available, forward declarations of all the message classes will suffice. Only the implementation part of the `Handler` class will require knowledge about the interface of the messages being handled. However, the public interface of the `Handler` class must be known when compiling `dispatchImpl()` member function of any `ActualMessageX` class.

Eliminating Boilerplate Code

You may also notice that the body of all `dispatchImpl()` member functions in all the `ActualMessageX` classes is going to be the same:

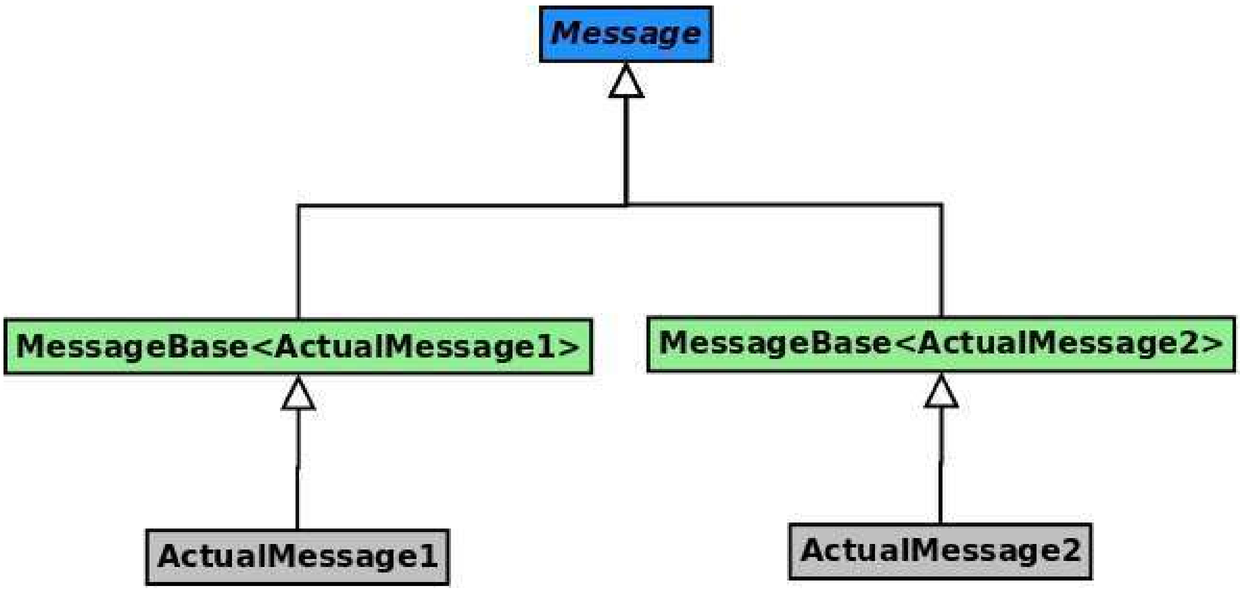
```
virtual void dispatchImpl(Handler& handler) override
{
    handler.handle(*this);
}
```

The problem is that `*this` expression in every function evaluates to the object of different type.

The apperent code duplication may be eliminated using Curiously Recurring Template Pattern idiom.

```
class Message
{
public:
    void dispatch(Handler& handler)
    {
        dispatchImpl(handler);
    }
    ...
protected:
    virtual void dispatchImpl(Handler& handler) = 0;
};

template <typename TDerived>
class MessageBase : public Message
{
protected:
    virtual void dispatchImpl(Handler& handler) override
    {
        handler.handle(static_cast<Derived&>(*this));
    }
}

class ActualMessage1 : public MessageBase<ActualMessage1>
{
    ...
};

class ActualMessage2 : public MessageBase<ActualMessage2>
{
    ...
};
```

Please note, that `ActualMessageX` provide their own type as a template parameter to their base class `MessageBase` and do not require to implement `dispatchImpl()` any more. The class hierarchy looks like this:



Handling Limited Number of Messages

What if there is a need to handle only limited number of messages, all the rest just need to be ignored. Let's assume the protocol defines 10 messages: `ActualMessage1` , `ActualMessage2` , ..., `ActualMessage10` . The messages that need to be handled are just `ActualMessage2` and `ActualMessage5` , all the rest ignored. Then the definition of the `Handler` class will look like this:

```
class Handler
{
public:
    void handle(ActualMessage2& msg) {...}
    void handle(ActualMessage5& msg) {...}

    void handle(Message& msg) {} // empty body
}
```

In this case, when compiling `dispatchImpl()` member function of `ActualMessage2` and `ActualMessage5` , the compiler will generate invocation code for appropriate `handle()` function. For the rest of the message classes, the best matching option will be invocation of `handle(Message&)` .

Polymorphic Handling

There may be a need to have multiple handlers for the same set of messages. It can easily be achieved by making the `Handler` an abstract interface class and defining its `handle()` member functions as virtual.

```
class Handler
{
public:
    virtual void handle(ActualMessage1& msg) = 0;
    virtual void handle(ActualMessage2& msg) = 0;
    ...
}

class ActualHandler1 : public Handler
{
public:
    virtual void handle(ActualMessage1& msg) override;
    virtual void handle(ActualMessage2& msg) override;
    ...
}

class ActualHandler2 : public Handler
{
public:
    virtual void handle(ActualMessage1& msg) override;
    virtual void handle(ActualMessage2& msg) override;
    ...
}
```

No other changes to dispatch functionality is required:

```
using MsgPtr = std::unique_ptr<Message>;

MsgPtr msg = ... // any custom message object;
AtualHandler1 handler1;
AtualHandler2 handler2;

// Works for any handler
msg->dispatch(handler1);
msg->dispatch(handler2);
```

Generic Handler

Now it's time to think about the required future effort of extending the handling functionality when new messages are added to the protocol and their respective classes are implemented. It is especially relevant when Polymorphic Handling is involved. There is a need to introduce new `virtual handle(...)` member function for every new message that is being added.

There is a way to delegate this job to the compiler using template specialisation. Let's assume, that all the message types, which need to be handled, are bundled into a simple declarative statement of `std::tuple` definition:

```
using AllMessages = std::tuple<
    ActualMessage1,
    ActualMessage2,
    ...
>;
```

Then the definition of the generic handling class will be as following:

```
// TCommon is common interface class for all the messages
// TAll is all the message types, that need to be handled, bundl
ed in std::tuple
template <typename TCommon, typename TAll>
class GenericHandler;

template <typename TCommon, typename TFirst, typename... TRest>
class GenericHandler<TCommon, std::tuple<TFirst, TRest...> > :
                        public GenericHandler<TCommon, std::tupl
e<TRest...> >
{
    using Base = GenericHandler<TCommon, std::tuple<TRest...> >;
public:
    using Base::handle; // Don't hide all handle() functions fro
m base classes
    virtual void handle(TFirst& msg)
    {
        // By default call handle(TCommon&)
        this->handle(static_cast<TCommon&>(msg));
    }
};

template <typename TCommon>
class GenericHandler<TCommon, std::tuple<> >
{
public:
    virtual ~GenericHandler() {}
    virtual void handle(TCommon&)
    {
        // Nothing to do
    }
};
```

The code above generates `virtual handle(TCommon&)` function for the common interface class, which does nothing by default. It also creates a separate `virtual handle(...)` function for every message type provided in `TAll` tuple. Every such function upcasts the message type to its interface class `TCommon` and invokes the `handle(TCommon&)` .

As the result simple declaration of

```
class Handler : public GenericHandler<Message, AllMessages> {};
```

is equivalent to having the following class defined:

```
class Handler
{
public:
    virtual void handle(ActualMessage1& msg)
    {
        this->handle(static_cast<Message&>(msg));
    }

    virtual void handle(ActualMessage2& msg)
    {
        this->handle(static_cast<Message&>(msg));
    }

    ...

    virtual void handle(Message& msg)
    {
        // do nothing
    }
```

From now on, when new message class is defined, just add it to the `AllMessages` tuple definition. If there is a need to override the default behaviour for specific message, override the appropriate message in the handling class:

```
class ActualHandler1 : public Handler
{
public:
    virtual void handle(ActualMessage2& msg) override
    {
        std::cout << "Handling ActualMessage2" << std::endl;
    }

    virtual void handle(Message& msg) override
    {
        std::cout << "Common handling function is invoked" << std
::endl;
    }
}
```

REMARK: Remember that the `Handler` class was forward declared when defining the `Message` interface class? Usually it looks like this:

```
class Handler;
class Message
{
public:
    void dispatch(Handler& handler) {...}
};
```

Note, that `Handler` is declared to be a `class`, which prevents it from being a simple `typedef` of `GenericHandler`. Usage of `typedef` will cause compilation to fail.

CAUTION: The implementation of the `GenericHandler` presented above creates a chain of **N + 1** inheritances for **N** messages defined in `AllMessages` tuple. Every new class adds a single virtual function. Many compilers will create a separate `vtable` for every such class. The size of every new `vtable` is greater by one entry than a previous one. Depending on total number of messages in that tuple, the code size may grow quite big due to growing number of `vtable`s generated by the compiler. It may be not suitable for some systems, especially bare-metal. It is possible to significantly reduce number of inheritances using more template specialisation classes. Below is an example of adding up to 3 virtual functions in a single class at once. You may easily extend the example to say 10 functions or more.

```
template <typename TCommon, typename TAll>
class GenericHandler;

template <typename TCommon, typename TFirst, TSecond, TThird, ty
pename... TRest>
class GenericHandler<TCommon, std::tuple<TFirst, TSecond, TThird
, TRest...> > :
                        public GenericHandler<TCommon, std::tupl
e<TRest...> >
{
    using Base = GenericHandler<TCommon, std::tuple<TRest...> >;
public:
    using Base::handle;
    virtual void handle(TFirst& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
    virtual void handle(TSecond& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
    virtual void handle(TThird& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
};

template <typename TCommon, typename TFirst, typename TSecond>
class GenericHandler<TCommon, std::tuple<TFirst, TSecond> >
{
public:
    virtual ~GenericHandler() {}
    virtual void handle(TFirst& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
    virtual void handle(TSecond& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
    virtual void handle(TCommon&)
    {
        // Nothing to do
    }
};

template <typename TCommon, typename TFirst>
class GenericHandler<TCommon, std::tuple<TFirst> >
{
public:
    virtual ~GenericHandler() {}
    virtual void handle(TFirst& msg)
    {
        this->handle(static_cast<TCommon&>(msg));
    }
    virtual void handle(TCommon&)
    {
        // Nothing to do
    }
};

template <typename TCommon>
class GenericHandler<TCommon, std::tuple<> >
{
public:
    virtual ~GenericHandler() {}
    virtual void handle(TCommon&)
    {
        // Nothing to do
    }
};
```

Extending Interface

Let's assume the protocol was initially developed for some embedded system which required very basic message interface of only **read** / **write** / **dispatch**.

The interface class definition was defined allowing iterators to be specified elsewhere:

```
class Handler;
template <typename TReadIterator, typename TWriteIterator>
class Message
{
public:
    using ReadIterator = TReadIterator;
    using WriteIterator = TWriteIterator;

    // Read the message
    ErrorStatus read(ReadIterator& iter, std::size_t len)
    {
        return readImpl(iter, len);
    }

    // Write the message
    ErrorStatus write(WriteIterator& iter, std::size_t len) const

    {
        return writeImpl(iter, len);
    }

    // Dispatch to handler
    void dispatch(Handler& handler)
    {
        dispatchImpl(handler);
    }

protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) = 0;
    virtual ErrorStatus writeImpl(WriteIterator& iter, std::size
_t len) const = 0;
    virtual void dispatchImpl(Handler& handler) = 0;
};
```

The intermediate class allowing common implementation of `dispatchImpl()` :

```
template <typename TReadIterator, typename TWriteIterator, typename TDerived>
class MessageBase : public Message<TReadIterator, TWriteIterator>
{
protected:
    virtual void dispatchImpl(Handler& handler) override {...};
};
```

And the actual message classes:

```
template <typename TReadIterator, typename TWriteIterator>
class ActualMessage1 : public MessageBase<TReadIterator, TWriteIterator, ActualMessage1>
{
    ...
};

template <typename TReadIterator, typename TWriteIterator>
class ActualMessage2 : public MessageBase<TReadIterator, TWriteIterator, ActualMessage2>
{
    ...
};
```

Then, after a while a new application needs to be developed, which monitors the I/O link and dumps all the message traffic into standard output and/or *.csv file. This application requires knowledge about names of the messages, and it would be convenient to add an appropriate function into the common message interface and reuse the existing implementation. There is one problem though, the code of the protocol is already written and used in the embedded system, which does not require this additional functionality and its binary code should not contain these extra functions.

One of the solutions can be to use preprocessor:

```
template <...>
class Message
{
public:
#ifdef HAS_NAME
    const char* name() const
    {
        return nameImpl();
    }
#endif

protected:
#ifdef HAS_NAME
    virtual const char* nameImpl() const = 0;
#endif
};

template <...>
class MessageBase : public Message<...> {...};

template <>
class ActualMessage1 : public MessageBase<...>
{
protected:
#ifdef HAS_NAME
    virtual const char* nameImpl() const
    {
        return "ActualMessage1";
    }
#endif
};
```

Such approach may work for some products, but not for others, especially ones that developed by multiple teams. If one team developed a reference implementation of the communication protocol being used and is an "owner" of the code, then it may be difficult and/or impractical for other team to push required changes upstream.

Another approach is to remove hard coded inheritance relationship between `Message` interface class and intermediate `MessageBase` class. Instead, provide the common interface class as a template parameter to the latter:

```
template <typename TIternface, typename TDerived>
class MessageBase : public TIternface
{
protected:
    virtual void dispatchImpl(Handler& handler) override {...};
}
```

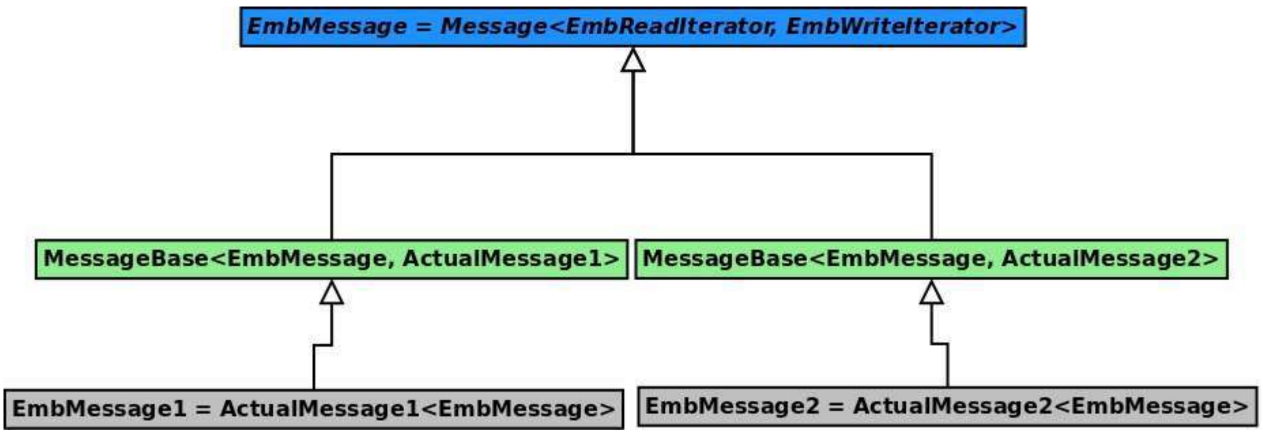
And the `ActualMessage*` classes will look like this:

```
template <typename TIternface>
class ActualMessage1 : public MessageBase<TIternface, ActualMess
age1>
{
    ...
};

template <typename TIternface>
class ActualMessage2 : public MessageBase<TIternface, ActualMess
age2>
{
    ...
};
```

Then, the initial embedded system may use the common protocol code like this:

```
using EmbReadIterator = ...;
using EmbWriteIterator = ...;
using EmbMessage = Message<EmbReadIterator, EmbWriteIterator>;
using EmbMessage1 = ActualMessage1<EmbMessage>;
using EmbMessage2 = ActualMessage2<EmbMessage>;
```

The original class hierarchy preserved intact:


EmbMessage = Message<EmbReadIterator, EmbWriteIterator>
MessageBase<EmbMessage, ActualMessage1>
MessageBase<EmbMessage, ActualMessage2>
EmbMessage1 = ActualMessage1<EmbMessage>
EmbMessage2 = ActualMessage2<EmbMessage>

And when extended interface and functionality are required, just use extra class inheritances:

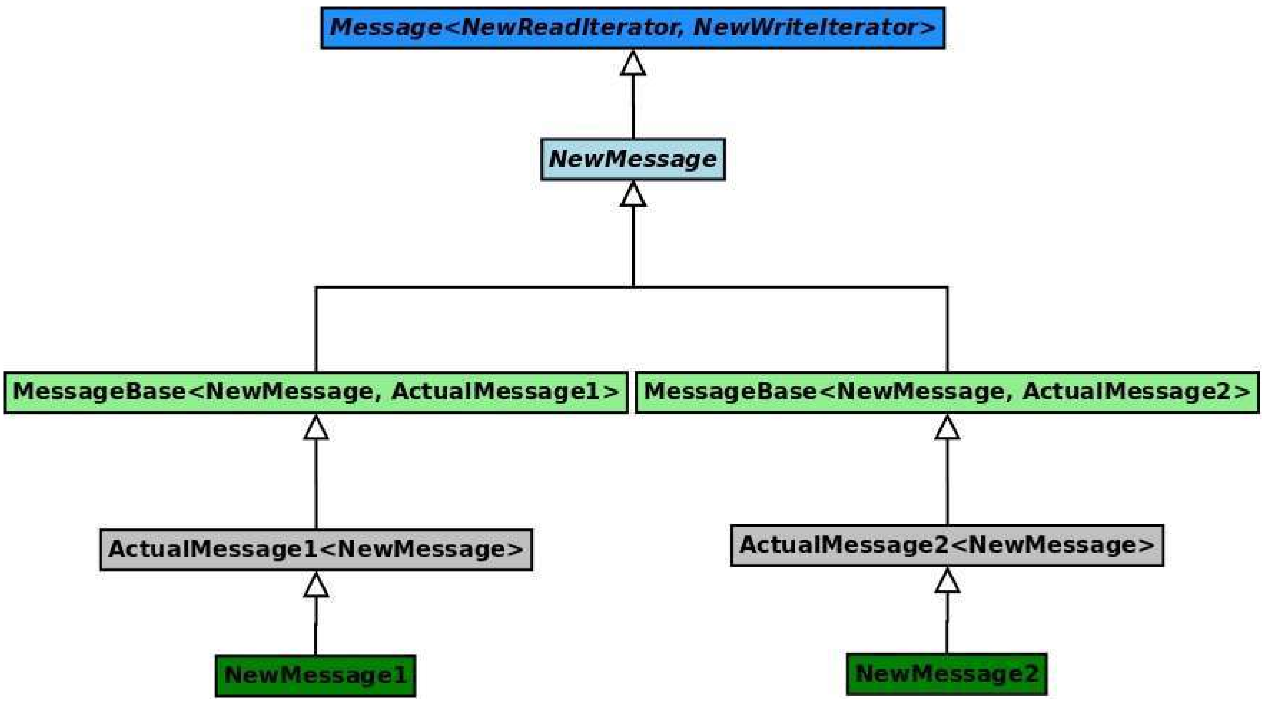
```
// Define extended interface
template <typename TReadIterator, typename TWriteIterator>
class ExtMessage : public Message<TReadIterator, TWriteIterator>
{
public:
    const char* name() const
    {
        return nameImpl();
    }

protected:
    virtual const char* nameImpl() const = 0;
}

// Define extended messages
<typename TInterface>
class ExtActualMessage1 : public ActualMessage1<TInterface>
{
protected:
    virtual const char* nameImpl() const
    {
        return "ActualMessage1";
    }

}
```

The new application that requires extended implementation may still reuse the common protocol code like this:

```
using NewReadIterator = ...;
using NewWriteIterator = ...;
using NewMessage = ExtMessage<NewReadIterator, NewWriteIterator>
;
using NewMessage1 = ExtActualMessage1<NewMessage>;
using NewMessage2 = ExtActualMessage2<NewMessage>;
```

As a result, no extra modifications to the original source code of the protocol implementation is required, and every team achieves their own goal. Everyone is happy!!!

