prophy Documentation

Release 1.1.2

Krzysztof Laskowski

Aug 02, 2018

Contents

Prophy is a statically typed, binary, unpacked serialization protocol. See *examples* to get started quickly.

It has a *schema language*, specified *wire representation* and compiler which generates codecs in *Python* and C++ [*full*, *raw*].

It's similar to XDR, ASN.1, Google Protobuf, Apache Thrift and Cap'n Proto.

CHAPTER 1

Installation

Prophy requires Python 2.7 or Python 3.4 (or newer). You can install it via PyPI:

```
pip install prophy
```

If you need *sack mode* in Prophy Compiler, you also need:

- libclang, at least 3.4
- Python libclang adapter with corresponding version

In order to compile C++ codecs and dependent code, you'll need to deploy in your distribution or build system directory with C++ prophy header-only library:

```
prophy_cpp/include/prophy
```

so that includes in generated code are found by compiler:

```
#include <prophy/prophy.hpp>
```

Project is hosted on github.

CHAPTER 2

Schema language

Let's call it *prophylang*. Prophy messages are meant to be held in .prophy files. These files contain structs and unions composed with numeric types, constants, enums, arrays and nested structs and unions. All these primitives are explained below.

Encoding section explains wire representation of these language constructs.

There are constraints on constructs composability, pointed out in notes.

2.1 Numeric types

Prophy has following numeric types:

i8	signed 8-bit integer
i16	signed 16-bit integer
i32	signed 32-bit integer
i64	signed 64-bit integer
u8	unsigned 8-bit integer
u16	unsigned 16-bit integer
u32	unsigned 32-bit integer
u64	unsigned 64-bit integer
float	32-bit floating single-precision number
double	64-bit floating double-precision number

2.2 Constant

Constants for use as enumerator values, array lengths or union discriminators may be defined this way:

```
const MY_MIN = -1;
const MY_MAX = 0xFFF;
const MY_AVG = (MY_MIN + MY_MAX) / 2;
```

Signed, unsigned decimal, octal, hexal numbers and expressions are allowed.

2.3 Enum

Enumerations may be defined in following manner:

```
enum MyEnum
{
    MyEnum_1 = 1,
    MyEnum_2 = 2,
    MyEnum_3 = (MyEnum_1 + MyEnum_2) << 2
};
```

Enumerator definitions follow the same rules as constants.

2.4 Typedef

Typedef is an alias for previously defined type. Numeric type, enum, struct, union or other typedef may be aliased:

```
typedef u32 my_aliased_int;
typedef X my_aliased_x;
```

2.5 Struct

Struct contains fields. Fields can be numeric types, enums, structs, unions, typedefs or arrays of any former. There are 4 kinds of arrays:

kind	schema	fixed-length on wire	variable number of elements
fixed	`Type x[Size]`	yes	no
dynamic	`Type x<>`	no	yes
limited	`Type x<Limit>`	yes	yes (up to limit)
greedy	`Type x<...>`	no	yes
ext. sized	`Type x<@sizer>`	no	yes

Note: Fixed and limited array cannot hold dynamic nor unlimited struct.

Note: Greedy array may only be used in the last field of struct.

Note: Externally sized array can be sized explicity by an exisiting field of a structure. A few arrays of different types can be sized by the same field.

This is how they can be used in a struct:

```
struct X
{
    u32 a;
    MyUserDefinedType b;
    u32 c[3]; // fixed array of length 3
    u32 d<>; // dynamic array
    u32 e<3>; // limited array of limit 3
    u32 f<...>; // greedy array
    u8 g<@a>; // External sized array 1, size in a
    u32 g<@a>; // External sized array 2, size in a
};
```

Field may be declared as bytes field, if it's meant to hold opaque binary data. Codec may use this information to manipulate its data in more effective manner. Same rules as with arrays apply:

```
struct X
{
    bytes a[3];
    bytes b<>;
    bytes c<3>;
    bytes d<...>;
};
```

Field may be optional:

```
struct X
{
    u32* x;
};
```

Note: Optional field cannot hold dynamic nor unlimited struct. There's no optional array.

Array field size may be given as expression:

```
const A = 10;
const B = 2;

struct X
{
    u32 a[A * B];
};
```

2.6 Union

Discriminated unions are defined like structs, but with unsigned discriminators at the beginning of each field:

```
union MyUnion
{
    1: i8 a;
    2: u64 b;
    3: SomeType c;
};
```

Discriminators may be literals or references to constants or enumerators.

Note: Union arm cannot hold dynamic nor unlimited struct, nor array.

2.7 Include

Prophy files may include definitions and constants from different files using the usual include syntax borrowed from C language:

```
#include "A_is_inside.prophy"

struct X
{
    A x;
}
```

File "A_is_inside.prophy" may exist in current directory of including file or in any directory provided as include dir in compiler invocation.

2.8 Limitations

Currently there are no scoped definitions in the language.

CHAPTER 3

Encoding

This section describes how serialized prophy messages look like.

Prophy message wire format features no field or type tags, no submessage delimiters and no integer packing.

Prophy ensures that each field in message is aligned. This allows to manipulate message directly in serialized buffer and contributes to encoding speed.

There are constraints related to array, struct and union composability pointed out in this section's notes.

If not stated otherwise all following examples will have numbers encoded in little endian for the sake of brevity.

3.1 Numeric types

Prophy encodes integral and floating point numbers according to their size and endianness. Enums are treated as 32-bit unsigned numbers.

Number 42 encodes	as little endian	as big endian
u8/i8	2a	2a
u16/i16	2a 00	00 2a
u32/i32	2a 00 00 00	00 00 00 2a
u64/i64	2a 00 00 00 00 00 00 00	00 00 00 00 00 00 00 2a
float	00 00 28 42	42 28 00 00
double	00 00 00 00 00 00 45 40	40 45 00 00 00 00 00 00
enum	2a 00 00 00	00 00 00 2a

Signed types treat most significant bit as sign bit following U2 encoding rules.

3.2 Array

Array is a sequence of elements of the same type. Elements may be numeric types, structs or unions.

Note: Dynamic struct may not be held in fixed or limited array.

Note: Unlimited (greedy) struct may not be held in any array.

There are so much as 5 different types of arrays in Prophy.

3.2.1 Fixed array

Fixed array has fixed number of elements, hence fixed size on wire. This array:

```
u16 x[4];
```

with elements set to 1, 2, 3 and 4 would encode as:

```
01 00 02 00 03 00 04 00
```

3.2.2 Dynamic array

Dynamic array has varying number of elements counted by 32-bit unsigned delimiter. This one:

```
u16 x<>;
```

with 2 elements set to 1 and 2 encodes as:

```
02 00 00 00 01 00 02 00
```

3.2.3 Limited array

A combination of fixed and dynamic one. Delimited by an element counter, has fixed size on wire. This requires it to have an upper limit. Such array:

```
u16 x<4>;
```

with 2 elements set to 1 and 2 encodes as:

```
02 00 00 00 01 00 02 00 00 00 00 00
```

3.2.4 Greedy array

Variable element array without element counter. This one:

```
u16 x<...>;
```

with 2 elements set to 1 and 2 encodes as:

```
01 00 02 00
```

Note: Greedy array can be used only in the last field of struct. Such struct may also be only the last field of any other struct.

3.2.5 Externally sized array

An array with externally, explicit defined size:

```
u8 size;         // = 2
u8 x<@size>;     // = [4, 5]
u16 y<@size>;    // = [6, 7]
```

with fields set to values from comments above - encodes as:

```
02 04 05 00 06 00 07
```

Note:

- One 'sizer' can describe element quantity of several arrays of different type.
- The 'sizer' doesn't have to be followed by its sized arrays. There can be anything in between (besides the greedy array).
- If there are many sized arrays - they can also be splited by other fields.

