USEMARCON

User Controlled Generic MARC Converter

Manual

USEMARCON - MARC Record Converter
Version 3.15
Adapted by Crossnet Systems Limited under contract to The British Library
Adapted by ATP Library Systems Ltd, Finland, 2002-2004
Adapted by The National Library of Finland, 2004-2011

January 2011

TABLE OF CONTENTS

1. INTRODUCTION

Background to the Project

Different national MARC (MAchine Readable Catalogue) standards are seen as barriers to wider exchange of bibliographic records in Europe and beyond. Throughout the world nearly 50 different MARC formats are currently in use, with 10 employed in the national libraries of European Community countries. Such variation is a fundamental problem for libraries wishing to obtain or supply data internationally and often results in the re-cataloguing of material for which records are readily available in formats other than the library's own. Lack of language expertise or knowledge of the context of publication can lead to the creation of records of a significantly inferior quality to those which might otherwise have been obtained from an agency in the country of publication.

The aim of the USEMARCON Project is to develop a generic conversion program capable of translating bibliographic records created in any ISO 2709 compatible MARC format (e.g UKMARC) into any other ISO 2709 MARC format (e.g. InterMARC) using UNIMARC as a central switching format. It is also possible to convert directly from a MARC format to another without using UNIMARC in between.

The development of the USEMARCON software was supported by the European Union's Telematics Applications Programme (DG XIII-E).

The partners of the USEMARCON consortium are drawn from a variety of library and information technology backgrounds and comprise the following:

Co-ordinating Partner
Koninklijke Bibliotheek, Holland

Full Partners
Instituto da Biblioteca Nacional e do Livro, Portugal
The British Library, UK

Associate Partner
Die Deutsche Bibliothek, Germany

Original Software Developer
Jouve, Systèmes d'Information, France

The original USEMARCON software was designed to provide users with two specific services.

1. The facility to convert MARC records compliant with a specified input format into MARC records compliant with a specified output format.

2. The facility to create and modify rules files, used to achieve MARC conversions, in order to meet specific local requirements. The present software is designed to be used by senior cataloguers or others with a detailed knowledge of the structure of the MARC formats they wish to convert between.

The USEMARCON Graphical User Interface (GUI) was designed for use with both MS Windows (3.1x and ‘95) and Unix Motif environments, in the latter case running under the Sunsoft Solaris operating system. In the Windows environment USEMARCON used a Multiple Display Interface.

Background to the Standalone Batch Version

The USEMARCON software developed on the USEMARCON project was heavily reliant on the XVT toolkit. Any modifications to the software by anyone required the XVT DSP++ development package.

The British Library published "A Feasibility Study To Investigate If The Usemarcon Software Can Be Adapted To A Command-Line Environment" in August 1999, which concluded that portability to command line scenario was possible, and this has been performed under contract. This version of the User Manual is applicable to the Standalone Batch version.

Initial Batch Version Software Developer
Crossnet Systems Limited, UK

Since the initial version the batch version was modified by ATP Library Systems Ltd. It is currently (as of 2005) maintained by Ere Maijala at the National Library of Finland. See the CHANGES file in the root directory of the package for more information.

Basic Concepts

Bibliographic data conversion requires the processing of:

- MARC bibliographic data, including related blocking and padding characters
- Conversion rules for specifying the conversion of each data element of the input format into each data element of the output format
- Coded data tables corresponding to sets of codes used by the format, e.g. language of publication, country of publication
- Character set tables specifying the translation of individual character codes between input and output formats
- Format checking tables specifying valid elements of input and output formats

MARC Formats

The MARC format family was created through a Library of Congress project, initiated in 1964, to prepare bibliographic information for automated processing. Different national MARC formats were created in response to specific national needs according to their local cataloguing rules and operating environments. The principles of MARC formats are to structure the bibliographic data into fields and subfields as in the following UNIMARC example:

Tag	Indicators	Subfield delimiter	Title proper	Subfield delimiter	Sub-title
200	1ƀ	$a	UNIMARC	$e	Cataloguing Manual

A tag is a three digit number that defines a bibliographic condition. Indicators are two single digit numbers which supply additional information about the contents of the field. Where an indicator is undefined the character position generally contains a blank (represented by ƀ). Subfield delimiters are introduced by a special control character (often conventionally represented by$) followed by a letter or a number qualifying the data element in the field. ($a, $b or $1, $2). Some fields contain a single fixed length subfield with the data coded in fixed positions. The format specifies whether each field and subfield are mandatory, optional or conditional upon another data element.

Brief Description of Functionality

Modular Design

The software has been developed to work on a modular basis allowing the user to perform at several levels of conversion. These can range from simple character set or exchange format translations using individual tables to full MARC format conversions using an entire 'cluster' of tables and rules.

USEMARCON reads all conversion parameters from an INI file. This INI file is a mandatory command line parameter to the program.

Bibliographic data conversion requires the processing of:

- MARC bibliographic data, including related blocking and padding characters
- Conversion rule for specifying the conversion of each data element of the input format into each data element of the output format
- Coded data tables corresponding to sets of codes used by the format, e.g. language of publication, country of publication
- Character set tables specifying the translation of individual character codes between input and output formats
- Format checking tables specifying valid elements of input and output formats

The software makes use of files containing simple ASCII tables rather than compiled files to carry out conversions. These files may therefore be created or customised to ensure conversions accurately reflect local requirements. All the tables may be created or edited using a text editor such as Windows Notepad. There is also a separate program developed by the National Library of Finland, USEMARCON GUI, that facilitates creating and running conversions. See http://www.kansalliskirjasto.fi/kirjastoala/formaatit/gui.html for more information (the page is currently only available in Finnish, but the program itself is in English).

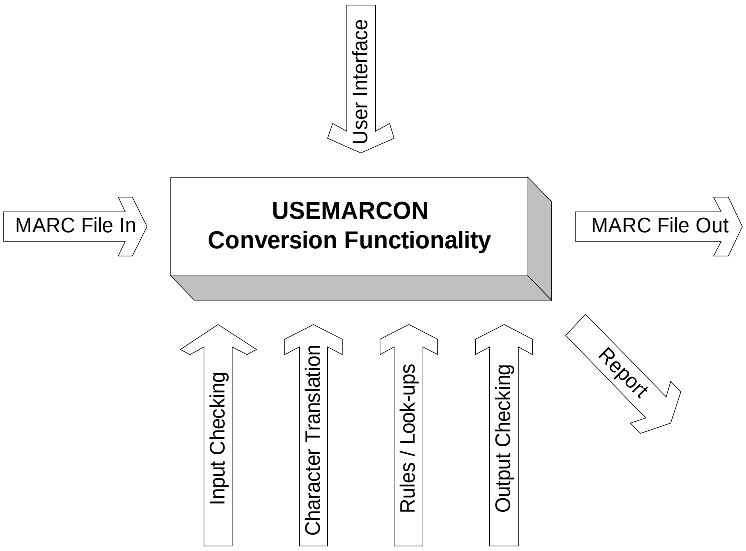
As the program is performing its conversion, it displays a running total of how many records have been read from the input file, and an approximate percentage of completion.

Details of processing problems are stored in a report file and classified by the following error types: input format, MARC checking in input, character translation, coded data translation, conversion, MARC checking in output and building of the output MARC file.

Diacritics and other extended characters are always displayed in hexadecimal code in order to allow users to check the accuracy of character translations.

Use of Tables

As the software makes use of simple ASCII tables rather than compiled rules to guide conversions, users can create or customise these files to ensure conversions accurately reflect local requirements. Visual representation of how USEMARCON makes use of tables to facilitate record conversion is shown below.



