ZKRD Zentrales
Knochenmarkspender-
Register Deutschland

FML

July 20, 2020

Flexible Message Language

Zentrales Knochenmarkspender-Register für die
Bundesrepublik Deutschland gemeinnützige GmbH
Handelsregister Ulm, HRB 2566

P.O.B. 4244, 89032 Ulm, Germany
Helmholtzstr. 10, 89081 Ulm, Germany
Phone: +49 731 1507-000

Contents

1. Revision History

Version	Date	Comment
1.00	30 Oct 2002	FML Flexible Message Language
2.00	03 Mar 2005	Incorporated FML Supplementary document by Werner Bochtler
2.01	05 Feb 2007	Replaced the term "low line" with underscore. Correctly specified the domain for data_value.
2.02	09 Feb 2007	Specified domain for count_value
2.03	09 Mar 2020	Added note regarding newlines

2. The FML Concept

The development of the European and the German national communication system for the unrelated bone marrow donor search (EMDIS[1] and GERMIS[2]) respectively, required the concept of a message language for the exchange of data between applications in a network (WAN, LAN). For that purpose, ZKRD developed the Flexible Message Language - short FML, which shall be explained in the following. The main goals for the language were:

- The language should be both human readable and understandable as well as "computer parseable" i.e. suitable for automated processing.
- It should be simple to generate FML messages. A programmer using this language should especially not be concerned about order of elements, line length etc.
- It should also be very easy to implement changes of a message definition as well as adding a new message type.

[1] European Marrow Donor Information System
[2] German Marrow Donor Information System

3. Syntax

This chapter will define the FML language in a somehow informal way. It will not give the complete formal definition of the language e.g. in Backus-Naur-Form, but it will use syntax diagrams to define the most important properties of the language[1].

3.1 General

FML is *not* line oriented. This simplifies the generation of FML text from within programs, because they do not have to care about where to put in newline characters for readability[2]. However, to avoid problems with email programs, the line length should be limited to a maximum of 80 characters.

Blanks and tabs are also not significant. They can be used to make the FML text more readable for humans.

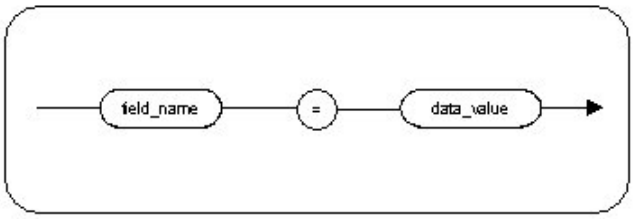
FML allows for so-called trailing comments, i.e. every character after and including the comment sign, which was chosen to be the pound sign "#", up to the end of the physical line is ignored.

The valid message types and their fields are defined in a so-called *Message Definition File* - short MDF. The build up of the MDF depends on the concrete realisation of the FML parser.

3.2 Assignment

Since FML was designed for data exchange, the fundamental importance of assignments is obvious. They are defined as follows:

Syntax diagram: *assignment*



field_names may only contain alphanumeric characters (incl. underscore). The valid field names for a message type are defined in the MDF. *data_values* are scalar values and may consist of any character but newline. However, they must be quoted with single or double quotes if they are not a character string of type WORD or NUMBER.

Definition: A string consisting of:

