The MultiFrame Metaformat Specification (2.2)

S. Anvar, R. Fox, B. Raine, F. Saillant, N. Usher

1 Goal and Features

The main goal of the MultiFrame metaformat (MFM) is to be able to define binary formats for data acquisition and serialization that are self-contained, layered, adapted to network transfers and evolving.

- Self-containment is achieved through the use of in-frame metadata information.
- Layering is achieved by allowing a data frame to include other data frames.
- Network adaptation is achieved by allowing limited necessary decoding (basically byte counts) for transmission purposes.
- Extensibility is achieved by including byte counts for frame elements (e.g. Headers), version fields and additive extensibility rules that allow both backward and forward compatibility with user software.

2 Format Specification

2.1 Numbers and Sizes

All sizes are in **BYTES** unless specified otherwise, a BYTE being an 8-bit information unit.

By default, all variables and fields are organized as Big Endian variables (most significant byte comes first). However, an endianness bit is foreseen as part of the Frame metadata.

2.2 Definitions

A "Frame" is a self-sufficient, continuous set of bytes composed of two sections:

1) A "Header" section,
2) A "Data" section.

2.2.1 The Data Section of a Frame

The Data section of a frame is composed of "Items" *of the same type* and a possible reserved ending section called the "Data Reserve." There are 3 basic types of Items:

1) The "Fixed size" Item,
2) The "Variable size" Item,
3) The "Blob" Item

A Fixed size Item is a block of data made of a fixed number of bytes. Typically, a C structure would appear as a Fixed size Item. A Frame whose Data section contains Fixed size Items is called a "Basic Frame." Such a Basic Frame contains any number of Items provided that they are all of the same size and appear consecutively in the byte stream of the Frame.

A Variable size Item corresponds to a block of data whose size is variable. In our case, *it is implemented as a Frame*, i.e. any Variable size Item is itself a Frame as defined by this specification.

A Blob Item corresponds to an opaque block of data whose internal structure is not described in the Header.

2.2.2 The Header Section of a Frame

The Header section of any Frame contains the information describing:

1) The endianness, "blob-ness" and unit block size (a power of 2 number of bytes) of the Frame .
2) The total size of the Frame *as a number of unit blocks*.
3) The data source id referring to the (type of) apparatus that has produced the Frame data.
4) The Frame type, encoding both the Header type and the data type that constitute the Frame.
5) The revision number for that Frame type.

The Header section of a non-blob Frame contains the former fields followed by:

6) The total size of the Header itself *as a number of unit blocks*.
7) The byte size of the Frame Item, size 0 corresponding to Variable size Items.
8) The number of Items contained within the Frame.

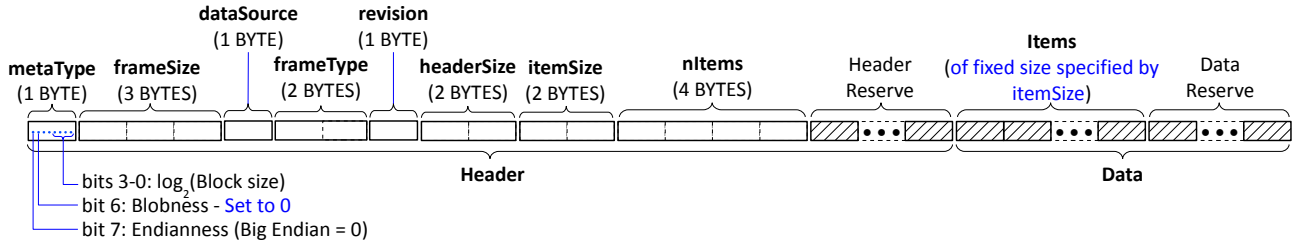
The Header Section of a non-blob Frame can also have an ending section called the “Header Reserve”.

2.3 Detailed Structure

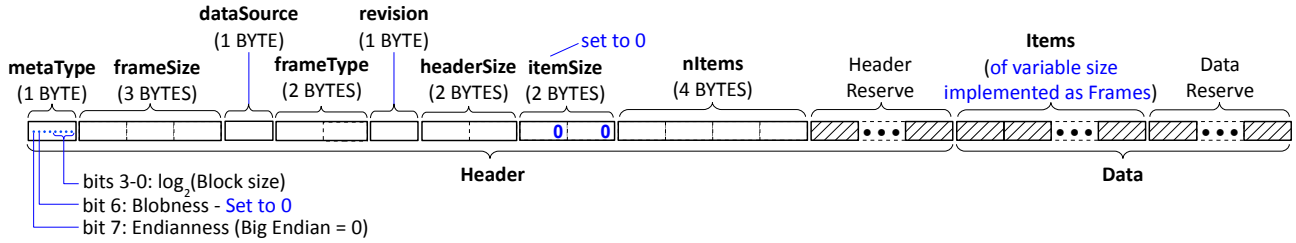
Blob Frames are characterized by a minimal Header and an opaque Data part. As a consequence, they do not address the software compatibility issues associated with the evolution of Header formats within a given application. For this reason, Basic and Layered Frames are recommended over Blob Frames. The use of Blob Frames should be reduced to frame types whose structure are not well adapted to Basic or Layered Frames, for instance, when the Data part of a frame type is too small as compared to the Header, so that a non-blob Header would be too much of an overhead.

