# Sim0MQ Message Structure

| | |
|---|---|
| *Author:* | *Alexander Verbraeck, Sibel Eker, TU Delft* |
| *Version:* | *1.4* |
| *Date:* | *21 April 2017* |

## Table of Changes:

| *Version* | *Date* | *Author* | *Major changes* |
|---|---|---|---|
| 1.0 | 04-03-2017 | Verbraeck | Initial version |
| 1.1 | 16-03-2017 | Eker | Message descriptions, update to Figure 1 and addition of Figure 2. |
| 1.2 | 19-4-2017 | Eker | Changes in the StartFederate and FederateStarted messages |
| 1.3 | 20-4-2017 | Verbraeck | REQ-ROUTER pattern in chapter 3 added and explained |
| 1.4 | 21-04-2017 | Eker | Changes in the StartFederate and FederateStarted messages related to the port number of model instances |

# Table of Contents


**1 Sim0MQ Basics ... 4**

1.1 ØMQ Message Bus ... 4

**2 Message Structure ... 5**

2.1 Typed Sim0MQ Messages ... 5

2.1.1 String types (#9 - #10) ... 7

2.1.2 Array types (#11 - #17) ... 7

2.1.3 Matrix types (#18 - #24) ... 8

2.2 Coding of units ... 8

2.2.1 Float with unit (#25) ... 9

2.2.2 Double with unit (#26) ... 9

2.2.3 Float array with unit (#27) ... 9

2.2.4 Double array with unit (#28) ... 9

2.2.5 Float matrix with unit (#29) ... 10

2.2.6 Double matrix with unit (#30) ... 10

2.2.7 Float matrix with unique units per column (#31) ... 10

2.2.8 Double matrix with unique units per column (#32) ... 11

2.2.9 Money units per quantity (unit types #101 - #106) ... 12

2.3 Sim0MQ Simulation Messages ... 13

2.3.1 Simulation run id ... 14

2.3.2 Sender id ... 14

2.3.3 Receiver id ... 14

2.3.4 Message type id ... 14

2.3.5 Message status id ... 14

2.3.6 Example ... 14

**3 Sim0MQ Components ... 16**

3.1 Federate Starter ... 16

3.2 Federation Manager ... 16

**4 Notes and possible extensions ... 20**

4.1 Notes ... 27

4.2 Possible extensions ... 27

**Appendix A. Unit display type coding ... 28**

0. Dimensionless ... 28

1. Acceleration ... 28

2. AngleSolid ... 28

3. Angle ... 28

4. Direction ... 28

5. Area ... 29

6. Density ... 29
7. ElectricalCharge ... 29
8. ElectricalCurrent ... 30
9. ElectricalPotential ... 30
10. ElectricalResistance ... 30
11. Energy ... 30
12. FlowMass ... 31
13. FlowVolume ... 31
14. Force ... 32
15. Frequency ... 32
16. Length ... 32
17. Position ... 33
18. LinearDensity ... 33
19. Mass ... 34
20. Power ... 34
21. Pressure ... 35
22. Speed ... 35
23. Temperature ... 35
24. AbsoluteTemperature ... 36
25. Duration ... 36
26. Time ... 36
27. Torque ... 36
28. Volume ... 36
100. Money ... 37
101. MoneyPerArea ... 41
102. MoneyPerEnergy ... 41
103. MoneyPerLength ... 41
104. MoneyPerMass ... 41
105. MoneyPerTime ... 41
106. MoneyPerVolume ... 41

# 1 Sim0MQ Basics

## 1.1 ØMQ Message Bus

Sim0MQ makes use of the ØMQ (or 0MQ or ZMQ) message bus, and contains a layer of simulation-specific components and messages to aid in creating distributed simulation execution.

# 2 Message Structure

Several types of messages can be distinguished: internal messages to ØMQ, such as the heartbeat; binary messages that have an internal structure that is described by external metadata, and formatted or typed messages that have an internal structure including structure metadata to be able to automatically parse the message. Finally, the actual messages could be generated by an external protocol such as Google's Protocol Buffers (protobuf), which is analogous to the IDL (Interface Description Language) in Corba and DDS (OMG's Data Distribution Service). We will focus here on the typed messages that contain internal structure metadata as part of the message, making parsing of the message easy.

Message structures can be characterized by the following aspects:

- **magic number**, aka header frame (or no header frame). The message might have a header frame at the start, identifying it is a Sim0MQ message to distinguish it from other messages using the bus. Messages without the magic number might be discarded.
- **structure metadata**. The message might have no structure metadata in the message (e.g., the first 4 bytes are a float in network byte order, followed by a 8-byte long in network byte order), or have structure metadata (e.g., a byte that precedes every field and indicates the type of field that follows. The advantage of structure metadata is that errors and incompatibilities between versions can be easily spotted (I expected an integer in field 4, but I got a String).
- **external metadata**. There might be a file that describes the structure of a message type that can be used to automatically parse the message. This could work both for messages with or without structure metadata. External metadata adds names to the types of the structure metadata (e.g., field 4 is the companyName, and it cannot be null or blank). Required fields and optional fields can be distinguished in many IDLs (Interface Description Languages).
- **generative metadata**. The file that describes the external metadata might be used for code generation to create named data structures or classes that contain the same information as the message type described by the external metadata.

## 2.1 Typed Sim0MQ Messages

Typed messages have a magic number, and contain structured metadata. Right now, no external metadata or generative metadata exists. The idea of typed messages is that every field in the message has a prefix that indicates the field type. Although this can be considered overhead, it makes it easy to quickly create a data structure from the message without having to know the exact naming of the fields.

A typed Sim0MQ message looks as follows:

- **Frame 0**. Magic number = "SMQ##" where ## stands for the version number, e.g., 01. The magic number is coded as a String, which means that the string type indicator and number of characters are the prefix for the magic number. Therefore, every Sim0MQ message starts with: |9|0|0|0|5|S|I|M|, followed by a 2-digit String version number, e.g.: |9|0|0|0|5|S|I|M|0|1|.
- **Frame 1-n**: Fields, where each field has a 1-byte prefix denoting the type of field. The standard way of communicating is big-Endian, also known as network byte order. Little endian can be supported as well, but will lead to additional translations in Java and Python

implementations. When multiple C or C++ components talk to each other, little endian communication might be a good idea, though. Here, we will focus in big endian implementations only. Every field is prefixed with a one-byte type code, e.g., 2 for a big endian 32 bit signed two's complement integer. An int with the value 824 will therefore be coded as: $|2|0|0|3|56|$ using decimal notation.

The following big endian datatypes have been defined:

| code | name | description |
|---|---|---|
| **0** | BYTE_8 | Byte, 8 bit signed two's complement integer |
| **1** | SHORT_16 | Short, 16 bit signed two's complement integer, big endian order |
| **2** | INT_32 | Integer, 32 bit signed two's complement integer, big endian order |
| **3** | LONG_64 | Long, 64 bit signed two's complement integer, big endian order |
| **4** | FLOAT_32 | Float, single-precision 32-bit IEEE 754 floating point, big endian order |
| **5** | DOUBLE_64 | Float, double-precision 64-bit IEEE 754 floating point, big endian order |
| **6** | BOOLEAN_8 | Boolean, sent / received as a byte; 0 = false, 1 = true |
| **7** | CHAR_8 | Char, 8-bit ASCII character |
| **8** | CHAR_16 | Char, 16-bit Unicode character, big endian order |
| **9** | STRING_8 | String, 32-bit number-preceded byte array of 8-bits characters |
| **10** | STRING_16 | String, 32-bit number-preceded char array of 16-bits characters, big-endian order |
| **11** | BYTE_8_ARRAY | Byte array, preceded by a 32-bit number indicating the number of bytes, big-endian order |
| **12** | SHORT_16_ARRAY | Short array, preceded by a 32-bit number indicating the number of shorts, big-endian order |
| **13** | INT_32_ARRAY | Integer array, preceded by a 32-bit number indicating the number of integers, big-endian order |
| **14** | LONG_64_ARRAY | Long array, preceded by a 32-bit number indicating the number of longs, big-endian order |
| **15** | FLOAT_32_ARRAY | Float array, preceded by a 32-bit number indicating the number of floats, big-endian order |
| **16** | DOUBLE_64_ARRAY | Double array, preceded by a 32-bit number indicating the number of doubles, big-endian order |
| **17** | BOOLEAN_8_ARRAY | Boolean array, preceded by a 32-bit number indicating the number of booleans, big-endian order |
| **18** | BYTE_8_MATRIX | Byte matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **19** | SHORT_16_MATRIX | Short matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **20** | INT_32_MATRIX | Integer matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **21** | LONG_64_MATRIX | Long matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **22** | FLOAT_32_MATRIX | Float matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **23** | DOUBLE_64_MATRIX | Double matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **24** | BOOLEAN_8_MATRIX | Boolean matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order |
| **25** | FLOAT_32_UNIT | Float stored internally as an SI unit, with unit type and display unit attached. The total size of the field is 6 bytes. See below. |
| **26** | DOUBLE_64_UNIT | Double stored internally as an SI unit, with unit type and display unit attached. The total size of the field is 10 bytes. See below. |

| code | name | description |
|---|---|---|
| **27** | FLOAT_32_UNIT_ARRAY | Dense float array, preceded by a 32-bit number indicating the number of floats, big-endian order, with unit type and display unit attached to the entire float array. See below. |
| **28** | DOUBLE_64_UNIT_ARRAY | Dense double array, preceded by a 32-bit number indicating the number of doubles, big-endian order, with unit type and display unit attached to the entire float array. See below. |
| **29** | FLOAT_32_UNIT_MATRIX | Dense float matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order, with unit type and display unit attached to the entire float matrix. See below. |
| **30** | DOUBLE_64_UNIT_MATRIX | Dense double matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order, with unit type and display unit attached to the entire double matrix. See below. |
| **31** | FLOAT_32_UNIT2_MATRIX | Dense float matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order, with a unique unit type and display unit per row of the float matrix. See below. |
| **32** | DOUBLE_64_UNIT2_MATRIX | Dense double matrix, preceded by a 32-bit number row count and a 32-bit number column count, big-endian order, with a unique unit type and display unit per row of the doublematrix. See below. |

As an example, this means that a message coding {"Hello world",24,TRUE} where the number 24 is an int, is coded as:

|9|0|0|0|5|S|I |M|0|1|9|0|0|0|11|H|e|l |l |o|  |w|o|r|l |d|2|0|0|0|24|6|1|

### 2.1.1 String types (#9 - #10)

The string types are preceded by a 32-bit int indicating the number of characters in the array that follows. This int is itself not preceded by a byte indicating it is an int. An ASCII string "Hello" is therefore coded as follows:

|9|0|0|0|5|H|e|l |l |o|

in UTF-8 and as

|10|0|0|0|5|0x00|H|0x00|e|0x00|l |0x00|l |0x00|o|

in UTF-16.