`[A-Za-z0-9_]`	is of type `WORD`
`[-+]?[0-9]+(\.[0-9]*)?`	is of type `NUMBER`

[1] Syntax diagrams and BNF have the same expressiveness

[2] Newlines are only allowed outside of *data_values*, see section 3.2

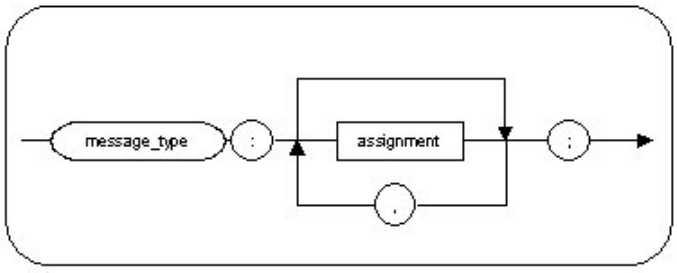
Examples:

```
city =                  # correct
city =    ""            # correct
city =    ''            # correct
city =    "'            # not correct
city =    '"            # not correct
city =    """"          # not correct
city =    "''"          # correct
city =    '""'          # correct
city =    Neu Ulm       # not correct
city =    Neu_Ulm       # correct
city =    "Neu Ulm"     # correct
city =    'Neu Ulm'     # correct
city =    'Neu Ulm"     # not correct
city =    "Neu Ulm'     # not correct
city =    ""Neu Ulm""   # not correct
city =    "'Neu Ulm'"   # correct
city =    '"Neu Ulm"'   # correct
```

3.3 Statement

A statement is defined as the concrete realisation of a certain message type. Note, that the order of the assignments in a statement do not necessarily have to correspond with the field order defined in the MDF.

Syntax diagramm: *statement (simple)*



message_types may only contain alphanumeric characters (incl. underscore). The valid message types are defined in the MDF. The part of the *message_type* up to the colon is called *Statement-Header*. The part after the colon is called *Statement-Body*.

Example:

```
address :  name = ZKRD, street = Helmholtzstrasse, city = Ulm;
```

The Statement-Header needs only to be written for the first statement in case of a sequence of several statements with the same message type. Statements without a header are called *pure statements*.

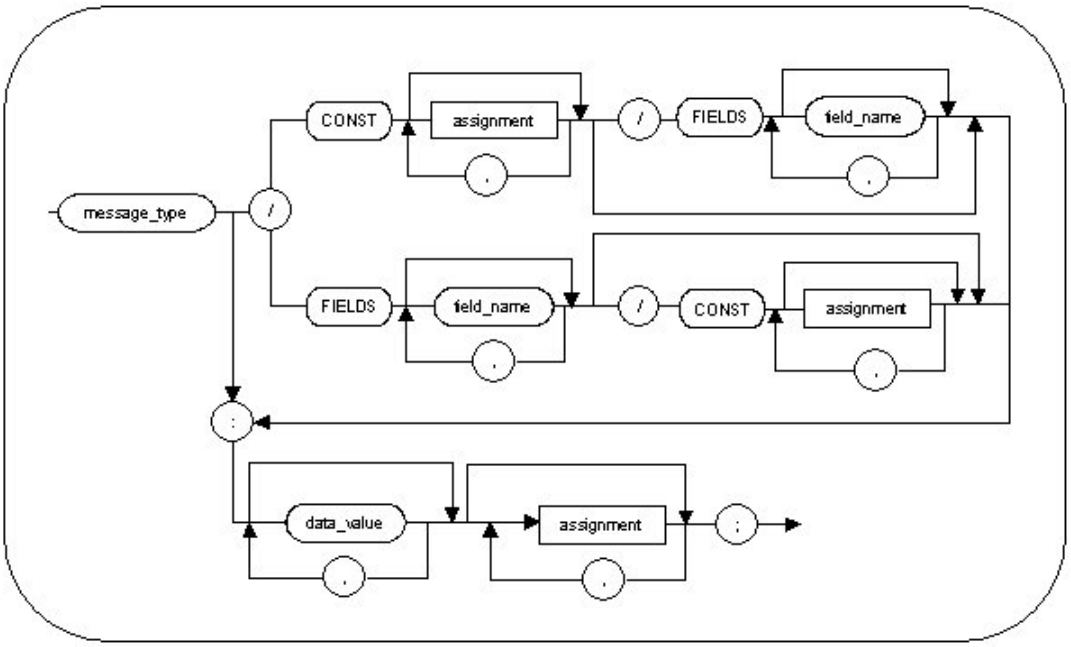
Example:

```
address:  name = ZKRD, street = Helmholtzstrasse, city = Ulm;

          name = DRK, street = "Helmholtzstrasse", city = Ulm;
```

The statement definition introduced above is called *simple form*. By adding the qualifiers *CONST* and *FIELDS* one gets the so-called *extended form*. This form allows a more compact and clear representation of the data and is particularly useful for the transfer of big amounts of data.