Warning:

The 'sizer' field needs to be defined:

- inside the same struct as its sized arrays and
- placed before these arrays.

The externally sized array is not supported by *c++ full* codec.

3.2.6 Bytes

Bytes field is an array of bytes, which can be handled by codec in more effective way than an array of u8. Wire format, however, is the same.

3.3 Optional

Optional is a fixed-size value prepended by a boolean value encoded as a 32-bit integer. If it's not set, it's filled with zeroes up to size. This one:

```
u32* x;
```

with x set to 1 would encode as:

```
01 00 00 00 01 00 00 00
```

and with x not set would encode as:

```
00 00 00 00 00 00 00 00
```

Note: Optional field may not contain unlimited nor dynamic struct.

3.4 Struct

A sequence of fields which get serialized in strict order. Following struct X:

```
struct Nested
{
    u16 n1;
    u16 n2;
};

struct X
{
    Nested x;
    u32 y;
};
```

with fields set to (1, 2) and 3 will yield:

```
01 00 02 00 03 00 00 00
```

3.4.1 Dynamic struct

Struct containing dynamic arrays directly or indirectly becomes dynamic itself - its wire representation size varies.

Note: Dynamic struct may not be held in fixed or limited array.

3.4.2 Unlimited struct

Struct which contains greedy array or unlimited struct in the last field becomes an unlimited struct.

Note: Unlimited struct may not be held in any array or non-last struct field.

3.5 Union

Union has fixed size, related to its largest arm size. It encodes single arm prepended by a field discriminator encoded as a 32-bit integer. This union:

```
struct TwoInts
{
    u16 a1;
    u16 a2;
};

union X
{
    0: u32 x;
    1: TwoInts y;
};
```

with first arm discriminated and set to 1 encodes as:

```
00 00 00 00 01 00 00 00
```

and with second arm discriminated and set to (2, 3) encodes as:

```
01 00 00 00 02 00 03 00
```

Note: Union arm may not contain unlimited nor dynamic struct, nor array.

3.6 Padding

Prior examples were deliberately composed of values tiled together without padding in-between. Facts that:

- different length integral values are allowed,
- any field in struct/union (recursively) needs to be aligned to address divisible by its alignment (assuming starting from 0).

makes it necessary to insert padding between struct fields, union discriminator and arm, optional flag and value, array delimiter and elements or at the end of struct.

Technically padding bytes can have any values, but canonically encoded messages should be padded with zeroes.

Let's go through a couple of examples.

3.6.1 Integer padding

In this struct:

```
struct
{
    u8 a;
    u16 b;
};
```

field b requires one byte of padding to be aligned:

```
01 [00] 02 00
```

3.6.2 Composite padding

Composite (struct or union) alignment is the greatest alignment of its fields. Optional flag, union discriminator, array delimiter all contribute to struct alignment. Furthermore - each composite byte-size must be a multiple of its alignment. In this example struct X:

```
struct Nested
{
    u16 n1;
    u32 n2;
    u16 n3;
};

struct X
{
    u64 x;
    u32 y;
    u8 z;
    Nested n;
};
```

illustrates four such paddings:

1. to align Nested field
2. to align n2 field
3. to align Nested struct
4. to align X struct

```
01  00  00  00  00  00  00  00
02  00  00  00  03 [00  00  00]
04  00 [00  00] 05  00  00  00
06  00 [00  00][00  00  00  00]
```

3.6.3 Dynamic array padding

Dynamic array is tricky - it requires padding depending on number of elements. Other than that - usual rules apply. Such struct:

```
struct X
{
    u8 x<>;
    u8 y<>;
};
```

encoded with [1] and [2, 3, 4] will be padded this way:

```
01 00 00 00 01 [00 00 00] 03 00 00 00 02 03 04 [00]
```

if [] and [1, 2, 3, 4] were chosen, there would be no padding at all:

```
00 00 00 00 04 00 00 00 01 02 03 04
```

Arrays with elements exceeding delimiter alignment may require padding:

```
struct X
{
    u64 x<>;
};
```

```
01 00 00 00 [00 00 00 00] 01 00 00 00 00 00 00 00
```

even if there are no elements (composite padding):

```
00 00 00 00 [00 00 00 00]
```

3.6.4 Optional padding

Optional fields don't follow the composite rule, their byte-size doesn't need to be a multiple of alignment. Thanks to that, second field in this example doesn't need to be padded (but struct as such is padded to multiple of 4 - flag alignment):

```
struct X
{
    u8* x;
    u8 y;
};
```

```
01 00 00 00 01 02 [00 00]
```

Optional fields can have padding between flag and value, if value has alignment greater than flag:

```
struct X
{
    u64* x;
};
```

```
01 00 00 00 [00 00 00 00] 01 00 00 00 00 00 00 00
```

3.6.5 Union padding

Unions follow composite rule of padding to multiple of alignment:

```
union X
{
    1: u8 x;
};
```

```
01 00 00 00 02 [00 00 00]
```

and - like optionals - can insert padding between discriminator and arm:

```
union X
{
    1: u64 x;
    2: u8 y;
};
```

```
01 00 00 00 [00 00 00 00] 02 00 00 00 00 00 00 00
```

Note that:

- such padding applies to other arms also,
- shorter arms are padded to largest arm size.

```
02 00 00 00 [00 00 00 00] 03 [00 00 00 00 00 00 00]
```

3.6.6 Fields following dynamic fields

We can split any struct to blocks which end with dynamic fields. In order to have paddings between non-dynamic fields in blocks stable regardless of dynamic fields byte-sizes, we need to propose an unusual rule: first field of such block has the greatest alignment of all block fields. In other words: block is treated like composite in that regard:

```
struct X
{
    u8 a<>; // = [1]
    u8 b;   // = 2
    u32 c;  // = 3
    u8 d<>; // = [4]
    u8 e;   // = 5
    u64 f;  // = 6
};
```

This one (both arrays set with 1 element only as in comments above) has four paddings:

1. dynamic padding to align b-d block
2. to align c field
3. dynamic padding to align e-f block
4. to align f field

```
01  00  00  00  01 [00  00  00]
02 [00  00  00] 03  00  00  00
01  00  00  00  04 [00  00  00]
05 [00  00  00  00  00  00  00]
06  00  00  00  00  00  00  00
```

CHAPTER 4

Examples

This section tries to present how Prophy language and codecs may be put to work on an example with moderately complex data structure.

In each tutorial we'll:

- write a .prophy file,
- use compiler to generate chosen codec,
- use this codec to write and read data.