The extended class hierarchy becomes:


Message<NewReadIterator, NewWriteIterator>
NewMessage
MessageBase<NewMessage, ActualMessage1>
MessageBase<NewMessage, ActualMessage2>
ActualMessage1<NewMessage>
ActualMessage2<NewMessage>
NewMessage1
NewMessage2

Fields

Every message in any communication protocol has zero or more internal fields, which get serialised in some predefined order and transferred as message payload over I/O link.

Usually developers implement some boilerplate code of explicitly reading and writing all message fields in appropriate functions, such as `readImpl()` and `writeImpl()` described in Reading and Writing chapter. The primary disadvantage of this approach is an increased development effort when contents of some message need to be modified, i.e. some new field is added, or existing one removed, even when the type of an existing field changes. Having multiple places in the code, that need to be updated, leads to an increased chance of forgetting to update one of the places, or introducing some silly error, which will take time to notice and fix.

This chapter describes how to automate basic operations, such as read and write, i.e. to make it a responsibility of the compiler to generate appropriate code. All the developer needs to do is to define the list of all the field types the message contains, and let the compiler do the job.

Automating Basic Operations

Let's start with automation of read and write. In most cases the `read()` operation of the message has to read **all** the fields the message contains, as well as `write()` operation has to write **all** the fields of the message.

In order to make the generation of appropriate read/write code to be a job of the compiler we have to:

- Provide the same interface for every message field.
- Introduce a meta-programming friendly structure to hold all the fields, such as `std::tuple`.
- Use meta-programming techniques to iterate over every field in the bundle and invoke the required read/write function of every field.

Let's assume, all the message fields provide the following interface:

```
class SomeField
{
public:
    // Value storage type definition
    using ValueType = ...;

    // Provide an access to the stored value
    ValueType& value();
    const ValueType& value() const;

    // Read (deserialise) and update internal value
    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len);

    // Write (serialise) internal value
    template <typename TIter>
    ErrorStatus write(TIter& iter, std::size_t len) const;

    // Get the serialisation length
    std::size_t length() const;

private:
    ValueType m_value;
}
```

The custom message class needs to define its fields bundled in `std::tuple`

```
class ActualMessage1 : public Message
{
public:
    using Field1 = ...
    using Field2 = ...
    using Field3 = ...

    using AllFields = std::tuple<
        Field1,
        Field2,
        Field3
    >;
    ...
protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) override
    {
        ...// invoke read() member function of every field
    }

    virtual ErrorStatus writeImpl(WriteIterator& iter, std::size
_t len) const override
    {
        ...// invoke write() member function of every field
    }

private:
    AllFields m_fields;
};
```

What remains is to implement automatic invocation of `read()` and `write()` member function for every field in `AllFields` tuple.

Let's take a look at standard algorithm std::for_each. Its last parameter is a functor object, which must define appropriate `operator()` member function. This function is invoked for every element being iterated over. What we need is something similar, but instead of receiving iterators, it must receive a full tuple object, and the `operator()` of provided functor must be able to receive any type, i.e. be a template function.

As the result the signature of such function may look like this:

```
template <typename TTuple, typename TFunc>
void tupleForEach(TTuple&& tuple, TFunc&& func);
```

where `tuple` is l- or r-value reference to any `std::tuple` object, and `func` is l- or r-value reference to a functor object that must define the following public interface:

```
struct MyFunc
{
    template <typename TTupleElem>
    void operator()(TTupleElem&& elem) {...}
};
```

Implementation of the `tupleForEach()` function described above can be a nice exercise for practising some meta-programming skills. Appendix A contains the required code if help is required.

Implementing Read

In order to implement read functionality there is a need to define proper reading functor class, which may receive any field:

```
class FieldReader
{
public:
    FieldReader(ErrorStatus& status, ReadIterator& iter, std::si
ze_t& len)
      : m_status(status),
        m_iter(iter),
        m_len(len)
    {
    }

    template <typename TField>
    void operator()(TField& field)
    {
        if (m_status != ErrorStatus::Success) {
            // Error occurred earlier, don't continue with read
            return;
        }
        m_status = field.read(m_iter, m_len);
        if (m_status == ErrorStatus::Success) {
            m_len -= field.length();
        }
    }

private:
    ErrorStatus& m_status;
    ReadIterator& m_iter;
    std::size_t& m_len;
}
```

Then the body of `readImpl()` member function of the actual message class may look like this:

```
class ActualMessage1 : public Message
{
public:
    using AllFields = std::tuple<...>;
protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) override
    {
        auto status = ErrorStatus::Success;
        tupleForEach(m_fields, FieldReader(status, iter, len));
        return status;
    }

private:
    AllFields m_fields;
};
```

From now on, any modification to the `AllFields` bundle of fields does NOT require any additional modifications to the body of `readImpl()` function. It becomes a responsibility of the compiler to invoke `read()` member function of all the fields.

Implementing Write

Implementation of the write functionality is very similar. Below is the implementation of the writer functor class:

```
class FieldWriter
{
public:
    FieldWriter(ErrorStatus& status, WriterIterator& iter, std::
size_t& len)
      : m_status(status),
        m_iter(iter),
        m_len(len)
    {
    }

    template <typename TField>
    void operator()(TField& field)
    {
        if (m_status != ErrorStatus::Success) {
            // Error occurred earlier, don't continue with write
            return;
        }
        m_status = field.write(m_iter, m_len);
        if (m_status == ErrorStatus::Success) {
            m_len -= field.length();
        }
    }

private:
    ErrorStatus& m_status;
    WriterIterator& m_iter;
    std::size_t& m_len;
}
```

Then the body of `writeImpl()` member function of the actual message class may look like this:

```
class ActualMessage1 : public Message
{
public:
    using AllFields = std::tuple<...>;

protected:
    virtual ErrorStatus writeImpl(WriterIterator& iter, std::siz
e_t len) const override
    {
        auto status = ErrorStatus::Success;
        tupleForEach(m_fields, FieldWriter(status, iter, len));
        return status;
    }

private:
    AllFields m_fields;
};
```

Just like with reading, any modification to the `AllFields` bundle of fields does NOT require any additional modifications to the body of `writeImpl()` function. It becomes a responsibility of the compiler to invoke `write()` member function of all the fields.

Eliminating Boilerplate Code

It is easy to notice that the body of `readImpl()` and `writeImpl()` of every `ActualMessage*` class looks the same. What differs is the tuple of fields which get iterated over.

It is possible to eliminate such duplication of boilerplate code by introducing additional class in the class hierarchy, which receives a bundle of fields as a template parameter and implements the required functions. The same technique was used to eliminate boilerplate code for message dispatching.

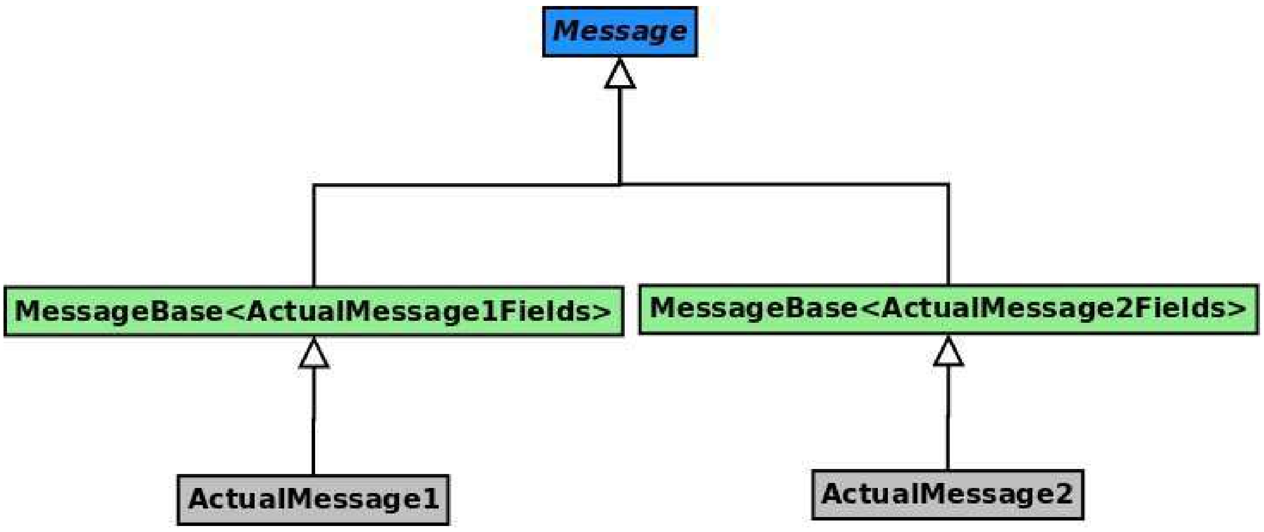
```
// Common interface class:
class Message {...};
```

```
template <typename TFields>
class MessageBase : public Message
{
public:
    using Message::ReadIterator;
    using Message::WriteIterator;
    using AllFields = TFields;

    // Access to fields bundle
    AllFields& fields() { return m_fields; }
    const AllFields& fields() const { return m_fields; }

protected:
   virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
len) override
    {
        auto status = ErrorStatus::Success;
        tupleForEach(m_fields, FieldReader(status, iter, len));
        return status;
    }

    virtual ErrorStatus writeImpl(WriterIterator& iter, std::siz
e_t len) const override
    {
        auto status = ErrorStatus::Success;
        tupleForEach(m_fields, FieldWriter(status, iter, len));
        return status;
    }

private:
    class FieldReader { ... /* same code as from earlier example
 */ };
    class FieldWriter { ... /* same code as from earlier example
 */ };

    AllFields m_fields;
}
```

All the `ActualMessage*` classes need to inherit from `MessageBase` while providing their own fields. The right implementation of `readImpl()` and `writeImpl()` is going to be generated by the compiler automatically for every custom message.

```
using ActualMessage1Fields = std::tuple<...>;
class ActualMessage1 : public MessageBase<ActualMessage1Fields>
{...};

using ActualMessage2Fields = std::tuple<...>;
class ActualMessage2 : public MessageBase<ActualMessage2Fields>
{...};

...
```

The class hierarchy looks like this:


Message
MessageBase<ActualMessage1Fields>
MessageBase<ActualMessage2Fields>
ActualMessage1
ActualMessage2

Other Basic Operations

In addition to read and write, there are other operations that can be automated. For example, the serialisation length of the full message is a summary of the serialisation lengths of all the fields. If every field can report its serialisation length, then the implementation may look like this:

```
class Message
{
public:
    std::size_t length() const
    {
        return lengthImpl();
    }

protected:
    virtual std::size_t lengthImpl() const = 0;
};

template <typename TFields>
class MessageBase : public Message
{
protected:
    virtual std::size_t lengthImpl() const override
    {
        return tupleAccumulate(m_fields, 0U, LengthCalc());
    }

private:
    struct LengthCalc
    {
        template <typename TField>
        std::size_t operator()(std::size_t size, const TField& f
ield) const
        {
            return size + field.length();
        }
    };

    AllFields m_fields;
}
```

NOTE, that example above used `tupleAccumulate()` function, which is similar to std::accumulate. The main difference is that binary operation function object, provided to the function, must be able to receive any type, just like with `tupleForEach()` described earlier. The code of `tupleAccumulate()` function can be found in Appendix B.

Another example is an automation of validity check. In most cases the message is considered to be valid if **all** the fields are valid. Let's assume that every fields can also provide an information about validity of its data:

```
class SomeField
{
public:
    // Get validity information
    bool valid() const;
    ...
}
```

The implementation of message contents validity check may look like this:

```
class Message
{
public:
    bool valid() const
    {
        return validImpl();
    }

protected:
    virtual bool validImpl() const = 0;
};

template <typename TFields>
class MessageBase : public Message
{
protected:
    virtual bool validImpl() constImpl() const override
    {
        return tupleAccumulate(m_fields, true, ValidityCalc());
    }

private:
    struct ValidityCalc
    {
        template <typename TField>
        bool operator()(bool valid, const TField& field) const
        {
            return valid && field.valid();
        }
    };

    AllFields m_fields;
}
```

Overriding Automated Default Behaviour

It is not uncommon to have some optional fields in the message, the existence of which depends on some bits in previous fields. In this case the default read and/or write behaviour generated by the compiler needs to be modified. Thanks to the inheritance relationship between the classes, nothing prevents us to overriding the `readImpl()` and/or `writeImpl()` function and provide the right behaviour:

```
using ActualMessage1Fields = std::tuple<...>;
class ActualMessage1 : public MessageBase<ActualMessage1Fields>
{
protected:
    virtual void readImpl(ReadIterator& iter, std::size_t len) override {...}
    virtual void writeImpl(WriteIterator& iter, std::size_t len) const override {...}
}
```

The `MessageBase<...>` class already contains the definition of `FieldReader` and `FieldWriter` helper classes, it can provide helper functions to read/write only several fields from the whole bundle. These functions can be reused in the overriding implementations of `readImpl()` and/or `writeImpl()` :

```
template <typename TFields>
class MessageBase : public Message
{
    ...
protected:
    template <std::size_t TFromIdx, std::size_t TUntilIdx>
    ErrorStatus readFieldsFromUntil(
        ReadIterator& iter,
        std::size_t& size)
    {
        auto status = ErrorStatus::Success;
        tupleForEachFromUntil<TFromIdx, TUntilIdx>(m_fields, Fie
ldReader(status, iter, size));
        return status;
    }

    template <std::size_t TFromIdx, std::size_t TUntilIdx>
    ErrorStatus writeFieldsFromUntil(
        WriteIterator& iter,
        std::size_t size) const
    {
        auto status = ErrorStatus::Success;
        tupleForEachFromUntil<TFromIdx, TUntilIdx>(m_fields, Fie
ldWriter(status, iter, size));
        return status;
    }

private:
    class FieldReader { ... };
    class FieldWriter { ... };

    AllFields m_fields;
}
```

The provided `readFieldsFromUntil()` and `writeFieldsFromUntil()` protected member functions use `tupleForEachFromUntil()` function to perform read/write operations on a group of selected fields. It is similar to `tupleForEach()` used earlier, but receives additional template parameters, that specify indices of the fields for which the provided functor object needs to be invoked. The code of `tupleForEachFromUntil()` function can be found in Appendix C.

Working With Fields

In order to automate some basic operations, all the fields had to provide the same basic interface. As the result the actual field values had to be wrapped in a class that defines the required public interface. Such class must also provide means to access/update the wrapped value. For example:

```
class SomeField
{
public:
    // Value storage type definition
    using ValueType = ...;

    // Provide an access to the stored value
    ValueType& value() { return m_value; }
    const ValueType& value() const { return m_value; }
    ...
private:
    ValueType m_value;
}
```

Let's assume the `ActualMessage1` defines 3 integer value fields with serialisation lengths of 1, 2, and 4 bytes respectively.

```
using ActualMessage1Fields = std::tuple<
    IntValueField<std::int8_t>,
    IntValueField<std::int16_t>
    IntValueField<std::int32_t>
>;
class ActualMessage1 : public MessageBase<ActualMessage1Fields>
{...};
```

The Dispatching and Handling chapter described the efficient way to dispatch message object to its handler. The appropriate handling function may access its field's value using the following code flow:

```
class Handler
{
public:
    void handle(ActualMessage1& msg)
    {
        // Get access to the field's bundle of type std::tuple
        auto& allFields = msg.fields();

        // Get access to the field abstractions.
        auto& field1 = std::get<0>(allFields);
        auto& field2 = std::get<1>(allFields);
        auto& field3 = std::get<2>(allFields);

        // Get access to the values themselves:
        std::int8_t val1 = field1.value();
        std::int16_t val2 = field2.value();
        std::int32_t val3 = field3.value();

        ... Do something with retrieved values
    }
};
```

When preparing message to send, the similar code sequence may be applied to update the values:

```
ActualMessage1 msg;

// Get access to the field's bundle of type std::tuple
auto& allFields = msg.fields();

// Get access to the field abstractions.
auto& field1 = std::get<0>(allFields);
auto& field2 = std::get<1>(allFields);
auto& field3 = std::get<2>(allFields);

// Update the values themselves:
field1.value() = ...;
field2.value() = ...;
field3.value() = ...;

// Serialise and send the message:
sendMessage(msg);
```

Common Field Types

The majority of communication protocols use relatively small set of various field types. However, the number of various ways used to serialise these fields, as well as handle them in different parts of the code, may be significantly bigger.