2.3.1 The Structure of the 3 Frame Meta-types

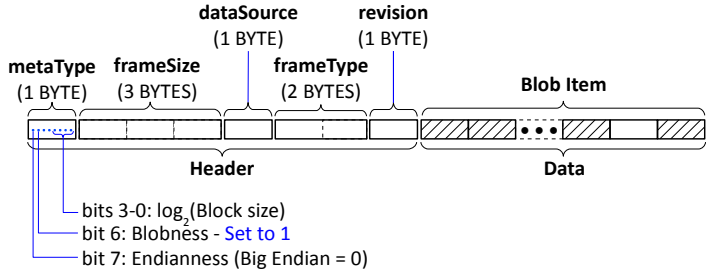
Basic Frame



Layered Frame



Blob Frame



2.3.2 Semantics of the Frame Fields

Field Name	Field Size	Field Description
metaType	**1 BYTE**	**`ISLEND`** ≡ Bit 7: endianness. Set to 1 if Frame fields are Little Endian. **`ISBLOB`** ≡ Bit 6: "blobness." Set to 1 if Frame is a Blob Frame. **`P2BLCK`** ≡ Bits 3-0: Unit block size (in bytes) as a power of 2. The frame size and header size are aligned on this.
frameSize	**3 BYTES**	This field represents the Frame size *expressed as an integer number of unit blocks*. This size is the addition of the sizes of the Header section and the Data section.
dataSource	**1 BYTE**	This field represents the source from which the data of this Frame were produced. It would typically be used to refer to the detector or detector type from which the Frame data have been extracted.
frameType	**2 BYTES**	This field is meant to receive the type of this Frame: Frames differ in type if their Items or Header or both differ in any way that breaks revision rules.
revision	**1 BYTE**	This field is meant to receive the Frame format revision: Two formats which differ only by revision have the same type and are both forward and backward compatible (e.g. old software based on format 2.3 can partially read format 2.5 without failing).
headerSize	**2 BYTES**	This field contains the total size of the Header section *expressed as an integer number of unit blocks*. The Header section consisting in the fields: *metaType, frameSize, dataSource, frameType, revision, headerSize* itself, *itemSize*, *nItems* and a possible "reserve" sub-section for the Header.
itemSize	**2 BYTES**	• set to 0 for layered Frames, i.e. when the Items are themselves Frames; • set to the Item size *in bytes* when the Frame is constituted of fixed size Items.
nItems	**4 BYTES**	This field receives the total number of Items in the Frame. WARNING: *itemSize* × *nItems* is not necessarily equal to the full size of the Data section; it can be smaller if the Data section ends with a "Data Reserve" sub-section.

2.3.3 Additional Remarks

The "reserve" sub-sections at the end of both Data and Header sections of non-blob Frames are not mandatory. They have been provided for two main purposes:

1) The user can extend the Header section in order to include application-specific fields, provided that the value of the *headerSize* field takes the extension into account. For any software that is not aware of these extensions, they appear as a mere "reserve" section to be skipped. See next section for format revision rules that allow backward and forward compatibility between different revisions of decoding software in an application.
2) The user can add padding in the Reserve subsection for alignment purposes or other low-level constraints typically encountered on embedded/real-time platforms. In particular, if the total byte size of the Data section or the Header section and its possible application-specific extensions do not add up to an exact multiple of the unit block size defined (as a power of 2) in the *metaType* field, the section must be padded accordingly (remember that the *headerSize* and *frameSize* fields are expressed as a number of unit blocks).

As explained in the Frame fields semantics table, each frameType refers to a unique couple of Header type and Item type. Consequently, two different frameTypes may refer to the same Item type and differ only because of the Header or vice-versa. However, the Header section of a Blob Frame is not allowed to evolve.

The size of an Item type as implemented by an application may be strictly smaller than the size indicated in the itemSize field, allowing any fixed Item to include trailing "reserved" bytes. This situation particularly arises when the actual Items of the frame belong to a revision that is higher than the revision for which the application was compiled (see Format Revisions further).

3 Format Revisions

The MFM has been designed with the purpose of allowing easy evolution in data formats so that users can provide for software and system enhancements without the need to force all related software to evolve at the same time. However, the feature can be implemented only if, from the beginning of software/hardware development, the following Format Revision Rules are enforced in Data format definitions for all Frame meta-types and and in Header format definition for all non-blob Frames.

3.1 Format Revision Rules

1) Format revisions can be implemented for both Header and Data sections in Basic and Layered Frames, but only in Data section for Blob Frames.
2) Format revisions apply only to application specific formatting: no change is allowed for the fields defined by MFM in section 2. In particular, the field structure *metaType, frameSize, dataSource, frameType, dataSource, revision, headerSize*, *itemSize* and *nItems* must remain untouched.
3) An increase in format ***version*** means that at least a new ***frame type*** has been added to the format, implying that old software will not be able to decode some Frame types in the new format, although it will be able to skip, store, transfer or even split such Frames, basically using all size information in the Header. Such changes must increment the format version, e.g. 2.6 to 3.0.
4) An increase in format ***revision*** means that new fields have been added (to Header or Item), former fields remaining identical to former revision, e.g. old software implementing format 2.6 will be able to decode (partially but coherently) any data in format 2.8.

4b) More precisely, in the case of a revision change, the format extension cannot remove any already defined field, nor can it modify its offset relative to the beginning of its section (Header or Data); in other words, a new field cannot be set to occupy bytes that are already occupied by a former field.

3.2 Additional Remarks

If the format revision rules are respected, then an old software that is able to decode version ***n.x*** data will still be able to safely decode version ***n.(x+1)*** data: to this old software, the new fields will be easily skipped as they will appear as “garbage” within the Reserve sections.