This is the .prophy input file used in all tutorials. It's sufficiently complex to express various Prophy features. Let's call it `values.prophy`:

```
struct Keys
{
    u32 key_a;
    u32 key_b;
    u32 key_c;
};

struct Nodes
{
    u32 nodes<3>;
};

union Token
{
    0: u32 id;
    1: Keys keys;
    2: Nodes nodes;
};

struct Object
{
    Token token;
```

(continues on next page)

(continued from previous page)

```
    i64 values<>;
    bytes updated_values<>;
};

struct Values
{
    u32 transaction_id;
    Object objects<>;
};
```

4.1 Python basics

4.1.1 Compilation

Prophy Compiler can be used to generate Python codec like this:

```
prophyc --python_out . values.prophy
```

Result is a file, which - together with Prophy Python library - forms a fully functional codec. It's called `values.py` and looks like this:

```
import prophy

class Keys(prophy.struct):
    __metaclass__ = prophy.struct_generator
    _descriptor = [('key_a', prophy.u32),
                   ('key_b', prophy.u32),
                   ('key_c', prophy.u32)]

class Nodes(prophy.struct):
    __metaclass__ = prophy.struct_generator
    _descriptor = [('num_of_nodes', prophy.u32),
                   ('nodes', prophy.array(prophy.u32, bound = 'num_of_nodes', size =
↪3))]

class Token(prophy.union):
    __metaclass__ = prophy.union_generator
    _descriptor = [('id', prophy.u32, 0),
                   ('keys', Keys, 1),
                   ('nodes', Nodes, 2)]

class Object(prophy.struct):
    __metaclass__ = prophy.struct_generator
    _descriptor = [('token', Token),
                   ('num_of_values', prophy.u32),
                   ('values', prophy.array(prophy.i64, bound = 'num_of_values')),
                   ('num_of_updated_values', prophy.u32),
                   ('updated_values', prophy.bytes(bound = 'num_of_updated_values'))]

class Values(prophy.struct):
    __metaclass__ = prophy.struct_generator
    _descriptor = [('transaction_id', prophy.u32),
                   ('num_of_objects', prophy.u32),
                   ('objects', prophy.array(Object, bound = 'num_of_objects'))]
```

4.1.2 Write and read

Now we'd need to write a small script to fill Values with some data. Values can be printed on screen as text, encoded as binary buffer. On the other communication end, this binary buffer can be used to retrieve the same data:

```
import values

x = values.Values()
x.transaction_id = 1234

empty_obj = x.objects.add()

obj = x.objects.add()
obj.token.discriminator = 'keys'
obj.token.keys.key_a = 1
obj.token.keys.key_b = 2
obj.token.keys.key_c = 3
obj.values[:] = [1, 2, 3, 4, 5]
obj.updated_values = '\x0e'

# human readable representation of data
print x

# this is how data can be encoded
data = x.encode('<')

from binascii import hexlify

print hexlify(data)

# this is how data can be decoded
x.decode(data, '<')
```

This is what print statement would generate:

```
transaction_id: 1234
objects {
  token {
    id: 0
  }
  updated_values: ''
}
objects {
  token {
    keys {
      key_a: 1
      key_b: 2
      key_c: 3
    }
  }
  values: 1
  values: 2
  values: 3
  values: 4
  values: 5
  updated_values: '\x0e'
}
```

This is how encoded data looks like:

```
d2040000 - transaction id
02000000 - number of objects

first, empty object
00000000   ...
00000000   ...
00000000   ...
00000000   ...
00000000   ...
00000000   ...
00000000   ...
00000000   ...

second, filled object
01000000 - token discriminated as keys
01000000 - key a
02000000 - key b
03000000 - key c
00000000   ...
05000000 - number of values
01000000 - value[0]
00000000   ...
02000000 - value[1]
00000000   ...
03000000 - value[2]
00000000   ...
04000000 - value[3]
00000000   ...
05000000 - value[4]
00000000   ...
01000000 - length of updated counters
0e000000 - updated counters
```

4.2 C++ full basics

4.2.1 Compilation

Prophy Compiler can be used to generate C++ full codec like this:

```
prophyc --cpp_full_out . values.prophy
```

Result is a pair of header and source files, which - together with Prophy C++ library - form a fully functional codec. They're called `values.ppf.hpp` and `values.ppf.cpp` and look like this:

```
#ifndef _PROPHY_GENERATED_FULL_values_HPP
#define _PROPHY_GENERATED_FULL_values_HPP

#include <stdint.h>
#include <numeric>
#include <vector>
#include <string>
#include <prophy/array.hpp>
#include <prophy/endianness.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <prophy/optional.hpp>
#include <prophy/detail/byte_size.hpp>
#include <prophy/detail/message.hpp>
#include <prophy/detail/mpl.hpp>

namespace prophy
{
namespace generated
{

struct Keys : public prophy::detail::message<Keys>
{
    enum { encoded_byte_size = 12 };

    uint32_t key_a;
    uint32_t key_b;
    uint32_t key_c;

    Keys(): key_a(), key_b(), key_c() { }
    Keys(uint32_t _1, uint32_t _2, uint32_t _3): key_a(_1), key_b(_2), key_c(_3) { }

    size_t get_byte_size() const
    {
        return 12;
    }
};

struct Nodes : public prophy::detail::message<Nodes>
{
    enum { encoded_byte_size = 16 };

    std::vector<uint32_t> nodes; /// limit 3

    Nodes() { }
    Nodes(const std::vector<uint32_t>& _1): nodes(_1) { }

    size_t get_byte_size() const
    {
        return 16;
    }
};

struct Token : public prophy::detail::message<Token>
{
    enum { encoded_byte_size = 20 };

    enum _discriminator
    {
        discriminator_id = 0,
        discriminator_keys = 1,
        discriminator_nodes = 2
    } discriminator;

    static const prophy::detail::int2type<discriminator_id> discriminator_id_t;
    static const prophy::detail::int2type<discriminator_keys> discriminator_keys_t;
    static const prophy::detail::int2type<discriminator_nodes> discriminator_nodes_t;
```

(continues on next page)

(continued from previous page)

```
    uint32_t id;
    Keys keys;
    Nodes nodes;

    Token(): discriminator(discriminator_id), id() { }
    Token(prophy::detail::int2type<discriminator_id>, uint32_t _1):␣
↪discriminator(discriminator_id), id(_1) { }
    Token(prophy::detail::int2type<discriminator_keys>, const Keys& _1):␣
↪discriminator(discriminator_keys), keys(_1) { }
    Token(prophy::detail::int2type<discriminator_nodes>, const Nodes& _1):␣
↪discriminator(discriminator_nodes), nodes(_1) { }

    size_t get_byte_size() const
    {
        return 20;
    }
};

struct Object : public prophy::detail::message<Object>
{
    enum { encoded_byte_size = -1 };

    Token token;
    std::vector<int64_t> values;
    std::vector<uint8_t> updated_values;

    Object() { }
    Object(const Token& _1, const std::vector<int64_t>& _2, const std::vector<uint8_t>
↪& _3): token(_1), values(_2), updated_values(_3) { }

    size_t get_byte_size() const
    {
        return prophy::detail::nearest<8>(
            values.size() * 8 + updated_values.size() * 1 + 28
        );
    }
};

struct Values : public prophy::detail::message<Values>
{
    enum { encoded_byte_size = -1 };

    uint32_t transaction_id;
    std::vector<Object> objects;

    Values(): transaction_id() { }
    Values(uint32_t _1, const std::vector<Object>& _2): transaction_id(_1), objects(_
↪2) { }

    size_t get_byte_size() const
    {
        return std::accumulate(objects.begin(), objects.end(), size_t(),␣
↪prophy::detail::byte_size()) + 8;
    }
};

} // namespace generated
```

(continues on next page)

(continued from previous page)

```
} // namespace prophy

#endif  /* _PROPHY_GENERATED_FULL_values_HPP */
```

```
#include "values.ppf.hpp"
#include <algorithm>
#include <prophy/detail/encoder.hpp>
#include <prophy/detail/decoder.hpp>
#include <prophy/detail/printer.hpp>
#include <prophy/detail/align.hpp>

using namespace prophy::generated;

namespace prophy
{
namespace detail
{

template <>
template <endianness E>
uint8_t* message_impl<Keys>::encode(const Keys& x, uint8_t* pos)
{
    pos = do_encode<E>(pos, x.key_a);
    pos = do_encode<E>(pos, x.key_b);
    pos = do_encode<E>(pos, x.key_c);
    return pos;
}
template uint8_t* message_impl<Keys>::encode<native>(const Keys& x, uint8_t* pos);
template uint8_t* message_impl<Keys>::encode<little>(const Keys& x, uint8_t* pos);
template uint8_t* message_impl<Keys>::encode<big>(const Keys& x, uint8_t* pos);

template <>
template <endianness E>
bool message_impl<Keys>::decode(Keys& x, const uint8_t*& pos, const uint8_t* end)
{
    return (
        do_decode<E>(x.key_a, pos, end) &&
        do_decode<E>(x.key_b, pos, end) &&
        do_decode<E>(x.key_c, pos, end)
    );
}
template bool message_impl<Keys>::decode<native>(Keys& x, const uint8_t*& pos, const␣
↪uint8_t* end);
template bool message_impl<Keys>::decode<little>(Keys& x, const uint8_t*& pos, const␣
↪uint8_t* end);
template bool message_impl<Keys>::decode<big>(Keys& x, const uint8_t*& pos, const␣
↪uint8_t* end);

template <>
void message_impl<Keys>::print(const Keys& x, std::ostream& out, size_t indent)
{
    do_print(out, indent, "key_a", x.key_a);
    do_print(out, indent, "key_b", x.key_b);
    do_print(out, indent, "key_c", x.key_c);
}
template void message_impl<Keys>::print(const Keys& x, std::ostream& out, size_t␣
↪indent);
```