Syntax-Diagramm: *statement (extended)*



The qualifier *CONST* starts a sequence of assignments which is valid for all following *pure statements*. This construction allows to remove possible redundancy from succeeding statements and makes them shorter. The assignments defined by ***CONST*** are valid until a new header is written.

The qualifier *FIELDS* allows the transfer of compact list in tabular form. The assignments of a statement are split up for this reason. The *field_names* are given in the list following the qualifier and the matching *data_values* are put after the header in the according sequence.

The order of the two qualifiers is arbitrary but each of them may only appear once in the header. The *field_names* of the *CONST*-list, the *FIELD*-list and the normal assignments at the end of the statement body have to be disjoint.

Examples:

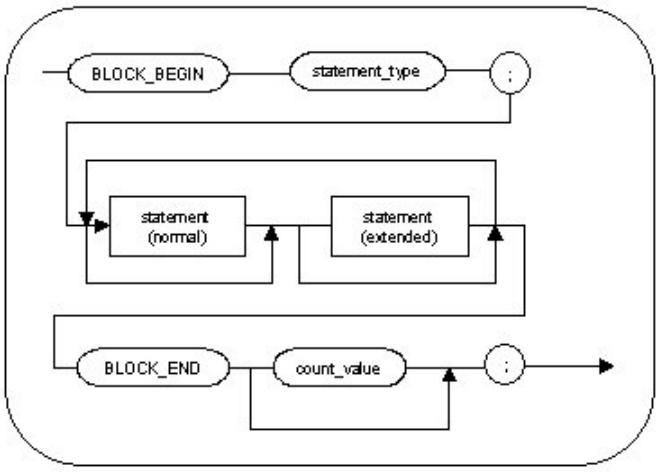
```
address /FIELDS name, street, city :
ZKRD, Helmholtzstrasse, Ulm;

address /CONST city = Ulm, street = Helmholtzstrasse /FIELDS name :
ZKRD;
DRK;
```

3.4 Statement block

The block construction was introduced for two reasons. First, the completeness of a list of statements of the same type should be checked and second, the grouping of logically related statements to a unit should be made possible.

Syntax diagramm: *block*



The value of the optional *count_value* gives the number of statements inside a block. Hence only unsigned integer values are allowed. It should be noticed, that a block may only contain statements of the message type which was specified at the beginning of the block. Further, nested blocks are inadmissible.

4. Semantics

Since the FML message set is not fixed, but is determined by an exchangeable or modifiable MDF, there is obviously no common semantics of the FML messages. Such a semantic has rather to be defined individually in dependence of the MDF used, the message types and their fields defined therein and under consideration of the requirements of the domain.

Regarding the message types and fields for GERMIS and EMDIS, documents containing the detailed semantics of these applications have been compiled but these are out of the scope of this document.

Furthermore, the following rules apply:

- Leading and trailing blanks in *data_values* are removed by the parser.
- An empty string (`""` or `''`) as *data_value* means that the field's content is empty and the corresponding value in the database has to be deleted or initially inserted as "empty" string.
- A single question mark (`"?"` or `'?'`) as *data_value* means that the field's content is undefined and the corresponding value in the database has to be left unchanged or initially inserted as *NULL* value. This so-called *undef-value* must under no circumstances be written into the database. The passing of the *undef-value* to the processing functions depends on the implementation of the parser.
- All missing fields of a statement are assigned the value `"?"` implicitly.
- The fields of assignments without *data_value* are assigned the value `"?"` implicitly.
- In case the realisation of the FML parser allows the specification of required or optional fields respectively in the MDF, the following applies:
  - `"?"` or `'?'` does not satisfy the required condition of a field.
  - `""` or `''` satisfies the required condition of a field.

Note further, that depending on the message type and the semantics, the chronological order of message generating and processing has to be kept.