Java uses UTF-16 internally, so Strings could be encoded in different ways, but the Sim0MQ message header fields from section 2.3 all use UTF-8. Other strings in the message can be encoded using UTF-16 or UTF-8, depending on the implementation of the message handler.

### 2.1.2 Array types (#11 - #17)

The array types are preceded by a 32-bit int indicating the number of values in the array that follows. This int is itself not preceded by a byte indicating it is an int. An array of 8 shorts with numbers 100 through 107 is therefore coded as follows:

|12|0|0|0|8|0|100|0|101|0|102|0|103|0|104|0|105|0|106|0|107|

### 2.1.3 Matrix types (#18 - #24)

The matrix types are preceded by a 32-bit int indicating the number of rows, followed by a 32-bit int indicating the number of columns. These integers are not preceded by a byte indicating it is an int. The number of values in the matrix that follows is rows * columns. The data is stored row by row, without a separator between the rows. A matrix with 2 rows and 3 columns of integers 1-2-4 6-7-8 is therefore coded as follows:

|20|0|0|0|2|0|0|0|3|0|0|0|1|0|0|0|2|0|0|0|4|0|0|0|6|0|0|0|7|0|0|0|8|

## 2.2 Coding of units

Units are coded with one byte indicating the unit type, and one byte indicating the display type of the unit. The SI unit or standard unit always has display type 0. **Appendix A** lists the display types for each unit type. The unit types as defined currently are:

| code | unit name | unit description | default (SI) unit |
|---|---|---|---|
| **0** | Dimensionless | Unit without a dimension | [] |
| **1** | Acceleration | Acceleration | [m/s^2] |
| **2** | AngleSolid | Solid angle | [steradian] |
| **3** | Angle | Angle (relative) | [rad] |
| **4** | Direction | Angle (absolute) | [rad] |
| **5** | Area | Area | [m^2] |
| **6** | Density | Density based on mass and length | [kg/m^3] |
| **7** | ElectricalCharge | Electrical charge (Coulomb) | [sA] |
| **8** | ElectricalCurrent | Electrical current (Ampere) | [A] |
| **9** | ElectricalPotential | Electrical potential (Volt) | [kgm^2/s^3A] |
| **10** | ElectricalResistance | Electrical resistance (Ohm) | [kgm^2/s^3A^2] |
| **11** | Energy | Energy (Joule) | [kgm^2/s^2] |
| **12** | FlowMass | Mass flow rate | [kg/s] |
| **13** | FlowVolume | Volume flow rate | [m^3/s] |
| **14** | Force | Force (Newton) | [kgm/s^2] |
| **15** | Frequency | Frequency (Hz) | [1/s] |
| **16** | Length | Length (relative) | [m] |
| **17** | Position | Length (absolute) | [m] |
| **18** | LinearDensity | Linear density | [1/m] |
| **19** | Mass | Mass | [kg] |
| **20** | Power | Power (Watt) | [kgm^2/s^3] |
| **21** | Pressure | Pressure (Pascal) | [kg/ms^2] |
| **22** | Speed | Speed | [m/s] |
| **23** | Temperature | Temperature (relative) | [K] |
| **24** | AbsoluteTemperature | Temperature (absolute) | [K] |
| **25** | Duration | Time (relative) | [s] |
| **26** | Time | Time (absolute) | [s] |
| **27** | Torque | Torque (Newton-meter) | [kgm^2/s^2] |
| **28** | Volume | Volume | [m^3] |
| **100** | Money | Money (cost in e.g., $, €, ...) | [$] |
| **101** | MoneyPerArea | Money/Area (cost/m^2) | [$/m^2] |
| **102** | MoneyPerEnergy | Money/Energy (cost/W) | [$s^3/kgm^2] |
| **103** | MoneyPerLength | Money/Length (cost/m) | [$/m] |
| **104** | MoneyPerMass | Money/Mass (cost/kg) | [$/kg] |
| **105** | MoneyPerTime | Money/Duration (cost/s) | [$/s] |
| **106** | MoneyPerVolume | Money/Volume (cost/m^3) | [$/m^3] |

Some of the unity types have a relative and an absolute variant. Relative scalars can be added to or subtracted from relative and absolute scalars; absolute scalars cannot be added, but can be subtracted, resulting in a relative scalar. As an example, one cannot add two times (3-1-2017, 5 o'clock + 3-1-2017, 3 o'clock = ??), but these values can be subtracted (3-1-2017, 5 o'clock – 3-1-2017, 3 o'clock = 2 hours). Absolute plus relative yields e.g., 3-1-2017, 17:00 + 2 hours = 3-1-2017, 19:00. Relative values can of course be added/subtracted: 2 hours + 30 minutes = 2.5 hours.

### 2.2.1 Float with unit (#25)

The internal storage of the value that is transmitted is always in the SI (or standard) unit, except for money where the display unit is used. The value is preceded by a one-byte unit type (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). As an example: suppose the unit indicates that the type is a length, whereas the display type indicates that the internally stored value 60000.0 should be displayed as 60.0 km, this is coded as follows:

|25|16|11|0x47|0x6A|0x60|0x00|

### 2.2.2 Double with unit (#26)

The internal storage of the value that is transmitted is always in the SI (or standard) unit, except for money where the display unit is used. The value is preceded by a one-byte unit type (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). As an example: suppose the unit indicates that the type is a length, whereas the display type indicates that the internally stored value 60000.0 should be displayed as 60.0 km, this is coded as follows:

|26|16|11|0x47|0x6A|0x60|0x00|0x00|0x00|0x00|0x00|

### 2.2.3 Float array with unit (#27)

After the byte with value 27, the array types have a 32-bit int indicating the number of values in the array that follows. This int is itself not preceded by a byte indicating it is an int. Then a one-byte unit type follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). The internal storage of the values that are transmitted after that always use the SI (or standard) unit, except for money where the display unit is used. As an example: when we send an array of two durations, 2.0 minutes and 2.5 minutes, this is coded as follows:

|27|0|0|0|2|25|7|0x40|0x00|0x00|0x00|0x40|0x20|0x00|0x00|

### 2.2.4 Double array with unit (#28)

After the byte with value 28, the array types have a 32-bit int indicating the number of values in the array that follows. This int is itself not preceded by a byte indicating it is an int. Then a one-byte unit type follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). The internal storage of the values that are transmitted after that always use

the SI (or standard) unit, except for money where the display unit is used. As an example: when we send an array of two durations, 21.2 minutes and 21.5 minutes, this is coded as follows:

```
|28|0|0|0|2|25|7|0x40|0x35|0x33|0x33|0x3|0x33|0x33|0x33|
|0x40|0x35|0x80|0x00|0x00|0x00|0x00|0x00|
```

### 2.2.5 Float matrix with unit (#29)

After the byte with value 29, the matrix types have a 32-bit int indicating the number of rows in the array that follows, followed by a 32-bit int indicating the number of columns. These integers are not preceded by a byte indicating it is an int. Then a one-byte unit type follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). The internal storage of the values that are transmitted after that always use the SI (or standard) unit, except for money where the display unit is used. Summarized, the coding is as follows:

```
|29|    |R|O|W|S|    |C|O|L|S|    |UT|    |DT|
|R|1|C|1|    |R|1|C|2|  ...  |R|1|C|n|
|R|2|C|1|    |R|2|C|2|  ...  |R|2|C|n|
...
|R|m|C|1|    |R|m|C|2|  ...  |R|m|C|n|
```

In the language sending or receiving a matrix, the rows are denoted by the outer index, and the columns by the inner index: matrix[row][col].

### 2.2.6 Double matrix with unit (#30)

After the byte with value 30, the matrix types have a 32-bit int indicating the number of rows in the array that follows, followed by a 32-bit int indicating the number of columns. These integers are not preceded by a byte indicating it is an int. Then a one-byte unit type follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type (see Appendix A). The internal storage of the values that are transmitted after that always use the SI (or standard) unit, except for money where the display unit is used. Summarized, the coding is as follows:

```
|30|    |R|O|W|S|    |C|O|L|S|    |UT|    |DT|
|R|1|C|1|.|.|.|.|    |R|1|C|2|.|.|.|.|  ...  |R|1|C|n|.|.|.|.|
|R|2|C|1|.|.|.|.|    |R|2|C|2|.|.|.|.|  ...  |R|2|C|n|.|.|.|.|
...
|R|m|C|1|.|.|.|.|    |R|m|C|2|.|.|.|.|  ...  |R|m|C|n|.|.|.|.|
```

In the language sending or receiving a matrix, the rows are denoted by the outer index, and the columns by the inner index: matrix[row][col].

### 2.2.7 Float matrix with unique units per column (#31)

After the byte with value 31, the matrix types have a 32-bit int indicating the number of rows in the array that follows, followed by a 32-bit int indicating the number of columns. These integers are not preceded by a byte indicating it is an int. Then a one-byte unit type for column 1 follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type for column 1 (see Appendix A). Then the unit type and display type for column 2, etc. The internal storage of the

values that are transmitted after that always use the SI (or standard) unit, except for money where the display unit is used. Summarized, the coding is as follows:

```
|31|   |R|O|W|S|   |C|O|L|S|
|UT1|DT1|   |UT2|DT2| ...  |UTn|DTn|
|R|1|C|1|   |R|1|C|2| ...  |R|1|C|n|
|R|2|C|1|   |R|2|C|2| ...  |R|2|C|n|
...
|R|m|C|1|   |R|m|C|2| ...  |R|m|C|n|
```

In the language sending or receiving a matrix, the rows are denoted by the outer index, and the columns by the inner index: matrix[row][col].