MARC Formats That May be Translated by USEMARCON

The USEMARCON program can deal with records compliant with the ISO 2709 standard for the exchange of bibliographic information (commonly known as the MARC Communications or Exchange format), MARCXML or MarcXchange. USEMARCON cannot process data in formats which do not conform to these standards (e.g. the German MAB or Dutch PICAPLUS formats).

In order to assist the processing of data from a variety of sources, users are provided with the facility to specify many details of structure used for ISO 2709 input and output files, including: blocking factors, use of segmentation characters, minimum size of usable data blocks and padding characters.

Conversions Available

USEMARCON does not come with prebuilt conversions. It only contains a sample UNIMARC to UKMARC conversion and provides the facility for creating such conversions. Many conversions have been developed by different organisations, so asking around might be beneficial. For example The National Library of Finland has built a conversion from MARC21-Fin to MARC 21 utilising many of the recent USEMARCON features, and while it wouldn’t be useful as is, it demonstrates many of the current capabilities.

2. INSTALLATION AND GETTING STARTED

Installation

The USEMARCON distribution zip package comes with a Windows executable version of USEMARCON. There is no installation program. It is sufficient to copy the program usemarcon.exe to a suitable directory and use it from there.

For Linux and UNIX systems an Autoconf/Automake build system is provided. See the accompanying BUILDING file for more instructions.

Running the USEMARCON Software

To run USEMARCON in Windows:

1) Open a command line window

2) Change directory to where the USEMARCON executable was installed or refer to the program using a valid path

3) Ensure that you have prepared an .ini file and that the associated conversion files are located as per the .ini file.

4) Run the software like:

```
  usemarcon <path>\my.ini [input file] [output file]
```

where path is the full or relative path name to where the .ini file is located.
You can substitute my.ini for your actual ini file name.
Input and output files are optional. If specified, they override the settings in the ini file. This could be used for example for batch processing of multiple files or creating a drag target on the desktop.

The software will then start the conversion and show status and possible errors.

You can also download and use USEMARCON GUI (see http://www.kansalliskirjasto.fi/kirjastoala/formaatit/gui.html). It provides a graphical user interface for Windows users.

Usage in Linux and UNIX systems is similar, but a slash must be used in the paths instead the backslash.

3. SYSTEM COMPONENTS

File Names

Creation and Maintenance of System Files

New directories for new conversions may be created at any time using any preferred file manager program or from the system command line. The files and tables required for the conversion may then be copied to or created in the new directory by the user. The system files supplied with the USEMARCON software are given defined suffixes detailed below in Description of System Files. In order to access easily the supplied system files it is recommended that the MS Windows Associate Application option is used to launch the preferred editor automatically after double clicking on a USEMARCON table or rules file. For further information on associating applications please refer to the MS Windows Help system.

A separate simple graphical user interface, USEMARCON GUI, is available from the National Library of Finland. It makes it somewhat easier to develop new conversions and test them. See http://www.lib.helsinki.fi/kirjastoala/formaatti/konversiot/gui.htm for more information (sorry, the page is currently only available in Finnish, but the program itself is in English).

File/Directory Naming Conventions

Since the program does not offer any form of rule editing, we advise that you retain any conventions for file names and extensions that you currently use.

Three letter filename suffixes must be used to denote the use of particular files. Suggested suffixes are:

`.ini` - program initialization file - *used with each conversion*

`.rul` - conversion rules file - *used with each conversion*

`.tbl` - conversion look up table - *only used in combination with a rules file.*

`.chk` - MARC format checking table - *(optional) used to validate MARC input and/or output files*

`.trs` - character set translation table - *(optional) used to convert from the character set of the input MARC format to that of the output format*

`.cnf` - MARC format configuration file - *only used in combination with MARC input and output files*

Description of System Files

Program Initialization Files

A program initialization (.ini) file lists all parameters necessary to perform a particular conversion. These include the names for each default system file, default parameters for processing input and output MARC files and preferred operation modes. The values contained in the .ini file may be modified using any Windows text editor such as Notepad. Beginning with version 1.3 the full paths for the files are not required anymore nor encouraged if they reside in the same directory as the INI file.

Rules Files

The rules governing a specific MARC conversion are held in a single rules file (.rul). The rules file describes precisely how the fields and subfields of the input MARC records should be converted into their counterparts in the output MARC format. This is achieved through the manipulation of input and output Content Designators (CDs), specific to each format, using a sophisticated rule syntax. The language used to create rules includes six main types of instruction:

- conditional
- boolean
- loop
- memory
- conversion

- translation

A rules file can be created or edited with any standard text processing program e.g. MS Windows Notepad or using the separate USEMARCON GUI.

Look Up Tables

Look up Tables (.tbl) are used to hold large amounts of tabular information (e.g. ISO Standard country codes) which are required by a conversion rules file. Data is held separately in a table to allow easier maintenance of the main rules file. Unlike other conversion files Look up Tables are commonly named to reflect their function e.g. ill.tbl for a table of illustration codes.

Format Checking Tables

A MARC Format Checking Table (.chk) describes the valid features of a particular MARC format. The table lists all fields and subfields of the format together with occurrence qualifiers and field indicator values. Use of these tables allows the input and output data to be checked for compliance with the specified format.

Character Set Translation Tables

Different MARC formats often require the use of different character sets. The USEMARCON system therefore incorporates the ability to convert between different specified character sets. This process is handled by the use of Character Set Translation Tables (.trs) which map individual input to output characters. The character set table need only deal with non-standard characters as conventional ASCII characters (e.g A-Z, a-z 1-9 etc.) are specified in a default character set table (standard.trs).

MARC Format Configuration Files

Format Configuration Files are used to signify fields in a particular MARC format which have no indicator values e.g. 001 or 008 in the UKMARC format.

Example Conversion

An example UNIMARC to UKMARC conversion can be found in *uni2uk* directory of USEMARCON distribution.

4. SYSTEM FILES

Overview

This Chapter describes the precise file formats for each of the system file types except the rule file which is dealt with in *Chapter 2.*

All the files are simple text files which may be created using a text editor such as MS Windows Notepad.

Conventional Abbreviations

In the summary of the format the following conventional abbreviations will be used:

```
Line         is a special part describing the whole line of a file
<name>       is a part of the description, which must be described
somewhere else
::=          describes the left part with the right part
             (<name> ::= <part1><part2>)
<p1><p2>     are two following parts of a description
<p1>|<p2>    are two possible descriptions of a part
[<p1>]       is an optional part
<p1>+        is a mandatory and repeatable part
<p1>*        is a repeatable but optional part
(<p1><p2>)+ () groups parts to give them an occurrence
'string'     is a literal character string
digit        is a decimal digit between 0 and 9
hdigit       is an hexadecimal digit between 0 and F
number       is a number
byte         is a number between 0 and 255
char         is a character other than a digit
string       is a character string
words        is a set of character strings
```

Initialization File (.ini)

The USEMARCON program uses a file to store a default set of system files and parameters to enable the user environment and system information to remain uniform between sessions. This file is used in MS Windows and Unix versions of USEMARCON. For example, the initialization file is used in the command line launching USEMARCON for a UK to UNIMARC conversion in the following way:

```
C:\usem\usemarcon.exe c:\usem\uk2uni.ini
```

The values contained in the .ini file may be modified using any text editor (such as Notepad in Windows). Please note that in some operating systems file names are case sensitive. Always use the correct case to ensure maximum compatibility between platforms.

