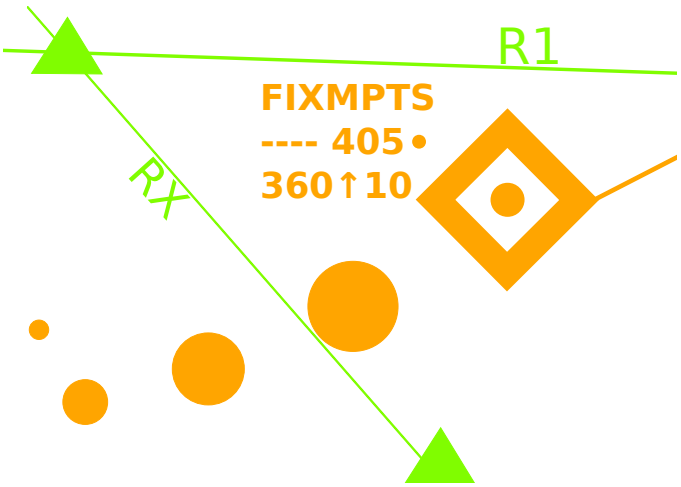

Document ID: FixMPTS-AsterixLib-ADD

Version 0.1

August 2021

Revision History

Name	Date	Reason For Change	Version
F. Kreuter	01.08.2021	Initial Version	V 0.1

Contents

1 Purpose

The aim of this document is to highlight the architectural design of the FixMPTS Aterix Library. It is intended for developers who want to make use of the library in their respective project and for those who plan to contribute or enhance the library as such. This document shall give only an overview. For a more in-depth understanding please refer to the introductory document which also contains some examples.

The document is split into two parts. The first focuses on the architectural design on the level of Asterix Categories, where as the second part will go into the design within the individual Asterix Categories.

1.1 Principles

To achieve a high level of compatibility the library is designed with as little usage of external resources as possible. Thus where ever possible it depends only on the C++ standard library.

2 References

List of documents this ADD has references to.

[1] Eurocontrol *EUROCONTROL Specification for Surveillance Data Exchange -Part 1.*
[*All Purpose Structured EUROCONTROL Surveillance Information Exchange(ASTERIX)*]
Edition 3.0, 07/12/2020, DocID: EUROCONTROL-SPEC-0149

3 Functional Overview

In the following chapters the different architectural aspects of the library will be highlighted.

3.1 Asterix Category Level

The main task for the Asterix Category class would be to encode and decode the individual messages. To make this statement the common base class to all Asterix Categories already provides those methods. This does not mean that this is all there is. The individual child classes are free and do implement convenience methods. This is, as only those child classes know the best data type to process it and therefore it is left to them to decide which other encode or decode method they want to provide.

In order to add processing of a new Asterix Category or a new version of an existing category, it is foreseen to either inherit from the Asterix Base class or the individual child class, which needs enhancement. This approach ensures tha any extension of the library is backward compatible and integrates seamlessly into existing applications.

The currently available structure of the Asterix Categories is visible in Fig. 1. For the child classes the diagram will only show those functions and attributes not inherited and those the author thought would carry a significant importance.