This data type is ideal for, for instance, sending a time series of values, where column1 indicates the time, and column 2 the value. Suppose that we have a time series of 4 values at t = {1, 2, 3, 4} hours and dimensionless values v = {20.0, 40.0, 50.0, 60.0}, then the coding is as follows:

```
|31|   |0|0|0|4|   |0|0|0|2|
|26|8|   |0|0|
|0x3F|0x80|0x00|0x00|   |0x41|0xA0|0x00|0x00|
|0x40|0x00|0x00|0x00|   |0x42|0x20|0x00|0x00|
|0x40|0x00|0x40|0x00|   |0x42|0x48|0x00|0x00|
|0x40|0x80|0x00|0x00|   |0x42|0x70|0x00|0x00|
```

### 2.2.8 Double matrix with unique units per column (#32)

After the byte with value 32, the matrix types have a 32-bit int indicating the number of rows in the array that follows, followed by a 32-bit int indicating the number of columns. These integers are not preceded by a byte indicating it is an int. Then a one-byte unit type for column 1 follows (see the table above) and a one-byte (or two-byte in case of the MoneyPerUnit) display type for column 1 (see Appendix A). Then the unit type and display type for column 2, etc. The internal storage of the values that are transmitted after that always use the SI (or standard) unit, except for money where the display unit is used. Summarized, the coding is as follows:

```
|32|   |R|O|W|S|   |C|O|L|S|
|UT1|DT1|   |UT2|DT2| ...  |UTn|DTn|
|R|1|C|1|.|.|.|.|   |R|1|C|2|.|.|.|.| ...  |R|1|C|n|.|.|.|.|
|R|2|C|1|.|.|.|.|   |R|2|C|2|.|.|.|.| ...  |R|2|C|n|.|.|.|.|
...
|R|m|C|1|.|.|.|.|   |R|m|C|2|.|.|.|.| ...  |R|m|C|n|.|.|.|.|
```

In the language sending or receiving a matrix, the rows are denoted by the outer index, and the columns by the inner index: matrix[row][col].

This data type is ideal for, for instance, sending a time series of values, where column1 indicates the time, and column 2 the value. Suppose that we have a time series of 4 values at dimensionless years {2010, 2011, 2012, 2013} and costs of dollars per acre of {415.7, 423.4, 428.0, 435.1}, then the coding is as follows:

|32| |0|0|0|4| |0|0|0|2|
|0|0| |101|150|18|
|0x40|0x9F|0x68|0x00|0x00|0x00|0x00|0x00|
|0x40|0x79|0xFB|0x33|0x33|0x33|0x33|0x33|
|0x40|0x9F|0x6C|0x00|0x00|0x00|0x00|0x00|
|0x40|0x7A|0x76|0x66|0x66|0x66|0x66|0x66|
|0x40|0x9F|0x70|0x00|0x00|0x00|0x00|0x00|
|0x40|0x7A|0xC0|0x00|0x00|0x00|0x00|0x00|
|0x40|0x9F|0x74|0x00|0x00|0x00|0x00|0x00|
|0x40|0x7A|0x91|0x99|0x99|0x99|0x99|0x9A|

### 2.2.9 Money units per quantity (unit types #101 - #106)

For MoneyPerUnit quantities such as MoneyPerArea, we need to send two bytes to indicate the display unit: one for the Money, and one foe the other unit. For Money per area, this means we first send a Money constant followed by an Area constant. As an example, if we want to send the price of land as € 2500 per hectare, transmitted as a double, this would be coded as:

|101|50|21|0x40|0xA3|0x88|0x00|0x00|0x00|0x00|0x00|

A double array of 200 cost elements in dollars would be coded as:

|28|0|0|0|200|100|150|...[200x8 bytes]...|

whereas a double array of 200 elements with dollars per liter would be coded as:

|28|0|0|0|200|106|150|17|...[200x8 bytes]...|

## 2.3 Sim0MQ Simulation Messages

In many cases, we want to distinguish between a definition of something, a subsequent change, and the deletion or termination of something. As an example, a generated entity can report its initial status, update state changes, and indicate when it leaves the simulation.

Furthermore, it is considered to be useful to know the message type, to avoid mistakes for parsing the wrong message. Although it adds a bit to the message overhead, the benefits of not parsing and interpreting a wrong message are clearly outweighing the transmission of a few bytes. In cases where many short messages of a certain type are sent, untyped messages could be preferred over typed simulation messages.

Finally, when multiple simulations are running in parallel, it is important to know for which running simulation the message is intended. In case it gets delivered to the wrong simulation, it can be discarded and potentially, the mistake can be logged.

The message structure of a typical typed Sim0MQ simulation message looks as follows:

**Frame 0. Magic number** = |9|0|0|0|5|S|I|M|#|#| where ## stands for the version number, e.g., 01. The magic number is **always** coded as a UTF-8 String, so it always starts with a byte equal to 9.

**Frame 1. Simulation run id**. Simulation run ids can be provided in different types. Examples are two 64-bit longs indicating a UUID, or a String with a UUID number, a String with meaningful identification, or a short or an int with a simulation run number. In order to check whether the right information has been received, the id can be translated to a String and compared with an internal string representation of the required simulation run id. The run id can be coded as UTF-8 or UTF-16.

**Frame 2. Sender id.** Sender ids can be provided in different types. Examples are two 64-bit longs indicating a UUID, or a String with a UUID number, a String with meaningful identification, or a short or an int with a sender id number. The sender id can be used to send back a message to the sender at some later time. The sender id can be coded as UTF-8 or UTF-16.

**Frame 3. Receiver id.** Receiver ids can be provided in different types. Examples are two 64-bit longs indicating a UUID, or a String with a UUID number, a String with meaningful identification, or a short or an int with a receiver id number. The receiver id can be used to check whether the message is meant for us, or should be discarded (or an error can be sent if we receive a message not meant for us). The receiver id can be coded as UTF-8 or UTF-16.

**Frame 4. Message type id**. Message type ids can be defined per type of simulation, and can be provided in different types. Examples are a String with a meaningful identification, or a short or an int with a message type number. For interoperability between different types of simulation, a String id with dot-notation (e.g., DSOL.1 for a simulator start message from DSOL or OTS.14 for a statistics message from OpenTrafficSim) would be preferred. The message type id can be coded as UTF-8 or UTF-16.

**Frame 5. Unique message number.** The unique message number will be sent as a long (64 bits), and is meant to confirm with a callback that the message has been received correctly. The number is unique for the sender, so not globally within the federation.

**Frame 6. Message status id.** Messages can be about something new (containing a definition that can be quite long), an update (which is often just an id followed by a single number), and a deletion (which is often just an id). The message status is is coded as a byte.

**Frame 7. Number of fields.** The number of fields in the payload is indicated to be able to check the payload and to avoid reading past the end. The number of fields can be encoded using byte, short, or int. A 32-bit int is the standard encoding.

**Frame 8-n. Payload**, where each field has a 1-byte prefix denoting the type of field.

### 2.3.1 Simulation run id

An example is to standardize on a String with meaningful information for the run id. The string contains a run prefix, with experiment number and replication number separated by dot-notation. This could be IDVV.14.2, indicating we run a simulation called 'IDVV', where the message is for scenario (experiment) 14, replication 2. Clients would know whether they would be part of experiment 14, replication 2, or not. A simple String comparison would yield whether they received a message that is meant for the particular simulation.

### 2.3.2 Sender id

An example is to standardize on a String with meaningful information for the sender id. The string contains a prefix identifying the type of component, e.g., "MM1" for an MM1 simulation model. A number after that will indicate a unique number of the instance of the running model or component, e.g. "MM1.4". When this model sends a message to the model controller, it will use "MM1.4" for this field as a sender.

### 2.3.3 Receiver id

An example is to standardize on a String with meaningful information for the receiver id. The string contains a prefix identifying the type of component, e.g., "MC" for the model controller. A number after that will indicate a unique number of the instance of the running model or component, e.g. "MC.1". When this model controller receives a message, it will test for this field as being "MC.1". We could allow wildcards, where all model controllers would be informed with a message, using "MC.*" to indicate that any model controller could receive this message, or even more extreme, "*" to indicate any component could receive this message.

### 2.3.4 Message type id

Analogous with the simulation run is, the message type id contains a String id with dot-notation. The first part of the message is the project the message belongs to (e.g., DSOL for the DSOL simulation package, or OTS for OpenTrafficSim), followed by a message type number that is maintained within that project (e.g., DSOL.1 for a simulator start message from DSOL or OTS.14 for a statistics message from OpenTrafficSim).

### 2.3.5 Message status id

Three different status messages are defined: **1 for new, 2 for change**, and **3 for delete**. These messages are coded as a byte.

### 2.3.6 Example

Suppose we have a simulation called IDVV.14.2 in which a message to change the (double) simulation speed to the value 0.2 is sent, of which the message type is DSOL.3. The message is sent by "MC.1"

and received by "MM1.4". Suppose the message number is 124. Then the message looks as follows (note that the double representation of 0.2 is 0x3FC999999999999A):

```
|9|0|0|0|5|S|I|M|0|1|9|0|0|0|9|I|D|V|V|.|1|4|.|2|
|9|0|0|0|4|M|C|.|1|9|0|0|0|5|M|M|1|.|4|9|0|0|0|6|D|S|0|L|.|3|
|3|0|0|0|0|0|0|0|124|0|2|1|0|1|
|5|0x3F|0xC9|0x99|0x99|0x99|0x99|0x99|0x9A|
```

# 3 Sim0MQ Components

Several components have been created to help in easy setting up of simulations that use the Sim0MQ message bus. Examples are a Federation Manager (aka Model Controller) and a Federate Starter. A Federate Starter is a lightweight executable to start a federate on a local node as a (sub)process. The Federate Starter listens to external messages, e.g. from a Federation Manager. The Federation Manager sends messages to Federate Starters to start model components, loggers, data collectors, etc. It can start a federation as soon as all required model components are on-line. It might kill model components that take too long to finish.

## 3.1 Federate Starter

A Federate Starter is a small program or daemon that listens on a certain port and that can start federates such as models, loggers, data providers and other federation components on a local machine. A Federation Manager (e.g., a model controller or a workbench) sends a `FederateStart` message to the Federate Starter, which starts the federate and provides it with enough information so it can report back to the federate starter that the start has succeeded. After that, the Federate Starter reports back to the Federation Manager that the federate is on-line with a `FederateStarted` message. After that, the Federate Starter resumes listening on the port for new messages to start a federate. See Figure 1 for more details.

Starting a program as a subprocess is sometimes done by forking. The disadvantage of forking is that the newly started program 'inherits' the state of the federate starter. In this case, we want to start a program fresh, that is independent of the parent process. In other words, if the parent process (the Federate Starter) dies, the federates and models that have been started by it should keep working. In Java 1.7 or higher, this can be achieved by a `ProcessBuilder` class that takes care of setting the working directory, setting the environment variables, redirecting the standard i/o (stdin, stdout, stderr) of the program to be started, and starts the program. It only takes a few lines of coding.