It would be impractical to create a separate class for each and every variant of the same type fields. That's why there is a need to use template parameters when defining a frequently used field type. The basic example would be implementing **numeric integral value** fields. Different fields of such type may have different serialisation lengths.

```
template <typename TValueType>
class IntValueField
{
public:
    using ValueType = TValueType;

    // Accessed stored value
    ValueType& value() { return m_value; }
    const ValueType& value() const { return m_value; }

    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len) {... /* read
m_value */ }

    template <typename TIter>
    ErrorStatus write(TIter& iter, std::size_t len) const {... /
* write m_value */ }

private:
    ValueType m_value = 0;
};
```

Below is a description of most common fields used by majority of binary communication protocols with the list of possible variations, that can influence how the field is serialised and/or handled.

The Generic Library chapter will concentrate on how to generalise development of any communication protocol by creating a generic library and reusing it in independent implementations of various protocols. It will also explain how to create generic field classes for the types listed below.

Numeric Integral Values

Used to operate with simple numeric integer values.

- May have different serialisation length: 1, 2, 3, 4, 5, ... bytes. Having basic types of `std::uint8_t` , `std::uint16_t` , `std::uint32_t` , ... may be not enough. Some extra work may be required to support lengths, such as 3, 5, 6, 7 bytes.
- May be signed and unsigned. Some protocols require different serialisation rules for signed values, such as adding some predefined offset prior to serialisation to make sure that the value being serialised is non-negative. When value deserialised, the same offset must be subtracted to get the actual value.
- May have variable serialisation length, based on the value being serialised, such as having Base-128 encoding.

Enumeration Values

Similar to Numeric Integral Values, but storing the value as enumeration type for easier access.

Bitmask Values

Similar to Numeric Integral Values, but with **unsigned** internal storage type and with each bit having separate meaning. The class definition should support having different serialisation lengths as well as provide a convenient interface to inquire about and update various bits' values.

Strings

Some protocols serialise strings by prefixing the string itself with its size, others have '\0' suffix to mark the end of the string. Some strings may be allocated a fixed size and require '\0' padding if its actual length is shorter.

Consider how the internal string value is stored. Usually `std::string` is used. However, what about the bare-metal embedded systems, that disallow usage of dynamic memory allocation and/or exceptions? There needs to be a way to substitute underlying `std::string` with a custom implementation of some `StaticString` that exposes similar interface, but receives a maximum storage size as a template parameter.

Lists

There may be lists of raw bytes, list of other fields, or even a group of fields. Similar to Strings, the serialisation of lists may differ. Lists of variable size may require a prefix with their size information. Other lists may have fixed (predefined) size and will not require any additional size information.

The internal storage consideration is applicable here as well. For most systems `std::vector` will do the job, but for bare-metal ones something else may be required. For example some custom implementation of `StaticVector` that exposes the same public interface, but receives a maximum storage size as a template parameter. There must be an easy way to substitute one with another.

Bundles

The group of fields sometimes needs to be bundled into a single entity and be treated as a single field. The good example would be having a list of complex structures (bundles).

Bitfields

Similar to Bundles, where every field member takes only limited number of bits instead of bytes. Usually the members of the bitfields are Numeric Integral Values, Enumeration Values, and Bitmask Values

Common Variations

All the fields stated above may require an ability to:

- set custom default value when the field object is created.
- have custom value validation logic.
- fail the read operation on invalid value.
- ignore the incoming invalid value, i.e. not to fail the read operation, but preserve the existing value if the value being read is invalid.

Generic Library

All the generalisation techniques, described so far, are applicable to most binary communication protocols. It is time to think about something generic - a library that can be reused between independent projects and facilitate a development of any binary communication protocol.

From now on, every generic, protocol independent class and/or function is going to reside in `comms` namespace in order to differentiate it from a protocol specific code.

Generalising Message Interface

The basic generic message interface may include the following operations:

- Retrieve the message ID.
- Read (deserialise) the message contents from raw data in a buffer.
- Write (serialise) the message contents into a buffer.
- Calculate the serialisation length of the message.
- Dispatch the message to an appropriate handling function.
- Check the validity of the message contents.

There may be multiple cases when not all of the operations stated above are needed for some specific case. For example, some sensor only reports its internal data to the outside world over some I/O link, and doesn't listen to the incoming messages. In this case the `read()` operation is redundant and its implementation should not take space in the produced binary code. However, the component that resides on the other end of the I/O link requires the opposite functionality, it only consumes data, without producing anything, i.e. `write()` operation becomes unnecessary.

There must be a way to limit the basic interface to a particular set of functions, when needed.

Also there must be a way to specify:

- type used to store and report the message ID.
- type of the read/write iterators
- endian used in data serialisation.
- type of the message handling class, which is used in `dispatch()` functionality.

The best way to support such variety of requirements is to use the variadic templates feature of C++11, which allows having non-fixed number of template parameters.

These parameters have to be parsed and used to define all the required internal functions and types. The common message interface class is expected to be defined like this:

```
namespace comms
{
template <typename... TOptions>
class Message
{
    ...
};
} // namespace comms
```

where `TOptions` is a set of classes/structs, which can be used to define all the required types and functionalities.

Below is an example of such possible option classes:

```
namespace comms
{
namespace option
{
// Define type used to store message ID
template <typename T>
struct MsgIdType{};

// Specify type of iterator used for reading
template <typename T>
struct ReadIterator {};

// Specify type of iterator used for writing
template <typename T>
struct WriteIterator {};

// Use little endian for serialisation (instead of default big)
struct LittleEndian {};

// Include serialisation length retrieval in public interface
struct LengthInfoInterface {};

// Include validity check in public interface
struct ValidCheckInterface {};

// Define handler class
template <typename T>
struct Handler{};
} // namespace option
} // namespace comms
```

Our **PRIMARY OBJECTIVE** for this chapter is to provide an ability to create a common message interface class with only requested functionality.

For example, the definition of `MyMessage` interface class below

```
class MyHandler;
using MyMessage = comms::Message<
    comms::option::MsgIdType<std::uint16_t>, // use std::uint16_t as message ID type
    comms::option::ReadIterator<const std::uint8_t*>, // use const std::uint8_t* as iterator for reading
    comms::option::WriteIterator<std::uint8_t*>, // use std::uint8_t* as iterator for writing
    comms::option::LengthInfoInterface, // add length() member function to interface
    comms::option::Handler<MyHandler> // add dispatch() member function with MyHandler as the handler class
>;
```

should be equivalent to defining:

```
class MyMessage
{
public:
    using MsgIdType = std::uint16_t;
    using ReadIterator = const std::uint8_t*;
    using WriteIterator = std::uint8_t*;
    using Handler = MyHandler;

    MsgIdType id() {...}
    ErrorStatus read(ReadIterator& iter, std::size_t len) {...}
    ErrorStatus write(WriteIterator& iter, std::size_t len) const {...}
    std::size_t length() const {...}
    void dispatch(Handler& handler) {...}
protected:
    template <typename T>
    static T readData(ReadIterator& iter) {...} // use big endian by default

    template <typename T>
    static void writeData(T value, WriteIterator& iter) {...}  // use big endian by default
    ...
};
```

And the following definition of `MyMessage` interface class

```
using MyMessage = comms::Message<
    comms::option::MsgIdType<std::uint8_t>, // use std::uint8_t as message ID type
    comms::option::LittleEndian, // use little endian in serialisation
    comms::option::ReadIterator<const std::uint8_t*> // use const std::uint8_t* as iterator for reading
>;
```

will be equivalent to:

```
class MyMessage
{
public:
    using MsgIdType = std::uint8_t;
    using ReadIterator = const std::uint8_t*;

    MsgIdType id() {...}
    ErrorStatus read(ReadIterator& iter, std::size_t len) {...}
protected:
    template <typename T>
    static T readData(ReadIterator& iter) {...} // use little en
dian

    template <typename T>
    static void writeData(T value, WriteIterator& iter) {...}  /
/ use little endian

    ...
};
```

Looks nice, isn't it? So, how are we going to achieve this? Any ideas?

That's right! We use **MAGIC**!

Sorry, I mean template meta-programming. Let's get started!

Parsing the Options

First thing, that needs to be done, is to parse the provided options and record them in some kind of a summary structure, with predefined list of `static const bool` variables, which indicate what options have been used, such as one below:

```
struct MessageInterfaceParsedOptions
{
    static const bool HasMsgIdType = false;
    static const bool HasLittleEndian = false;
    static const bool HasReadIterator = false;
    static const bool HasWriteIterator = false;
    static const bool HasHandler = false;
    static const bool HasValid = false;
    static const bool HasLength = false;
}
```

If some variable is set to `true`, the *summary structure* may also contain some additional relevant types and/or more variables.

For example the definition of

```
class MyHandler;
using MyMessage = comms::Message<
    comms::option::MsgIdType<std::uint16_t>, // use std::uint16_t

    comms::option::ReadIterator<const std::uint8_t*>, // use con
st std::uint8_t* as iterator for reading
    comms::option::WriteIterator<std::uint8_t*>, // use std::uin
t8_t* as iterator for writing
    comms::option::LengthInfoInterface, // add length() member f
unction to interface
    comms::option::Handler<MyHandler> // add dispatch() member f
unction with MyHandler as the handler class
>;
```

should result in

```
struct MessageInterfaceParsedOptions
{
    static const bool HasMsgIdType = true;
    static const bool HasLittleEndian = false;
    static const bool HasReadIterator = true;
    static const bool HasWriteIterator = true;
    static const bool HasHandler = true;
    static const bool HasValid = false;
    static const bool HasLength = true;

    using MsgIdType = std::uint16_t;
    using ReadIterator = const std::uint8_t*;
    using WriteIterator = std::uint8_t*;
    using Handler = MyHandler;
}
```

Here goes the actual code.

First, there is a need to define an initial version of such summary structure:

```
namespace comms
{
template <typename... TOptions>
class MessageInterfaceParsedOptions;

template <>
struct MessageInterfaceParsedOptions<>
{
    static const bool HasMsgIdType = false;
    static const bool HasLittleEndian = false;
    static const bool HasReadIterator = false;
    static const bool HasWriteIterator = false;
    static const bool HasHandler = false;
    static const bool HasValid = false;
    static const bool HasLength = false;
}
} // namespace comms
```

Then, handle the provided options one by one, while replacing the initial values and defining additional types when needed.

```
namespace comms
{
template <typename T, typename... TOptions>
struct MessageInterfaceParsedOptions<comms::option::MsgIdType<T>
, TOptions...> :
                                        public MessageInterfaceP
arsedOptions<TOptions...>
{
    static const bool HasMsgIdType = true;
    using MsgIdType = T;
};

template <typename... TOptions>
struct MessageInterfaceParsedOptions<comms::option::LittleEndian
, TOptions...> :
                                        public MessageInterfaceP
arsedOptions<TOptions...>
{
    static const bool HasLittleEndian = true;
};

template <typename T, typename... TOptions>
struct MessageInterfaceParsedOptions<comms::option::ReadIterator
<T>, TOptions...> :
                                        public MessageInterfaceP
arsedOptions<TOptions...>
{
    static const bool HasReadIterator = true;
    using ReadIterator = T;
};

... // and so on
} // namespace comms
```

Note, that inheritance relationship is used, and according to the C++ language specification the new variables with the same name hide (or replace) the variables defined in the base class.

Also note, that the order of the options being used to define the interface class does NOT really matter. However, it is recommended, to add some `static_assert()` statements in, to make sure the same options are not used twice, or no contradictory ones are used together (if such exist).

Assemble the Required Interface

The next stage in the **defining message interface** process is to define various chunks of interface functionality and connect them via inheritance.

```
namespace comms
{
// ID retrieval chunk
template <typename TBase, typename TId>
class MessageInterfaceIdTypeBase : public TBase
{
public:
    using MsgIdType = TId;
    MsgIdType getId() const
    {
        return getIdImpl();
    }

protected:
    virtual MsgIdType getIdImpl() const = 0;
};

// Big endian serialisation chunk
template <typename TBase>
class MessageInterfaceBigEndian : public TBase
{
protected:
    template <typename T>
    static T readData(ReadIterator& iter) {...} // use big endian
    template <typename T>
    static void writeData(T value, WriteIterator& iter) {...}  /
/ use big endian
};

// Little endian serialisation chunk
template <typename TBase>
class MessageInterfaceLittleEndian : public TBase
{
protected:
    template <typename T>
    static T readData(ReadIterator& iter) {...} // use little en
dian

    template <typename T>
    static void writeData(T value, WriteIterator& iter) {...}  /
/ use little endian
};

// Read functionality chunk
template <typename TBase, typename TReadIter>
class MessageInterfaceReadBase : public TBase
{
public:
    using ReadIterator = TReadIter;
    ErrorStatus read(ReadIterator& iter, std::size_t size)
    {
        return readImpl(iter, size);
    }

protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 size) = 0;
};

... // and so on
} // namespace comms
```

Note, that the interface chunks receive their base class through template parameters. It will allow us to connect them together using inheritance. Together they can create the required custom interface.

There is a need for some extra helper classes to implement such connection logic which chooses only requested chunks and skips the others.

The code below chooses whether to add `MessageInterfaceIdTypeBase` into the inheritance chain of interface chunks.

```
namespace comms
{
template <typename TBase, typename TParsedOptions, bool THasMsgI
dType>
struct MessageInterfaceProcessMsgId;

template <typename TBase, typename TParsedOptions>
struct MessageInterfaceProcessMsgId<TBase, TParsedOptions, true>
{
    using Type = MessageInterfaceIdTypeBase<TBase, typename TPar
sedOptions::MsgIdType>;
};

template <typename TBase, typename TParsedOptions>
struct MessageInterfaceProcessMsgId<TBase, TParsedOptions, false
>
{
    using Type = TBase;
};
} // namespace comms
```

Let's assume that the interface options were parsed and typedef-ed into some `ParsedOptions` type:

```
using ParsedOptions = comms::MessageInterfaceParsedOptions<TOpti
ons...>;
```

Then after the following definition statement

```
using NewBaseClass =
    comms::MessageInterfaceProcessMsgId<
        OldBaseClass,
        ParsedOptions,
        ParsedOptions::HasMsgIdType
    >::Type;
```

the `NewBaseClass` is the same as `OldBaseClass`, if the value of `ParsedOptions::HasMsgIdType` is `false` (type of message ID wasn't provided via options), otherwise `NewBaseClass` becomes `comms::MessageInterfaceIdTypeBase`, which inherits from `OldBaseClass`.

Using the same pattern the other helper wrapping classes must be implemented also.

Choose right chunk for endian:

```
namespace comms
{
template <typename TBase, bool THasLittleEndian>
struct MessageInterfaceProcessEndian;

template <typename TBase>
struct MessageInterfaceProcessEndian<TBase, true>
{
    using Type = MessageInterfaceLittleEndian<TBase>;
};

template <typename TBase>
struct MessageInterfaceProcessEndian<TBase, false>
{
    using Type = MessageInterfaceBigEndian<TBase>;
};
} // namespace comms
```

Add read functionality if required:

```
namespace comms
{
template <typename TBase, typename TParsedOptions, bool THasRead
Iterator>
struct MessageInterfaceProcessReadIterator;

template <typename TBase, typename TParsedOptions>
struct MessageInterfaceProcessReadIterator<TBase, TParsedOptions
, true>
{
    using Type = MessageInterfaceReadBase<TBase, typename TParse
dOptions::ReadIterator>;
};

template <typename TBase, typename TParsedOptions>
struct MessageInterfaceProcessReadIterator<TBase, TParsedOptions
, false>
{
    using Type = TBase;
};
} // namespace comms
```

And so on...

The interface building code just uses the helper classes in a sequence of type definitions:

```
namespace comms
{
class EmptyBase {};

template <typename... TOptions>
struct MessageInterfaceBuilder
{
    // Parse the options
    using ParsedOptions = MessageInterfaceParsedOptions<TOptions
...>;

    // Add ID retrieval functionality if ID type was provided
    using Base1 = typename MessageInterfaceProcessMsgId<
            EmptyBase, ParsedOptions, ParsedOptions::HasMsgIdTyp
e>::Type;

    // Add readData() and writeData(), that use the right endian
    using Base2 = typename MessageInterfaceProcessEndian<
            Base1, ParsedOptions::HasLittleEndian>::Type;


    // Add read functionality if ReadIterator type was provided
    using Base3 = typename MessageInterfaceProcessReadIterator<
            Base2, ParsedOptions, ParsedOptions::HasReadIterator
>::Type;

    // Add write functionality if WriteIterator type was provided

    using Base4 = typename MessageInterfaceProcessWriteIterator<
            Base3, ParsedOptions, ParsedOptions::HasWriteIterato
r>::Type;

    // And so on...
    ...
    using BaseN = ...;

    // The last BaseN must be taken as final type.
    using Type = BaseN;
};
} // namespace comms
```

Once all the required definitions are in place, the common dynamic message interface class `comms::Message` may be defined as:

```
namespace comms
{
template <typename... TOptions>
class Message : public typename MessageInterfaceBuilder<TOptions
...>::Type
{
};
} // namespace comms
```

As the result, any distinct set of options provided as the template parameters to `comms::Message` class will cause it to have the required types and member functions.

Now, when the interface is in place, it is time to think about providing common `comms::MessageBase` class which is responsible to provide default implementation for functions, such as `readImpl()`, `writeImpl()`, `dispatchImpl()`, etc...