Format

The file has a typical MS Windows structure:

```
[section]
entry=string
      .
      .
      .
```

Example:

```
[DEFAULT_FILES]
ErrorLogFile=errlog.txt
RuleFile=uni2uk.rul
TranscodingCharacterTable=iso2blec.trs
InputFormatCheckingTable=uni.chk
OutputFormatCheckingTable=uk.chk
MarcInputFile=UNI.MRC
```

```
MarcOutputFile=uk.mrc
InputMarcEditConfigurationFile=uni.cnf
OutputMarcEditConfigurationFile=uk.cnf

[DEFAULT_MARC_ATTRIBUTES]
IsInputBlockSegmentedChecked=
IsInputLastBlockPaddedChecked=
InputMarcSizeBlock=2048
InputMarcMinDataFree=5
InputMarcPaddingChar=5E
InputFileFormat=ISO2709

IsOutputBlockSegmentedChecked=FALSE
IsOutputLastBlockPaddedChecked=FALSE
OutputMarcSizeBlock=2048
OutputMarcMinDataFree=5
OutputMarcPaddingChar=5E
OutputFileFormat=ISO2709
OutputXMLRecordFormat=MARC21
OutputXMLRecordType=Bibliographic

[DEFAULT_VALUES]
MaxErrorsToBeEncountered=19999
OrdinalNumber=1
UpdateOrdinalNumber=FALSE

[DEFAULT_STATES]
IsVerboseExecutionModeChecked=
UTF8Mode=FALSE
ConvertInFieldOrder=TRUE

[DEBUG]
IsDebugExecutionModeChecked=FALSE
```

NB: Beginning with version 1.3 it is no longer necessary or encouraged to include full paths in the file names. USEMARCON can find the files if they reside in the same directory as the ini file.

The values in [DEFAULT_FILES] section define the files used in the conversion. RuleFile is the only mandatory file.

The values in [DEFAULT_MARC_ATTRIBUTES] section describe the layout of the MARC files. Segmented files seem to be rarely encountered.

InputFileFormat tells USEMARCON what the expected format of the input file is. Default is ISO2709. In most cases USEMARCON (as of version 3.13) can automatically detect if the input file is XML even when ini file has setting InputFileFormat=ISO2709. This can also be forced by setting InputFileFormat=MARCXML.

To make USEMARCON output MARCXML, set OutputFileFormat=MARCXML. Default for output is ISO2709. To process MarcXchange records, use MARCXCHANGE instead and set OutputXMLRecordFormat and OutputXMLRecordType to values to be written into the resulting XML. XML can also be converted to ISO2709 and vice versa by only specifying the input or output format. For example to convert MARCXML records to MarcXchange use the following settings:

```
InputFileFormat=MARCXML
OutputFileFormat=MARCXCHANGE
OutputXMLRecordFormat=MARC21
```

```
OutputXMLRecordType=Bibliographic
```

To convert ISO2709 to MARCXML:

```
InputFileFormat=ISO2709
OutputFileFormat=MARCXML
```

Note that the input XML is not validated for correctness. Either it can be successfully read or USEMARCON fails to find any records or fields.

[DEFAULT_VALUES] section controls different values.

- MaxErrorsToBeEncountered: maximum number of errors that can occur before the conversion is aborted.
- OrdinalNumber: a number that can be used during the conversion (see *Ordinal* instruction). It is automatically incremented for each record. **Unsafe** in situations where multiple conversions could be done simultaneously.
- UpdateOrdinalNumber: Controls whether OrdinalNumber in the .ini file can be updated after the conversion. This procedure **is not safe** if the conversion is used in an environment where multiple conversion can happen at the same time and defaults to being disabled.
- DuplicateSubfields: Controls how duplicate subfields are processed. Possible values are:
  - **leave** (or empty): **default**, no action is taken
  - **delete:** matching subfields in a field are deleted (case-sensitive)
  - **delete_ignore_case:** matching subfields in a field are deleted (case-insensitive)
  - **delete_smart**: matching subfields in a field are deleted (case-insensitive). The better-looking subfield (read: less capital letters) is kept.

  Note: case-insensitivity only works with UTF-8 records.
- DuplicateFields: Controls how duplicate fields are processed. Possible values are:
  - **leave** (or empty): **default**, no action is taken
  - **delete:** matching fields in a record are deleted (case-sensitive)
  - **delete_ignore_case:** matching fields in a record are deleted (case-insensitive)
  - **delete_smart**: matching fields in a record are deleted (case-insensitive). The better-looking field (read: less capital letters) is kept.

  Note: case-insensitivity only works with UTF-8 records.

[DEFAULT_STATES] section controls the following:

- IsVerboseExecutionModeChecked: Controls whether messages are printed when errors etc. happen.
- UTF8Mode: Controls whether string functions should treat strings as UTF-8. If TRUE, UTF-8 multibyte characters are considered a single character in for example *From* and *To* to make the string functions work properly when characters are counted. Defaults to FALSE.
- ConvertInFieldOrder: Controls whether the conversion is done in field order or the traditional rule order. Field order makes it easier to for example handle repeated fields that are not mapped 1 to 1 in a smart way. It also implies smart wildcard matching that doesn’t match a field already matched by a non-wildcard rule. **It is suggested that ConvertInFieldOrder be set TRUE for any new conversions.** Here is an example of a case where field order is necessary:

  Source data:

500 $a First note $b Additional info
500 $a Second note
500 $a Third note

Rules:

500$a | 500(1)$a | + If (D != '') Then '. ' + S Else S
500$b | 500(1)$a | + ' (' + S + ')'

Result:

500 $a First note (Additional info). Second note. Third note.

If using rule order the result might be something like this:

500 $a First note. Second note. Third note. (Additional info)

- HandleLinkedFields: Controls whether linked 880 fields are handled side by side with the original fields avoiding the need to duplicate rules for them. If TRUE, the special handling is enabled. Defaults to FALSE. Subfield $6 is automatically retained and updated. Side by side handling is only available for a subset of functions and won't work properly for control fields. Currently the following operations support linked fields:
    - Concatenation (+)
    - From
    - To
    - Between
    - Replace
    - ReplaceOcc
    - RegReplace
    - RegReplaceTable
    - MoveBefore
    - MoveAfter
    - MoveFirst
    - MoveLast
- ConvertSubfieldCodesToLowercase: If TRUE, all subfield codes from A to Z are converted to lowercase. Default is FALSE.
- SortRecord: If FALSE, does not sort the fields in the output records (except those created during the conversion using <XYZ notation). Default TRUE.

Look Up Table (.tbl)

Look Up Tables (.tbl) are used to hold large amounts of tabular information (e.g. ISO Standard country codes) which are required by a conversion rule file. Data is held separately in a table to allow easier maintenance of the main rule file. A rule can therefore be written which will 'call' the table rather than actually containing the information in the table.

Example:

```
008/16-17/  | 102$b     | Table ('Region.tbl')
```

Unlike other conversion files Look Up Tables are commonly named to reflect their function e.g. ill.tbl for a table of illustration codes.

Each Look Up table used in a Rule file is defined in a separate file. Each file bears the name of the table used in the Rule file.

Format

The first and second lines of the file may contain comments such as the name of the file, or a description of its function. The table itself must start on the third line of the file.

Each line of the table can be either:

- An #include line. This enables the loading of another table of coded data as part of the current file.

- A translation line. This contains the characters in the input record and the characters to which they are to be translated in the output record, separated by a pipe character.

The format of the translation line is as follows:

Character (or sequence of characters), space or tab, pipe character (|), space or tab, character (or sequence of characters).

i.e. `<characters in>   |     <characters out>`

Example:

```
pubcode.tbl

s | d
r | e
m | a
z | e
```

Optionally, a comment may be added to the end of the line. The comment must be preceded by two slashes (//).

i.e. `<characters in>   |     <characters out>//<comment>`

Example:

```
Genre littéraire Interuni

0x30 | a    //Fiction adulte
0x31 | a    //fiction adolescent
0x32 | a    //fiction enfants
0x33 | y    //document pédagogique enseignement supérieur
0x34 | y    //document pédagogique enseignement primaire
...
```

Each character can be described as its ASCII representation, as its hexadecimal value (preceded by 0x), or its decimal value.