One of the important things to take into account is that all arguments to the program need to be split as separate arguments, so "java -jar test.jar" should be split into 3 arguments. Another point of attention is the redirection of input and output. When it is not redirected, it might stay in a buffer within the program that consumes memory until it is full or until memory runs out. Therefore, adequate handling of stdout and stderr is needed.

## 3.2 Federation Manager

The Federation Manager is responsible for the management of the execution of one or more models. It asks the Federate Starter to start models on its behalf on other servers (or could do so itself if all models run on the same computer). An extensive set of messages has been devised to communicate with one or more Federate Starters on multiple computers to start models, and to communicate with the models to set the parameters and experimental conditions, and to gather the statistics afterward. Multiple Federation Managers can work in parallel and communicate with their own set of models. Figure 1 shows the basic communication pattern, and Figure 2 shows a simple implementation of the connections between the models.

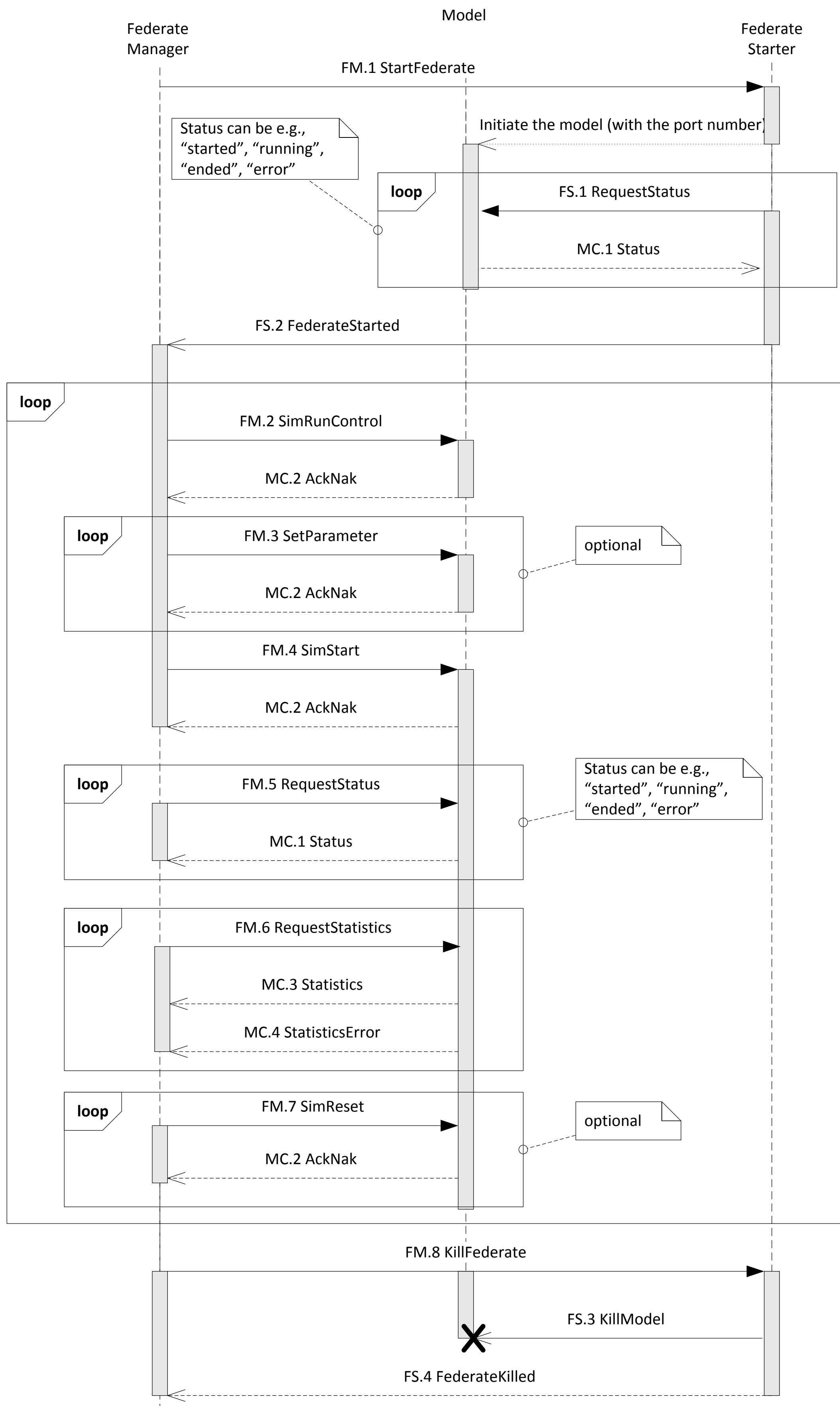


**Figure 1. Federate Starter and Model Run control flow**

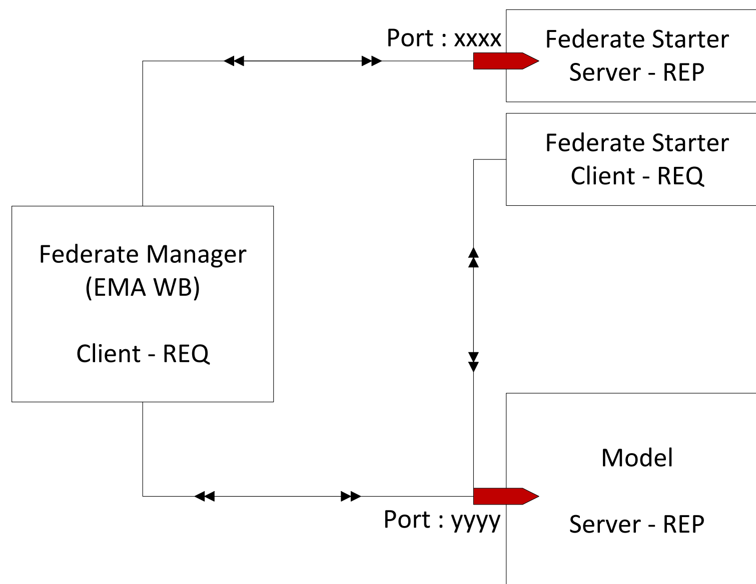


**Figure 2: ZeroMQ Simple REQ-REP Messaging Pattern between Federation Manager, Model and Federate Starter**

## 3.3 REQ-ROUTER pattern

In Figure 2, it is shown that both the Federation Manager and the Federate Starter communicate with the Model. This cannot be done with a REQ-REP framework, as REQ-REP is a 1:1 connection. Instead, the model can implement a ROUTER socket, which allows multiple connections. The ROUTER socket is a bit more complicated, however. The following changes have to be implemented in the model:

- Each client should have a unique identity. This is done with the socket.setIdentity(String identity) method in the Federation Manager and the Federate Starter, right after the socket has been created, and before binding to a port. In the reference implementation, a unique UUID is generated for each identity.
- The model should use the identity to talk back to the clients, so it will use the right channel.
- The model is a bit more complicated in terms of communication, as it explicitly receives the identity and the envelope separator (the REP receives these fields as well, but strips the identity and separator from the message so the Model does not have to act on it).

The communication takes place as follows:

```
Model (ROUTER)                        FS/FM Client (REQ)
socket(ZMQ.ROUTER)                    socket(ZMQ.REQ)
                                      setIdentity(String uniqueId)
recv(identity)         <-----         sendMessage
recv(delimiter)
recv(message)
[process the data]
sendMore(identity)
sendMore(delimiter)
send(reply_message)    ----->         recv
```

The connections with the REQ-ROUTER pattern are given in Figure 3.