Generalising Message Implementation

Previous chapters described `MessageBase` class, which provided implementation for some portions of polymorphic behaviour defined in the the interface class `Message`. Such implementation eliminated common boilerplate code used in every `ActualMessage*` class.

This chapter is going to generalise the implementation of `MessageBase` into the generic `comms::MessageBase` class, which is communication protocol independent and can be re-used in any other development.

The generic `comms::MessageBase` class must be able to:

- provide the ID of the message, i.e. implement the `idImpl()` virtual member function, when such ID is known at compile time.
- provide common dispatch functionality, i.e. implement `dispatchImpl()` virtual member function, described in Message / Dispatching and Handling chapter.
- support extension of the default message interface, described in Message / Extending Interface chapter.
- automate common operations on fields, i.e. implement `readImpl()`, `writeImpl()`, `lengthImpl()`, etc..., described in Fields / Automating Basic Operations chapter.

Just like common `comms::Message` interface class, the `comms::MessageBase` will also receive options to define its behaviour.

```
namespace comms
{
template <typename TBase, typename... TOptions>
class MessageBase : public TBase
{
    ...
};
} // namespace comms
```

Note, that the `comms::MessageBase` class receives its base class as a template parameter. It is expected to be any variant of `comms::Message` or any extended interface class, which inherits from `comms::Message` .

The supported options may include:

```
namespace comms
{
namespace option
{
// Provide static numeric ID, to facilitate implementation of id
Impl()
template <std::intmax_t TId>
struct StaticNumIdImpl {};

// Facilitate implementation of dispatchImpl()
template <typename TActual>
struct DispatchImpl {};

// Provide fields of the message, facilitate implementation of
// readImpl(), writeImpl(), lengthImpl(), validImpl(), etc...
template <typename TFields>
struct FieldsImpl {};

} // namespace option
} // namespace comms
```

Parsing the Options

The options provided to the `comm::MessageBase` class need to be parsed in a very similar way as it was with `comms::Message` in the previous chapter.

Starting with initial version of the options struct:

```
namespace comms
{
template <typename... TOptions>
class MessageImplParsedOptions;

template <>
struct MessageImplParsedOptions<>
{
    static const bool HasStaticNumIdImpl = false;
    static const bool HasDispatchImpl = false;
    static const bool HasFieldsImpl = false;
}
} // namespace comms
```

and replacing the initial value of the appropriate variable with new ones, when appropriate option is discovered:

```
namespace comms
{
template <std::intmax_t TId, typename... TOptions>
struct MessageImplParsedOptions<option::StaticNumIdImpl<TId>, TO
ptions...> :
        public MessageImplParsedOptions<TOptions...>
{
    static const bool HasStaticNumIdImpl = true;
    static const std::intmax_t MsgId = TID;
};

template <typename TActual, typename... TOptions>
struct MessageImplParsedOptions<option::DispatchImpl<TActual>, T
Options...> :
        public MessageImplParsedOptions<TOptions...>
{
    static const bool HasDispatchImpl = true;
    using ActualMessage = TActual;
};

template <typename TFields, typename... TOptions>
struct MessageImplParsedOptions<option::FieldsImpl<TFields>, TOp
tions...> :
        public MessageImplParsedOptions<TOptions...>
{
    static const bool HasFieldsImpl = true;
    using Fields = TFields;
};
} // namespace comms
```

Assemble the Required Implementation

Just like with building custom message interface, there is a need to create chunks of implementation parts and connect them using inheritance based on used options.

```
namespace comms
{
// ID information chunk
template <typename TBase, std::intmax_t TId>
class MessageImplStaticNumIdBase : public TBase
{
public:
    // Reuse the message ID type defined in the interface
    using MsgIdType = typename Base::MsgIdType;

protected:
    virtual MsgIdType getIdImpl() const override
    {
        return static_cast<MsgIdType>(TId);
    }
};

// Dispatch implementation chunk
template <typename TBase, typename TActual>
class MessageImplDispatchBase : public TBase
{
public:
    // Reuse the Handler type defined in the interface class
    using Handler = typename Base::Handler;

protected:
    virtual void dispatchImpl(Handler& handler) const override
    {
        handler.handle(static_cast<TActual&>(*this));
    }
};
} // namespace comms
```

NOTE, that single option `comms::option::FieldsImpl<>` may facilitate implementation of multiple functions: `readImpl()`, `writeImpl()`, `lengthImpl()`, etc... Every such function was declared due to using a separate option when defining the interface. We'll have to cherry-pick appropriate implementation parts, based on the interface options. As the result, these implementation chunks must be split into separate classes.

```
namespace comms
{
template <typename TBase, typename TFields>
class MessageImplFieldsBase : public TBase
{
public:
    using AllFields = TFields;

    AllFields& fields() { return m_fields; }
    const AllFields& fields() const { return m_fields; }
private:
    TFields m_fields;
};

template <typename TBase>
class NessageImplFieldsReadBase : public TBase
{
public:
    // Reuse ReadIterator definition from interface class
    using ReadIterator = typename TBase::ReadIterator;
protected:
    virtual ErrorStatus readImpl(ReadIterator& iter, std::size_t
 len) override
    {
        // Access fields via interface provided in previous chunk

        auto& allFields = TBase::fields();
        ... // read all the fields
    }
};

... // and so on
} // namespace comms
```

All these implementation chunks are connected together using extra helper classes in a very similar way to how the interface chunks where connected:

Add `idImpl()` if needed

```
namespace comms
{
template <typename TBase, typename ParsedImplOptions, bool TImpl
ement>
struct MessageImplProcessStaticNumId;

template <typename TBase, typename ParsedImplOptions>
struct MessageImplProcessStaticNumId<TBase, ParsedImplOptions, t
rue>
{
    using Type = MessageImplStaticNumIdBase<TBase, ParsedImplOpt
ions::MsgId>;
};

template <typename TBase, typename ParsedImplOptions>
struct MessageInterfaceProcessEndian<TBase, false>
{
    using Type = TBase;
};
} // namespace comms
```

Add `dispatchImpl()` if needed

```cpp
namespace comms
{
template <typename TBase, typename ParsedImplOptions, bool TImplement>
struct MessageImplProcessDispatch;

template <typename TBase, typename ParsedImplOptions>
struct MessageImplProcessDispatch<TBase, ParsedImplOptions, true>
{
    using Type = MessageImplDispatchBase<TBase, typename ParsedImplOptions::ActualMessage>;
};

template <typename TBase, typename ParsedImplOptions>
struct MessageImplProcessDispatch<TBase, false>
{
    using Type = TBase;
};
} // namespace comms
```

Add `fields()` access if needed

```
namespace comms
{
template <typename TBase, typename ParsedImplOptions, bool TImpl
ement>
struct MessageImplProcessFields;

template <typename TBase, typename ParsedImplOptions>
struct MessageImplProcessFields<TBase, ParsedImplOptions, true>
{
    using Type = MessageImplFieldsBase<TBase, typename ParsedImp
lOptions::Fields>;
};

template <typename TBase, typename ParsedImplOptions>
struct MessageImplProcessFields<TBase, false>
{
    using Type = TBase;
};
} // namespace comms
```

Add `readImpl()` if needed

```
namespace comms
{
template <typename TBase, bool TImplement>
struct MessageImplProcessReadFields;

template <typename TBase>
struct MessageImplProcessReadFields<TBase, true>
{
    using Type = NessageImplFieldsReadBase<TBase>;
};

template <typename TBase>
struct MessageImplProcessReadFields<TBase, false>
{
    using Type = TBase;
};

} // namespace comms
```

And so on for all the required implementation chunks: `writeImpl()` , `lengthImpl()` , `validImpl()` , etc...

The final stage is to connect all the implementation chunks together via inheritance and derive `comms::MessageBase` class from the result.

NOTE, that existence of the implementation chunk depends not only on the implementation options provided to `comms::MessageBase` , but also on the interface options provided to `comms::Message` . For example, `writeImpl()` must be added only if `comms::Message` interface includes `write()` member function (`comms::option::WriteIterator<>` option was used) and implementation option which adds support for fields (`comms::option::FieldsImpl<>`) was passed to `comms::MessageBase` .

The implementation builder helper class looks as following:

```
namespace comms
{
// TBase is interface class
// TOptions... are the implementation options
```

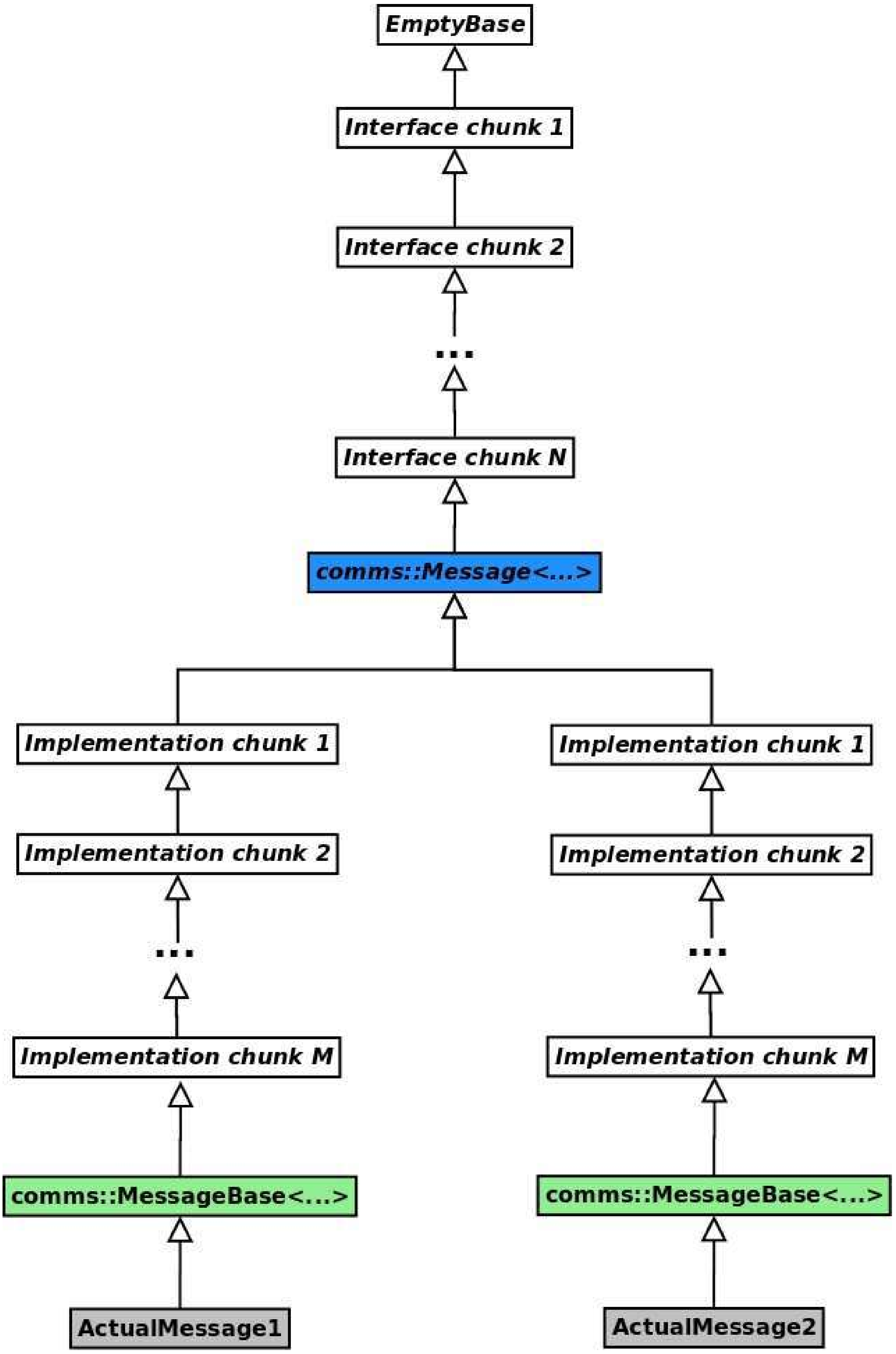
```
template <typename TBase, typename... TOptions>
struct MessageImplBuilder
{
    // ParsedOptions class is supposed to be defined in comms::M
essage class
    using InterfaceOptions = typename TBase::ParsedOptions;

    // Parse implementation options
    using ImplOptions = MessageImplParsedOptions<TOptions...>;

    // Provide idImpl() if possible
    static const bool HasStaticNumIdImpl =
        InterfaceOptions::HasMsgIdType && ImplOptions::HasStatic
NumIdImpl;
    using Base1 = typename MessageImplProcessStaticNumId<
            TBase, ImplOptions, HasStaticNumIdImpl>::Type;

    // Provide dispatchImpl() if possible
    static const bool HasDispatchImpl =
        InterfaceOptions::HasHandler && ImplOptions::HasDispatch
Impl;
    using Base2 = typename MessageImplProcessDispatch<
            Base1, ImplOptions, HasDispatchImpl>::Type;

    // Provide access to fields if possible
    using Base3 = typename MessageImplProcessFields<
            Base2, ImplOptions, ImplOptions::HasFieldsImpl>::Typ
e;

    // Provide readImpl() if possible
    static const bool HasReadImpl =
        InterfaceOptions::HasReadIterator && ImplOptions::HasFie
ldsImpl;
    using Base4 = typename MessageImplProcessReadFields<
            Base3, HasReadImpl>::Type;

    // And so on...
    ...
    using BaseN = ...;
    // The last BaseN must be taken as final type.
    using Type = BaseN;
};
} // namespace comms
```

Defining the generic `comms::MessageBase` :

```
namespace comms
{
template <typename TBase, typename... TOptions>
class MessageBase : public typename MessageImplBuilder<TBase, TO
ptions>::Type
{
    ...
};
} // namespace comms
```

Please note, that `TBase` template parameter is passed to `MessageImplBuilder<>` , which in turn passes it up the chain of possible implementation chunks, and at the end it turns up to be the base class of the whole hierarchy.

The full hierarchy of classes presented at the image below.



The total number of used classes may seem scary, but there are only two, which are of any particular interest to us when implementing communication protocol. It's `comms::Message` to specify the interface and `comms::MessageBase` to provide default implementation of particular functions. All the rest are just implementation details.

Summary

After all this work our library contains generic `comms::Message` class, that defines the interface, as well as generic `comms::MessageBase` class, that provides default implementation for required polymorphic functionality.

Let's define a custom communication protocol which uses little endian for data serialisation and has numeric message ID type defined with the enumeration below:

```
enum MyMsgId
{
    MyMsgId_Msg1,
    MyMsgId_Msg2,
    ...
};
```

Assuming we have relevant field classes in place (see Fields chapter), let's define custom `ActualMessage1` that contains two integer value fields: 2 bytes unsigned value and 1 byte signed value.

```
using ActualMessage1Fields = std::tuple<
    IntValueField<std::uint16_t>,
    IntValueField<std::int8_t>
>;
template <typename TMessageInterface>
class ActualMessage1 : public
    comms::MessageBase<
        comms::option::StaticNumIdImpl<MyMsgId_Msg1>, // provide
 idImpl() if needed
        comms::option::DispatchImpl<ActualMessage1>, // provide
dispatchImpl() if needed
        comms::option::FieldsImpl<ActualMessage1Fields> // provi
de access to fields and
                                                        // readI
mpl(), writeImpl(),
                                                        // lengt
hImpl(), validImpl()
                                                        // funct
ions if needed
    >
{
};
```

That's it, no extra member functions are needed to be implemented, unless the message interface class is extended one. Note, that the implementation of the `ActualMessage1` is completely generic and doesn't depend on the actual message interface. It can be reused in any application with any runtime environment that uses our custom protocol.

The interface class is defined according to the requirements of the application, that uses the implementation of the defined protocol.

```
class MyHandler; // forward declaration of the handler class.
using MyMessage = comms::Message<
    comms::option::MsgIdType<MyMsgId>, // add id() operation
    comms::option::ReadIterator<const std::uint8_t*>, // add rea
d() operation
    comms::option::WriteIterator<std::uint8_t*> // add write() o
peration
    comms::option::Handler<MyHandler>, // add dispatch() operati
on
    comms::option::LengthInfoInterface, // add length() operation

    comms::option::ValidCheckInterface, // add valid() operation
    comms::option::LittleEndian // use little endian for seriali
sation
>;
```

For convenience the protocol messages should be redefined with appropriate interface:

```
using MyActualMessage1 = ActualMessage1<MyMessage>;
using MyActualMessage2 = ActualMessage2<MyMessage>;
...
```

Generalising Fields Implementation

The automation of read/write operations of the message required fields to expose predefined minimal interface:

```
class SomeField
{
public:
    // Value storage type definition
    using ValueType = ...;

    // Provide an access to the stored value
    ValueType& value();
    const ValueType& value() const;

    // Read (deserialise) and update internal value
    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len);

    // Write (serialise) internal value
    template <typename TIter>
    ErrorStatus write(TIter& iter, std::size_t len) const;

    // Get the serialisation length
    std::size_t length() const;

private:
    ValueType m_value;
}
```

The read/write operations will probably require knowledge about the serialisation endian used for the protocol. We need to come up with the way to convey the endian information to the field classes. I would recommend doing it by having common base class for all the fields:

```
namespace comms
{
template <bool THasLittleEndian>
class Field
{
protected:
    // Read value using appropriate endian
    template <typename T, typename TIter>
    static T readData(TIter& iter) {...}

    // Read partial value using appropriate endian
    template <typename T, std::size_t TSize, typename TIter>
    static T readData(TIter& iter) {...}

    // Write value using appropriate endian
    template <typename T, typename TIter>
    static void writeData(T value, TIter& iter) {...}

    // Write partial value using appropriate endian
    template <std::size_t TSize, typename T, typename TIter>
    static void writeData(T value, TIter& iter)
}
} // namespace comms
```

The choice of the right endian may be implemented using Tag Dispatch Idiom.

```
namespace comms
{
template <bool THasLittleEndian>
class Field
{
protected:
    // Read value using appropriate endian
    template <typename T, typename TIter>
    static T readData(TIter& iter)
    {
        // Dispatch to appropriate read function
        return readDataInternal<T>(iter, Tag());
    }
    ...
private:
    BigEndianTag {};
    LittleEndianTag {};

    using Tag = typename std::conditional<
        THasLittleEndian,
        LittleEndianTag,
        BigEndianTag
    >::type;

    // Read value using big endian
    template <typename T, typename TIter>
    static T readBig(TIter& iter) {...}

    // Read value using little endian
    template <typename T, typename TIter>
    static T readLittle(TIter& iter) {...}

    // Dispatch to readBig()
    template <typename T, typename TIter>
    static T readDataInternal(TIter& iter, BigEndianTag)
    {
        return readBig<T>(iter);
    }

    // Dispatch to readLittle()
    template <typename T, typename TIter>
    static T readDataInternal(TIter& iter, LittleEndianTag)
    {
        return readLittle<T>(iter);
    }
};
} // namespace comms
```

Every field class should receive its base class as a template parameter and may use available `readData()` and `writeData()` static member functions when serialising/deserialising internal value in `read()` and `write()` member functions.