(continues on next page)

(continued from previous page)

```
template <>
template <endianness E>
uint8_t* message_impl<Nodes>::encode(const Nodes& x, uint8_t* pos)
{
    pos = do_encode<E>(pos, uint32_t(std::min(x.nodes.size(), size_t(3))));
    do_encode<E>(pos, x.nodes.data(), uint32_t(std::min(x.nodes.size(), size_t(3))));
    pos = pos + 12;
    return pos;
}
template uint8_t* message_impl<Nodes>::encode<native>(const Nodes& x, uint8_t* pos);
template uint8_t* message_impl<Nodes>::encode<little>(const Nodes& x, uint8_t* pos);
template uint8_t* message_impl<Nodes>::encode<big>(const Nodes& x, uint8_t* pos);

template <>
template <endianness E>
bool message_impl<Nodes>::decode(Nodes& x, const uint8_t*& pos, const uint8_t* end)
{
    return (
        do_decode_resize<E, uint32_t>(x.nodes, pos, end, 3) &&
        do_decode_in_place<E>(x.nodes.data(), x.nodes.size(), pos, end) &&
        do_decode_advance(12, pos, end)
    );
}
template bool message_impl<Nodes>::decode<native>(Nodes& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Nodes>::decode<little>(Nodes& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Nodes>::decode<big>(Nodes& x, const uint8_t*& pos, const␣
↪uint8_t* end);

template <>
void message_impl<Nodes>::print(const Nodes& x, std::ostream& out, size_t indent)
{
    do_print(out, indent, "nodes", x.nodes.data(), std::min(x.nodes.size(), size_
↪t(3)));
}
template void message_impl<Nodes>::print(const Nodes& x, std::ostream& out, size_t␣
↪indent);

template <>
template <endianness E>
uint8_t* message_impl<Token>::encode(const Token& x, uint8_t* pos)
{
    pos = do_encode<E>(pos, x.discriminator);
    switch (x.discriminator)
    {
        case Token::discriminator_id: do_encode<E>(pos, x.id); break;
        case Token::discriminator_keys: do_encode<E>(pos, x.keys); break;
        case Token::discriminator_nodes: do_encode<E>(pos, x.nodes); break;
    }
    pos = pos + 16;
    return pos;
}
template uint8_t* message_impl<Token>::encode<native>(const Token& x, uint8_t* pos);
template uint8_t* message_impl<Token>::encode<little>(const Token& x, uint8_t* pos);
template uint8_t* message_impl<Token>::encode<big>(const Token& x, uint8_t* pos);
```

(continues on next page)

(continued from previous page)

```
template <>
template <endianness E>
bool message_impl<Token>::decode(Token& x, const uint8_t*& pos, const uint8_t* end)
{
    if (!do_decode<E>(x.discriminator, pos, end)) return false;
    switch (x.discriminator)
    {
        case Token::discriminator_id: if (!do_decode_in_place<E>(x.id, pos, end))␣
↪return false; break;
        case Token::discriminator_keys: if (!do_decode_in_place<E>(x.keys, pos, end))␣
↪return false; break;
        case Token::discriminator_nodes: if (!do_decode_in_place<E>(x.nodes, pos,␣
↪end)) return false; break;
        default: return false;
    }
    return do_decode_advance(16, pos, end);
}
template bool message_impl<Token>::decode<native>(Token& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Token>::decode<little>(Token& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Token>::decode<big>(Token& x, const uint8_t*& pos, const␣
↪uint8_t* end);

template <>
void message_impl<Token>::print(const Token& x, std::ostream& out, size_t indent)
{
    switch (x.discriminator)
    {
        case Token::discriminator_id: do_print(out, indent, "id", x.id); break;
        case Token::discriminator_keys: do_print(out, indent, "keys", x.keys); break;
        case Token::discriminator_nodes: do_print(out, indent, "nodes", x.nodes);␣
↪break;
    }
}
template void message_impl<Token>::print(const Token& x, std::ostream& out, size_t␣
↪indent);

template <>
template <endianness E>
uint8_t* message_impl<Object>::encode(const Object& x, uint8_t* pos)
{
    pos = do_encode<E>(pos, x.token);
    pos = do_encode<E>(pos, uint32_t(x.values.size()));
    pos = do_encode<E>(pos, x.values.data(), uint32_t(x.values.size()));
    pos = do_encode<E>(pos, uint32_t(x.updated_values.size()));
    pos = do_encode<E>(pos, x.updated_values.data(), uint32_t(x.updated_values.
↪size()));
    pos = align<8>(pos);
    return pos;
}
template uint8_t* message_impl<Object>::encode<native>(const Object& x, uint8_t* pos);
template uint8_t* message_impl<Object>::encode<little>(const Object& x, uint8_t* pos);
template uint8_t* message_impl<Object>::encode<big>(const Object& x, uint8_t* pos);

template <>
```

(continues on next page)

(continued from previous page)

```
template <endianness E>
bool message_impl<Object>::decode(Object& x, const uint8_t*& pos, const uint8_t* end)
{
    return (
        do_decode<E>(x.token, pos, end) &&
        do_decode_resize<E, uint32_t>(x.values, pos, end) &&
        do_decode<E>(x.values.data(), x.values.size(), pos, end) &&
        do_decode_resize<E, uint32_t>(x.updated_values, pos, end) &&
        do_decode<E>(x.updated_values.data(), x.updated_values.size(), pos, end) &&
        do_decode_align<8>(pos, end)
    );
}
template bool message_impl<Object>::decode<native>(Object& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Object>::decode<little>(Object& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Object>::decode<big>(Object& x, const uint8_t*& pos, const␣
↪uint8_t* end);

template <>
void message_impl<Object>::print(const Object& x, std::ostream& out, size_t indent)
{
    do_print(out, indent, "token", x.token);
    do_print(out, indent, "values", x.values.data(), x.values.size());
    do_print(out, indent, "updated_values", std::make_pair(x.updated_values.data(), x.
↪updated_values.size()));
}
template void message_impl<Object>::print(const Object& x, std::ostream& out, size_t␣
↪indent);

template <>
template <endianness E>
uint8_t* message_impl<Values>::encode(const Values& x, uint8_t* pos)
{
    pos = do_encode<E>(pos, x.transaction_id);
    pos = do_encode<E>(pos, uint32_t(x.objects.size()));
    pos = do_encode<E>(pos, x.objects.data(), uint32_t(x.objects.size()));
    return pos;
}
template uint8_t* message_impl<Values>::encode<native>(const Values& x, uint8_t* pos);
template uint8_t* message_impl<Values>::encode<little>(const Values& x, uint8_t* pos);
template uint8_t* message_impl<Values>::encode<big>(const Values& x, uint8_t* pos);

template <>
template <endianness E>
bool message_impl<Values>::decode(Values& x, const uint8_t*& pos, const uint8_t* end)
{
    return (
        do_decode<E>(x.transaction_id, pos, end) &&
        do_decode_resize<E, uint32_t>(x.objects, pos, end) &&
        do_decode<E>(x.objects.data(), x.objects.size(), pos, end)
    );
}
template bool message_impl<Values>::decode<native>(Values& x, const uint8_t*& pos,␣
↪const uint8_t* end);
template bool message_impl<Values>::decode<little>(Values& x, const uint8_t*& pos,␣
↪const uint8_t* end);
```