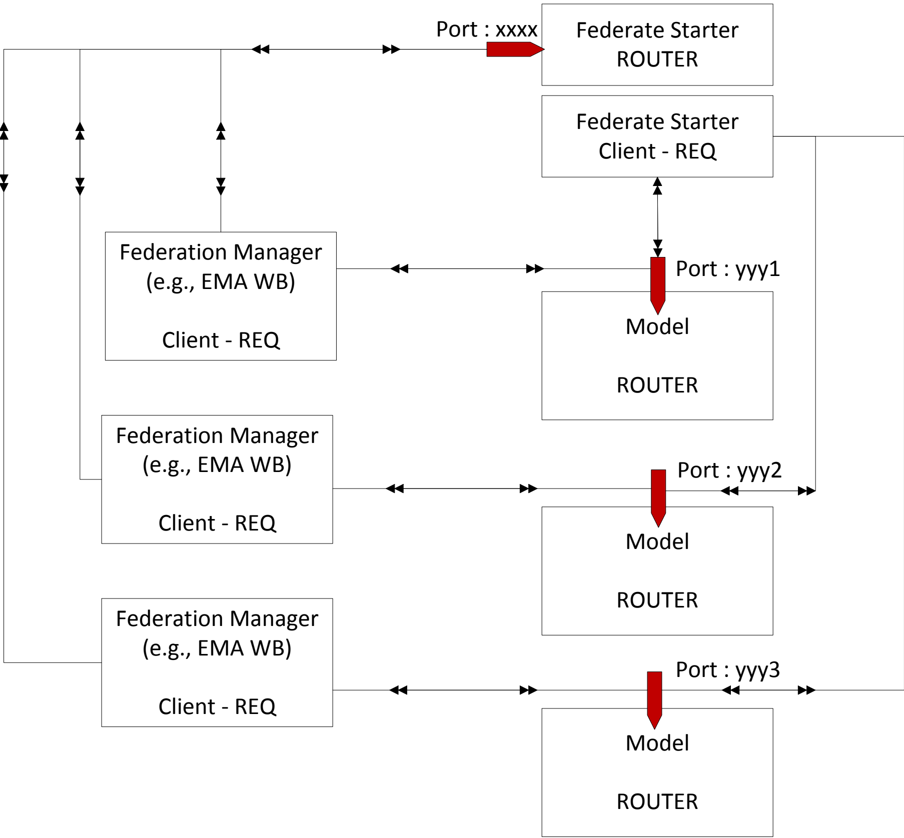


**Figure 3. ZeroMQ REQ-ROUTER Messaging Pattern between Federation Manager, Model and Federate Starter**

# 4 Messages

## 4.1 StartFederate (message type id = FM.1)

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| instanceId | String(9) | Id to identify the callback to know which model instance has been started, e.g. "IDVV.14". The model instance will use this as its sender id. The string cannot be empty. |
| softwareCode | String(9) | Code for the software to run, will be looked up in a table on the local computer to determine the path to start the software on that computer. Example: "java". The string cannot be empty. |
| argsBefore | String(9) | Arguments that the software needs, before the model file path and name; e.g. "–Xmx2G -jar" in case of a Java model. This String can be empty (0 characters). |
| modelPath | String(9) | The actual path on the target computer where the model resides, including the model that needs to be run. This String cannot be empty. |
| argsAfter | String(9) | Arguments that the software or the model needs, after the model file path and name; e.g. arguments for the model itself to run like a data file or a data location . This String can be empty (0 characters), but usually we would want to send the port number(s) or a location where the model can find it as well as the name under which the model was registered. |
| workingDirectory | String(9) | Full path on the target computer that will be used as the working directory. Some files may be temporarily stored there. If the working directory does not exist yet, it will be created. The string cannot be empty. |
| redirectStdin | String(9) | Place to get user input from in case a model asks for it (it shouldn't, by the way). The string can be empty (0 characters). |
| redirectStdout | String(9) | Place to send the output to that the model normally displays on the console. If this is not redirected, the memory buffer for the stdout might get full, and the model might stop as a result. On Linux systems, this often redirected to /dev/null. On Windows systems, this can e.g., be redirected to a file "out.txt" in the current working directory. For now, it has to be a path name (including /dev/null as being acceptable). If no full path is given, the filename is relative to the working directory. The string cannot be empty. |
| redirectStderr | String(9) | Place to send the error messages to that the model normally displays on the console. If this is not redirected, the memory buffer for the stderr might get full, and the model might stop as a result. On Linux systems, this often redirected to /dev/null. On Windows systems, this can e.g., be redirected to a file "err.txt" in the current working directory. For now, it has to be a path name (including /dev/null as being acceptable). If no full path is given, the filename is relative to the working directory. The string cannot be empty. |
| deleteWorkingDirectory | Boolean(6) | Whether to delete the working directory after the run of the model or not. |

| deleteStdout | Boolean(6) | Whether to delete the redirected stdout after running or not (in case it is stored in a different place than the working directory) |
|---|---|---|
| deleteStderr | Boolean(6) | Whether to delete the redirected stderr after running or not (in case it is stored in a different place than the working directory) |

**CHANGE**
**Not sent**

**DELETE**
Not sent

A number of standard types of software to look up and their respective codes are:

- java for the latest Java version
- java7, java8, java7+, etc. for a specific version of Java
- python for the latest python version
- python2, python3, python2+, etc. for a specific version of Python
- if necessary, special Strings could be created for 32-bit and 64-bit versions of the software. Preferably, "x64" is added at the end of the String to denote a 64-bit version
- if a specific version is needed of software, either extra entries can be created, or the actual path on the computer can be specified instead of the type code.

### 4.1.1 Federate Starter's instantiation of a model:

When it receives a StartFederate message, the Federate starter creates a process to run the model with the specifications given in the message, such as the working directory, model file, output and error files etc. Creating a model instance in this way also requires a port number, to which the model instance should bind as a ROUTER. This port number is assigned by the Federate Starter. Federate Starter picks an available port from a range of ports on the machine it is running (which must be open to outside connection) and gives this to the model as an argument. If the binding is not successful, the Federate Starter creates generates a new port number.

## 4.2 RequestStatus (message type id = FS.1)

(The id can be different, because the very same message is sent by the Federation Manager, too.)

This message is sent by the Federate Starter to the Model until a "started" response is received from the Model. Since the message type id clarifies the function of this message and no information exchange is necessary, the payload field can be empty (number of fields = 0).

Message status id : NEW

| *Variable* | *Type* | *Comments* |
|---|---|---|

## 4.3 Status (message type id = MC.1)

The Model sends this message as a response to RequestStatus messages sent by the Federate Starter or the Federation Manager.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| uniqueId | Long(3) | The unique message id (Frame 5) of the sender for which this is the reply. |
| status | String(9) | A string that refers to the model status. Four options: “started”, “running”, “ended”, “error”. |
| error | String(9) | Optional. If there is an error, the error message is sent as well. Otherwise this field is an empty string. |

## 4.4 FederateStarted (message type id = FS.2)

Message sent by the Federate Starter to the Federation Manager in response to message FM.1.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| instanceId | String(9) | The sender id of the model that was started or had an error while starting. This is exactly the same as the instanceId sent by the Federation Manager in the Start Federate message. |
| status | String(9) | A string that refers to the model status. Four options: “started”, “running”, “ended”, “error”. |
| modelPortNumber | short(1) | Port number of the model, so the FederateManager can connect to the model on this port for further simulation messages. |
| error | String(9) | Optional. If there is an error, the error message is sent as well. Otherwise this field is an empty string. |

## 4.5 SimRunControl (message type id = FM.2)

Message sent by the Federation Manager to the Model to initiate a simulation.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| runTime | Any numeric type (1-5) or Float or Double with Unit (25, 26) of type Duration (25) | Duration of the run of a single replication, **including** the warmup time, if present. |
| warmupTime | Any numeric type (1-5) or Float or Double with Unit (25, 26) of type Duration (25) | Warmup time of  the model in time units that the model uses. |
| offsetTime | Any numeric type (1-5) or Float or Double with Unit (25, 26) of type Time (26) | Offset of the time (e.g., a model time of 0 is the year 2016, or 1-1-2015). |
| speed | Double(5) | Speed as the number of times real-time the model should run; Double.INFINITY means as fast as possible. |
| noReplications | Integer(2) | Number of replications for stochastic uncertainties in the model. |
| noRandomStreams | Integer(2) | Number of random streams that follow |
| streamId.1 | basic type (1,2,3,9) | Identifier of random stream 1 |

| | | |
|---|---|---|
| seed.1 | Long(3) | Seed for random stream 1 |
| … | | |
| streamId.n | basic type (1,2,3,9) | Identifier of random stream n |
| seed.n | Long(3) | Seed for random stream n |

## 4.6 AckNak (message type id = MC.2)

Message sent by the Model to acknowledge the reception and implementation of a message sent by the Federation Manager.

This type of message is sent in response to many messages of the FM. That could create confusion if there were multiple model instances, and one sending an acknowledgement e.g.for SimRunControl, the other for SetParameter. However, since a different port number will be assigned to each model and these acknowledgment messages will be sent only after a command, and include the uniqueId of the request, such a confusion is not expected.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| uniqueId | Long(3) | The unique message id (Frame 5) of the sender for which this is the reply. |
| status | Boolean(6) | A boolean that indicates whether the command sent by the FM has been successfully implemented, e.g. whether the run control parameters are set successfully. |
| error | String(9) | If 'status' is False, an error message that indicates which parameter could not be set and why. Otherwise, an empty string. |

## 4.7 SetParameter (message type id = FM.3)

Message sent by the FederateManager to the Model for setting the parameter values. Parameters are set one by one (but can be a Vector or Matrix).

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| parameterName | String(9) | Name of the parameter as it is in the model. |
| parameterValue | any type (0-32) | Value of the parameter assigned for a specific simulation. The type depends on the parameter. It could, e.g., be *long* or *double*. [1] |

## 4.8 SimStart (message type id = FM.4)

Message sent by the Federation Manager to start the simulation.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|

[1] The EMA workbench knows beforehand the type of these parameters, samples accordingly and sends the sampled values. However, since all integer types are 'long' by default and all floats are 'double', they will be sent in that format. Message decoding and encoding handles this type specification.

## 4.9 RequestStatus (message type id = FM.5)

Message sent by the Federation Manager to enquire the status of the simulation. The answer to this message is MC.1 "Status" (discussed above).

Since the message type id clarifies the function of this message and no information exchange is necessary, the payload field can be empty.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|

## 4.10 RequestStatistics (message type id = FM.6)

Message sent by the Federation Manager to collect the output.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| variableName | String(9) | The name of the output variable whose value is requested. That should match with the name in the model. For a tallied variable, several statistics are possible, e.g., average, variance, minimum, maximum, time series, etc. The name should clearly indicate what the Model Controller expects and what the model should produce. |

## 4.11 Statistics (message type id = MC.3)

Message sent by the Model to give the model output, if there is output generated for the specified variable.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| variableName | String(9) | The name of the output variable whose value is requested. That should match with the name in the model. |
| variableValue | Any type (0-32) | If variableType is scalar, the data type is e.g., an integer, float etc. and the value generated in the model.<br>If variableType is timeseries, the data type is an 'array' (type 11-16 or 27/28) or a time series (type 31/32). |

## 4.12 StatisticsError (message type id = MC.4)

Message sent by the Model to indicate that there is an error with the output. MC.3 and MC.4 are alternative to each other.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| variableName | String(9) | The name of the output variable whose value is requested. That should match with the name in the model. |
| error | String(9) | Three types of error can occur:<br>- If the variableName does not exist in the model, *error* = "name"<br>- If the simulation did not generate a value for this variable, e.g. NaN or division by zerro, *error*= "novalue" |

## 4.13 SimReset (message type id = FM.7)

**CHANGE**

| *Variable* | *Type* | *Comments* |
|---|---|---|

## 4.14 KillFederate (message type id = FM.8)

Since the items to be deleted were specified in the StartFederate message, no input is required. The payload can be an empty string.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| instanceId | String(9) | Id to identify the model instance that has to be killed, e.g. "IDVV.14". |

## 4.15 KillAll (message type id = FM.9)

The message sent by the Federation Manager to the Federate Starter to kill all running model instances. There is no payload.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| | | |

## 4.16 KillModel (message type id = FS.3)

The message is sent by the federate starter to a model instance.

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| | | |

## 4.17 FederateKilled (message type id = FS.4)

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| instanceId | String(9) | Id to identify the model instance that was killed, e.g. "IDVV.14". |
| status | Boolean(6) | A boolean that indicates whether the federate has been successfully terminated. |
| error | String(9) | If 'status' is False, an error message that specifies the problem. Otherwise, an empty string. |

## 4.18 FederatesKilled (message type id = FS.5)

**NEW**

| *Variable* | *Type* | *Comments* |
|---|---|---|
| status | Boolean(6) | A boolean that indicates whether all federates has been successfully terminated. |
| error | String(9) | If 'status' is False, an error message that specifies the problem. Otherwise, an empty string. |

# 5 Example: MM1-model with EMA workbench

Magic number is SIM01.

Federation Manager uses id EMA.

Federate Starter id is FS, listens on port 5555.

FM.1, StartFederate: java-jar mm1.jar MM1.1 5556
MM1 listens on port 5556, I/O gets redirected to out.txt, err.txt, and will be deleted.
Model id will be MM1.# where # indicates the instance number.
Each model instance has its own port number.

FM.2 RunControl: run time: 100(double), warmup time 0(double), speed infinite(double), start time 0(double), no of replications 1(integer), 1 seed for the model, name "default", to be set by EMA.