Numbers must be described by their decimal or hexadecimal value e.g. 0x31 for 1, because 1 does not mean ‘1’ but the character whose ASCII value is decimal 1.

When typing words in a Look Up Table, each character of each word must be separated by a space. A set of words, normally separated by a blank character should be shown as separate words by the hexadecimal of the blank character i.e. 0x20.

Example:

```
538.tbl

0x30  |     I s s u e 0x20 n o . // 30 is the hex for ‘0’, 20 is the hex for
‘space’
0x31  |     M a t r i x 0x20 n o .
0x32  |     P l . 0x20 n o .
0x33  |     P u b l i s h e r ' s 0x20 n o .
```

Each character or sequence to be translated will either match or not match one character or sequence of the first column of the table. The translation software searches for the longest matching sequence. In order to prevent ambiguity with shorter sequences, the longest one will be chosen.

If a character or sequence in the input record matches a character or sequence in the first column of the table, it will be replaced in the output record by the character or sequence in the second column of the table.

A default may also be set to take care of cases where no sequence in the input record matches a sequence in the input column.

The #DEFAULT entry must be entered in the form:

#DEFAULT, space or tab, pipe, space or tab, character (or sequence of characters)

i.e. `#DEFAULT    |     <characters out>`

Example:

```
aud.tbl

a | j
b | j
c | j
d | j
#DEFAULT | 0X20
```

In the example above, if the input record does not contain a, b, c or d then the blank, represented by the hex code 20, will be inserted in the appropriate position in the output record.

A magic token COPY can be used as the characters out. This will make the default action copy the input directly to output without conversion. This way only the characters that need conversion need to be listed and others can be copied as is.

Example:

```
aud.tbl

a | j
```

```
b | j
c | j
d | j
#DEFAULT | COPY
```

If no sequence in the first column matches and a #DEFAULT entry does not exist, the input sequence of word(s) will be suppressed in the output and an error message will be logged in the report file.

Summary:

```
Line ::= <include line> | <translation line>
<include line> ::= '#include  'string' '
<translation line> ::= <words in> '|' <words> [// words]
<words in> ::= <word>+ | #DEFAULT
<words> ::= <word>+
<word> ::=  (<character> ‘ƀ’)*
```

Format Checking Table (.chk)

A MARC Format Checking Table (.chk) describes the valid features of a particular MARC format. The table lists all the fields and subfields allowed in records, their occurrences and the ordinal number of fixed length fields or subfields.

For each MARC format conversion two tables are required: the first to check input records (with a particular format) and the second to check output records

Use of these tables allows the input and output data to be checked for compliance with the specified format.

The UNIMARC to UKMARC Format Checking Tables can be found in *Appendix 4*.

Format

A Format Checking Table is a simple text file. The first line may contain a comment such as the name of the format.
Each following line can be either:

- An #include line. This enables the loading of another checking table as part of the current file.
- A description line. This describes a field, its occurrence, its indicators and its subfields.

Each line must have the following structure :

1. A field tag e.g. 245
2. A character from the following list:
   - _ means that the field is mandatory, not repeatable,
   - + means that the field is mandatory and repeatable,
   - ? means that the field is neither mandatory nor repeatable,
   - * means that the field is not mandatory but can be repeatable.
3. Space or tab, pipe character (|), space or tab.
4. I1=
   This should be followed by a list of valid values for the first indicator. Blank should be represented by _. The fill character (|) is represented by a space then the hex value e.g. 0x7C
5. Space or tab, pipe character (|), space or tab.
6. I2=
   This should be followed by a list of valid values for the second indicator. Blank should be represented by _. The fill character (|) is represented by a space then the hex value e.g. 0x7C
7. Space or tab, pipe character (|), space or tab.
8. A list of the subfield delimiters allowable for the tag.
   The delimiters are introduced by a control character, usually represented by $. This is followed by a letter or digit identifying the subfield e.g. $a.
   The delimiter is then followed by a character from the list at 2 above, to indicate its repeatability, etc.
   The delimiters must be separated from each other by space or tab, pipe character (|), space or tab.
9. An optional text comment preceded by two slashes (//).
   This may also be inserted above the field tag. Note that // is very strong, and if it’s needed in a string literal, the slashes must be escaped with backslashes (e.g. 'http:\/\/www..').

This file enables the verification of the occurrence of fields or subfields, but does not verify that a particular field or subfield may only be repeated a particular number of times e.g. twice. This file only verifies fixed structures and if the occurrence of a field/subfield is linked to particular Indicator values these cannot be checked.

Each field found in a record is compared with this table and if an error is detected, it will be logged in the Error Log File (see Chapter 3).

Example:

```
UNIMARC format
…
#include  deb_uni.fmt
…
001_  |      I1=    |      I2=    |

100_  |  I1=_    |  I2=_     |  $a_

101_  |  I1=012 0X7C  |  I2=_ |$a*  |  $b*  |  $c*  |  $d*  |  $e*  | $f*  |  $h*  |  $i*  |
$g?  // Mandatory, not repeatable. I1 can be fill character. $g optional not repeatable, rest
optional & repeatable
…
200_  |  I1=01  |  I2=_     |  $a+ | $b* | $c* …                        //
Title
700*  |  I1=_    |  I2=012 |  $a_ | $b? | $c? | $d? | $f? …             //
Author
```

Summary:

```
Line ::= <include line> | <checking line>
<include line> ::= '#include  'string' '
<checking line> ::= <field tag> '|' <I1> '|' <I2> '|' <subfields> [// words]
<field tag> ::= <tag><occurrence>
<tag> ::= number
<occurrence> ::= _|+|?|*
<I1> ::= 'I1='<indicator value>*
<I2> ::= 'I2='<indicator value>*
<indicator value> ::=_|digit
<subfields> ::= <subfield>['|'<subfields>]
<subfield> ::=<subfield tag><occurrence>
<subfield tag> ::= '$'char|digit
```

Character Set Translation Table (.trs)

Different MARC formats often require the use of different character sets. The USEMARCON system therefore incorporates the ability to convert between different specified character sets. This process is handled by the use of Character Set Translation Tables (.trs) which map individual input to output characters. Conversion for basic ASCII characters can be included from a default character set table (standard.trs). A #DEFAULT rule can also be used (see below).

The UNIMARC to UKMARC Conversion Character Set Translation Tables can be found in *Appendix 5*.

Format

The first and second lines of the file may contain comments such as the name of the file, or a description of its function. The table itself must start on the third line of the file.

Each line of the table can be either:

- An #include line. This enables the loading of another table of coded data as part of the current file.
- A translation line. This contains the characters in the input record and the characters to which they are to be translated in the output record, separated by a pipe character.

The format of the translation line is as follows:

Character (or sequence of characters), space or tab, pipe character (|), space or tab, character (or sequence of characters). Optionally, a comment may be added to the end of the line , which must be preceded by two slashes (//).

i.e. `<characters in>   |     <characters out>//<comment>`

Each character can be described as its ASCII representation, as its hexadecimal value (preceded by 0x), or its decimal value.

Numbers must be described by their decimal or hexadecimal value (0x31 for 1), because 1 does not mean ‘1’ but the character whose ASCII value is 1.

Each character or sequence to be translated will either match or not match one character or sequence of the first column of the table. The translation software searches for the longest matching sequence. In order to prevent ambiguity with shorter sequences, the longest one will be chosen (see example below).