(continues on next page)

(continued from previous page)

```
template bool message_impl<Values>::decode<big>(Values& x, const uint8_t*& pos, const␣
↪uint8_t* end);

template <>
void message_impl<Values>::print(const Values& x, std::ostream& out, size_t indent)
{
    do_print(out, indent, "transaction_id", x.transaction_id);
    do_print(out, indent, "objects", x.objects.data(), x.objects.size());
}
template void message_impl<Values>::print(const Values& x, std::ostream& out, size_t␣
↪indent);

} // namespace detail
} // namespace prophy
```

4.2.2 Write and read

We can create a program which fills message with data, encodes it, then decodes buffer to another instance of message and prints it:

```
#include <stdio.h>
#include <iostream>
#include "values.ppf.hpp"

void print_bytes(const void* opaque_data, size_t size)
{
    const uint8_t* data = static_cast<const uint8_t*>(opaque_data);
    for (int i = 0; i < size; i++)
    {
        if (i && (i % 4 == 0))
        {
            printf("\n");
        }
        printf("%02x", data[i]);
    }
    printf("\n");
}

using namespace prophy::generated;

int main()
{
    Values msg;
    msg.transaction_id = 1234;
    msg.objects.emplace_back();
    msg.objects.emplace_back(Object{{Token::discriminator_keys_t, {1, 2, 3}}, {1, 2,␣
↪3, 4, 5}, {'\x0e'}});

    std::vector<uint8_t> data = msg.encode();
    print_bytes(data.data(), data.size());

    Values msg2;
    msg2.decode(data);
    std::cout << msg2.print();
    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

Program outputs:

```
d2040000
02000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01000000
01000000
02000000
03000000
00000000
05000000
01000000
00000000
02000000
00000000
03000000
00000000
04000000
00000000
05000000
00000000
01000000
0e000000
transaction_id: 1234
objects {
  token {
    id: 0
  }
  updated_values: ''
}
objects {
  token {
    keys {
      key_a: 1
      key_b: 2
      key_c: 3
    }
  }
  values: 1
  values: 2
  values: 3
  values: 4
  values: 5
  updated_values: '\x0e'
}
```

4.3 C++ raw basics

4.3.1 Compilation

Prophy Compiler can be used to generate C++ raw codec like this:

```
prophyc --cpp_out . values.prophy
```

Result is a file, which contains C++ structs with layout intended to be identical to Prophy wire format. It's `values.pp.hpp` and looks like this:

```
#ifndef _PROPHY_GENERATED_values_HPP
#define _PROPHY_GENERATED_values_HPP

#include <prophy/prophy.hpp>

PROPHY_STRUCT(4) Keys
{
    uint32_t key_a;
    uint32_t key_b;
    uint32_t key_c;
};

PROPHY_STRUCT(4) Nodes
{
    uint32_t num_of_nodes;
    uint32_t nodes[3]; /// limited array, size in num_of_nodes
};

PROPHY_STRUCT(4) Token
{
    enum _discriminator
    {
        discriminator_id = 0,
        discriminator_keys = 1,
        discriminator_nodes = 2
    } discriminator;

    union
    {
        uint32_t id;
        Keys keys;
        Nodes nodes;
    };
};

PROPHY_STRUCT(8) Object
{
    Token token;
    uint32_t num_of_values;
    int64_t values[1]; /// dynamic array, size in num_of_values

    PROPHY_STRUCT(4) part2
    {
        uint32_t num_of_updated_values;
        uint8_t updated_values[1]; /// dynamic array, size in num_of_updated_values
    } _2;
```

(continues on next page)

(continued from previous page)

```
};

PROPHY_STRUCT(8) Values
{
    uint32_t transaction_id;
    uint32_t num_of_objects;
    Object objects[1]; /// dynamic array, size in num_of_objects
};

namespace prophy
{

template <> Keys* swap<Keys>(Keys*);
template <> Nodes* swap<Nodes>(Nodes*);
template <> Token* swap<Token>(Token*);
template <> Object* swap<Object>(Object*);
template <> Values* swap<Values>(Values*);

} // namespace prophy

#endif  /* _PROPHY_GENERATED_values_HPP */
```

Warning: C++ raw codec assumes specific struct padding heuristics (natural alignment and special rules for nested dynamic fields) and requires enum to be represented as a 32-bit integral value. It's tested on gcc, clang and ti cgt on a couple of 32- and 64-bit platforms, but your platform ABI may break these rules.

It's accompanied by `values.pp.cpp` with endianness swap algorithms for structs and unions:

```
#include <prophy/detail/prophy.hpp>

#include "values.pp.hpp"

using namespace prophy::detail;

namespace prophy
{

template <>
Keys* swap<Keys>(Keys* payload)
{
    swap(&payload->key_a);
    swap(&payload->key_b);
    swap(&payload->key_c);
    return payload + 1;
}

template <>
Nodes* swap<Nodes>(Nodes* payload)
{
    swap(&payload->num_of_nodes);
    swap_n_fixed(payload->nodes, payload->num_of_nodes);
    return payload + 1;
}
```

(continues on next page)

(continued from previous page)

```
template <>
Token* swap<Token>(Token* payload)
{
    swap(reinterpret_cast<uint32_t*>(&payload->discriminator));
    switch (payload->discriminator)
    {
        case Token::discriminator_id: swap(&payload->id); break;
        case Token::discriminator_keys: swap(&payload->keys); break;
        case Token::discriminator_nodes: swap(&payload->nodes); break;
        default: break;
    }
    return payload + 1;
}

inline Object::part2* swap(Object::part2* payload)
{
    swap(&payload->num_of_updated_values);
    return cast<Object::part2*>(swap_n_fixed(payload->updated_values, payload->num_of_
↪updated_values));
}

template <>
Object* swap<Object>(Object* payload)
{
    swap(&payload->token);
    swap(&payload->num_of_values);
    Object::part2* part2 = cast<Object::part2*>(swap_n_fixed(payload->values, payload-
↪>num_of_values));
    return cast<Object*>(swap(part2));
}

template <>
Values* swap<Values>(Values* payload)
{
    swap(&payload->transaction_id);
    swap(&payload->num_of_objects);
    return cast<Values*>(swap_n_dynamic(payload->objects, payload->num_of_objects));
}

} // namespace prophy
```

4.3.2 Write and read

We can create a program to write data to buffer and read from it:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "values.pp.hpp"

void print_bytes(const void* opaque_data, size_t size)
{
    const uint8_t* data = static_cast<const uint8_t*>(opaque_data);
```

(continues on next page)