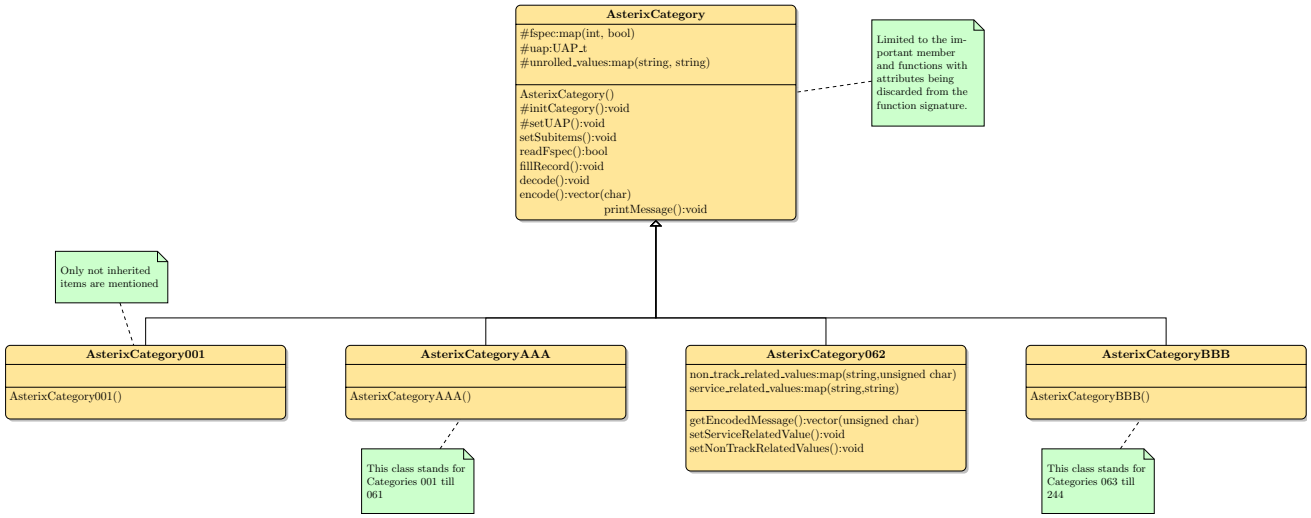

Figure 1: Asterix Category Class Diagram

3.2 Asterix Item Level

Looking down from the Asterix Category, the next level is the Asterix Item. Each Asterix Category consists of a defined set of Asterix Items. Within the Asterix Category each Asterix Item is defined by its number of order. In addition the item comes with a unique name. The uniqueness is not ensured by the item itself, but is up to the developer. The naming schema in place is thee digits for the category the item is in, plus a slash, plus three digits for the item itself (e.g. 001/101). The general architecture of the Asterix Items can be seen in Fig. 2.

As with the Asterix Categories there is a so called base item, which shall be the root parent to all individual Asterix Items. It contains already the name, which is mandatory, along with the read function. This read function must be implemented by the derived classes. The read function is supposed to digest the binary data that represents the item from the binary input stream and to return only those bytes belonging to this item. Purpose of the item is not to interpret the binary data but just to extract it from the message. Interpretation will be the workload of the Asterix Subitem classes.

From Fig. 2 it can be seen that a set of different Asterix Items already exists. These are all items currently defined in [1]. There purpose, in what the architectural overview is concerned, will be be detailed in the paragraph below.

For every item, handling of the input data is different. There are Asterix Items where the length of the bytes to read is known beforehand. Others are either a small Asterix Category with FSPEC and sub items in itself. And yet another kind is of variable length of repeated length not known during compile time but calculated during runtime from the message itself. The focus clearly was to bundle every item with equal byte reading into one class.

These ready to use Asterix Items are now added to the Asterix Category by populating the so called *uap* variable. The Asterix Category Base class will then take care of calling the read function of the item as soon as the input data is available. It will also take of presenting the input data to the Asterix Items in the way it would expect the data. This reduces the task of implementing a new category basically to defining how the category looks like. Most of the reading, encoding and decoding will then already be implemented by the parent class. Only in very view cases the developer might have the need to override the behaviour of the parent class.

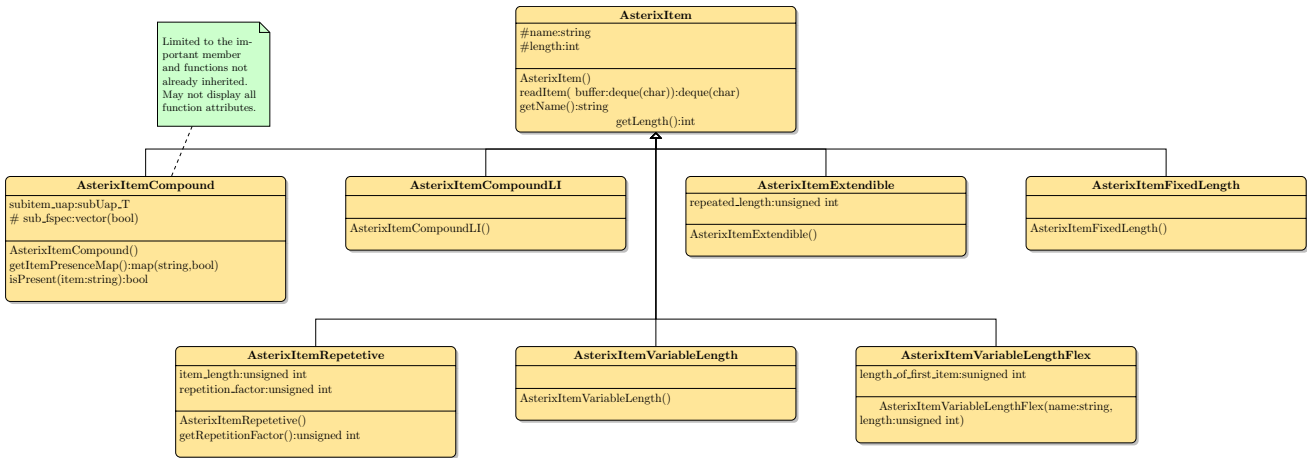

Figure 2: Asterix Item Class Diagram

3.3 Asterix Sub-Item Level

Purpose of the Asterix Sub-Item is to provide the interpretation of the Asterix Item. The Asterix Items work on byte level, whereas the Asterix Sub-Item work on bit level. Thus one Asterix Item can be comprised of several sub items. So the sub item is the smallest level of granularity. The architectural structure the all available sub items can be found in Fig. 3.