5. FML - a relational language

The attentive reader may have noticed that there is a strong affinity between FML statements and DML[1] statements of relational database management systems (RDBMS). This is not surprising since FML is intended to be a communication language between instances of relational databases and the corresponding applications. In the following the relational origin of FML considering EMDIS as example will be elaborated.

From an abstract level the EMDIS communication system is nothing else but a homogeneous distributed database in which every single communication partner corresponds to a node of the database. The abstract database structure of the single node is defined by the set of EMDIS data messages and their fields which both are defined in the Message Definition File (MDF). The node's local data is manipulated by the partners via FML statements. We shall use a simple example to show that the FML concept follows exactly the rules of relational databases:

Given a relational database consisting of one table according to the database schema below:

```
CREATE TABLE t_patient (
pat_id       char(17)   Not NULL,
a1           char(4)    NOT NULL,
a2           char(4)    NOT NULL,
b1           char(4)    NOT NULL,
b2           char(4)    NOT NULL,
c1           char(4)             ,
c2           char(4)             ,
dr1          char(4)             ,
dr2          char(4)
)
```

The following primitive SQL data manipulations are possible regarding this table:

1. Insert a new record
   - providing all fields:

     ```
     INSERT INTO t_patient
     VALUES (1000, 1, "", 5, 7, NULL, NULL, 13, 14)
     ```

     respectively in long form:

     ```
     INSERT INTO t_patient
      (pat_id, al, a2, b1, b2, c1, c2, dr1, dr2)
     VALUES (1000, 1, "", 5, 7, NULL, NULL, 13, 14)
     ```

   - omitting some fields:

     ```
     INSERT INTO t_patient
      (pat_id, a1, a2, b1, b2, dr1, dr2)
     VALUES (1000, 1, "", 5, 7, 13, 14)
     ```

[1] data manipulation language

2. Update an existing record (all changes in boldface)
   - providing all fields:

     ```
     UPDATE t_patient
     SET pat_id = 1000,
         a1     = 1,
         a2     = "",
         b1     = 5,
         b2     = 7,
         c1     = NULL,
         c2     = NULL,
         dr1    = 13,
         dr2    = 15
     WHERE pat_id = 1000
     ```

   - omitting some fields:

     ```
     UPDATE t_patient
     SET dr2 = 15
     WHERE pat_id = 1000
     ```

3. delete an existing record

   ```
   DELETE FROM t_patient
   WHERE pat_id = 1000
   ```

Observations:

- SQL is an abstract language, i.e. the set of syntactically correct SQL statements is determined only by the tables and fields given in the database schema.
- Omitted fields are implicitly assigned the NULL value for INSERT. Fields that are defined as NOT NULL must neither be omitted nor explicitly be assigned the NULL value.
- Omitted fields are implicitly assigned with the already existing value for UPDATE i.e. these fields are not changed in the updated record. Fields that are defined as NOT NULL must not be assigned NULL in the update statement.

The database schema corresponds with the MDF in FML:

```
T_PATIENT
PAT_ID      REQ     STRING  17
A1          REQ     STRING   4
A2          REQ     STRING   4
B1          REQ     STRING   4
B2          REQ     STRING   4
C1          OPT     STRING   4
C2          OPT     STRING   4
DR1         OPT     STRING   4
DR2         OPT     STRING   4
```

Observations:
NOT NULL fields from the database schema become required fields in the MDF.
NULL fields from the database schema become optional fields in the MDF.

The data manipulation in EMDIS with regards to the abstract database defined by the MDF is limited to inserting new records and updating existing records. Deletions of records are not possible in the abstract database under consideration. One could think of using SQL itself for data manipulation in EMDIS. However, since only primitive, structural very similar INSERT and UPDATE operations are needed to manipulate the abstract database, SQL would be over complex and FML was designed as specific SQL replacement.