(continued from previous page)

```
    for (int i = 0; i < size; i++)
    {
        if (i && (i % 4 == 0))
        {
            printf("\n");
        }
        printf("%02x", data[i]);
    }
    printf("\n");
}

void print_values(Values* x, int index)
{
    Object* obj = x->objects;
    while(index)
    {
        Object::part2* obj_part2 = prophy::cast<Object::part2*>(
                obj->values + obj->num_of_values);
        obj = prophy::cast<Object*>(
                obj_part2->updated_values +
                obj_part2->num_of_updated_values);
        --index;
    }
    printf("number of values: %d\n", obj->num_of_values);
    for (int i = 0; i < obj->num_of_values; i++)
    {
        printf("value: %d\n", obj->values[i]);
    }
}

int main()
{
    void* data = malloc(1024);
    memset(data, 0, 1024);

    Values* x = static_cast<Values*>(data);
    x->transaction_id = 1234;
    x->num_of_objects = 2;

    Object* obj = x->objects;
    obj->token.discriminator = Token::discriminator_id;
    obj->token.id = 0;
    obj->num_of_values = 0;
    Object::part2* obj_part2 = prophy::cast<Object::part2*>(obj->values);
    obj_part2->num_of_updated_values = 0;

    obj = prophy::cast<Object*>(obj_part2->updated_values);
    obj->token.discriminator = Token::discriminator_keys;
    obj->token.keys.key_a = 1;
    obj->token.keys.key_b = 2;
    obj->token.keys.key_c = 3;
    obj->num_of_values = 5;
    obj->values[0] = 1;
    obj->values[1] = 2;
    obj->values[2] = 3;
    obj->values[3] = 4;
    obj->values[4] = 5;
```

(continues on next page)

(continued from previous page)

```
    obj_part2 = prophy::cast<Object::part2*>(obj->values + 5);
    obj_part2->num_of_updated_values = 1;
    obj_part2->updated_values[0] = 0x0e;

    size_t byte_size =
        reinterpret_cast<uint8_t*>(prophy::cast<Values*>(obj_part2->updated_values +␣
↪1)) -
        reinterpret_cast<uint8_t*>(x);

    printf("byte size: %d\n", byte_size);
    print_bytes(x, byte_size);
    print_values(x, 0);
    print_values(x, 1);

    return 0;
}
```

This program outputs:

```
byte size: 112
d2040000
02000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01000000
01000000
02000000
03000000
00000000
05000000
01000000
00000000
02000000
00000000
03000000
00000000
04000000
00000000
05000000
00000000
01000000
0e000000
number of values: 0
number of values: 5
value: 1
value: 2
value: 3
value: 4
value: 5
```

CHAPTER 5

Python codec

This page describes how to encode, decode and manipulate prophy messages in Python.

5.1 Compilation

Prophy Compiler can be used to generate Python codec source code from .prophy files. This generated code together with Python prophy library forms a fully functional codec.

Example compiler invocation:

```
prophyc --python_out . test.prophy
```

will result in creating `test.py`.

5.2 Generated code

File generated by Prophy Compiler defines classes representing enums, structs and unions.

Enumerators can be accessed:

```
enum Test1
{
    Test1_1 = 1,
    Test1_2 = 2,
    Test1_3 = 3
};
```

```
>>> import test
>>> test.Test1_1
1
```

Structs can be instantiated, written or read and encoded or decoded:

```
struct Test2
{
    u32 a;
};
```

```
>>> import test
>>> x = test.Test2()
>>> x.a = 42
>>> x.a
42
>>> print x
a: 42
>>> x.encode('>')
'\x00\x00\x00*'
>>> x.decode('\x00\x00\x00*', '>')
4
```

Struct enum field can be set by value or name, name can be extracted from it:

```
struct Test3
{
    Test1 a;
};
```

```
>>> import test
>>> x = test.Test3()
>>> x.a = 2
>>> x.a = 'Test1_2'
>>> x.a
2L
>>> x.a.name
'Test1_2'
```

Arrays (all kinds) may be indexed, sliced and iterated:

```
struct Test4
{
    i32 a[3];
};
```

```
>>> import test
>>> x = test.Test4()
>>> x.a[0] = 42
>>> x.a
[42, 0, 0]
>>> x.a[:] = [1, 2, 3]
>>> for value in x.a:
...     print value
...
1
2
3
```

Arrays of structs or unions can add new elements or be extended by iterables of them:

```
struct Test5
{
    Test2 a<>;
};
```

```
>>> import test
>>> x = test.Test5()
>>> y = x.a.add()
>>> y.a = 42
>>> print x
a {
  a: 42
}

>>> x.a.extend([y])
>>> print x
a {
  a: 42
}
a {
  a: 42
}
```

Optional struct fields may be set or cleared by setting with True and None:

```
struct Test6
{
    u32* a;
    Test2* b;
};
```

```
>>> import test
>>> x = test.Test6()
>>> x.a = 42
>>> x.a
42
>>> x.a = None
>>> x.a
>>> x.b = True
>>> print x.b
a: 0

>>> x.b = None
>>> print x.b
None
```

Union arm is chosen by setting discriminator with arm number or name:

```
union Test7
{
    0: u32 a;
    1: Test2 b;
};
```

```
>>> import test
>>> x = test.Test7()
```

(continues on next page)

(continued from previous page)

```
>>> x.discriminator = 0
>>> x.a = 42
>>> x.discriminator = 'b'
>>> x.b.a = 42
```

5.3 Introspection

Struct and union can be introspected by `get_descriptor` and `get_discriminated` methods:

```
struct Struct
{
    u32 a;
    u8 b[4];
};

union Union
{
    0: u32 a;
    1: u16 b;
};
```

```
>>> import test
>>> test.Struct.get_descriptor()
[<a, ('INT', 0), <class 'prophy.scalar.u32'>>, <b, ('ARRAY', 3), <class 'prophy.
↪container._array'>>]
>>> test.Struct().get_descriptor()
[<a, ('INT', 0), <class 'prophy.scalar.u32'>>, <b, ('ARRAY', 3), <class 'prophy.
↪container._array'>>]
>>> test.Union.get_descriptor()
[<a, ('INT', 0), <class 'prophy.scalar.u32'>>, <b, ('INT', 0), <class 'prophy.scalar.
↪u16'>>]
>>> test.Union().get_descriptor()
[<a, ('INT', 0), <class 'prophy.scalar.u32'>>, <b, ('INT', 0), <class 'prophy.scalar.
↪u16'>>]
>>> test.Union().get_discriminated()
<a, ('INT', 0), <class 'prophy.scalar.u32'>>

>>> field_desc = test.Struct.get_descriptor()[0]
>>> field_desc.name
'a'
>>> field_desc.kind
('INT', 0)
>>> field_desc.type
<class 'prophy.scalar.u32'>
>>> import prophy
>>> field_desc.kind == prophy.kind.INT
True
```

5.4 Packed mode

Python generated message descriptors may be altered to inhibit padding by inheriting from `struct_packed` instead of `struct`. Following message would be encoded as 6 bytes:

```
class PackedMessage(prophy.struct_packed):
    __metaclass__ = prophy.struct_generator
    _descriptor = [('x', prophy.u8),
                   ('y', prophy.u32),
                   ('z', prophy.u8)]
```

Warning: Mixing `struct_packed` with nested `struct` and otherwise yields undefined behavior.

CHAPTER 6

C++ full codec

This page describes how to manipulate prophy messages using C++ codec. Codec uses value-semantics types with integral, `array`, `vector` and `optional` fields to represent prophy structs and unions. These types have means to size, encode, decode and print represented messages.

Codec is nicknamed "full" as opposed to "raw" one, which allows to form an encoded message way faster but requires moderate amount of user code and attention.

6.1 Compilation

Prophy Compiler can be used to generate C++ full codec source code from .prophy files. This generated code together with C++ prophy header-only library can be used to handle Prophy messages.

Example compiler invocation:

```
prophyc --cpp_full_out . test.prophy
```

will result in creating `test.ppf.hpp` and `test.ppf.cpp`.

6.2 Generated code

All types in generated C++ code are enclosed in `prophy::generated` namespace.

Note: `array` and `optional` class templates shipped with C++ prophy library are derived from `std` and `boost` implementations, for the sake of making library independent of C++11 std lib or boost libraries.