If a character or sequence in the input record matches a character or sequence in the first column of the table, it will be replaced in the output record by the character or sequence in the second column of the table.

A default may also be set to take care of cases where no character sequence in the input record matches a sequence in the input column.

The #DEFAULT entry must be entered in the form:

#DEFAULT, space or tab, pipe, space or tab, character (or sequence of characters)

i.e. `#DEFAULT    |     <characters out>`

Example:

```
char.tbl

a | a
b | b
c | c
d | d
#DEFAULT | 0X20
```

In the example above, if the input character sequence is not a, b, c or d then it will be replaced with the blank, represented by the hex code 20.

A magic token COPY can be used as the characters out. This will make the default action copy the input directly to output without conversion. This way only the characters that need conversion need to be listed and others can be copied as is.

Example:

```
aud.tbl

ä | -
#DEFAULT | COPY
```

If no character or sequence in the input record matches a character or sequence in the first column and #default is not specified, the current input character is written in the output record and an error message is generated.

Example:

```
iso5426
ASCII
#include  basic.trf
0xA3        |     0x9C  // Pound character - simple translation
0xCA        |     0xF8  // Degree - if 0xCA appears before an A see below
0xCA A      |     0x8F  // A with circle above
0xBF        |           // The inverted question mark is not
            // translated.
```

Summary:

```
Line ::= <include line> | <translation line>
<include line> ::= '#include  'string' '
<translation line> ::= <chars> '|' <chars> [// words ]
<chars> ::= <char>+
<char> ::= <hexa>|<deci>| char
<hexa> ::= 0xhdigit hdigit
<deci> ::= byte
```

Configuration File (.cnf)

This file describes fields that have no indicator in the corresponding MARC file. There is one configuration file for the Input MARC file and one for the Output MARC file.

A wildcard character (?) can be used to replace one character. For Example 00? means any field with a tag beginning with 00.

See *Appendix 6* for the UNIMARC to UKMARC Conversion Configuration Files.

Format and Example

```
00?
```

```
010
998
999
```

If no configuration file is provided, a MARC field is treated as having no indicators if a subfield delimiter ($) is found within the first two positions of the field contents. If the tag is of the format 00X i.e. is a control field, and no subfield delimiter appears in the field contents, it is treated as having no indicator. If the tag is of the 00X format but a subfield delimiter is found after the first two positions of the field contents, the first two positions are treated as being indicators. Every field which begins with anything other than 00X and which does not have $ in the first two positions of the field contents, are treated as having indicators.

Rule Files (.rul)

In order to convert records from one format to another the software must be given instructions on what to do with each part of each field and subfield of the input record, in order to create the output record.

These instructions are written in a rule file. There is no generic rule file. The instructions written in a rule file are specific to the format of the records in the database to be converted, and the required format of the converted records in the database.

The rules governing a specific MARC conversion are held in a single rule file (.rul). The rule file describes precisely how the fields and subfields of the input MARC records should be converted into their counterparts in the output MARC format. This is achieved through the manipulation of input and output Content Designators (CDs), specific to each format, using a sophisticated rule syntax. The language used to create rule includes six main types of instruction:

- conditional
- boolean
- loop
- memory
- conversion
- translation

5. RULE FILES

Overview

This chapter gives the details of how rule files are created. The UNIMARC to UKMARC sample can be found in *uni2uk* directory of the distribution.

Format

The first and second lines of the file may contain comments such as the name of the file, or a description of its function. The actual rules must start on the third line of the file.

Each line of the file can be either:

- An #include line. This enables the loading of another file as part of the current file.
- A #define line. This enables defining the following (see the example below for example of the syntax):
  - Macros that can be used in the rules. Macros can be used anywhere in the file after their definition. Use distinct macro names that cannot be confused with anything else in the rule file. **With** command can be used to set parameters inside the macro. See the example below. Macros are expanded in the order they are entered in the rule file.
  - **Note**: If you use a macro with multiple instructions inside an **If, While, For or With** clause, surround the instructions in the macro with block markers **{** and **}**. Otherwise only the first instruction is taken as the one associated with the condition.

    Example:

    ```
    ...
    #define macro FieldRef | { Mem(99);\
                               If (FLD=880) Then {\
                                 Sto(99, Val(S) + 1); Mem(99);\
                                 If (S < 10) Then 'FLD-0' + S Else 'FLD-' +
    S;\
                               }\
                               else\
                               {\
                                 If (S < 10) Then 'FLD-0' + S + '/(N'\ Else
    'FLD-' + S + '/(N';\
                               }\
                             }\

    // Clear memory, do nothing else
    000 | 000 | '0'; Sto(99); If (0=1) Then S

    022I1       | 022I1         | S
    022I2       | 022I2         | S
    022         | 022$6         | With (FLD=880) FieldRef
    022$a       | 022$a         | Table ('kyril2latinma21.tbl')
    022$y       | 022$y         | Table ('kyril2latinma21.tbl')
    022$z       | 022$z         | Table ('kyril2latinma21.tbl')

    022I1       | <AC2I1        | S
    022I2       | <AC2I2        | ' '
    022         | <AC2$6        | With (FLD=022) FieldRef
    022         | <AC2          | + S
    AC2I1       | 880(new)I1    | S
    AC2I2       | 880(newest)I2 | S
    AC2         | 880(newest)   | S
    ```

  - Required rule version number. This can be used verify that the USEMARCON version used actually does what is expected in the rules. Version number is an integer value, e.g. 300 (the currently supported version).

    Example:

    ```
    ...
    #define version 300
    ```
- An #if line together with #endif line. This designates a group of rules that are executed conditionally. Only a single level is currently supported, so #if’s cannot be nested.

  Example:

```
#if (Tag='600' And I1='2')
600I1        | 500I1        | S
600I2        | 500I2        | S
600$a        | 500$a        | S
#endif
```

- A translation line. This contains the characters in the input record and the characters to which they are to be translated in the output record, separated by a pipe character.

If a backslash is appended to the end of the line, it will be merged with the next one. For example:

```
First line\
Second line
```

Is parsed to:

```
First line Second line
```

The format of the translation line is as follows:

1. A source CD description.
2. Space or tab, pipe character (|), space or tab.
3. A destination CD description.
4. Space or tab, pipe character (|), space or tab.
5. The description of the rule which will enable the conversion of the source CD(s) into the destination CD
6. An optional text comment preceded by two slashes (//)

(See 'Content Designators' in *Chapter 2* for the definition of a CD)

N.B. Backslash is the escape character. If you need literal **$** in the rule, it needs to be escaped by prepending it with a backslash: **\$** (otherwise it is translated to a subfield delimiter). Any character can be presented in hexadecimal format by prepending the hex value with a backslash (e.g. n = **\6e**). A backslash is presented with \\.

```
#define macro Replace_A() | { Replace('A' By 'XAN');\
                              Replace('-' By '/'); }

245    | 200I2               | ' '
245I1  | 200I1               | If (n=1 Or n=3) Then 1 Else 0
245$a  | 200$a               | If (I2 = '0') Then S
245$a  | 200$a               | If (I2 != '0') Then Sto(0);
                               Between(1,Val(I2), Strict); Sto(1);
                               Mem(0); From(Val(I2)); Sto(2);
                                  '\88'+Mem(1)+'\89'+Mem(2)
245$b  | 200$e               |
245$e  | 200$f               | If (n=1) Then S
245$e  | 200$g               | If (n>1) Then S
245    | 200                 | Sort ('abefg'); With(XAN=B) Replace_A();  //
Replaces 'A' with 'B'
```

```
...
```

Summary:

```
Line ::= <include line> | <translation line>
<include line> ::= '#include  'string' '
<translation line> ::= <source CD> '|' <destination CD> '|' <rule> [// words ]
<source CD> ::=  described in Chapter 2
<destination CD> ::=  described in Chapter 2
<rule> ::=  described in Chapter 2
```

Contents of a Bibliographic Record

A bibliographic record contains :

- A **record label** (or Leader) containing fixed position information. For convenience this is treated as an ordinary field with the arbitrary field tag 000. (In MARC records conforming to ISO 2709, the record label has no tag, indicators or subfields.)
- A set of **fields** each of which is identified by an *aaa* tag (where *a* is either a numeric or an uppercase letter).