The SQL statements given above can be translated to FML as follows:

1. Insert a new record
   - providing all fields:

     ```
     T_PATIENT
     /FIELDS PAT_ID, A1, A2, B1, B2, C1, C2, DR1, DR2:
     1000, "1", "", "5", "7", "?", "?", "13", "14";
     ```

     respectively in long form:

     ```
     T_PATIENT:
     PAT_ID = 1000, A1 = "1", A2 = "", B1 = "5", B2 = "7",
     C1 = "?", C2 = "?", DR1 = "13", DR2 = "14";
     ```

   - omitting some fields:

     ```
     T_PATIENT
     /FIELDS PAT_ID, A1, A2, B1, B2, DR1, DR2:
     1000, "1", "", "5", "7", "13", "14";
     ```

     respectively in long form:

     ```
     T_PATIENT:
     PAT_ID = 1000, A1 = "1", A2 = "", B1 = "5", B2 = "7",
     DR1 = "13", DR2 = "14";
     ```

2. Update an existing record (all changes in boldface)
   - providing all fields:

     ```
     T_PATIENT
     /FIELDS PAT_ID, A1, A2, B1, B2, C1, C2, DR1, DR2:
     1000, "1", "", "5", "7", "?", "?", "13", "15";
     ```

     respectively in long form:

     ```
     T_PATIENT:
     PAT_ID = 1000, A1 = "1", A2 = "", B1 = "5", B2 = "7",
     C1 ="?", C2 = "?", DR1 = "13", DR2 = "15";
     ```

   - omitting some fields:

     ```
     T_PATIENT
     /FIELDS PAT_ID, A1, A2, B1, B2, DR1, DR2:
     1000, "1", "", "5", "7", "13", "15";
     ```

     respectively in long form:

     ```
     PAT_ID = 1000, A1 = "1", A2 = "", B1 = "5", B2 = "7",
     DR1 = "13", DR2 = "15";
     ```

Observations:

- FML is an abstract language, i.e. the set of syntactically correct FML statements is determined by the message types and fields given in the MDF.
- INSERT and UPDATE statements have identical syntax in FML. The recipient has to decide according to his local data whether he has to perform an INSERT or an UPDATE statement.

- The FML UNDEF value has to be inserted as NULL in the abstract database for INSERT. Omitted fields are implicitly assigned the UNDEF value (”?”). Required fields can therefore neither be omitted nor be assigned the UNDEF value for INSERT.
- A field with a FML UNDEF value in an UPDATE has to be assigned the already existing value in the abstract database (might be NULL). Omitted fields are implicitly assigned the existing database value for UPDATE i.e. these fields remain unchanged in the record to be updated.
- The FML Parser has no information whatsoever about the records in the abstract database. That is the reason why it can’t be distinguished between INSERT and UPDATE at parse time and that’s the reason why required fields must neither be assigned the UNDEF value nor be omitted.
- The primary keys of the abstract database have to be defined as required and have to be assigned allowed values in every FML statement. That’s the only way to search for an existing record in the database.

Remarks:

The concept of the abstract EMDIS database is of paramount importance. It is by no means required that the real database structure has to correspond with the structure of the abstract database. Non administrative FML statements however have to be interpreted as if a physically identical database structure would exist.

FML statements are nothing else but remote SQL statements which have to be executed by the recipient like the sender expects. The sender must rely on the abstract database structure at the recipient’s side since he has no knowledge about the physical database of the latter. It’s a major design goal of EMDIS to hide the real existing differences between physical databases by the concept of an abstract database (MDF). The recipient has to interpret and process the FML statements received in that sense and independent from his own physical database. Basically, the EMDIS core is nothing else but the implementation of the abstract EMDIS database. Core-less implementations have to provide that functionality at least on a logical level.

The FML UNDEF value is only allowed in the context of the abstract database.

Merging of data of different MDF entities is only allowed after their processing and ”logical” storage in the abstract database (or the core database respectively). On the level of the abstract database merging of different entities e.g. TYP_RES and DONOR_CB is not allowed.

Be aware that after merging of data of different FML entities in the physical database, the reconstruction of the original data, which is mandatory for further processing of subsequent FML messages, can be non trivial. A non trivial example is storing HLA values of a typing result message in the according donor record. Therefore the semantics should specify which messages - e.g. typing result and donor update - belong logically together in order to avoid the above mentioned non trivial problems.