For example:

```
namespace comms
{
template <typename TBase, typename TValueType>
class IntValueField : public TBase
{
    using Base = TBase;
public:
    using ValueType = TValueType;
    ...
    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len)
    {
        if (len < length()) {
            return ErrorStatus::NotEnoughData;
        }

        m_value = Base::template read<ValueType>(iter);
        return ErrorStatus::Success;
    }

    template <typename TIter>
    ErrorStatus write(TIter& iter, std::size_t len) const
    {
        if (len < length()) {
            return ErrorStatus::BufferOverflow;
        }

        Base::write(m_value, iter);
        return ErrorStatus::Success
    }

    static constexpr std::size_t length()
    {
        return sizeof(ValueType);
    }
private:
    ValueType m_value
};
} // namespace comms
```

When the endian is known and fixed (for example when implementing third party protocol according to provided specifications), and there is little chance it's ever going to change, the base class for all the fields may be explicitly defined:

```
using MyProjField = comms::Field<false>; // Use big endian for f
ields serialisation
using MyIntField = comms::IntValueField<MyProjField>;
```

However, there may be the case when the endian information is not known up front, and the one provided to the message interface definition (`comms::Message`) must be used. In this case, the message interface class may define common base class for all the fields:

```
namespace comms
{
template <typename... TOptions>
class Message : public typename MessageInterfaceBuilder<TOptions
...>::Type
{
    using Base = typename MessageInterfaceBuilder<TOptions...>::
Type;
pablic:
    using ParsedOptions = typename Base::ParsedOptions ;
    using Field = comms::Field<ParsedOptions::HasLittleEndian>;
    ...
};
} // namespace comms
```

As the result the definition of the message's fields must receive a template parameter of the base class for all the fields:

```
template <typename TFieldBase>
using ActualMessage1Fields = std::tuple<
    comms::IntValueField<TFieldBase>,
    comms::IntValueField<TFieldBase>,
    ...
>:

template <typename TMsgInterface>
class ActualMessage1 : public
    comms::MessageBase<
        comms::option::FieldsImpl<ActualMessage1Fields<typename
TMsgInterface::Field> >,
        ...
    >
{
};
```

Multiple Ways to Serialise Fields

The Common Field Types chapter described most common types of fields with various serialisation and handling nuances, which can be used to implement a custom communication protocol.

Let's take the basic integer value field as an example. The most common way to serialise it is just read/write its internally stored value as is. However, there may be cases when serialisation takes limited number of bytes. Let's say, the protocol specification states that some integer value consumes only 3 bytes in the serialised bytes sequence. In this case the value will probably be be stored using `std::int32_t` or `std::uint32_t` type. The field class will also require different implementation of read/write/length functionality.

Another possible case is a necessity to add/subtract some predefined offset to/from the value being serialised and subtracting/adding the same offset when the value is deserialised. Good example of such case would be the serialisation of a **year** information, which is serialised as an offset from year 2000 and consumes only 1 byte. It is possible to store the value as a single byte (`comms::IntValueField<..., std::uint8_t>`), but it would be very inconvenient. It is much better if we could store a normal year value (`2015` , `2016` , etc ...) using `std::uint16_t` type, but when serialising, the values that get written are `15` , `16` , etc... **NOTE**, that such field requires two steps in its serialisation logic:

- add required offset (`-2000` in the example above)
- limit the number of bytes when serialising the result

Another popular way to serialise integer value is to use Base-128 encoding. In this case the number of bytes in the serialisation sequence is not fixed.

What if some protocol decides to serialise the same offset from year 2000, but using the **Base-128** encoding? It becomes obvious that having a separate field class for every possible variant is impractical at least. There must be a way to split the serialisation logic into small chunks, which can be applied one on top of another.

Using the same idea of the *options* and adapting the behaviour of the field class accordingly, we can generalise all the fields into a small subset of classes and make them also part of our generic library.

The options described earlier may be defined using following option classes:

```
namespace comms
{
namespace option
{
// Provide fixed serialisation length
template<std::size_t TLen>
struct FixedLength {};

// Provide numeric offset to be added to the value before serial
isation
template<std::intmax_t TOffset>
struct NumValueSerOffset {};

// Force using variable length (base-128 encoding) while providi
ng
// minimal and maximal allowed serialisation lengths.
template<std::size_t TMin, std::size_t TMax>
struct VarLength {};

} // namespace option
} // namespace comms
```

Parsing the Options

In a very similar way to parsing options of the message interface (`comms::Message`) and message implementation (`comms::MessageBase`) described in earlier chapters, we will create a struct, that will contain all the provided information to be used later.

```
namespace comms
{
template <typename... TOptsion>
struct FieldParsedOptions;

template <>
struct FieldParsedOptions<>
{
    static const bool HasSerOffset = false;
    static const bool HasFixedLengthLimit = false;
    static const bool HasVarLengthLimits = false;
}

template <std::size_t TLen, typename... TOptsion>
struct FieldParsedOptions<option::FixedLength<TLen>, TOptions...
> :
    public FieldParsedOptions<TOptions...>
{
    static const bool HasFixedLengthLimit = true;
    static const std::size_t FixedLengthLimit = TLen;
};

template <std::intmax_t TOffset, typename... TOptions>
struct FieldParsedOptions<option::NumValueSerOffset<TOffset>, TO
ptions...> :
    public FieldParsedOptions<TOptions...>
{
    static const bool HasSerOffset = true;
    static const auto SerOffset = TOffset;
};

template <std::size_t TMinLen, std::size_t TMaxLen, typename...
TOptions>
struct FieldParsedOptions<VarLength<TMinLen, TMaxLen>, TOptions.
..> :
    public FieldParsedOptions<TOptions...>
{
    static const bool HasVarLengthLimits = true;
    static const std::size_t MinVarLength = TMinLen;
    static const std::size_t MaxVarLength = TMaxLen;
};
} // namespace comms
```

Assemble the Required Functionality

Before parsing the options and assembling the right functionality there is a need to start with basic integer value functionality:

```
namespace comms
{
template <typename TFieldBase, typename TValueType>
class BasicIntValue : public TFieldBase
{
public:
    using ValueType = TValueType;

    ... // rest of the interface
private:
    ValueType m_value;
};
} // namespace comms
```

Such field receives its base class and the type of the value it stores. The implementation of read/write/length functionalities are very basic and straightforward.

Now, we need to prepare various adaptor classes that will wrap or replace the existing interface functions:

```
namespace comms
{
template <std::intmax_t TOffset, typename TNext>
class SerOffsetAdaptor
{
public:
    ... // public interface
private:
    TNext m_next;
};

template <std::size_t TLen, typename TNext>
class FixedLengthAdaptor
{
public:
    ... // public interface
private:
    TNext m_next;
};

... // and so on
} // namespace comms
```

NOTE, that the adaptor classes above wrap one another (`TNext` template parameter) and either replace or forward the read/write/length operations to the next adaptor or final `BasicIntValue` class, instead of using inheritance as it was with message interface and implementation chunks. The overall architecture presented in this book doesn't require the field classes to exhibit polymorphic behaviour. That's why using inheritance between adaptors is not necessary, although not forbidden either. Using inheritance instead of containment has its pros and cons, and at the end it's a matter of personal taste of what to use.

Now it's time to use the parsed options and wrap the `BasicIntValue` with required adaptors:

Wrap with `SerOffsetAdaptor` if needed

```
namespace comms
{
template <typename TField, typename TOpts, bool THasSerOffset>
struct AdaptBasicFieldSerOffset;

template <typename TField, typename TOpts>
struct AdaptBasicFieldSerOffset<TField, TOpts, true>
{
    using Type = SerOffsetAdaptor<TOpts::SerOffset, TField>;
};

template <typename TField, typename TOpts>
struct AdaptBasicFieldSerOffset<TField, TOpts, false>
{
    using Type = TField;
};
} // namespace comms
```

Wrap with `FixedLengthAdaptor` if needed

```
namespace comms
{
template <typename TField, typename TOpts, bool THasFixedLength>
struct AdaptBasicFieldFixedLength;

template <typename TField, typename TOpts>
struct AdaptBasicFieldFixedLength<TField, TOpts, true>
{
    using Type = FixedLengthAdaptor<TOpts::FixedLength, TField>;
};

template <typename TField, typename TOpts>
struct AdaptBasicFieldFixedLength<TField, TOpts, false>
{
    using Type = TField;
};
} // namespace comms
```

And so on for all other possible adaptors.

Now, let's bundle all the required adaptors together:

```
namespace comms
{
template <typename TBasic, typename... TOptions>
sturct FieldBuilder
{
    using  ParsedOptions = FieldParsedOptions<TOptions...>;

    using Field1 = typename AdaptBasicFieldSerOffset<
        TBasic, ParsedOptions, ParsedOptions::HasSerOffset>::Typ
e;

    using Field2 = typename AdaptBasicFieldFixedLength<
        Field1, ParsedOptions, ParsedOptions::HasFixedLengthLimi
t>::Type;

    using Field3 = ...
    ...
    using FieldN = ...
    using Type = FieldN;
};
} // namespace comms
```

The final stage is to actually define final `IntValueField` type:

```
namespace comms
{
template <typename TBase, typename TValueType, typename... TOpti
ons>
class IntValueField
{
    using Basic = BasicIntValue<TBase, TValueType>;
    using Adapted = typename FieldBuilder<Basic, TOptions...>::T
ype;
public:
    using ValueType = typename Adapted::ValueType;

    // Just forward all the API requests to the adapted field.
    ValueType& value()
    {
        return m_adapted.value();
    }

    const ValueType& value() const
    {
        return m_adapted.value();
    }

    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len)
    {
        return m_adapted.read(iter, len);
    }

    ...
private:
    Adapted m_adapted;
};
} // namespace comms
```

The definition of the **year** field which is serialised using offset from year `2000` may be defined as:

```
using MyFieldBase = comms::Field<false>; // use big endian
using MyYear = comms::IntValueField<
    MyFieldBase,
    std::uint16_t, // store as 2 bytes unsigned value
    comms::option::NumValueSerOffset<-2000>,
    comms::option::FixedLength<1>
>;
```

Other Options

In addition to options that regulate the read/write behaviour, there can be options which influence how the field is created and/or handled afterwards.

For example, there may be a need to set a specific value when the field object is created (using default constructor). Let's introduce a new options for this purpose:

```
namespace comms
{
namespace option
{
template <typename T>
struct DefaultValueInitialiser{};
} // namespace option
} // namespace comms
```

The template parameter provided to this option is expected to be a class/struct with the following interface:

```
struct DefaultValueSetter
{
    template <typename TField>
    void operator()(TField& field) const
    {
        field.value() = ...; // Set the custom value
    }
}
```

Then the relevant adaptor class may set the default value of the field using the provided setter class:

```
namespace comms
{
template <typename TSetter, typename TNext>
class DefaultValueInitAdaptor
{
public:
    using ValueType = typename TNext::ValueType;

    DefaultValueInitAdaptor()
    {
        TSetter()(*this);
    }

    ValueType& value()
    {
        return m_next.value();
    }

    ...

private:
    TNext m_next;
};
} // namespace comms
```

Please note, that both `comms::option::DefaultValueInitialiser` option and `DefaultValueInitAdaptor` adaptor class are completely generic, and they can be used with any type of the field.

For numeric fields, such as `IntValueField` defined earlier, the generic library may provide built-in setter class:

```
namespace comms
{
template<std::intmax_t TVal>
struct DefaultNumValueInitialiser
{
    template <typename TField>
    void operator()(TField& field)
    {
        using FieldType = typename std::decay<TField>::type;
        using ValueType = typename FieldType::ValueType;
        field.value() = static_cast<ValueType>(TVal);
    }
};
} // namespace comms
```

And then, create a convenience alias to `DefaultValueInitialiser` option which receives a numeric value as its template parameter and insures that the field's value is initialised accordingly:

```
namespace comms
{
namespace option
{
template<std::intmax_t TVal>
using DefaultNumValue = DefaultValueInitialiser<details::Default
NumValueInitialiser<TVal> >;
} // namespace option
} // namespace comms
```

As the result, the making the **year** field to be default constructed with value `2016` may look like this:

```
using MyFieldBase = comms::Field<false>; // use big endian
using MyYear = comms::IntValueField<
    MyFieldBase,
    std::uint16_t, // store as 2 bytes unsigned value
    comms::option::NumValueSerOffset<-2000>,
    comms::option::FixedLength<1>,
    comms::option::DefaultNumValue<2016>
>;
```

Other Fields

The Common Field Types chapter mentions multiple other fields and several different ways to serialise them. I'm not going to describe each and every one of them here. Instead, I'd recommend taking a look at the documentation of the COMMS library which was implemented using ideas from this book. It will describe all the fields it implements and their options.

Eliminating Dynamic Memory Allocation

Fields like **String** or **List** may contain variable number of characters/elements. The default internal value storage type for such fields will probably be `std::string` or `std::vector` respectively. It will do the job, mostly. However, they may not be suitable for bare-metal products that cannot use dynamic memory allocation and/or exceptions. In this case there must be a way to easily substitute these types with alternatives, such as custom `StaticString` or `StaticVector` types.

Let's define a new option that will provide fixed storage size and will force usage of these custom types instead of `std::string` and `std::vector`.

```
namespace comms
{
namespace option
{
template <std::size_t TSize>
struct FixedSizeStorage {};
} // namespace option
} // namespace comms
```

The parsed option structure needs to be extended with new information:

```
namespace comms
{
template <typename... TOptsion>
struct FieldParsedOptions;

template <>
struct FieldParsedOptions<>
{
    ...
    static const bool HasFixedSizeStorage = false;
}

template <std::size_t TSize, typename... TOptsion>
struct FieldParsedOptions<option::FixedSizeStorage<TSize>, TOpti
ons...> :
    public FieldParsedOptions<TOptions...>
{
    static const bool HasFixedSizeStorage = true;
    static const std::size_t FixedSizeStorage = TSize;
};

} // namespace comms
```

Now, let's implement the logic of choosing `StaticString` as the value storage type if the option above is used and choosing `std::string` if not.

```
// TOptions is a final variant of FieldParsedOptions<...>
template <typename TOptions, bool THasFixedStorage>
struct StringStorageType;

template <typename TOptions>
struct StringStorageType<TOptions, true>
{
    typedef comms::util::StaticString<TOptions::FixedSizeStorage
> Type;
};

template <typename TOptions>
struct StringStorageType<TOptions, false>
{
    typedef std::string Type;
};

template <typename TOptions>
using StringStorageTypeT =
    typename StringStorageType<TOptions, TOptions::HasFixedSizeS
torage>::Type;
```

`comms::util::StaticString` is the implementation of a string management class, which exposes the same public interface as `std::string`. It receives the fixed size of the storage area as a template parameter, uses `std::array` or similar as its private data member the store the string characters.

The implementation of the **String** field may look like this:

```
template <typename TBase, typename... TOptions>
class StringField
{
public:
    // Parse the option into the struct
    using ParsedOptions = FieldParsedOptions<TOptions...>;

    // Identify storage type: StaticString or std::string
    using ValueType = StringStorageTypeT<ParsedOptions>;

    // Use the basic field and wrap it with adapters just like I
ntValueField earlier
    using Basic = BasicStringValue<TBase, ValueType>;
    using Adapted = typename FieldBuilder<Basic, TOptions...>::T
ype;

    // Just forward all the API requests to the adapted field.
    ValueType& value()
    {
        return m_adapted.value();
    }

    const ValueType& value() const
    {
        return m_adapted.value();
    }

    template <typename TIter>
    ErrorStatus read(TIter& iter, std::size_t len)
    {
        return m_adapted.read(iter, len);
    }

    ...
private:
    Adapted m_adapted;
};
} // namespace comms
```

As the result the definition of the message with a string field that doesn't use dynamic memory allocation may look like this:

```
template <typename TFieldBase>
using ActualMessage3Fields = std::tuple<
    comms::StringField<TFieldBase, comms::option::FixedStorageSi
ze<128> >,
    ...
>:

template <typename TMsgInterface>
class ActualMessage3 : public
    comms::MessageBase<
        comms::option::FieldsImpl<ActualMessage3Fields<typename
TMsgInterface::Field> >,
        ...
    >
{
};
```

And what about the case, when there is a need to create a message with a string field, but substitute the underlying default `std::string` type with `StaticString` **only** when compiling the bare-metal application? In this case the `ActualMessage3` class may be defined to have additional template parameter which will determine the necessity to substitute the storage type.

```
template <bool THasFixedSize>
struct StringExtraOptions
{
    using Type = comms::option::EmtpyOption; // doesn't do anything
};

template <>
struct StringExtraOptions<false>
{
    using Type = comms::option::FixedStorageSize<128> >; // forces static storage
};

template <typename TFieldBase, bool THasFixedSize>
using ActualMessage3Fields = std::tuple<
    comms::StringField<TFieldBase, typename StringExtraOptions<THasFixedSize>::Type>,
    ...
>:

template <typename TMsgInterface, bool THasFixedSize = false>
class ActualMessage3 : public
    comms::MessageBase<
        comms::option::FieldsImpl<ActualMessage3Fields<typename TMsgInterface::Field, THasFixedSize> >,
        ...
    >
{
};
```

Thanks to the fact that `StaticString` and `std::string` classes expose the same public interface, the message handling function doesn't need to worry about actual storage type. It just uses public interface of `std::string` :

```
class MsgHandler
{
public:
    void handle(ActualMessage3& msg)
    {
        auto& fields = msg.fields();
        auto& stringField = std::get<0>(fields);

        // The type of the stringVal is either std::string or St
aticString
        auto& stringVal = stringField.value();
        if (stringVal == "string1") {
            ... // do something
        }
        else if (stringVal == "string2") {
            ... // do something else
        }
    }
};
```

Choosing internal value storage type for **List** fields to be `std::vector` or `StaticVector` is very similar.