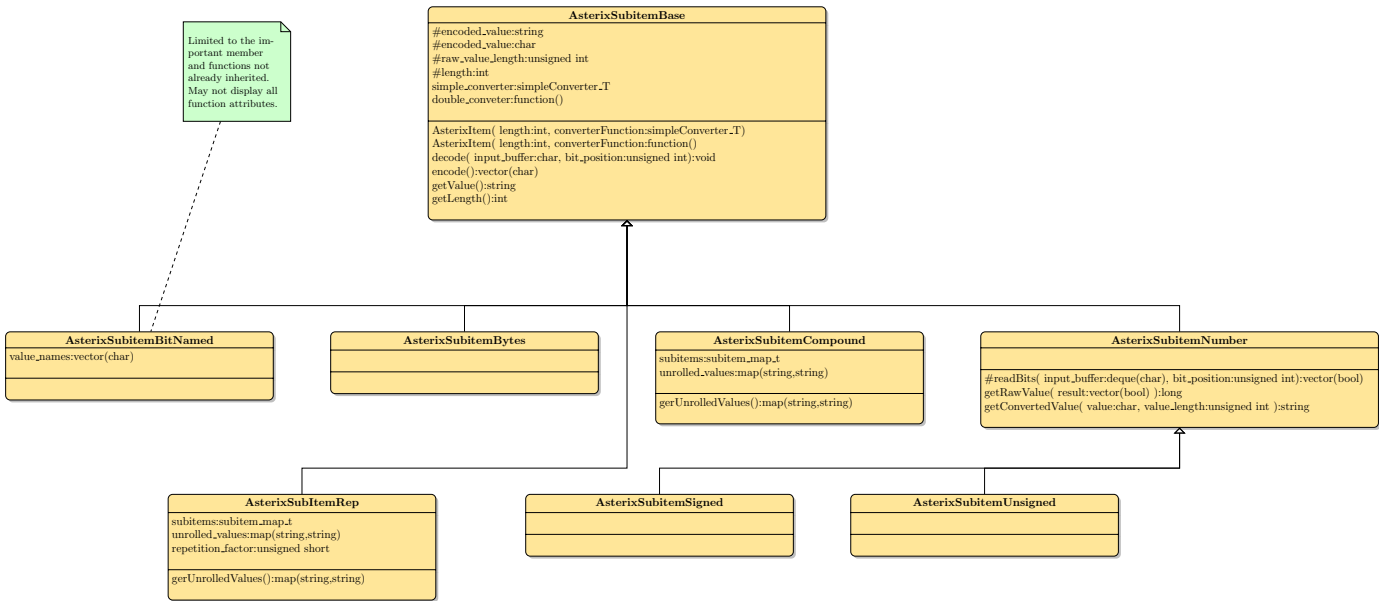

Figure 3: Asterix Sub-Item Class Diagram

Again, the AsterixSubItemBase class provides all the functions and attributes common to all Asterix Sub-Items. As can be seen from Fig. 3 this is the pace where the actual value is stored. After calling encode method, the encoded and ready to use value is stored in *encoded_value* variable. In order to get the interpretation of the raw bits right, it is important to define the length of the item and its interpretation first. In interpretation of the raw data is defined by the means of converter function, detailed in chapter 4.4. Depending on the complexity of the conversion this can either be a *simple_converter* or a *double_double* converter. To convert the value even further into a human readable format, this is the place where the different Sub-Item classes come into play.

The derived classes have two purposes. On is to define the way of reading the raw bits of the Asterix item and the other is to define the human readable representation of the value.

3.4 Sub-Item Converter Functions

The converter are the means to translate the binary input data to the expected output format. There are converters for nearly every aspect of the Asterix Protocol. They are grouped by the output type. Thus all conversions resulting in a floating

point number are currently grouped into the *DoubleConveter*. A detailed overview of what they provide will be presented in the following sections. For all converter the design is to have all conversion methods static and to avoid having any state. Thus the idea is to just provide the translation without anything on top.

3.4.1 Common Converter

The CommonConverter as shown in fig. 4 is the most basic converter available. Its sole purpose is to do nothing. It just reads the input bits and converts them to a string. This converter is to used if no interpretation of the input data is required.