A bibliographic record can therefore be considered as a set of fields from 000 to ZZZ. Fields may be either repeatable or not and they may also be mandatory or not.

In addition, fields may be divided into **control fields** (tags numbered 000-009) and **data fields** (tags numbered 010-ZZZ). Control fields contain no indicators and no subfields.

Each data field contains:

- A set of indicators each of one character or one digit. Most formats have two. Throughout each indicator is treated as a particular subfield named I1 or I2.
- A set of subfields: each subfield begins with a control character followed by a character or digit indicating the ‘name’ of the subfield e.g. $a. Subfields may be repeatable or not and they may also be mandatory or not.

A data field can be therefore be treated as a set of subfields like I1, I2 or $*x* (where *x* is a digit or a letter).

Each field or subfield can be either:

- A 'normal' field or subfield, containing text.
- A 'fixed position' field or subfield. When constructing rules, character positions within a ‘fixed position’ field or subfield must be numbered from 1 and not from 0 as in some formats. For example, the ‘Date entered on file’ in UNIMARC 100$a is described as 100$a/1-8/ not 100$a/0-7/ as it is in the *UNIMARC Manual.*

Content Designators

To create a rule which will convert a particular part of a record, you must describe in which Tag, and in which occurrence of that Tag the part is.

If you do not wish to access a complete Tag, you must describe which subfield and which occurrence of that subfield you want to access.

If the (sub)field is a fixed position subfield, you must describe which position of that (sub)field you wish to access.

The description above, enables access to one particular part of a record. The description of this particular part will be named the **Content Designator(CD)**. The CDIn is the Content Designator for the input file and the CDOut is the Content Designator for the output file.

The structure of a rule file line follows the pattern:

CDIn | CDOut | conversion rule

The three elements are separated by a pipe (|).

In a rule file, a CD may be defined as follows:

[@|%][TTT[(n_1)]] [SS[(n_2)]] [/n_3[-n_4]/]

@ Is only used in conversion rules, not in the CDIn or the CDOut. It is used to avoid numerical ambiguity in a conversion rule i.e. 100 could either mean the value 100, or the field 100. @100 makes it clear that the rule is referring to the field 100.

% Is used to denote a converted (output) field. Normally, a CD references the input data, but prepending it with a percentage sign can be used to peek at the conversion results.

TTT Represents the field tag:
000 for the record label
001 to **ZZZ** for other fields
0?? to match all fields beginning with 0
If this part of CD is omitted, the current tag is assumed

(n_1) Represents the tag occurrence number; see chapter Rule File Structure for more information.

SS Represents the subfield mark:
I1 for first indicator
I2 for second indicator
$a to **$z**, **$A** to **$Z**, and **$0** to **$9** for other subfields
$? for any subfield
If this part of CD is omitted, the complete field is assumed

(n_2) Represents the subfield occurrence number; see chapter Rule File Structure for more information.

/[n_3[-]n_4]/ Represents the character positions in the (sub)field:
n_3 means only character position n_3
n_3- means character position n_3 to the end of the (sub)field
-n_4 means character positions from beginning of the (sub)field to n_4
If both n_3 and n_4 are present, characters from position n_3 to n_4 are assumed
In rule files, 1 is always used to describe the first character position, not 0.
If this part of CD is omitted, the complete (sub)field is assumed

Examples:

```
001                 The complete field 001
000/5/              Character position 5 of the record label
000/11-12/          Character positions 11 to 12 of the record
                    label
010$a               The (first) subfield $a of field 010
100$a/-8/           The first 8 character positions of the $a of
                    field 100
```

`123I1`	The first indicator of field 123
`701(`*n*`)$a`	The $a of *n*th field 701
`101$a(`*n*`)`	The *n*th $a of field 101
`123(`*nt*`)$a(`*ns*`)`	The *ns*th $a of *nt*th field 123
`???$?`	Any field with any subfield
`If (Exists(@040))`	Tests for the presence of tag 040

N.B. When converting in field order (ConvertInFieldOrder=TRUE), wildcard matching is only done for a field until a precise match is found. Therefore it’s possible to create rules for converting a single a field to another while preserving all other fields. In traditional rule order this smart matching is not done.

Link between Input CD(s) and Output CD(s)

A CD enables the user to address a particular part of a record.

To convert records from one format to another, a set of CDs in the input record must be linked to a set of CDs in the output record and a set of conversion rules.

Each output CD must be defined together with a set of input CDs and a conversion rule. In fact, one output CD is usually linked with only one input CD.

Each line in a rule file has the following format:

[CDIn] | [<][CDOut] | Conversion rule

CDIn is the most important CD for the creation of the CDOut.

A Rule will be evaluated only if a CDIn exists in the input record.
If the CDIn is omitted, the last CDIn in the file is assumed.

CDOut is the CD to be created (or modified).

If the CDOut is omitted, the last CDOut in the file is assumed.
If a ‘<‘ is placed before the CDOut, the CDOut will be added to the input record (in the memory), creating a pseudo-CDIn. This allows the user to re-convert the newly created CDIn, to manage complex rules, as rules cannot manipulate CDOut data. Normally i.e. when ‘<‘ is not present, the CDOut is added to the output record.

Rules describe how to create or modify the CDOut from the CDIn and other CDs, if necessary. If other CDs are necessary to create the CDOut, they can be accessed from within the rule.

Many rules are simple translations e.g.

```
111$a | 710$a |
```

```
111$a | 710$a | S
```

Either of the above can be used to place the S(ource) data of UKMARC 111$a into UNIMARC 710$a.

The 'pipe' symbol '|' must have at least one space or tab on each side of it. The two lines below are identical as far as the software is concerned.

```
111$a[space]|[space][space][tab]710$a[space]|[tab][tab]
```

```
111$a[tab]|[tab]710$a[tab]|[tab]
```

The following is a slightly more complex example:

```
[earlier commands]
660 | <66A$2 | 'COMPASS'
66A | 606 |
```

Here the conversion takes place in two stages: a temporary CDIn is created from the original CDIn then that CDIn is used to create a complete CDOut field

Here is an example of a simple conversion that removes field 509, moves 508 to 501 and leaves everything else as is:

```
Simple conversion to remove field 509 and move field 508 to 501.
Just an example.

508        | 501          | S
509        | <DEL         | S

#if (TAG != 'DEL')
???I1      | ???I1        | S
???I2      | ???I2        | S
???        | ???          | S
#endif
```

For the above to work properly, ConvertInFieldOrder must be TRUE in the ini file.

Occurrences

Each CD can be represented by three occurrence numbers :

- ***nt***: (no. of tag) represents the occurrence of the tag (field)
- ***ns***: (no. of subfield) represents the occurrence of the subfield in the tag
- ***n***: represents the global occurrence of the CD (field + subfield in the record)

Example:

	nt=1
	ns=1
	n=1
		nt=1
		ns=2
		n=2
			nt=1
			ns=1
			n=1
				nt=1
				ns=3
				n=3

123	$a...	$a...	$b...	$a...
123	$c...	$a...	$a...	

	nt=2
	ns=1
	n=1
		nt=2
		ns=1
		n=4
			nt=2
			ns=2
			n=5	

For input CDs, occurrences are accessed via *nt*, *ns* and *n*.
For output CDs, occurrences are accessed via *nto, nso* and *no*. These are normally used to create new a new field or subfield in the output.