FM.3 parameterr: "iat", double, value, standard value is 1.0.
"servicetime", double, standard value is 0.8.

FM.6 RequestStatistics: "dN", "qN", "uN" are the main statistics. Of each you can ask:

- .average
- .stdev
- .variance
- .sum
- .min
- .max
- .halfwidth(alpha) where alpha is a real number like 0.05 for a 95% confidence interval
- .n

# 6 Notes and possible extensions

## 6.1 Notes

## 6.2 Possible extensions

# Appendix A. Unit display type coding

## 0. Dimensionless

The Dimensionless unit does not have any display codes, except the default one, indicated with code number 0.

## 1. Acceleration

| | | |
|---|---|---|
| 0 | METER_PER_SECOND_2 (SI) | $m/s^2$ |
| 1 | KM_PER_HOUR_2 | $km/h^2$ |
| 2 | INCH_PER_SECOND_2 | $in/s^2$ |
| 3 | FOOT_PER_SECOND_2 | $ft/s^2$ |
| 4 | MILE_PER_HOUR_2 | $mi/h^2$ |
| 5 | MILE_PER_HOUR_PER_SECOND | mi/h/s |
| 6 | KNOT_PER_SECOND | kt/s |
| 7 | GAL | gal |
| 8 | STANDARD_GRAVITY | g |

## 2. AngleSolid

| | | |
|---|---|---|
| 0 | STERADIAN (SI) | sr |
| 1 | SQUARE_DEGREE | sq.deg |

## 3. Angle

| | | |
|---|---|---|
| 0 | RADIAN (SI) | rad |
| 1 | ARCMINUTE | arcmin / ' |
| 2 | ARCSECOND | arcsec / " |
| 3 | CENTESIMAL_ARCMINUTE | centesimal_arcmin |
| 4 | CENTESIMAL_ARCSECOND | centesimal_arcsec |
| 5 | DEGREE | deg |
| 6 | GRAD | grad |

## 4. Direction

| | | |
|---|---|---|
| 0 | RADIAN (SI) | rad |
| 1 | ARCMINUTE | arcmin / ' |
| 2 | ARCSECOND | arcsec / " |
| 3 | CENTESIMAL_ARCMINUTE | centesimal_arcmin |
| 4 | CENTESIMAL_ARCSECOND | centesimal_arcsec |
| 5 | DEGREE | deg |
| 6 | GRAD | grad |

## 5. Area

| | | |
|---|---|---|
| 0 | SQUARE_METER (SI) | $m^2$ |
| 1 | SQUARE_ATTOMETER | $am^2$ |
| 2 | SQUARE_FEMTOMETER | $fm^2$ |
| 3 | SQUARE_PICOMETER | $pm^2$ |
| 4 | SQUARE_NANOMETER | $nm^2$ |
| 5 | SQUARE_MICROMETER | $\mu m^2$ |
| 6 | SQUARE_MILLIMETER | $mm^2$ |
| 7 | SQUARE_CENTIMETER | $cm^2$ |
| 8 | SQUARE_DECIMETER | $dm^2$ |
| 9 | SQUARE_DEKAMETER | $dam^2$ |
| 10 | SQUARE_HECTOMETER | $hm^2$ |
| 11 | SQUARE_KILOMETER | $km^2$ |
| 12 | SQUARE_MEGAMETER | $Mm^2$ |
| 13 | SQUARE_INCH | $in^2$ |
| 14 | SQUARE_FOOT | $ft^2$ |
| 15 | SQUARE_YARD | $yd^2$ |
| 16 | SQUARE_MILE | $mi^2$ |
| 17 | SQUARE_NAUTICAL_MILE | $NM^2$ |
| 18 | ACRE | acre |
| 19 | ARE | a |
| 20 | CENTIARE | ca |
| 21 | HECTARE | ha |

## 6. Density

| | | |
|---|---|---|
| 0 | KG_PER_METER_3 (SI) | $kg/m^3$ |
| 1 | GRAM_PER_CENTIMETER_3 | $g/cm^3$ |

## 7. ElectricalCharge

| | | |
|---|---|---|
| 0 | COULOMB | C |
| 1 | PICOCOULOMB | pC |
| 2 | NANOCOULOMB | nC |
| 3 | MICROCOULOMB | µC |
| 4 | MILLICOULOMB | mC |
| 5 | ABCOULOMB | abC |
| 6 | ATOMIC_UNIT | au |
| 7 | EMU | emu |
| 8 | ESU | esu |
| 9 | FARADAY | F |
| 10 | FRANKLIN | Fr |
| 11 | STATCOULOMB | statC |
| 12 | MILLIAMPERE_HOUR | mAh |
| 13 | AMPERE_HOUR | Ah |
| 14 | KILOAMPERE_HOUR | kAh |
| 15 | MEGAAMPERE_HOUR | MAh |
| 16 | MILLIAMPERE_SECOND | mAs |

### 8. ElectricalCurrent

| 0 | AMPERE (SI) | A |
|---|---|---|
| 1 | NANOAMPERE | nA |
| 2 | MICROAMPERE | µA |
| 3 | MILLIAMPERE | mA |
| 4 | KILOAMPERE | kA |
| 5 | MEGAAMPERE | MA |
| 6 | ABAMPERE | abA |
| 7 | STATAMPERE | statA |

### 9. ElectricalPotential

| 0 | VOLT (SI) | V |
|---|---|---|
| 1 | NANOVOLT | nV |
| 2 | MICROVOLT | µV |
| 3 | MILLIVOLT | mV |
| 4 | KILOVOLT | kV |
| 5 | MEGAVOLT | MV |
| 6 | GIGAVOLT | GV |
| 7 | ABVOLT | abV |
| 8 | STATVOLT | statV |

### 10. ElectricalResistance

| 0 | OHM (SI) | Ω |
|---|---|---|
| 1 | NANOOHM | nΩ |
| 2 | MICROOHM | µΩ |
| 3 | MILLIOHM | mΩ |
| 4 | KILOOHM | kΩ |
| 5 | MEGAOHM | MΩ |
| 6 | GIGAOHM | GΩ |
| 7 | ABOHM | abΩ |
| 8 | STATOHM | statΩ |

### 11. Energy

| 0 | JOULE (SI) | J |
|---|---|---|
| 1 | PICOJOULE | pJ |
| 2 | NANOJOULE | mJ |
| 3 | MICROJOULE | µJ |
| 4 | MILLIJOULE | mJ |
| 5 | KILOJOULE | kJ |
| 6 | MEGAJOULE | MJ |
| 7 | GIGAJOULE | GJ |
| 8 | TERAJOULE | TJ |
| 9 | PETAJOULE | PJ |
| 10 | ELECTRONVOLT | eV |
| 11 | MICROELECTRONVOLT | µeV |
| 12 | MILLIELECTRONVOLT | meV |
| 13 | KILOELECTRONVOLT | keV |

| 14 | MEGAELECTRONVOLT | MeV |
|---|---|---|
| 15 | GIGAELECTRONVOLT | GeV |
| 16 | TERAELECTRONVOLT | TeV |
| 17 | PETAELECTRONVOLT | PeV |
| 18 | EXAELECTRONVOLT | EeV |
| 19 | WATT_HOUR | Wh |
| 20 | FEMTOWATT_HOUR | fWh |
| 21 | PICOWATT_HOUR | pWh |
| 22 | NANOWATT_HOUR | mWh |
| 23 | MICROWATT_HOUR | µWh |
| 24 | MILLIWATT_HOUR | mWh |
| 25 | KILOWATT_HOUR | kWh |
| 26 | MEGAWATT_HOUR | MWh |
| 27 | GIGAWATT_HOUR | GWh |
| 28 | TERAWATT_HOUR | TWh |
| 29 | PETAWATT_HOUR | PWh |
| 30 | CALORIE | cal |
| 31 | KILOCALORIE | kcal |
| 32 | CALORIE_IT | cal(IT) |
| 33 | INCH_POUND_FORCE | in lbf |
| 34 | FOOT_POUND_FORCE | ft lbf |
| 35 | ERG | erg |
| 36 | BTU_ISO | BTU(ISO) |
| 37 | BTU_IT | BTU(IT) |
| 38 | STHENE_METER | sth.m |

## 12. FlowMass

| 0 | KG_PER_SECOND (SI) | kg/s |
|---|---|---|
| 1 | POUND_PER_SECOND | lb/s |

## 13. FlowVolume

| 0 | CUBIC_METER_PER_SECOND (SI) | $m^3$/s |
|---|---|---|
| 1 | CUBIC_METER_PER_MINUTE | $m^3$/min |
| 2 | CUBIC_METER_PER_HOUR | $m^3$/h |
| 3 | CUBIC_METER_PER_DAY | $m^3$/day |
| 4 | CUBIC_INCH_PER_SECOND | $in^3$/s |
| 5 | CUBIC_INCH_PER_MINUTE | $in^3$/min |
| 6 | CUBIC_FEET_PER_SECOND | $ft^3$/s |
| 7 | CUBIC_FEET_PER_MINUTE | $ft^3$/min |
| 8 | GALLON_PER_SECOND | gal/s |
| 9 | GALLON_PER_MINUTE | gal/min |
| 10 | GALLON_PER_HOUR | gal/h |
| 11 | GALLON_PER_DAY | gal/day |
| 12 | LITER_PER_SECOND | l/s |
| 13 | LITER_PER_MINUTE | l/min |
| 14 | LITER_PER_HOUR | l/h |
| 15 | LITER_PER_DAY | l/day |

### 14. Force

| | | |
|---|---|---|
| 0 | NEWTON (SI) | N |
| 1 | KILOGRAM_FORCE | kgf |
| 2 | OUNCE_FORCE | ozf |
| 3 | POUND_FORCE | lbf |
| 4 | TON_FORCE | tnf |
| 5 | DYNE | dyne |
| 6 | STHENE | sth |

### 15. Frequency

| | | |
|---|---|---|
| 0 | HERTZ (SI) | Hz |
| 1 | KILOHERTZ | kHz |
| 2 | MEGAHERTZ | MHz |
| 3 | GIGAHERTZ | GHz |
| 4 | TERAHERTZ | THz |
| 5 | PER_SECOND | 1/s |
| 6 | PER_ATTOSECOND | 1/as |
| 7 | PER_FEMTOSECOND | 1/fs |
| 8 | PER_PICOSECOND | 1/ps |
| 9 | PER_NANOSECOND | 1/ns |
| 10 | PER_MICROSECOND | 1/µs |
| 11 | PER_MILLISECOND | 1/ms |
| 12 | PER_MINUTE | 1/min |
| 13 | PER_HOUR | 1/hr |
| 14 | PER_DAY | 1/day |
| 15 | PER_WEEK | 1/wk |
| 16 | RPM | rpm |