Enums are represented as regular C++ enums:

```
enum Test1
{
    Test1_1 = 1,
    Test1_2 = 2,
    Test1_3 = 3
};
```

```
enum Test1
{
    Test1_1 = 1,
    Test1_2 = 2,
    Test1_3 = 3
};
```

Structs and unions are represented as value-semantics types inheriting self-specialized `prophy::detail::message` class template (CRTP), thus fulfill the API of prophy message:

```
template <class T>
struct message
{
    template <endianness E>
    size_t encode(void* data) const;

    size_t encode(void* data) const;

    template <endianness E>
    std::vector<uint8_t> encode() const;

    std::vector<uint8_t> encode() const;

    template <endianness E>
    bool decode(const void* data, size_t size);

    bool decode(const void* data, size_t size);

    template <endianness E>
    bool decode(const std::vector<uint8_t>& data);

    bool decode(const std::vector<uint8_t>& data);

    std::string print() const;
};
```

Encode and decode can be used as method templates to choose specific `prophy::endianness` to process data or as regular methods, which use `native` endianness (actual machine endianness):

```
enum endianness
{
    native,
    little,
    big
};
```

Structs are represented as C++ structs with corresponding fields:

```
struct Test2
{
```

(continues on next page)

(continued from previous page)

```
    u32 a;
};
```

```
struct Test2 : public prophy::detail::message<Test2>
{
    enum { encoded_byte_size = 4 };

    uint32_t a;

    Test2(): a() { }
    Test2(uint32_t _1): a(_1) { }

    size_t get_byte_size() const
    {
        return 4;
    }
};
```

Arrays are represented as `array` or `vector`. In case of limited arrays - exceeding limit is not prohibited, but encoding will serialize only elements up to limit:

```
struct Test8
{
    i32 a[3];
    i32 b<>;
    i32 c<3>;
    i32 d<...>;
};
```

```
struct Test8 : public prophy::detail::message<Test8>
{
    enum { encoded_byte_size = -1 };

    array<int32_t, 3> a;
    std::vector<int32_t> b;
    std::vector<int32_t> c; /// limit 3
    std::vector<int32_t> d; /// greedy

    Test8(): a() { }
    Test8(const array<int32_t, 3>& _1, const std::vector<int32_t>& _2, const␣
↪std::vector<int32_t>& _3, const std::vector<int32_t>& _4): a(_1), b(_2), c(_3), d(_
↪4) { }

    size_t get_byte_size() const
    {
        return b.size() * 4 + d.size() * 4 + 32;
    }
};
```

Optional fields are represented by `optional` template class:

```
struct Test6
{
    u32* a;
    Test2* b;
};
```

```
struct Test6 : public prophy::detail::message<Test6>
{
    enum { encoded_byte_size = 16 };

    optional<uint32_t> a;
    optional<Test2> b;

    Test6() { }
    Test6(const optional<uint32_t>& _1, const optional<Test2>& _2): a(_1), b(_2) { }

    size_t get_byte_size() const
    {
        return 16;
    }
};
```

Union representation is similar to struct one - it contains all arms as independent fields. Depending on current value of discriminator, chosen arm will be encoded or printed. Decoding overwrites discriminator as well as decoded arm:

```
union Test7
{
    0: u32 a;
    1: Test2 b;
};
```

```
struct Test7 : public prophy::detail::message<Test7>
{
    enum { encoded_byte_size = 8 };

    enum _discriminator
    {
        discriminator_a = 0,
        discriminator_b = 1
    } discriminator;

    static const prophy::detail::int2type<discriminator_a> discriminator_a_t;
    static const prophy::detail::int2type<discriminator_b> discriminator_b_t;

    uint32_t a;
    Test2 b;

    Test7(): discriminator(discriminator_a), a() { }
    Test7(prophy::detail::int2type<discriminator_a>, uint32_t _1):␣
↪discriminator(discriminator_a), a(_1) { }
    Test7(prophy::detail::int2type<discriminator_b>, const Test2& _1):␣
↪discriminator(discriminator_b), b(_1) { }

    size_t get_byte_size() const
    {
        return 8;
    }
};
```

`discriminator_<field_name>_t` variables are meant to facilitate C++11 brace-enclosed initialization:

```
Test7 x{Test7::discriminator_a_t, 42};
Test7 y{Test7::discriminator_b_t, {13}};
```

CHAPTER 7

C++ raw codec

This page describes how to manipulate prophy messages using C++ structs. This is the fastest option, since it allows to read fields directly from datagram. It's also most environment-tolerant, since it reduces necessary operations to memory reads and writes, and pointer arithmetics.

Codec is compliant with C++98 and later.

Warning: C++ raw codec assumes specific struct padding heuristics (natural alignment and special rules for nested dynamic fields) and requires enum to be represented as a 32-bit integral value. It's tested on gcc, clang and ti cgt on a couple of 32- and 64-bit platforms, but your platform ABI may break these rules.

7.1 Compilation

Prophy Compiler can be used to generate C++ raw codec source code from .prophy files. This generated code together with C++ raw prophy header-only library can be used to write and read Prophy messages.

Example compiler invocation:

```
prophyc --cpp_out . test.prophy
```

will result in creating `test.pp.hpp` and `test.pp.cpp`. Header file contains struct definitions. Implementation file contains endianness swap algorithms for structs and unions.

7.2 Generated code

Enums are represented as regular C++ enums:

```
enum Test1
{
```

(continues on next page)

(continued from previous page)

```
    Test1_1 = 1,
    Test1_2 = 2,
    Test1_3 = 3
};
```

```
enum Test1
{
    Test1_1 = 1,
    Test1_2 = 2,
    Test1_3 = 3
};
```

Structs are also represented as regular structs:

```
struct Test2
{
    u32 a;
};
```

```
PROPHY_STRUCT(4) Test2
{
    uint32_t a;
};
```

Arrays are represented as optional element counter and C++ array depending on type:

- fixed: C++ arrays,
- dynamic: uint32_t element counter followed by a 1-size C++ array representing variable length array,
- limited: uint32_t element counter followed by a C++ array,
- greedy: 1-size C++ array representing variable length array.

```
struct Test8
{
    i32 a[3];
    i32 b<>;
    i32 c<3>;
    i32 d<...>;
};
```

```
PROPHY_STRUCT(4) Test8
{
    int32_t a[3];
    uint32_t num_of_b;
    int32_t b[1]; /// dynamic array, size in num_of_b

    PROPHY_STRUCT(4) part2
    {
        uint32_t num_of_c;
        int32_t c[3]; /// limited array, size in num_of_c
        int32_t d[1]; /// greedy array
    } _2;
};
```

Above snippet shows how Prophy handles multiple dynamic fields in single struct: it defines inner structs with remaining fields. Field _2 is not meant to be written, it's there merely to get the main struct alignment right.