Occurrences in CDIn and CDOut

As has been described previously, a rule is evaluated when the corresponding CDIn is found in the input record.

Each corresponding CDIn has particular occurrences (*nt*, *ns* and *n*).

From each CDIn the software must create or modify a particular CDOut and (*nt*, *ns* and *n*) must be linked with (*nto*, *nso* and *no*).

Beginning with version 1.70 it's possible to specify the absolute field or subfield index (occurrence) in the input or output CD. For example the following rule would $a from first 245 to the second 240 in the output:

```
245(1)$a | 240(2)$a | S
```

To describe this in a rule file, the easiest way is to describe CDIn with no occurrence number, CDOut with occurrence numbers (*nt, ns, n, nto, nso* or *no*), and allow rules to access these occurrence numbers.

Beginning with version 2.00 there is a proper way to always force USEMARCON to create a new field. This can be accomplished by using occurrence number *new*:

```
856$u | 856(new)$u | S
```

Further on, the latest occurrence can be referenced with *newest*:

```
650I1 | 650(new)I1 | S
650I2 | 650(newest)I2 | S
650$a | 650(newest)$a | S

652I1 | 650(new)I1 | S
652I2 | 650(newest)I2 | S
652$a | 650(newest)$a | S
```

The table below (not for the faint of heart) explains how each type of input occurrence can be linked to each kind of output occurrence:

R	Means the tag/subfield is repeatable,
NR	Means the tag/subfield is not repeatable.
BRule	Is the first character of the rule

CD In		CD Out				B	
Tag	Subfield	Tag	Occ	Subfield	Occ	Rule	COMMENT
NR	NR	NR		NR			The CDIn goes in CDOut
NR	NR	NR		R			Idem, only one occurrence of CDOut will exist
NR	NR	R		NR			Idem, only one occurrence of CDOut will exist
NR	NR	R		R			Idem, only one occurrence of CDOut will exist
NR	R	NR		NR		+	Each subfield of CD In will go in the same subfield of CD Out (they must be merged : + at beginning of rule signifies Destination+…).
NR	R	NR		R	*n*		Each occurrence of subfield in input will create a new occurrence of subfield in output.
NR	R	R	*N*	NR			Each occurrence of subfield in input will create a new occurrence of field in output.
NR	R	R		R	*n*		Each occurrence of subfield in input will create a new occurrence of subfield in output
NR	R	R	*N*	R			Each occurrence of subfield in input will create a new occurrence of field in output.
R	NR	NR		NR		+	Each occurrence of field in input will be merged in the same subfield (if + is omitted at the beginning of the rule, an error of format can occur in output).
R	NR	NR		R	*n*		Each occurrence of field in input will create a new subfield in the same field in output
R	NR	R	*N*	NR			Each occurrence of field in input will create a new occurrence of the field in output
R	NR	R		R	*n*		Each occurrence of field in input will create a new subfield in the same field in output
R	NR	R	*N*	R			Each occurrence of field in input will create a new occurrence of the field in output
R	R	NR		NR		+	Each occurrence of field and/or subfield will be merged in the same subfield in output (if + is omitted at the beginning of the rule, an error of format can occur).
R	R	NR		R	*n*		Each occurrence of field and/or subfield in input will create a new occurrence of subfield in output
R	R	R	*N*	NR			Each occurrence of field and/or subfield in input will create a new occurrence of field in output
R	R	R		R	*n*		Each occurrence of field and/or subfield in input will create a new occurrence of subfield in output
R	R	R	*N*	R			Each occurrence of field and/or subfield in input will create a new occurrence of field in output
R	R	R	*Nt*	R	*ns*		Each occurrence of field (numbered *nt*) will create a new occurrence of field. Each occurrence of subfield (numbered *ns*) will create a new occurrence of subfield within current field.

The table shows how the link between one particular CDIn to one other particular CDOut can be described, with n, ns and nt appearing in CDOut occurrences.

Three configurations can occur:

- One particular CDIn is linked to a particular CDOut (1-1). This case is treated in the table.
- A set of particular CDIn are linked to only one CDOut (n-1). This case has two different configurations:
    1. The number of CDIn to be merged is known; it is possible, in the rule, to access each CDIn to create the appropriate CDOut.
    2. The number of CDIn to be merged is unknown. In this case (see table) each

occurrence in the input can be merged in the output. The + at the beginning of the rule means that every time the rule is evaluated (for each occurrence of each CDIn), the resulting rule will be merged in the contents of CDOut.

- A particular CDIn must be used to create a set of CDOut (1-n). This case also has two different configurations:

  1. The number of CDOut to be created is known. It is possible to create as many lines in the rule file as will be created for CDOut, together with appropriate rules.
  2. The number of CDOut to be created is unknown. In this case you will be required to use the **Redo** command. The part of the rule before **Redo** is a normal rule used to generate CDOut. The part of the rule after **Redo** creates a pseudo-CDIn which will be used to re-evaluate the rule as many times as necessary (the *no* is incremented, like *nso* or *nto etc*) until the pseudo-CDIn created is empty. In this particular case the occurrence number of CDOut will not match any occurrence number of CDIn. In the rule file *nso, nto* or *no* must appear.

- An unknown number of different CDIn's are used to create a set of CDOut (1-n). In this case occurrence specifiers **new** and **newest** can be used to create and access a new occurrence at will.

Rule File Structure

Each line of a rule file must have the following structure:

[TTT$_{in}$] [(*n$_1$*)][SS$_{in]}$[(*n$_2$*)][/*n$_3$*[-*n$_4$*]/] | [<][TTT$_{out[}$(*n$_5$*)]]][SS$_{out}$[(*n$_6$*)]][/*n$_7$*[-*n$_8$*]/] | Rule

TTT$_{in}$ Represents the field tag of the CDIn which will start the evaluation of the rule:
000 for the record label.
001 to **ZZZ** for all other fields.
If this part of the CD is omitted the previous **TTT$_{in}$** is assumed.

(n$_1$) Represents the tag occurrence number. The values which can be used are:
n, ns, nt, no, nto or a number to specify absolute occurrence. If ***n$_1$*** and ***n$_2$*** are omitted, the default value of ***n$_1$*** is ***nt***, and the default value of ***n$_2$*** is ***ns***. If only ***n$_1$*** is omitted, its default value is 1.

Ss$_{in}$ Represents the subfield mark:
I1 for first indicator.
I2 for second indicator.
$a to **$z**, **$A** to **$Z**, and **$0** to **$9** for other subfields.
If this part of CD is omitted, the complete field is assumed.

(n$_2$) Represents the subfield occurrence number. The values which can be used are:
n, ns, nt, no, nso or a number to specify absolute occurrence. If ***n$_1$*** and ***n$_2$*** are omitted, the default value of ***n$_1$*** is ***nt***, and the default value of ***n$_2$*** is ***ns***. If only ***n$_1$*** is omitted, its default value is 1.

/n$_3$[-n$_4$]/ Represents the character positions in the (sub)fields in the CDIn.
n$_3$ means only character position ***n$_3$***

n_3- means character positions n_3 to the end of the (sub)field.
-n_4 means the character positions from beginning of the (sub)field to n_4
If both n_3 and n_4 are present, characters from position n_3 to n_4 are assumed.
In rule files, 1 is always used to describe the first character position, not 0.