Transport

In addition to definition of the messages and their contents, every communication protocol must ensure that the message is successfully delivered over the I/O link to the other side. The serialised message payload must be wrapped in some kind of transport information, which usually depends on the type and reliability of the I/O link being used. For example, protocols that are designed to be used over TCP/IP connection, such as MQTT, may omit the whole packet synchronisation and checksum logic, because TCP/IP connection ensures that the data is delivered correctly. Such protocols are usually defined to use only message ID and remaining size information to wrap the message payload:

```
ID | SIZE | PAYLOAD
```

Other protocols may be designed to be used over less reliable RS-232 link, which may require a bit better protection against data loss or corruption:

```
SYNC | SIZE | ID | PAYLOAD | CHECKSUM
```

The number of most common types of the wrapping "chunks" is quite small. However, different protocols may have different rules of how these values are serialised. Very similar to Fields.

The main logic of processing the incoming raw data remains the same for all the protocols, though. It is to read and process the transport information "chunks" one by one:

- SYNC - check the next one or more bytes for an expected predefined value. If the value is as expected proceed to the next "chunk". If not, drop one byte from the front of the incoming data queue and try again.
- SIZE - compare the remaining expected data length against actually available. If there is enough data, proceed to the next "chunk". If not report, to the caller, that more data is required.
- ID - read the message ID value and create appropriate message object, then proceed to the next "chunk".

- PAYLOAD - let the created message object to read its payload data.
- CHECKSUM - read the expected checksum value and calculate the actual one. If the checksums don't match, discard the created message and report error.

Each and every "chunk" operates independently, regardless of what information was processed before and/or after it. When some operation seems to repeat itself several times, it should be generalised and become part of our Generic Library.

So, how is it going to be implemented? My advice is to use independent "chunk" classes, that expose predefined interface, wrap one another, and forward the requested operation to the next "chunk" when appropriate. As was stated earlier, the transport information values are very similar to Fields, which immediately takes us to the direction of reusing Field classes to handle these values:

```
template <typename TField, typename TNextChunk, ... /* some othe
r template parameters */>
class SomeChunk
{
public:
    // Iterator used for reading
    using ReadIterator = typename TNextChunk::ReadIterator;

    // Iterator used for writing
    using WriteIterator = typename TNextChunk::WriteIterator;

    // Type of the common message interface class
    using Message = typename TNextChunk::Message;

    // Smart pointer used to hold newly created message object
    using MsgPtr = typename TNextChunk::MsgPtr;

    ErrorStatus read(MsgPtr& msg, ReadIterator& iter, std::size_
t len)
    {
        TField field;
        auto es = field.read(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }

        ... process field value.
        return m_next.read(msg, iter, len - field.length());
    }

    ErrorStatus write(const Message& msg, WriteIterator& iter, s
td::size_t len)
    {
        TField field;
        field.value() = ...; // set required value
        auto es = field.write(iter, len);
        if (es != ErrorStatus) {
            return es;
        }
        return m_next.write(msg, iter, len - field.length());
    }

private:
    TNextChunk m_next;
}
```

Please note that `ReadIterator` and `WriteIterator` are taken from the next chunk. One of the chunks, which is responsible for processing the `PAYLOAD` will receive the class of the message interface as a template parameter, will retrieve the information of the iterators' types, and redefine them as its internal types. Also, this class will define the type of the message interface as its internal `Message` type. All other wrapping chunk classes will reuse the same information.

Also note, that one of the chunks will have to define pointer to the created message object (`MsgPtr`). Usually it is the chunk that is responsible to process `ID` value.

The sequential processing the the transport information "chunks", and stripping them one by one before proceeding to the next one, may remind of OSI Conceptual Model, where a layer serves the layer above it and is served by the layer below it.

From now on, I will use a term **layer** instead of the **chunk**. The combined bundle of such layers will be called **protocol stack** (of layers).

Let's take a closer look at all the layer types mentioned above.

PAYLOAD Layer

Processing of the `PAYLOAD` is always the last stage in the protocol stack. All previous layers have successfully processed their transport data, the message object was created and is ready to read its fields encoded in the PAYLOAD.

Such layer must receive type of the message interface class as a template parameter and redefine read/write iterator types.

```
namespace comms
{
template <typename TMessage>
class MsgDataLayer
{
public:
    // Define type of the message interface
    using Message = TMessage;

    // Type of the pointer to message is not defined yet, will b
e defined in
    // the layer that processes message ID
    using MsgPtr = void;

    // ReadIterator is the same as Message::ReadIterator if such
 exists, void
    // otherwise.
    using ReadIterator = typename std::conditional<
            Message::InterfaceOptions::HasReadIterator,
            typename TMessage::ReadIterator,
            void
        >::type;

    // WriteIterator is the same as Message::WriteIterator if su
ch exists, void
    // otherwise.
    using WriteIterator = typename std::conditional<
            Message::InterfaceOptions::HasWriteIterator,
            typename TMessage::WriteIterator,
            void
        >::type WriteIterator;
    ...
};
} // namespace comms
```

The read/write operations just forward the request the message object.

```
namespace comms
{
template <typename TMessage>
class MsgDataLayer
{
public:

    template <typename TMsgPtr>
    static ErrorStatus read(
        TMsgPtr& msgPtr,
        ReadIterator& iter,
        std::size_t len)
    {
        return msgPtr->read(iter, len);
    }

    static ErrorStatus write(
        const Message& msg,
        WriteIterator& iter,
        std::size_t len)
    {
        return msg.write(iter, len);
    }
};
} // namespace comms
```

Please note that `read()` member function expects to receive a reference to the smart pointer, which holds allocated message object, as the first parameter. The type of the pointer is not known yet. As the result, type of such pointer is provided via template parameter.

ID Layer

The job of this layer is handle the message ID information.

- When new message is received, appropriate message object needs to be created, prior to invoking read operation of the next (wrapped) layer.
- When any message is about to get sent, just get the ID information from the message object and serialise it prior to invoking the write operation of the next layer.

The code of the layer is pretty straightforward:

```
namespace comms
{
// TField is type of the field used to read/write message ID
// TNext is the next layer this one wraps
template <typename TField, typename TNext, ... /* other paramete
rs */>
class MsgIdLayer
{
public:
    // Type of the field object used to read/write message ID va
lue.
    using Field = TField;

    // Take type of the ReadIterator from the next layer
    using ReadIterator = typename TNext::ReadIterator;

    // Take type of the WriteIterator from the next layer
    using WriteIterator = typename TNext::WriteIterator;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Type of the message ID
    using MsgIdType = typename Message::MsgIdType;
    // Redefine pointer to message type (described later)
    using MsgPtr = ...;

    ErrorStatus read(MsgPtr& msgPtr, ReadIterator& iter, std::size_t len)
    {
        Field field;
        auto es = field.read(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        msgPtr = createMsg(field.value()); // create message object based on ID
        if (!msgPtr) {
            // Unknown ID
            return ErrorStatus::InvalidMsgId;
        }
        es = m_next.read(iter, len - field.length());
        if (es != ErrorStatus::Success) {
            msgPtr.reset(); // Discard allocated message;
        }
        return es;
    }

    ErrorStatus write(const Message& msg, WriteIterator& iter, std::size_t len) const
    {
        Field field;
        field.value() = msg.id();
        auto es = field.write(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        return m_next.write(msg, iter, len - field.length());
    }
private:
    MsgPtr createMsg(MsgIdType id)
    {
        ... // TODO: create message object (described later)
    }
    TNext m_next;
};
} // namespace comms
```

To properly finalise the implementation above we need to resolve two main challenges:

- Implement `createMsg()` function which receives ID of the message and creates the message object.
- Define the `MsgPtr` smart pointer type, which is responsible to hold the allocated message object. In most cases defining it to be `std::unique_ptr<Message>` will do the job. However, the main problem here is usage of dynamic memory allocation. Bare metal platform may not have such luxury. There must be a way to support "in place" allocation as well.

Creating Message Object

Let's start with creation of proper message object, given the **numeric** message ID. It must be as efficient as possible.

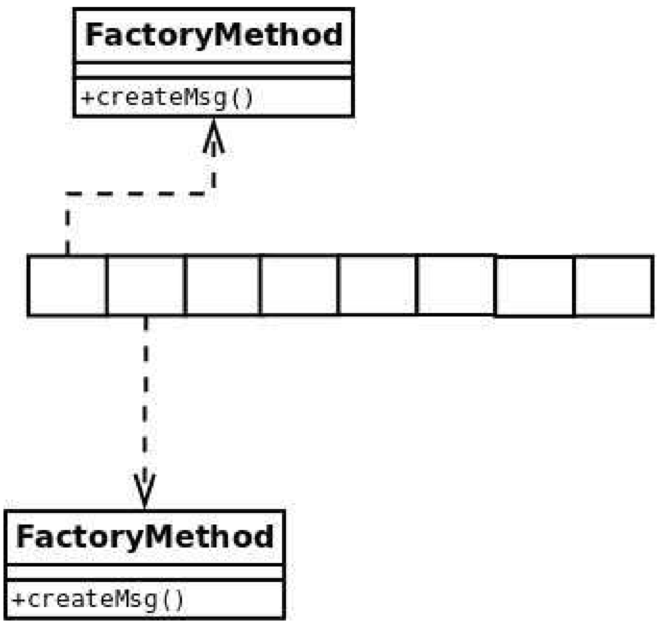
In many cases the IDs of the messages are sequential ones and defined using some enumeration type.

```
enum MsgId
{
    MsgId_Message1,
    MsgId_Message2,
    ...
    MsgId_NumOfMessages
};
```

Let's assume that we have `FactoryMethod` class with polymorphic `createMsg()` function, that returns allocated message object wrapped in a `MsgPtr` smart pointer.

```
class FactoryMethod
{
public:
    MsgPtr createMsg() const
    {
        return createMsgImpl();
    }

protected:
    virtual MsgPtr createMsgImpl() const = 0;
};
```

In this case, the most efficient way is to have an array of pointers to polymorphic class `FactoryMethod`. The index of the array cell corresponds to a message ID.



The code of `MsgIdLayer::createMsg()` function is quite simple:

```
namespace comms
{
template <...>
class MsgIdLayer
{
private:
    MsgPtr createMsg(MsgIdType id)
    {
        auto& registry = ...; // reference to the array of point
ers to FactoryMethod-s
        if ((registry.size() <= id) ||
            (registry[id] == nullptr)){
            return MsgPtr();
        }

        return registry[id]->createMsg();
    }
};
} // namespace comms
```

The runtime complexity of such code is `O(1)`.

However, there are many protocols that their ID map is quite sparse and it is impractical to use an array for direct mapping:

```
enum MsgId
{
    MsgId_Message1 = 0x0101,
    MsgId_Message2 = 0x0205,
    MsgId_Message3 = 0x0308,
    ...
    MsgId_NumOfMessages
};
```

In this case the array of `FactoryMethod`s described earlier must be packed and binary search algorithm used to find required method. To support such search, the `FactoryMethod` must be able to report ID of the messages it creates.

```
class FactoryMethod
{
public:
    MsgIdType id() const
    {
        return idImpl();
    }

    MsgPtr createMsg() const
    {
        return createMsgImpl();
    }

protected:
    virtual MsgIdType idImpl() const = 0;
    virtual MsgPtr createMsgImpl() const = 0;
};
```

Then the code of `MsgIdLayer::createMsg()` needs to apply binary search to find the required method:

```
namespace comms
{
template <...>
class MsgIdLayer
{
private:
    MsgPtr createMsg(MsgIdType id)
    {
        auto& registry = ...; // reference to the array of point
ers to FactoryMethod-s
        auto iter =
            std::lower_bound(
                registry.begin(), registry.end(), id,
                [](FactoryMethod* method, MsgIdType idVal) -> bo
ol
                {
                    return method->id() < idVal;
                });

        if ((iter == registry.end()) ||
            ((*iter)->id() != id)) {
            return MsgPtr();
        }
        return (*iter)->createMsg();
    }
};
} // namespace comms
```

Note, that std::lower_bound algorithm requires `FactoryMethod` s in the "registry" to be sorted by the message ID. The runtime complexity of such code is `O(log(n))` , where `n` is size of the registry.

Some communication protocols define multiple variants of the same message, which are differentiated by some other means, such as serialisation length of the message. It may be convenient to implement such variants as separate message classes, which will require separate `FactoryMethod` s to instantiate them. In this case, the `MsgIdLayer::createMsg()` function may use std::equal_range algorithm instead of std::lower_bound, and use additional parameter to specify which of the methods to pick from the equal range found:

```
namespace comms
{
template <...>
class MsgIdLayer
{
private:
    MsgPtr createMsg(MsgIdType id, unsigned idx = 0)
    {
        auto& registry = ...; // reference to the array of point
ers to FactoryMethod-s
        auto iters = std::equal_range(...);

        if ((iters.first == iters.second) ||
            (iters.second < (iters.first + idx))) {
            return MsgPtr();
        }

        return (*(iter.first + idx))->createMsg();
    }
};
} // namespace comms
```

Please note, that `MsgIdLayer::read()` function also needs to be modified to support multiple attempts to create message object with the same id. It must increment the `idx` parameter, passed to `createMsg()` member function, on every failing attempt to read the message contents, and try again until the found equal range is exhausted. I leave the implementation of this extra logic as an exercise to the reader.

To complete the message allocation subject we need to come up with an automatic way to create the registry of `FactoryMethod` s used earlier. Please remember, that `FactoryMethod` was just a polymorphic interface. We need to implement actual method that implements the virtual functionality.

```
template <typename TActualMessage>
class ActualFactoryMethod : public FactoryMethod
{
protected:
    virtual MsgIdType idImpl() const
    {
        return TActualMessage::ImplOptions::MsgId;
    }

    virtual MsgPtr createMsgImpl() const
    {
        return MsgPtr(new TActualMessage());
    }
}
```

Note, that the code above assumes that `comms::option::StaticNumIdImpl` option (described in Generalising Message Implementation chapter) was used to specify numeric message ID when defining the `ActualMessage*` class.

Also note, that the example above uses dynamic memory allocation to allocate actual message object. This is just for idea demonstration purposes. The Allocating Message Object section below will describe how to support "in-place" allocation.

The types of the messages, that can be received over I/O link, are usually known at compile time. If we bundle them together in `std::tuple`, it is easy to apply already familiar meta-programming technique of iterating over the provided types and instantiate proper `ActualFactoryMethod<>` object.

```
using AllMessages = std::tuple<
    ActualMessage1,
    ActualMessage2,
    ActualMessage3,
    ...
>;
```

The size of the *registry* can easily be identified using std::tuple_size.

```
static const RegistrySize = std::tuple_size<AllMessages>::value;
using Registry = std::array<FactoryMethod*, RegistrySize>;
Registry m_registry; // registry object
```

Now it's time to iterate (at compile time) over all the types defined in the `AllMessages` tuple and create separate `ActualFactoryMethod<>` for each and every one of them. Remember tupleForEach? We need something similar here, but missing the tuple object itself. We are just iterating over types, not the elements of the tuple object. We'll call it `tupleForEachType()`. See Appendix D for implementation details.

We also require a functor class that will be invoked for every message type and will be responsible to fill the provided registry:

```
class MsgFactoryCreator
{
public:
    MsgFactoryCreator(Registry& registry)
      : registry_(registry)
    {
    }

    template <typename TMessage>
    void operator()()
    {
        static const ActualFactoryMethod<TMessage> Factory;
        registry_[idx_] = &Factory;
        ++idx_;
    }

private:
    Registry& registry_;
    unsigned idx_ = 0;
};
```

The initialisation function may be as simple as:

```
void initRegistry()
{
    tupleForEachType<AllMessages>(MsgFactoryCreator(m_registry))
;
}
```

NOTE, that `ActualFactoryMethod<>` factories do not have any internal state and are defined as static objects. It is safe just to store pointers to them in the *registry* array.

To summarise this section, let's redefine `comms::MsgIdLayer` and add the message creation functionality.

```
namespace comms
{
// TField is type of the field used to read/write message ID
// TAllMessages is all messages bundled in std::tuple.
// TNext is the next layer this one wraps
template <typename TField, typename TAllMessages, typename TNext
>
class MsgIdLayer
{
public:
    // Type of the field object used to read/write message ID va
lue.
    using Field = TField;

    // Take type of the ReadIterator from the next layer
    using ReadIterator = typename TNext::ReadIterator;

    // Take type of the WriteIterator from the next layer
    using WriteIterator = typename TNext::WriteIterator;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Type of the message ID
    using MsgIdType = typename Message::MsgIdType;
    // Redefine pointer to message type:
    using MsgPtr = ...;

    // Constructor
    MsgIdLayer()
    {
        tupleForEachType<AllMessages>(MsgFactoryCreator(m_regist
ry));
    }

    // Read operation
    ErrorStatus read(MsgPtr& msgPtr, ReadIterator& iter, std::si
ze_t len) {...}

    // Write operation
    ErrorStatus write(const Message& msg, WriteIterator& iter, s
td::size_t len) const {...}

private:
    class FactoryMethod {...};

    template <typename TMessage>
    class ActualFactoryMethod : public FactoryMethod {...};

    class MsgFactoryCreator {...};

    // Registry of Factories
    static const auto RegistrySize = std::tuple_size<TAllMessage
s>::value;
    using Registry = std::array<FactoryMethod*, RegistrySize>;


    // Create message
    MsgPtr createMsg(MsgIdType id, unsigned idx = 0)
    {
        auto iters = std::equal_range(m_registry.begin(), m_regi
stry.end(), ...);
        ...
    }
    Registry m_registry;
    TNext m_next;

};
} // namespace comms
```

Allocating Message Object

At this stage, the only missing piece of information is definition of the smart pointer type responsible to hold the allocated message object (`MsgPtr`) and allowing "in place" allocation instead of using dymaic memory.