Optional struct fields are represented by uint32_t alias indicating value presence, and value itself. Fact that optional field type may not be dynamic simplifies things:

```
struct Test6
{
    u32* a;
    Test2* b;
};
```

```
PROPHY_STRUCT(4) Test6
{
    prophy::bool_t has_a;
    uint32_t a;
    prophy::bool_t has_b;
    Test2 b;
};
```

Union representation is similar to optional field, with discriminator in place of presence indicator. Union arms are accessible as members of unnamed inner union:

```
union Test7
{
    0: u32 a;
    1: Test2 b;
};
```

```
PROPHY_STRUCT(4) Test7
{
    enum _discriminator
    {
        discriminator_a = 0,
        discriminator_b = 1
    } discriminator;

    union
    {
        uint32_t a;
        Test2 b;
    };
};
```

7.3 How is natual alignment layout ensured?

By the means of explicit filling fields and compiler attributes setting type alignments.

```
union PaddedUnion
{
    0: u32 a;
    1: u64 b;
};

struct PaddedStruct
{
    u8 x;
```

(continues on next page)

(continued from previous page)

```
    PaddedUnion y;
};
```

```
PROPHY_STRUCT(8) PaddedUnion
{
    enum _discriminator
    {
        discriminator_a = 0,
        discriminator_b = 1
    } discriminator;

    uint32_t _padding0; /// manual padding to ensure natural alignment layout

    union
    {
        uint32_t a;
        uint64_t b;
    };
};

PROPHY_STRUCT(8) PaddedStruct
{
    uint8_t x;
    uint8_t _padding0; /// manual padding to ensure natural alignment layout
    uint16_t _padding1; /// manual padding to ensure natural alignment layout
    uint32_t _padding2; /// manual padding to ensure natural alignment layout
    PaddedUnion y;
};
```

7.4 How to size message?

Codec lacks support for sizing now. You need to be creative. You have a couple of options:

- you can allocate buffer large enough to hold any of your messages, write message and see where you are,
- you can also calculate exact size using sizeof operator, but need to be *careful with padding*:

```
PROPHY_STRUCT(4) X
{
    uint32_t num_of_x;
    uint32_t x[1]; /// dynamic array, size in num_of_x
};

/// assuming you want to write 5 elements
size_t msg_size = sizeof(X) - sizeof(uint32_t) + 5 * sizeof(uint32_t);
```

7.5 How to get past dynamic fields?

You'll want to use `prophy::cast` function to get a pointer of next field's type, aligned to that type. Othwerise you'll have problems either with alignment or fulfilling *wire format expectations*:

```
struct X
{
    u32 a<>;
    u32 b<>;
};
```

```
PROPHY_STRUCT(4) X
{
    uint32_t num_of_a;
    uint32_t a[1]; /// dynamic array, size in num_of_a

    PROPHY_STRUCT(4) part2
    {
        uint32_t num_of_b;
        uint32_t b[1]; /// dynamic array, size in num_of_b
    } _2;
};
```

```
X* x = static_cast<X*>(malloc(1024));
x->num_of_a = 3;
x->a[0] = 1;
x->a[1] = 2;
x->a[2] = 3;
X::part2* xp2 = prophy::cast<X::part2*>(x->a + 3);
xp2->num_of_b = 2;
xp2->b[0] = 4;
xp2->b[1] = 5;
```

7.6 How to swap message endianness?

Problem exists e.g. when you try to read message encoded on big endian system on little endian system. It won't work without swapping *multi-byte numeric values*.

This is what the implementation file and `prophy::swap` function are for:

```
EndiannessSensitive* msg = ...
prophy::swap(msg);
```

Warning: Swapping works only from foreign endianness to native. Swapping the other way around results in undefined behavior. It needs to be an interface agreement or extraneous data which lets receiver know endianness of data before reading it.

If you don't need endianness swapping in your application, disregard the implementation file altogether.

CHAPTER 8

Other schema languages

Prophy Compiler may parse other schemas than the regular one.

8.1 Patch

Since other schemas are limited to a subset of Prophy schema features, they may require *patching* in order to achieve expected form.

Patch file can have patch rules and blank lines. If message is not found, compilation is still successful. If message is found but rule does not apply, compilation fails.

If one of rules changes name of node, all rules for original name still apply, but rules for new name do not apply.

There are following patch rules:

- `<MESSAGE_NAME> type <FIELD_NAME> <NEW_TYPE>`

  Changes type of message field.

- `<MESSAGE_NAME> insert <FIELD_INDEX> <FIELD_NAME> <FIELD_TYPE>`

  Inserts a new field in message. Index 0 puts field at the beginning, index larger than number of fields, e.g. 999 puts field at the end. Newly inserted field is a scalar, not array. Turning it into an array requires another instruction.

- `<MESSAGE_NAME> remove <FIELD_NAME>`

  Removes field from message.

- `<MESSAGE_NAME> dynamic <FIELD_NAME> <SIZE_FIELD_NAME>`

  Makes field a dynamic array by associating it with a size field.

- `<MESSAGE_NAME> greedy <FIELD_NAME>`

  Makes field a greedy array. Greedy array doesn't have a size field, codecs deduce such array size by parsing message until all bytes are exhausted. There can be only one greedy field in any message as last field.

- `<MESSAGE_NAME> static <FIELD_NAME> <ARRAY_SIZE>`

  Makes field a fixed array. Only fixed size types can be fixed arrays.

- `<MESSAGE_NAME> limited <FIELD_NAME> <SIZE_FIELD_NAME>`

  Makes field a limited array, a combination of fixed and dynamic array. Field needs to be a fixed array to begin with. Limited array may have varying number of elements - up to limit - but it always has fixed size.

- `<MESSAGE_NAME> struct`

  Changes union to struct with fields of respective types.

- `<NODE_NAME> rename <NODE_NAME>`

  Changes node name.

- `<MESSAGE_NAME> rename <FIELD_NAME> <NEW_FIELD_NAME>`

  Changes struct or union member name.

8.2 Sack schema

Prophy messages can be defined in a C++ language subset, called Sack.

In this mode Prophy messages are defined by C++ structs and classes, which may contain enums, typedefs and unions.

With this definition:

```
// test.hpp
#include <stdint.h>
struct Test {
    uint32_t num_of_x;
    uint32_t x[1];
};
```

and this patch:

```
//patch.txt
Test dynamic x num_of_x
```

this command:

```
prophyc --sack --patch patch.txt --python_out . test.hpp
```

creates a Python codec equivalent to:

```
struct Test
{
    u32 x<>;
};
```

8.3 Isar schema

This schema is based on xml.

Isar xml may contain definitions of messages, enums, constants, typedefs and unions.

With this definition:

```
// test.xml
<xml>
    <struct name="Test">
        <member name="x" type="u32">
            <dimension isVariableSize="true"/>
        </member>
    </struct>
</xml>
```

this command:

```
prophyc --isar --python_out . test.xml
```

generates identical codec to one from previous example.

8.4 Mixing Isar and Sack

Since version `1.1.0` there is a possibility to implicitly include Isar definitions to Sack schema compilation. That's useful if the Sack code (c++) depends on many definitions originated in Isar XMLs. Probably in normal way these are supplied in C++ form by monstrual build system. This feature allows to skip the 'build system' stage and supplement definitions directly from Isar XMLs.

Having Isar code defining `Test` type from previous subchapter (the `test.xml` file) you can successfully compile it with such a Sack code:

```
// mixing.hpp
#include <stdint.h>

struct MixingTest
{
    int32_t field_a;
    Test field_b;
};
```

Note: There is missing definition of `Test` and that would fail to compile with any c/c++ compiler. No includes are defined, nor exising in filesystem, besides the `test.xml`. It's up to user to prepare c/c++ file, e.g. remove all problematic inclusions (if any). Types from isar doesn't need to be explicitly included in the compiled C/C++ code.

Such a prophyc call:

```
prophyc --sack --include_isar test.xml --python_out . mixing.hpp
```

... will generate two python files.

1. Comming from Isar: `test.xml` -> `test.py`:

```
# test.py
import prophy

class Test(prophy.with_metaclass(prophy.struct_generator, prophy.struct)):
    _descriptor = [('x_len', prophy.u32),
                   ('x', prophy.array(prophy.u32, bound = 'x_len'))]
```

2. And from the sack code `mixing.hpp` -> `mixing.py`:

```
# mixing.py
import prophy

from test import *

class MixingTest(prophy.with_metaclass(prophy.struct_generator, prophy.struct)):
    _descriptor = [('field_a', prophy.i32),
                   ('field_b', Test)]
```