### 16. Length

| | | |
|---|---|---|
| 0 | METER (SI) | m |
| 1 | ATTOMETER | am |
| 2 | FEMTOMETER | fm |
| 3 | PICOMETER | pm |
| 4 | NANOMETER | nm |
| 5 | MICROMETER | µm |
| 6 | MILLIMETER | mm |
| 7 | CENTIMETER | cm |
| 8 | DECIMETER | dm |
| 9 | DEKAMETER | dam |
| 10 | HECTOMETER | hm |
| 11 | KILOMETER | km |
| 12 | MEGAMETER | Mm |
| 13 | INCH | in |
| 14 | FOOT | ft |
| 15 | YARD | yd |
| 16 | MILE | mi |
| 17 | NAUTICAL_MILE | NM |
| 18 | ASTRONOMICAL_UNIT | au |

| | | |
|---|---|---|
| 19 | PARSEC | pc |
| 20 | LIGHTYEAR | ly |
| 21 | ANGSTROM | Å |

## 17. Position

| | | |
|---|---|---|
| 0 | METER (SI) | m |
| 1 | ATTOMETER | am |
| 2 | FEMTOMETER | fm |
| 3 | PICOMETER | pm |
| 4 | NANOMETER | nm |
| 5 | MICROMETER | µm |
| 6 | MILLIMETER | mm |
| 7 | CENTIMETER | cm |
| 8 | DECIMETER | dm |
| 9 | DEKAMETER | dam |
| 10 | HECTOMETER | hm |
| 11 | KILOMETER | km |
| 12 | MEGAMETER | Mm |
| 13 | INCH | in |
| 14 | FOOT | ft |
| 15 | YARD | yd |
| 16 | MILE | mi |
| 17 | NAUTICAL_MILE | NM |
| 18 | ASTRONOMICAL_UNIT | au |
| 19 | PARSEC | pc |
| 20 | LIGHT_YEAR | ly |
| 21 | ANGSTROM | Å |

## 18. LinearDensity

| | | |
|---|---|---|
| 0 | PER_METER (SI) | 1/m |
| 1 | PER_ATTOMETER | 1/am |
| 2 | PER_FEMTOMETER | 1/fm |
| 3 | PER_PICOMETER | 1/pm |
| 4 | PER_NANOMETER | 1/nm |
| 5 | PER_MICROMETER | 1/µm |
| 6 | PER_MILLIMETER | 1/mm |
| 7 | PER_CENTIMETER | 1/cm |
| 8 | PER_DECIMETER | 1/dm |
| 9 | PER_DEKAMETER | 1/dam |
| 10 | PER_HECTOMETER | 1/hm |
| 11 | PER_KILOMETER | 1/km |
| 12 | PER_MEGAMETER | 1/Mm |
| 13 | PER_INCH | 1/in |
| 14 | PER_FOOT | 1/ft |
| 15 | PER_YARD | 1/yd |
| 16 | PER_MILE | 1/mi |
| 17 | PER_NAUTICAL_MILE | 1/NM |
| 18 | PER_ASTRONOMICAL_UNIT | 1/au |

| 19 | PER_PARSEC | 1/pc |
|---|---|---|
| 20 | PER_LIGHT_YEAR | 1/ly |
| 21 | PER_ANGSTROM | 1/Å |

## 19. Mass

| 0 | KILOGRAM (SI) | kg |
|---|---|---|
| 1 | FEMTOGRAM | fg |
| 2 | PICOGRAM | pg |
| 3 | NANOGRAM | mg |
| 4 | MICROGRAM | μg |
| 5 | MILLIGRAM | mg |
| 6 | GRAM | kg |
| 7 | MEGAGRAM | Mg |
| 8 | GIGAGRAM | Gg |
| 9 | TERAGRAM | Tg |
| 10 | PETAGRAM | Pg |
| 11 | MICROELECTRONVOLT | μeV |
| 12 | MILLIELECTRONVOLT | meV |
| 13 | KILOELECTRONVOLT | keV |
| 14 | MEGAELECTRONVOLT | MeV |
| 15 | GIGAELECTRONVOLT | GeV |
| 16 | TERAELECTRONVOLT | TeV |
| 17 | PETAELECTRONVOLT | PeV |
| 18 | EXAELECTRONVOLT | EeV |
| 19 | OUNCE | oz |
| 20 | POUND | lb |
| 21 | DALTON | Da |
| 22 | TON_LONG | ton (long) |
| 23 | TON_SHORT | ton (short) |
| 24 | TONNE | tonne |

## 20. Power

| 0 | WATT (SI) | W |
|---|---|---|
| 1 | FEMTOWATT | fW |
| 2 | PICOWATT | pW |
| 3 | NANOWATT | mW |
| 4 | MICROWATT | μW |
| 5 | MILLIWATT | mW |
| 6 | KILOWATT | kW |
| 7 | MEGAWATT | MW |
| 8 | GIGAWATT | GW |
| 9 | TERAWATT | TW |
| 10 | PETAWATT | PW |
| 11 | ERG_PER_SECOND | erg/s |
| 12 | FOOT_POUND_FORCE_PER_SECOND | ft.lbf/s |
| 13 | FOOT_POUND_FORCE_PER_MINUTE | ft.lbf/min |
| 14 | FOOT_POUND_FORCE_PER_HOUR | ft.lbf/h |
| 15 | HORSEPOWER_METRIC | hp / PS |

| 16 | STHENE_METER_PER_SECOND | sth/s |
|---|---|---|

## 21. Pressure

| 0 | PASCAL (SI) | Pa |
|---|---|---|
| 1 | HECTOPASCAL | hPa |
| 2 | KILOPASCAL | kPa |
| 3 | ATMOSPHERE_STANDARD | atm |
| 4 | ATMOSPHERE_TECHNICAL | at |
| 5 | MILLIBAR | mbar |
| 6 | BAR | bar |
| 7 | BARYE | Ba |
| 8 | MILLIMETER_MERCURY | mmHg |
| 9 | CENTIMETER_MERCURY | cmHg |
| 10 | INCH_MERCURY | inHg |
| 11 | FOOT_MERCURY | ftHg |
| 12 | KGF_PER_SQUARE_MM | kgf/mm$^2$ |
| 13 | PIEZE | pz |
| 14 | POUND_PER_SQUARE_INCH | lb/in$^2$ |
| 15 | POUND_PER_SQUARE_FOOT | lb/ft$^2$ |
| 16 | TORR | torr |

## 22. Speed

| 0 | METER_PER_SECOND (SI) | m/s |
|---|---|---|
| 1 | METER_PER_HOUR | m/h |
| 2 | KM_PER_SECOND | km/s |
| 3 | KM_PER_HOUR | km/h |
| 4 | INCH_PER_SECOND | in/s |
| 5 | INCH_PER_MINUTE | in/min |
| 6 | INCH_PER_HOUR | in/h |
| 7 | FOOT_PER_SECOND | ft/s |
| 8 | FOOT_PER_MINUTE | ft/min |
| 9 | FOOT_PER_HOUR | ft/h |
| 10 | MILE_PER_SECOND | mi/s |
| 11 | MILE_PER_MINUTE | mi/min |
| 12 | MILE_PER_HOUR | mi/h |
| 13 | KNOT | kt |

## 23. Temperature

| 0 | KELVIN (SI) | K |
|---|---|---|
| 1 | DEGREE_CELSIUS | $^{O}$C |
| 2 | DEGREE_FAHRENHEIT | $^{O}$F |
| 3 | DEGREE_RANKINE | $^{O}$R |
| 4 | DEGREE_REAUMUR | $^{O}$Ré |

## 24. AbsoluteTemperature

| 0 | KELVIN (SI) | K |
|---|---|---|
| 1 | DEGREE_CELSIUS | $^{o}$C |
| 2 | DEGREE_FAHRENHEIT | $^{o}$F |
| 3 | DEGREE_RANKINE | $^{o}$R |
| 4 | DEGREE_REAUMUR | $^{o}$Ré |

## 25. Duration

| 0 | SECOND (SI) | s |
|---|---|---|
| 1 | ATTOSECOND | as |
| 2 | FEMTOSECOND | fs |
| 3 | PICOSECOND | ps |
| 4 | NANOSECOND | ns |
| 5 | MICROSECOND | µs |
| 6 | MILLISECOND | ms |
| 7 | MINUTE | min |
| 8 | HOUR | hr |
| 9 | DAY | day |
| 10 | WEEK | wk |

## 26. Time

| 0 | SECOND (SI) | s |
|---|---|---|
| 1 | ATTOSECOND | as |
| 2 | FEMTOSECOND | fs |
| 3 | PICOSECOND | ps |
| 4 | NANOSECOND | ns |
| 5 | MICROSECOND | µs |
| 6 | MILLISECOND | ms |
| 7 | MINUTE | min |
| 8 | HOUR | hr |
| 9 | DAY | day |
| 10 | WEEK | wk |

## 27. Torque

| 0 | NEWTON_METER (SI) | Nm |
|---|---|---|
| 1 | POUND_FOOT | lb.ft |
| 2 | POUND_INCH | lb.in |
| 3 | METER_KILOGRAM_FORCE | m.kgf |

## 28. Volume

| 0 | CUBIC_METER (SI) | $m^3$ |
|---|---|---|
| 1 | CUBIC_ATTOMETER | $am^3$ |
| 2 | CUBIC_FEMTOMETER | $fm^3$ |
| 3 | CUBIC_PICOMETER | $pm^3$ |
| 4 | CUBIC_NANOMETER | $nm^3$ |
| 5 | CUBIC_MICROMETER | $µm^3$ |

| 6 | CUBIC_MILLIMETER | $mm^3$ |
|---|---|---|
| 7 | CUBIC_CENTIMETER | $cm^3$ |
| 8 | CUBIC_DECIMETER | $dm^3$ |
| 9 | CUBIC_DEKAMETER | $dam^3$ |
| 10 | CUBIC_HECTOMETER | $hm^3$ |
| 11 | CUBIC_KILOMETER | $km^3$ |
| 12 | CUBIC_MEGAMETER | $Mm^3$ |
| 13 | CUBIC_INCH | $in^3$ |
| 14 | CUBIC_FOOT | $ft^3$ |
| 15 | CUBIC_YARD | $yd^3$ |
| 16 | CUBIC_MILE | $mi^3$ |
| 17 | LITER | l |
| 18 | GALLON_IMP | gal (imp) |
| 19 | GALLON_US_FLUID | gal (US) |
| 20 | OUNCE_IMP_FLUID | oz (imp) |
| 21 | OUNCE_US_FLUID | oz (US) |
| 22 | PINT_IMP | pt (imp) |
| 23 | PINT_US_FLUID | pt (US) |
| 24 | QUART_IMP | qt (imp) |
| 25 | QUART_US_FLUID | qt (US) |
| 26 | CUBIC_PARSEC | $pc^3$ |
| 27 | CUBIC_LIGHT_YEAR | $ly^3$ |