When dynamic memory allocation is allowed, everything is simple, just use `std::unique_ptr` with standard deleter. However, it is a bit more difficult when such allocations are not allowed.

Let's start with the calculation of the buffer size which is big enough to hold any message in the provided `AllMessages` bundle. It is similar to the size of the `union` below.

```
union AllMessagesU
{
    ActualMessage1 msg1;
    ActualMessage2 msg2;
    ...
};
```

However, all the required message types are provided as `std::tuple` , not as `union` . What we need is something like std::aligned_union, but for the the types already bundled in `std::tuple` . It turns out it is very easy to implement using template specialisation:

```
template <typename TTuple>
struct TupleAsAlignedUnion;

template <typename... TTypes>
struct TupleAsAlignedUnion<std::tuple<TTypes...> >
{
    using Type = typename std::aligned_union<0, TTypes...>::type
;
};
```

NOTE, that some compilers (gcc v5.0 and below) may not implement `std::aligned_union` type, but they do implement std::aligned_storage. The Appendix E shows how to implement aligned union functionality using `std::aligned_storage` .

The "in place" allocation area, that can fit in any message type listed in `AllMessages` tuple, can be defined as:

```
using InPlaceStorage = typename TupleAsAlignedUnion<AllMessages>
::Type;
```

The "in place" allocation is simple:

```
InPlaceStorage inPlaceStorage;
new (&inPlaceStorage) TMessage(); // TMessage is type of the mes
sage being created.
```

The "in place" allocation requires "in place" deletion, i.e. destruction of the allocated element.

```
template <typename T>
struct InPlaceDeleter
{
    void operator()(T* obj) {
        obj->~T();
    }
};
```

The smart pointer to `Message` interface class may be defined as `std::unique_ptr<Message, InPlaceDeleter<Message> >`.

Now, let's define two independent allocation policies with the similar interface. One for dynamic memory allocation, and the other for "in place" allocation.

```
template <typename TMessageInterface>
struct DynMemAllocationPolicy
{
    using MsgPtr = std::unique_ptr<TMessageInterface>;

    template <typename TMessage>
    MsgPtr allocMsg()
    {
        return MsgPtr(new TMessage());
    }
}

template <typename TMessageInterface, typename TAllMessages>
class InPlaceAllocationPolicy
{
public:
    template <typename T>
    struct InPlaceDeleter {...};

    using MsgPtr = std::unique_ptr<TMessageInterface, InPlaceDel
eter<TMessageInterface> >;

    template <typename TMessage>
    MsgPtr allocMsg()
    {
        new (&m_storage) TMessage();
        return MsgPtr(
            reinterpret_cast<TMessageInterface*>(&m_storage),
            InPlaceDeleter<TMessageInterface>());
    }

private:
    using InPlaceStorage = typename TupleAsAlignedUnion<TAllMess
ages>::Type;
    InPlaceStorage m_storage;
}
```

Please pay attention, that the implementation of `InPlaceAllocationPolicy` is the simplest possible one. In production quality code, it is recommended to insert protection against double allocation in the used storage area, by introducing boolean flag indicating, that the storage area is or isn't free. The pointer/reference to such flag must also be passed to the deleter object, which is responsible to update it when deletion takes place.

The choice of the allocation policy used in `comms::MsgIdLayer` may be implemented using the already familiar technique of using options.

```
namespace comms
{
template <
    typename TField,
    typename TAllMessages,
    typename TNext,
    typename... TOptions>
class MsgIdLayer
{
    ...
};
} // namespace comms
```

If no option is specified, the `DynMemAllocationPolicy` must be chosen. To force "in place" message allocation a separate option may be defined and passed as template parameter to `comms::MsgIdLayer` .

```
namespace comms
{
namespace option
{
struct InPlaceAllocation {};
} // namespace option
} // namespace comms
```

Using the familiar technique of options parsing, we can create a structure, where a boolean value `HasInPlaceAllocation` defaults to `false` and can be set to `true` , if the option mentioned above is used. As the result, the policy choice may be implemented as:

```
namespace comms
{
template <
    typename TField,
    typename TAllMessages,
    typename TNext,
    typename... TOptions>
class MsgIdLayer
{
public:
    // TOptions parsed into struct
    using ParsedOptions = ...;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Choice of the allocation policy
    using AllocPolicy = typename std::conditional<
        ParsedOptions::HasInPlaceAllocation,
        InPlaceAllocationPolicy<Message, TAllMessages>,
        DynMemAllocationPolicy<Message>
    >::type;

    // Redefine pointer to message type
    using MsgPtr = typename AllocPolicy::MsgPtr;
    ...
private:
    AllocPolicy m_policy;
};
} // namespace comms
```

What remains to be done is to provide the `ActualFactoryMethod<>` class with an ability to use allocation policy for allocating the message. Please remember, that `ActualFactoryMethod<>` objects are stateless static ones. It means that the allocation policy object needs to passed as the parameter to its allocation function.

```
namespace comms
{
template <
    typename TField,
    typename TAllMessages,
    typename TNext,
    typename... TOptions>
class MsgIdLayer
{
public:
    // Choice of the allocation policy
    using AllocPolicy = ...;

    // Redefine pointer to message type
    using MsgPtr = typename AllocPolicy::MsgPtr;
    ...
private:
    class FactoryMethod
    {
    public:
        MsgPtr createMsg(AllocPolicy& policy) const
        {
            return createMsgImpl(policy);
        }

    protected:
        virtual MsgPtr createMsgImpl(AllocPolicy& policy) const
= 0;
    };

    template <typename TActualMessage>
    class ActualFactoryMethod : public FactoryMethod
    {
    protected:
        virtual MsgPtr createMsgImpl(AllocPolicy& policy) const
        {
            return policy.allocMsg<TActualMessage>();
        }
    }
    AllocPolicy m_policy;
};
} // namespace comms
```

Summary

The final implementation of the ID Layer (`comms::MsgIdLayer`) is a generic piece of code. It receives a list of message classes, it must recognise, as a template parameter. The whole logic of creating the right message object given the numeric ID of the message is automatically generated by the compiler using only static memory. When new message is added to the protocol, what needs to be updated is the bundle of available message classes (`AllMessages`). Nothing else is required. Recompilation of the sources will generate a code that supports new message as well. The implementation of `comms::MsgIdLayer` above has `O(log(n))` runtime complexity of finding the right factory method and creating appropriate message object. It also supports multiple variants of the same message which are implemented as different message classes, but report the same message ID. By default `comms::MsgIdLayer` uses dynamic memory to allocate new message object. It can easily be changed by providing `comms::option::InPlaceAllocation` option to it, which will force usage of "in place" allocation. The "in place" allocation may create one message at a time. In order to be able to create a new message object, the previous one must be destructed and de-allocated before.

SIZE Layer

This layer is responsible to handle the remaining length information.

- During read operation it reads the information about number of bytes required to complete the message deserialisation and compares it to the number of bytes available for reading. If input buffer has enough data, the read operation of the next (wrapped) layer is invoked.
- During write operation, the layer must calculate and write the number of bytes required to serialise the message prior to invoking the write operation of the next (wrapped) layer.

```
namespace comms
{
// TField is type of the field used to read/write SIZE informati
on
// TNext is the next layer this one wraps
template <typename TField, typename TNext>
class MsgSizeLayer
{
public:
    // Type of the field object used to read/write SIZE informat
ion.
    using Field = TField;

    // Take type of the ReadIterator from the next layer
    using ReadIterator = typename TNext::ReadIterator;

    // Take type of the WriteIterator from the next layer
    using WriteIterator = typename TNext::WriteIterator;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Take type of the message interface pointer from the next
layer
    using MsgPtr = typename TNext::MsgPtr;
    template <typename TMsgPtr>
    ErrorStatus read(TMsgPtr& msgPtr, ReadIterator& iter, std::s
ize_t len)
    {
        Field field;
        auto es = field.read(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        auto actualRemainingSize = (len - field.length());
        auto requiredRemainingSize = static_cast<std::size_t>(fi
eld.value());
        if (actualRemainingSize < requiredRemainingSize) {
            return ErrorStatus::NotEnoughData;
        }
        es = reader.read(msgPtr, iter, requiredRemainingSize);
        if (es == ErrorStatus::NotEnoughData) {
            return ErrorStatus::ProtocolError;
        }
        return es;
    }

    ErrorStatus write(const Message& msg, WriteIterator& iter, s
td::size_t len) const
    {
        Field field;
        field.value() = m_next.length(msg);
        auto es = field.write(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        return m_next.write(msg, iter, len - field.length());
    }

private:
    TNext m_next;
};
} // namespace comms
```

Please note, that reference to the smart pointer holding the message object is passed to the `read()` function using *undefined* type (template parameter) instead of using the `MsgPtr` internal type. Some communication protocols may serialise `SIZE` information before the `ID`, others may do the opposite. The `SIZE` layer is not aware of what other layers it wraps. If `ID` information is serialised before the `SIZE`, the `MsgPtr` type definition is probably taken from PAYLOAD Layer, which is defined to be `void`.

Also note, that `write()` function requires knowledge of how many bytes it will take to the next layer to serialise the message. It requires every layer to define `length(...)` member function in addition to `read()` and `write()`.

The `length()` member function of the PAYLOAD Layer may be defined as:

```
namespace comms
{
template <typename TMessage>
class MsgDataLayer
{
public:
    static constexpr std::size_t length(const TMessage& msg)
    {
        return msg.length();
    }
};
} // namespace comms
```

The `length()` member function of the ID Layer may be defined as:

```
namespace comms
{
template <
    typename TField,
    typename TAllMessages,
    typename TNext,
    typename... TOptions>
class MsgIdLayer
{
public:
    std::size_t length(const Message& msg) const
    {
        TField field;
        field.value() = msg.id();
        return field.length() + m_next.length(msg);
    }
};
} // namespace comms
```

And the `length()` member function of the SIZE Layer itself may be defined as:

```
namespace comms
{
template <typename TField, typename TNext>
class MsgSizeLayer
{
public:
    std::size_t length(const Message& msg) const
    {
        TField field;
        field.value() = m_next.length(msg);
        return field.length() + field.value();
    }
};
} // namespace comms
```

SYNC Layer

This layer is responsible to find and validate the synchronisation prefix.

```
namespace comms
{
// TField is type of the field used to read/write SYNC prefix
// TNext is the next layer this one wraps
template <typename TField, typename TNext>
class SyncPrefixLayer
{
public:
    // Type of the field object used to read/write SYNC prefix.
    using Field = TField;

    // Take type of the ReadIterator from the next layer
    using ReadIterator = typename TNext::ReadIterator;

    // Take type of the WriteIterator from the next layer
    using WriteIterator = typename TNext::WriteIterator;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Take type of the message interface pointer from the next
layer
    using MsgPtr = typename TNext::MsgPtr;

    template <typename TMsgPtr>
    ErrorStatus read(TMsgPtr& msgPtr, ReadIterator& iter, std::s
ize_t len)
    {
        Field field;
        auto es = field.read(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        if (field.value() != Field().value()) {
            // doesn't match expected
            return ErrorStatus::ProtocolError;
        }
        return m_next.read(msgPtr, iter, len - field.length());
    }

    ErrorStatus write(const Message& msg, WriteIterator& iter, std::size_t len) const
    {
        Field field;
        auto es = field.write(iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        return m_next.write(msg, iter, len - field.length());
    }

    std::size_t length(const TMessage& msg) const
    {
        return Field().length() + m_next.length(msg);
    }

private:
    TNext m_next;
};
} // namespace comms
```

Note, that the value of the `SYNC` prefix is expected to be equal to the value of the default constructed `TField` field type. The default construction value may be set using `comms::option::DefaultNumValue` option described in Generalising Fields Implementation chapter.

For example, 2 bytes synchronisation prefix `0xab 0xcd` with big endian serialisation may be defined as:

```
using CommonFieldBase = comms::Field<false>; // big endian seria
lisation base
using SyncPrefixField =
    comms::IntValueField<
        CommonFieldBase,
        std::uint16_t,
        comms::option::DefaultNumValue<0xabcd>
    >;
```

CHECKSUM Layer

This layer is responsible to calculate and validate the checksum information.

- During read operation it remembers the initial value of the read iterator, then invokes the read operation of the next (wrapped) layer. After the next layer reports successful completion of its read operation, the expected checksum value is read. Then, the real checksum on the read data bytes is calculated and compered to the expected one. If the values match, the read operation is reported as successfully complete. If not, the created message object is deleted and error reported.
- During write operation it lets the next (wrapped) layer to finish its writing, calculates the checksum value on the written data bytes, and writes the result into output buffer.

Before jumping into writing the code, there is a need to be aware of couple of issues:

- The generic code of the **CHECKSUM Layer** mustn't depend on any particular checksum calculation algorithm. I'd recommend providing the calculator class as a template parameter, `operator()` of which is responsible to implement the checksum calculation logic.
- The checksum calculation after write operation requires the iterator to go back and calculate the checksum on the written data bytes. It can easily be done when used iterator is random access one. Sometimes it may not be the case (for example the output data is written into std::vector using std::back_insert_iterator). There is a need to have a generic way to handle such cases.

Implementing Read

Let's start with implementing the read first.

```
namespace comms
{
// TField is type of the field used to read/write checksum value
// TCalc is generic class that is responsible to implement check
sum calculation logic
// TNext is the next layer this one wraps
template <typename TField, typename TCalc, typename TNext>
class ChecksumLayer
{
public:
    // Type of the field object used to read/write SYNC prefix.
    using Field = TField;

    // Take type of the ReadIterator from the next layer
    using ReadIterator = typename TNext::ReadIterator;

    // Take type of the WriteIterator from the next layer
    using WriteIterator = typename TNext::WriteIterator;

    // Take type of the message interface from the next layer
    using Message = typename TNext::Message;

    // Take type of the message interface pointer from the next
layer
    using MsgPtr = typename TNext::MsgPtr;

    template <typename TMsgPtr>
    ErrorStatus read(TMsgPtr& msgPtr, ReadIterator& iter, std::s
ize_t len)
    {
        Field field;
        if (len < field.length()) {
            return ErrorStatus::NotEnoughData;
        }
        auto fromIter = iter; // The read is expected to use ran
dom-access iterator
        auto es = m_next.read(iter, len - field.length());
        if (es != ErrorStatus::Success) {
            return es;
        }
        auto consumedLen = static_cast<std::size_t>(std::distanc
e(fromIter, iter));
        auto remLen = len - consumedLen;
        es = field.read(iter, remLen);
        if (es != ErrorStatus::Success) {
            msgPtr.reset();
            return es;
        }
        auto checksum = TCalc()(fromIter, consumedLen);
        auto expectedValue = field.value();
        if (expectedValue != checksum) {
            msgPtr.reset(); // Delete allocated message
            return ErrorStatus::ProtocolError;
        }
        return ErrorStatus::Success;
    }
private:
    TNext m_next;
};
} // namespace comms
```

It is clear that `TCalc` class is expected to have `operator()` member function, which receives the iterator for reading and number of bytes in the buffer.

As an example let's implement generic total sum of bytes calculator:

```
namespace comms
{
template <typename TResult = std::uint8_t>
class BasicSumCalc
{
public:
    template <typename TIter>
    TResult operator()(TIter& iter, std::size_t len) const
    {
        using ByteType = typename std::make_unsigned<
            typename std::decay<decltype(*iter)>::type
        >::type;

        auto checksum = TResult(0);
        for (auto idx = 0U; idx < len; ++idx) {
            checksum += static_cast<TResult>(static_cast<ByteTyp
e>(*iter));
            ++iter;
        }
        return checksum;
    }
};
} // namespace comms
```

Implementing Write

Now, let's tackle the write problem. As it was mentioned earlier, there is a need to recognise the type of the iterator used for writing and behave accordingly. If the iterator is properly defined, the std::iterator_traits class will define `iterator_category` internal type.

```
using WriteIteratorCategoryTag =
    typename std::iterator_traits<WriteIterator>::iterator_categ
ory;
```

For random access iterators the `WriteIteratorCategoryTag` class will be either std::random_access_iterator_tag or any other class that inherits from it. Using this information, we can use Tag Dispatch Idiom to choose the right writing functionality.

```
namespace comms
{
template <...>
class ChecksumLayer
{
public:
    using WriteIteratorCategoryTag =
        typename std::iterator_traits<WriteIterator>::iterator_c
ategory;

    ErrorStatus write(const Message& msg, WriteIterator& iter, s
td::size_t len) const
    {
        return writeInternal(msg, iter, len, WriteIteratorCatego
ryTag());
    }

private:
    ErrorStatus writeInternal(
        const Message& msg,
        WriteIterator& iter,
        std::size_t len,
        const std::random_access_iterator_tag&) const
    {
        return writeRandomAccess(msg, iter, len);
    }

    ErrorStatus writeInternal(
        const Message& msg,
        WriteIterator& iter,
        std::size_t len,
        const std::output_iterator_tag&) const
    {
        return writeOutput(msg, iter, len);
    }

    ErrorStatus writeRandomAccess(const Message& msg, WriteItera
tor& iter, std::size_t len) const
    {
        auto fromIter = iter;
        auto es = m_next.write(msg, iter, len);
        if (es != ErrorStatus::Success) {
            return es;
        }
        auto consumedLen = static_cast<std::size_t>(std::distanc
e(fromIter, iter));
        auto remLen = len - consumedLen;
        Field field;
        field.value() = TCalc()(fromIter, consumedLen);
        return field.write(iter, remLen);
    }

    ErrorStatus writeOutput(const Message& msg, WriteIterator& i
ter, std::size_t len) const
    {
        TField field;
        auto es = m_next.write(msg, iter, len - field.length());
        if (es != ErrorStatus::Success) {
            return es;
        }
        field.write(iter, field.length());
        return ErrorStatus::UpdateRequired;
    }

    TNext m_next;
};
} // namespace comms
```

Please pay attention, that `writeOutput()` function above is unable to properly calculate the checksum of the written data. There is no way the iterator can be reversed back and used as input instead of output. As the result the function returns special error status: `ErrorStatus::UpdateRequired`. It is an indication that the write operation is not complete and the output should be updated using random access iterator.