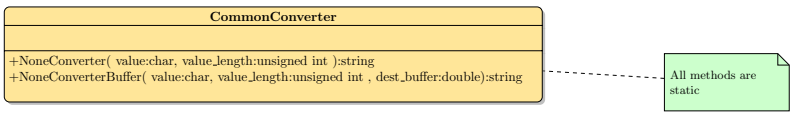

Figure 4: Common Converter

3.4.2 Double Converter

As visible from fig. 5, the DoubleConverter already provides a whole lot of functions. Its aim is to convert the input bits into whatever double precision output is needed. It already contains static conversion functions for example for a quarter flight level, various WGS84 converter, or fractional values. The details on how the different converter worked and what their output will look like, can be found in the code documentation. The DoubleConverter class shall hold all converter functions which have the output type double.

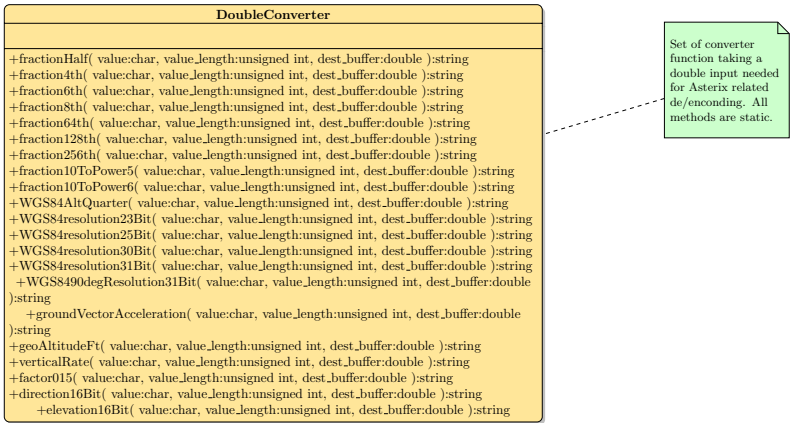

Figure 5: Double Converter

As this converter will deal with signed double values a dedicated converter for unsigned double values is presented in the next section.

3.4.3 Unsigned Double Converter

In case the the converted value shall be of type double but the valid range is only within the positive number area, the UnsignedDoubleConverter is the one to use. This is enforced by interpreting the input bits as unsigned number before applying the conversion. Shown fig. 6 one can see all static converter functions already available. Most of them deal with fraction of various input bit length.

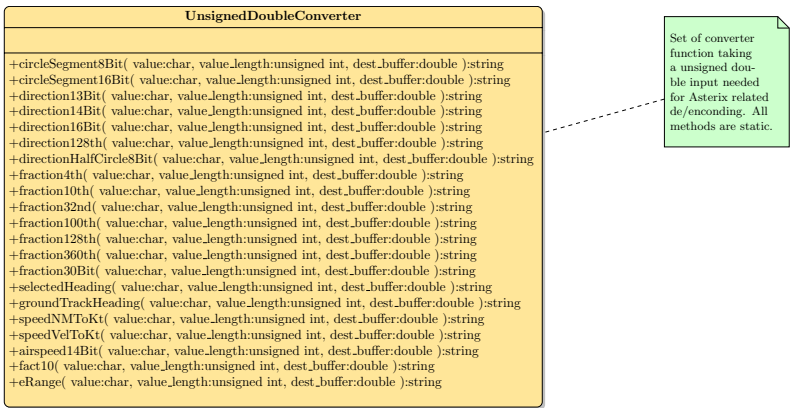

Figure 6: Unsigned Double Converter

3.4.4 Integer Converter

For all input bits that are expected to be converted to SignedIntegers, the IntegerConverter class is the place to put them. Fig. 7 shows the current static functions available. The details on the two currently available functions can be found in the code documentation.

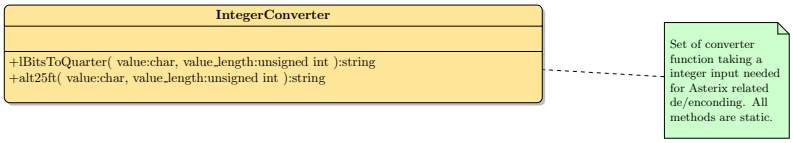

Figure 7: Integer Converter

3.4.5 String Converter

For the very few cases where the input bit cannot be converted directly into characters, the StringConverter exists. Visible in fig. 8, it contains functions for output alphabets which do not necessarily follow the standard ASCII alphabet. The details of the converter functions are available from the code documentation.

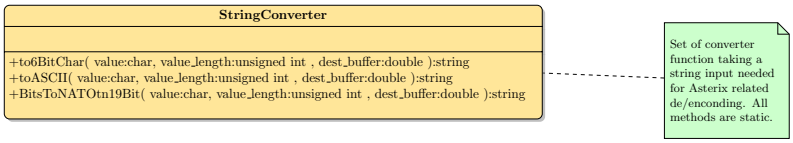

Figure 8: String Converter