### 100. Money

Money is in essence dimensionless. The following currencies are supported, according to ISO 4217[2]:

| 1 | AED | United Arab Emirates dirham |
|---|---|---|
| 2 | AFN | Afghan afghani |
| 3 | ALL | Albanian lek |
| 4 | AMD | Armenian dram |
| 5 | ANG | Netherlands Antillean guilder |
| 6 | AOA | Angolan kwanza |
| 7 | ARS | Argentine peso |
| 8 | AUD | Australian dollar |
| 9 | AWG | Aruban florin |
| 10 | AZN | Azerbaijani manat |
| 11 | BAM | Bosnia and Herzegovina convertible mark |
| 12 | BBD | Barbados dollar |
| 13 | BDT | Bangladeshi taka |
| 14 | BGN | Bulgarian lev |
| 15 | BHD | Bahraini dinar |
| 16 | BIF | Burundian franc |
| 17 | BMD | Bermudian dollar |
| 18 | BND | Brunei dollar |
| 19 | BOB | Boliviano |
| 20 | BOV | Bolivian Mvdol (funds code) |
| 21 | BRL | Brazilian real |
| 22 | BSD | Bahamian dollar |

[2] https://en.wikipedia.org/wiki/ISO_4217

| 23 | BTN | Bhutanese ngultrum |
|---|---|---|
| 24 | BWP | Botswana pula |
| 25 | BYN | New Belarusian ruble |
| 26 | BYR | Belarusian ruble |
| 27 | BZD | Belize dollar |
| 28 | CAD | Canadian dollar |
| 29 | CDF | Congolese franc |
| 30 | CHE | WIR Euro (complementary currency) |
| 31 | CHF | Swiss franc |
| 32 | CHW | WIR Franc (complementary currency) |
| 33 | CLF | Unidad de Fomento (funds code) |
| 34 | CLP | Chilean peso |
| 35 | CNY | Chinese yuan |
| 36 | COP | Colombian peso |
| 37 | COU | Unidad de Valor Real (UVR) (funds code) |
| 38 | CRC | Costa Rican colon |
| 39 | CUC | Cuban convertible peso |
| 40 | CUP | Cuban peso |
| 41 | CVE | Cape Verde escudo |
| 42 | CZK | Czech koruna |
| 43 | DJF | Djiboutian franc |
| 44 | DKK | Danish krone |
| 45 | DOP | Dominican peso |
| 46 | DZD | Algerian dinar |
| 47 | EGP | Egyptian pound |
| 48 | ERN | Eritrean nakfa |
| 49 | ETB | Ethiopian birr |
| 50 | EUR | Euro |
| 51 | FJD | Fiji dollar |
| 52 | FKP | Falkland Islands pound |
| 53 | GBP | Pound sterling |
| 54 | GEL | Georgian lari |
| 55 | GHS | Ghanaian cedi |
| 56 | GIP | Gibraltar pound |
| 57 | GMD | Gambian dalasi |
| 58 | GNF | Guinean franc |
| 59 | GTQ | Guatemalan quetzal |
| 60 | GYD | Guyanese dollar |
| 61 | HKD | Hong Kong dollar |
| 62 | HNL | Honduran lempira |
| 63 | HRK | Croatian kuna |
| 64 | HTG | Haitian gourde |
| 65 | HUF | Hungarian forint |
| 66 | IDR | Indonesian rupiah |
| 67 | ILS | Israeli new shekel |
| 68 | INR | Indian rupee |
| 69 | IQD | Iraqi dinar |
| 70 | IRR | Iranian rial |
| 71 | ISK | Icelandic króna |
| 72 | JMD | Jamaican dollar |

| 73 | JOD | Jordanian dinar |
|---|---|---|
| 74 | JPY | Japanese yen |
| 75 | KES | Kenyan shilling |
| 76 | KGS | Kyrgyzstani som |
| 77 | KHR | Cambodian riel |
| 78 | KMF | Comoro franc |
| 79 | KPW | North Korean won |
| 80 | KRW | South Korean won |
| 81 | KWD | Kuwaiti dinar |
| 82 | KYD | Cayman Islands dollar |
| 83 | KZT | Kazakhstani tenge |
| 84 | LAK | Lao kip |
| 85 | LBP | Lebanese pound |
| 86 | LKR | Sri Lankan rupee |
| 87 | LRD | Liberian dollar |
| 88 | LSL | Lesotho loti |
| 89 | LYD | Libyan dinar |
| 90 | MAD | Moroccan dirham |
| 91 | MDL | Moldovan leu |
| 92 | MGA | Malagasy ariary |
| 93 | MKD | Macedonian denar |
| 94 | MMK | Myanmar kyat |
| 95 | MNT | Mongolian tögrög |
| 96 | MOP | Macanese pataca |
| 97 | MRO | Mauritanian ouguiya |
| 98 | MUR | Mauritian rupee |
| 99 | MVR | Maldivian rufiyaa |
| 100 | MWK | Malawian kwacha |
| 101 | MXN | Mexican peso |
| 102 | MXV | Mexican Unidad de Inversion (UDI) (funds code) |
| 103 | MYR | Malaysian ringgit |
| 104 | MZN | Mozambican metical |
| 105 | NAD | Namibian dollar |
| 106 | NGN | Nigerian naira |
| 107 | NIO | Nicaraguan córdoba |
| 108 | NOK | Norwegian krone |
| 109 | NPR | Nepalese rupee |
| 110 | NZD | New Zealand dollar |
| 111 | OMR | Omani rial |
| 112 | PAB | Panamanian balboa |
| 113 | PEN | Peruvian Sol |
| 114 | PGK | Papua New Guinean kina |
| 115 | PHP | Philippine peso |
| 116 | PKR | Pakistani rupee |
| 117 | PLN | Polish zloty |
| 118 | PYG | Paraguayan guaraní |
| 119 | QAR | Qatari riyal |
| 120 | RON | Romanian leu |
| 121 | RSD | Serbian dinar |
| 122 | RUB | Russian ruble |

| | | |
|---|---|---|
| 123 | RWF | Rwandan franc |
| 124 | SAR | Saudi riyal |
| 125 | SBD | Solomon Islands dollar |
| 126 | SCR | Seychelles rupee |
| 127 | SDG | Sudanese pound |
| 128 | SEK | Swedish krona/kronor |
| 129 | SGD | Singapore dollar |
| 130 | SHP | Saint Helena pound |
| 131 | SLL | Sierra Leonean leone |
| 132 | SOS | Somali shilling |
| 133 | SRD | Surinamese dollar |
| 134 | SSP | South Sudanese pound |
| 135 | STD | São Tomé and Príncipe dobra |
| 136 | SVC | Salvadoran colón |
| 137 | SYP | Syrian pound |
| 138 | SZL | Swazi lilangeni |
| 139 | THB | Thai baht |
| 140 | TJS | Tajikistani somoni |
| 141 | TMT | Turkmenistani manat |
| 142 | TND | Tunisian dinar |
| 143 | TOP | Tongan pa?anga |
| 144 | TRY | Turkish lira |
| 145 | TTD | Trinidad and Tobago dollar |
| 146 | TWD | New Taiwan dollar |
| 147 | TZS | Tanzanian shilling |
| 148 | UAH | Ukrainian hryvnia |
| 149 | UGX | Ugandan shilling |
| 150 | USD | United States dollar |
| 151 | USN | United States dollar (next day) (funds code) |
| 152 | UYI | Uruguay Peso en Unidades Indexadas (URUIURUI) (funds code) |
| 153 | UYU | Uruguayan peso |
| 154 | UZS | Uzbekistan som |
| 155 | VEF | Venezuelan bolívar |
| 156 | VND | Vietnamese dong |
| 157 | VUV | Vanuatu vatu |
| 158 | WST | Samoan tala |
| 159 | XAF | CFA franc BEAC |
| 160 | XAG | Silver (one troy ounce) |
| 161 | XAU | Gold (one troy ounce) |
| 162 | XBA | European Composite Unit (EURCO) (bond market unit) |
| 163 | XBB | European Monetary Unit (E.M.U.-6) (bond market unit) |
| 164 | XBC | European Unit of Account 9 (E.U.A.-9) (bond market unit) |
| 165 | XBD | European Unit of Account 17 (E.U.A.-17) (bond market unit) |
| 166 | XCD | East Caribbean dollar |
| 167 | XDR | Special drawing rights |
| 168 | XOF | CFA franc BCEAO |
| 169 | XPD | Palladium (one troy ounce) |
| 170 | XPF | CFP franc (franc Pacifique) |
| 171 | XPT | Platinum (one troy ounce) |
| 172 | XSU | SUCRE |

| 173 | XTS | Code reserved for testing purposes |
|---|---|---|
| 174 | XUA | ADB Unit of Account |
| 175 | XXX | No currency |
| 176 | YER | Yemeni rial |
| 177 | ZAR | South African rand |
| 178 | ZMW | Zambian kwacha |
| 179 | ZWL | Zimbabwean dollar A/10 |
| 180 | XBT | Bitcoin |

### 101. MoneyPerArea

For MoneyPerArea, it is suggested to send two bytes: one for the Money, followed by an Area constant. As an example, if we want to send the price of land as € 2500 per hectare, transmitted as a double, this would be coded as:

| 101 | 50 | 21 | 0x40 | 0xA3 | 0x88 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 |

### 102. MoneyPerEnergy

For MoneyPerEnergy, it is suggested to send two bytes: one for the Money, followed by an Energy constant.

### 103. MoneyPerLength

For MoneyPerLength, it is suggested to send two bytes: one for the Money, followed by a Length constant.

### 104. MoneyPerMass

For MoneyPerMass, it is suggested to send two bytes: one for the Money, followed by a Mass constant.

### 105. MoneyPerTime

For MoneyPerTime, it is suggested to send two bytes: one for the Money, followed by a Duration constant.

### 106. MoneyPerVolume

For MoneyPerVolume, it is suggested to send two bytes: one for the Money, followed by a Volume constant.