Implementing Update

```
namespace comms
{
template <...>
class ChecksumLayer
{
public:

    template <typename TIter>
    ErrorStatus update(TIter& iter, std::size_t len) const
    {
        Field field;
        auto fromIter = iter;
        auto es = m_next.update(iter, len - field.length());
        if (es != ErrorStatus::Success) {
            return es;
        }

        auto consumedLen = static_cast<std::size_t>(std::distanc
e(fromIter, iter));
        auto remLen = len - consumedLen;
        field.value() = TCalc()(fromIter, consumedLen);
        es = field.write(iter, remLen);
        return es;
    }

private:
    TNext m_next;
};
} // namespace comms
```

Please note, that every other layer must also implement the `update()` member function, which will just advance the provided iterator by the number of bytes required to write its field and invoke `update()` member function of the next (wrapped) layer.

```
namespace comms
{
template <typename TMessage>
class MsgDataLayer
{
public:
    template <typename TIter>
    ErrorStatus update(TIter& iter, std::size_t len) const
    {
        std::advance(iter, len);
        return ErrorStatus::Success;
    }
};

} // namespace comms
```

```
namespace comms
{
template <...>
class MsgIdLayer
{
public:
    template <typename TIter>
    ErrorStatus update(TIter& iter, std::size_t len) const
    {
        TField field;
        std::advance(iter, field.length());
        return m_next.update(iter, len - field.length());
    }

private:
    TNext m_next;
};

} // namespace comms
```

And so on for the rest of the layers. Also note, that the code above will work, only when the field has the **same serialisation length for any value**. If this is not the case (Base-128 encoding is used), the previously written value needs to be read, instead of just advancing the iterator, to make sure the iterator is advanced right amount of bytes:

```
template <typename TIter>
ErrorStatus update(TIter& iter, std::size_t len) const
{
    TField field;
    auto es = field.read(iter);
    if (es != ErrorStatus::Success) {
        return es;
    }
    return m_next.update(iter, len - field.length());
}
```

The variable serialisation length encoding will be forced using some kind of special option. It can be identified at compile time and Tag Dispatch Idiom can be used to select appropriate `update` functionality.

The caller, that requests protocol stack to serialise a message, must check the error status value returned by the `write()` operation. If it is `ErrorStatus::UpdateRequired`, the caller must create random-access iterator to the already written buffer and invoke `update()` function with it, to make sure the written information is correct.

```
using ProtocolStack = ...;
ProtocolStack protStack;

ErrorStatus sendMessage(const Message& msg)
{
    ProtocolStack::WriteIterator writeIter = ...;
    auto es = protStack.write(msg, writeIter, bufLen);
    if (es == ErrorStatus::UpdateRequired) {
        auto updateIter = ...; // Random access iterator to writ
ten data
        es = protStack.update(updateIter, ...);
    }
    return es;
}
```

Defining Protocol Stack

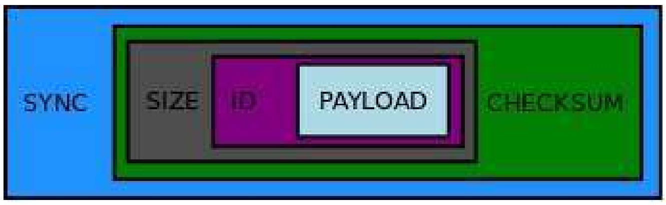
To summarise the protocol transport wrapping subject, let's define a custom protocol that wraps the message payload in the following way:

```
SYNC | SIZE | ID | PAYLOAD | CHECKSUM
```

where:

- All the fields are serialised using BIG endian.
- SYNC - 2 bytes of synchronisation value to indicate beginning of the message, must be "0xab 0xcd"
- SIZE - 2 bytes, length of remaining data, **including checksum** and not including SIZE field itself.
- ID - 1 byte, numeric ID of the message.
- PAYLOAD - any number of bytes, serialised message data
- CHECKSUM - 2 bytes, arithmetic summary of all bytes starting (and including) from SIZE field and ending after PAYLOAD field.

The protocol layer should wrap one another in the following way:



Please note, that `CHECKSUM` layer doesn't wrap `SYNC` because synchronisation prefix is not included in the checksum calculation.

Common Protocol Definitions

```
// BIG endian fields serialisation
using MyField = comms::Field<false>;

// Message IDs
enum MyMsgId : std::uint16_t
```

```
{
    MyMsgId_ActualMessage1,
    MyMsgId_ActualMessage2,
    MyMsgId_ActualMessage3,
    ...
};

// Forward declaration of MyHandler handling class
class MyHandler;

// Message interface
// NOTE: write operation will write data into a vector using pus
h_back() calls
using MyMessage =
    comms::Message<
        comms::option::MsgIdType<MyMsgId>
        comms::option::ReadIterator<const std::uint8_t*>
        comms::option::WriteIterator<std::back_insert_iterator<s
td::vector<std::uint8_t> > >,
        comms::option::LengthInfoInterface,
        comms::option::Handler<MyHandler>
    >;

// Definition of the messages
class ActualMessage1Fields = std::tuple<...>;
template <typename TMessage>
class ActualMessage1 : public
    comms::MessageBase<
        comms::option::StaticNumIdImpl<MyMsgId_ActualMessage1>,
        comms::option::FieldsImpl<ActualMessage1Fields>,
        comms::option::DispatchImpl<ActualMessage1<TMessage>
    >
{
};

class ActualMessage2Fields = std::tuple<...>;
template <typename TMessage>
class ActualMessage1 : public comms::MessageBase<...> {};
...
```

```
// Bundling all messages together
template <typename TMessage>
using AllMessages = std::tuple<
    ActualMessage1<TMessage>,
    ActualMessage2<TMessage>,
    ...
>;
```

PAYLOAD Layer

```
using MyPayloadLayer = comms::MsgDataLayer<MyMessage>;
```

ID Layer

```
using MyMsgIdField = comms::EnumValueField<MyField, MyMsgId>
using MyIdLayer = comms::MsgIdLayer<MyMsgIdField, AllMessages, M
yPayloadLayer>;
```

SIZE Layer

```
using MySizeField =
    comms::IntValueField<
        MyField,
        std::uint16_t,
        comms::option::NumValueSerOffset<sizeof(std::uint16_t)>
    >;
using MySizeLayer = comms::MsgSizeLayer<MySizeField, MyIdLayer>;
```

Please note, that `SIZE` field definition uses `comms::option::NumValueSerOffset` option, which effectively adds `2` when size value is serialised, and subtracts it when remaining length is deserialised. It must be done, because `SIZE` value specifies **number of remaining bytes**, including the `CHECKSUM` value at the end.

CHECKSUM Layer

```
using MyChecksumField = comms::IntValueField<MyField, std::uint1
6_t>;
using MyChecksumLayer = comms::ChecksumLayer<
    MyChecksumField,
    comms::BasicSum<std::uint16_t>
    MySizeLayer
>;
```

SYNC Layer

```
using MySyncField = comms::IntValueField<
    MyField,
    std::uint16_t,
    comms::option::DefaultNumValue<0xabcd>
>;
using MySyncPrefix = comms::SyncPrefixLayer<SyncField, MyChecksu
mLayer>;
```

Processing Loop

The outermost layer defines a full protocol stack. It should be typedef-ed to avoid any confusion:

```
using MyProtocolStack = MySyncPrefix;
```

The processing loop may look like this:

```
// Protocol stack
MyProtocolStack protStack;

// Message handler object
MyHandler handler;
```

```
// Input data storage, the data received over I/O link is appended here
std::vector<std::uint8_t> inData;

void processData()
{
    while (!inData.empty()) {
        MyProtocolStack::ReadIterator readIter = &inData[0];
        MyProtocolStack::MsgPtr msg;
        auto es = protStack.read(msg, readIter, inData.size());
        if (es == comms::ErrorStatus::NotEnoughData) {
            // More data is required;
            return;
        }
        if (es == comms::ErrorStatus::Success) {
            assert(msgPtr); // Must hold the valid message object

            msgPtr->dispatch(handler); // Process message, dispatch to handling function

            // Erase consumed bytes from the buffer
            auto consumedBytes =
                std::distance(ProtocolStack::ReadIterator(&inData[0]), readIter);
            inData.erase(inData.begin(), inData.begin() + consumedBytes);
            continue;
        }
        // Some error occurred, pop only one first byte and try to process again
        inData.erase(inData.begin());
    }
}
```

The processing loop above is not the most efficient one, but it demonstrates what needs to be done and how our generic library can be used to identify and process the received message.

Writing Message

The write logic is even simpler.

```
void sendMessage(const MyMessage& msg)
{
    // Output buffer
    std::vector<std::uint8_t> outData;
    // Reserve enough space in output buffer
    outData.reserve(protStack.length(msg));
    auto writeIter = std::back_inserter(outData);
    auto es = protStack.write(msg, writeIter, outData.max_size()
);
    if (es == comms::ErrorStatus::UpdateRequired) {
        auto updateIter = &outData[0];
        es = protStack.update(updateIter, outData.size());
    }

    if (es != comms::ErrorStatus::Success) {
        ... // report error
        return;
    }
    ... // Send written data over I/O link
}
```

Achievements

After all this effort of creating the generic `comms` library, let's summarise what has been achieved.

- The communication protocol implementation becomes easy and straightforward process, using mostly declarative statements of classes and types definitions without unnecessary boilerplate code. The C++ compiler does all the dirty and boring work of generating the required code.
- The default logic, provided by the library, can easily be extended and/or overridden using class inheritance and virtual functions.
- The protocol implementation doesn't enforce any restrictions on data structures being used, and as a result it can be reused in any system, including bare-metal ones.
- There is no dependency on any specific I/O link and the way the data is being communicated.
- The application level messages and transport data are completely independent, which allows usage of the same application level messages over different I/O links, which require different transport wrapping, at the same time.

Appendices

Appendices contain some extra code examples mentioned in this book and can be used for references.

Appendix A - tupleForEach

Implementation of `tupleForEach()` function. Namespace `details` contains some helper classes.

```
namespace details
{
template <std::size_t TRem>
class TupleForEachHelper
{
public:
    template <typename TTuple, typename TFunc>
    static void exec(TTuple&& tuple, TFunc&& func)
    {
        using Tuple = typename std::decay<TTuple>::type;
        static const std::size_t TupleSize = std::tuple_size<Tup
le>::value;
        static_assert(TRem <= TupleSize, "Incorrect parameters")
;

        // Invoke function with current element
        static const std::size_t Idx = TupleSize - TRem;
        func(std::get<Idx>(std::forward<TTuple>(tuple)));

        // Compile time recursion - invoke function with the rem
aining elements
        TupleForEachHelper<TRem - 1>::exec(
            std::forward<TTuple>(tuple),
            std::forward<TFunc>(func));
    }
};

template <>
class TupleForEachHelper<0>
{
public:
    // Stop compile time recursion
    template <typename TTuple, typename TFunc>
    static void exec(TTuple&& tuple, TFunc&& func)
    {
        static_cast<void>(tuple);
        static_cast<void>(func);
    }
};
}  // namespace details

template <typename TTuple, typename TFunc>
void tupleForEach(TTuple&& tuple, TFunc&& func)
{
    using Tuple = typename std::decay<TTuple>::type;
    static const std::size_t TupleSize = std::tuple_size<Tuple>:
:value;

    details::TupleForEachHelper<TupleSize>::exec(
        std::forward<TTuple>(tuple),
        std::forward<TFunc>(func));
}
```

Appendix B - tupleAccumulate

Implementation of `tupleAccumulate()` function. Namespace `details` contains some helper classes.

```
namespace details
{

template <std::size_t TRem>
class TupleAccumulateHelper
{
public:
    template <typename TTuple, typename TValue, typename TFunc>
    static constexpr TValue exec(TTuple&& tuple, const TValue& v
alue, TFunc&& func)
    {
        using Tuple = typename std::decay<TTuple>::type;
        static_assert(TRem <= std::tuple_size<Tuple>::value, "In
correct TRem");

        return TupleAccumulateHelper<TRem - 1>::exec(
                    std::forward<TTuple>(tuple),
                    func(value, std::get<std::tuple_size<Tuple>:
:value - TRem>(std::forward<TTuple>(tuple))),
                    std::forward<TFunc>(func));
    }
};

template <>
class TupleAccumulateHelper<0>
{

public:
    template <typename TTuple, typename TValue, typename TFunc>
    static constexpr TValue exec(TTuple&& tuple, const TValue& v
alue, TFunc&& func)
    {
        return value;
    }
};

}  // namespace details

template <typename TTuple, typename TValue, typename TFunc>
constexpr TValue tupleAccumulate(TTuple&& tuple, const TValue& v
alue, TFunc&& func)
{
    using Tuple = typename std::decay<TTuple>::type;

    return details::TupleAccumulateHelper<std::tuple_size<Tuple>
::value>::exec(
                std::forward<TTuple>(tuple),
                value,
                std::forward<TFunc>(func));
}
```

Appendix C - tupleForEachFromUntil

Implementation of `tupleAccumulate()` function. Namespace `details` contains some helper classes.

```
namespace details
{

template <std::size_t TRem, std::size_t TOff = 0>
class TupleForEachFromUntilHelper
{
public:
    template <typename TTuple, typename TFunc>
    static void exec(TTuple&& tuple, TFunc&& func)
    {
        using Tuple = typename std::decay<TTuple>::type;
        static const std::size_t TupleSize = std::tuple_size<Tup
le>::value;
        static const std::size_t OffsetedRem = TRem + TOff;
        static_assert(OffsetedRem <= TupleSize, "Incorrect param
eters");

        static const std::size_t Idx = TupleSize - OffsetedRem;
        func(std::get<Idx>(std::forward<TTuple>(tuple)));
        TupleForEachFromUntilHelper<TRem - 1, TOff>::exec(
            std::forward<TTuple>(tuple),
            std::forward<TFunc>(func));
    }
};

template <std::size_t TOff>
class TupleForEachFromUntilHelper<0, TOff>
{
public:
    template <typename TTuple, typename TFunc>
    static void exec(TTuple&& tuple, TFunc&& func)
    {
        static_cast<void>(tuple);
        static_cast<void>(func);
    }
};

}  // namespace details

// Invoke provided functor for every element in the tuple which
indices
//     are in range [TFromIdx, TUntilIdx).
template <std::size_t TFromIdx, std::size_t TUntilIdx, typename
TTuple, typename TFunc>
void tupleForEachFromUntil(TTuple&& tuple, TFunc&& func)
{
    using Tuple = typename std::decay<TTuple>::type;
    static const std::size_t TupleSize = std::tuple_size<Tuple>:
:value;
    static_assert(TFromIdx < TupleSize,
        "The from index is too big.");

    static_assert(TUntilIdx <= TupleSize,
        "The until index is too big.");

    static_assert(TFromIdx < TUntilIdx,
        "The from index must be less than until index.");

    static const std::size_t FieldsCount = TUntilIdx - TFromIdx;

    details::TupleForEachFromUntilHelper<FieldsCount, TupleSize
- TUntilIdx>::exec(
        std::forward<TTuple>(tuple),
        std::forward<TFunc>(func));
}
```

Appendix D - tupleForEachType

Implementation of `tupleForEachType()` function. Namespace `details` contains some helper classes.

```
namespace details
{
template <std::size_t TRem>
class TupleForEachTypeHelper
{
public:
    template <typename TTuple, typename TFunc>
    static void exec(TFunc&& func)
    {
        using Tuple = typename std::decay<TTuple>::type;
        static const std::size_t TupleSize = std::tuple_size<Tup
le>::value;
        static_assert(TRem <= TupleSize, "Incorrect TRem");

        static const std::size_t Idx = TupleSize - TRem;
        using ElemType = typename std::tuple_element<Idx, Tuple>
::type;
        func.template operator()<ElemType>();
        TupleForEachTypeHelper<TRem - 1>::template exec<TTuple>(
            std::forward<TFunc>(func));
    }
};

template <>
class TupleForEachTypeHelper<0>
{

public:
    template <typename TTuple, typename TFunc>
    static void exec(TFunc&& func)
    {
        // Nothing to do
    }
};
}  // namespace details

template <typename TTuple, typename TFunc>
void tupleForEachType(TFunc&& func)
{
    using Tuple = typename std::decay<TTuple>::type;
    static const std::size_t TupleSize = std::tuple_size<Tuple>:
:value;

    details::TupleForEachTypeHelper<TupleSize>::template exec<Tu
ple>(
        std::forward<TFunc>(func));
}
```

Appendix E - AlignedUnion

Implementation of `AlignedUnion` type.

```
template <typename TType, typename... TTypes>
class AlignedUnion
{
    using OtherStorage = typename AlignedUnion<TTypes...>::Type;
    static const std::size_t OtherSize = sizeof(OtherStorage);
    static const std::size_t OtherAlignment = std::alignment_of<
OtherStorage>::value;
    using FirstStorage = typename AlignedUnion<TType>::Type;
    static const std::size_t FirstSize = sizeof(FirstStorage);
    static const std::size_t FirstAlignment = std::alignment_of<
FirstStorage>::value;
    static const std::size_t MaxSize = FirstSize > OtherSize ? F
irstSize : OtherSize;
    static const std::size_t MaxAlignment =
        FirstAlignment > OtherAlignment ? FirstAlignment : Other
Alignment;
public:

    /// Type that has proper size and proper alignment to keep a
ny of the
    /// specified types
    using Type = typename std::aligned_storage<MaxSize, MaxAlign
ment>::type;
};

template <typename TType>
class AlignedUnion<TType>
{
public:
    using Type =
        typename std::aligned_storage<sizeof(TType), std::alignm
ent_of<TType>::value>::type;
};
```