< Means that the CDOut which is created by the rule will be added to the input record, not to the output record as it normally would.

TTT$_{out}$ Represents the field tag of the CDOut. It is created or updated during the evaluation of the rule.
000 for the record label.
001 to **ZZZ** for all other fields.
??? means that the CDIn is intentionally not converted, otherwise an error would be generated (this requires CDIn to not contain wildcards, otherwise ??? is treated as a wildcard).
This part of the CD is mandatory.

(n_5) Represents the tag occurrence number. The values which can be used are:
n, ns, nt, no, nto or a number to specify absolute occurrence. If n_5 and n_6 are omitted, the default value of n_5 is ***nt***, and the default value of n_6 is ***ns***. If only n_5 is omitted, its default value is 1.

SS$_{out}$ Represents the subfield mark.
I1 for first indicator.
I2 for second indicator.
$a to **$z**, **$A** to **$Z**, and **$0** to **$9** for other subfields.
If this part of CD is omitted, the complete field is assumed.

(n_6) Represents the subfield occurrence number. The values which can be used are:
n, ns, nt, no, nso or a number to specify absolute occurrence. If n_5 and n_6 are omitted, the default value of n_5 is ***nt***, and the default value of n_6 is ***ns***. If only n_5 is omitted, its default value is 1.

/**n_7**[-**n_8**]/ Represents the character positions in the (sub)fields in the CDOut.
n_7 means only character position n_7
n_7- means character position n_7 to the end of the (sub)field.
-n_8 means character positions from beginning of the (sub)field to n_8
If both n_7 and n_8 are present, characters from position n_7 to n_8 are assumed.
In rule files, 1 is always used to describe the first character position, not 0.

Rules

Basic Concept

To convert input records into output records, each CDIn must be linked to a CDOut with a rule. This rule must enable the software to make a lot of string operations. The basic concept is as follows.

When you use a pocket calculator the current result is on the screen and it can be modified using operator keys (+,-, etc.). The final result is calculated step by step. For example, if you want to calculate the result of 1/(7+(4+3)*5), you can do:

```
4
+
3
=          →     7
×
5          →     35
+
7
=          →     42
1/x        →     0.02380952380952
```

This can also be described as :

```
4+3;S×5+7;1/S
```

Here';' is used to indicate the end of a sequence of commands, and **S** represents the result of the last operation.

This basic concept is used in USEMARCON i.e. a set of basic operators is used to create the final result which will be the content of the CDOut (essentially string manipulation).

The rule syntax is:

[+][instruction$_1$; [instruction$_2$; [… instruction$_n$]…]

- The + at the beginning means the actual content of CDOut is merged with result of the following rule.
- The default instruction i.e. when no instruction is specified in the rule, is to copy CDIn to CDOut (or merge it if a + appears at the beginning of the rule).

Note: In all instructions, upper case and lower case characters can be used.

Variables

CD variables

There are three Content Designator variables :

- **S** represents the result of the last instruction; it is the Source of the current operation. It can be either a text string or a number, depending on the result of the last instruction.
- **D** represents the current contents of the Destination (output). This allows for adding or modifying the contents of a specific field when used with e.g. absolute indexes (occurrences).
- **[@][TTT[(n_1)]] [SS[(n_2)] [/[n_3[-]n_4]/])** which represents any valid CD. If TTT is omitted, the current CDIn Tag is assumed. If SS is omitted, the whole Tag (or fixed position of it) is assumed. SS can be either $*x* or I1 or I2.

Occurrence variables

There are six occurrence variables:

- ***n*** Represents the occurrence of CDIn in input record
- ***nt*** Represents the occurrence of the current tag of CDIn in the input record
- ***ns*** Represents the occurrence of the current subfield within the current tag
- ***no*** Represents the occurrence of CDOut in the output record
- ***nto*** Represents the occurrence of the current tag of CDOut in the output record
- ***nso*** Represents the occurrence of the current subfield of CDOut within the current tag

Other variables

- **TAG** represents the tag name of the field to be converted.
- **SUB** represents the subfield code of the field to be converted (without $ sign).
- **UTF8** indicates whether UTF-8 mode is active (e.g. *If (UTF8) Then '\c3\a4' Else 'ä'*)

Literal values

There are two kinds of literal values :

- **'…'** Represents a literal string (enclosed with a single quotation marks).

Example:

```
'This is a string'
```

Each character can be represented by its hexadecimal value preceded by a backslash (\). Some special characters are managed in strings:

- the '\' which must be coded as '\\' or its hexadecimal value,
- the subfield delimiter which can be coded '$' or its hexadecimal value (0x1F),
- the character '$' which must be coded as '\$' or its hexadecimal value.

Example:

```
≠NSB≠ is '\88'
```

- ***ii…*** Represents a literal numeric (where *i* is a digit).

Example: 123 is a numeric

Date and Time variables

There are six date and time variables:

- **Year** Contains the current year in the format YYYY
- **Month** Contains the current month (from 1 to 12)
- **Day** Contains the current day in the month (from 1 to 31)
- **Hour** Contains the current hour (from 0 to 23)
- **Minute** Contains the current minute within the hour (from 0 to 59)
- **Second** Contains the current second within the minute (from 0 to 59)

Instructions

There are six kinds of instruction: conditional, boolean, loop, memory, conversion and translation. Each instruction can begin with an upper case letter, the following letters must be in lower case.

Conditional instructions

These are the possible conditional instructions:

- **If (…) Then**

The syntax of this instruction is:

If **(***boolean instruction***)** **Then** *translation instruction*

If the result of a boolean instruction is TRUE then the instruction after **Then** is evaluated, as well as any subsequent instructions contained in the rule. If the result of a boolean instruction is FALSE then the CDOut is not created. **If** conditions where **Then** is omitted don't stop at FALSE but will evaluate any subsequent rules too and create CDOut.

Example (UNIMARC to UKMARC):

```
101$a    | 008/36-38/      | If (n=1) Then S
```

Character positions 36-38 in the 008 are only filled in if the occurrence number of 101$a is 1. In other cases, this rule does not do anything with 008.

`101$aeng`

becomes:

`008` with `eng` in character positions 36-38

Example (UNIMARC to UKMARC):

```
101$a    | 041$a     | + S
```

041$a is merged with 101$a each time a 101$a appears in the input record.

```
        101$aeng$afre$ager
```

becomes:

```
        041$aengfreger
```

Example (UNIMARC to UKMARC):

```
215$a    | 300(n)$a  | … ; From(First(Number)); If (S!='')
                             Then From(First(String)); If …
```

After evaluation of the From(First(Number)) instruction, the result (**S**) can be either empty or not empty. If **S** is not empty, then the evaluation of the rule continues, else 300(*n*)$a is not created. *Note* that != is used to represent 'not equal to'.

- **If (…) Then … Else**

The syntax of this instruction is :

If (*boolean instruction***) Then** *translation instruction* **Else** *translation instruction*

If the result of a boolean instruction is TRUE then the instruction after **Then** is evaluated. If the result of a boolean instruction is FALSE then the instruction after **Else** is evaluated.

In both cases any subsequent instructions of the rule will be evaluated.

Example (UNIMARC to UKMARC):

```
071I1 | 538(n)$a |  Table('538.tbl'); If (Exists($b)) Then S+':ƀ'
         +$b Else S
071$a | 538(n)$b |
```

The table (538.tbl) converts the value of indicator 1, e.g. 2, into an expression, e.g. 'Pl. no.'; if there is a $b in the 071 containing the name of the agency, e.g. 'Breitkopf & Härtel'; 538$a will contain 'Pl. no.: Breitkopf & Härtel' else it will simply contain 'Pl. no.'. The command line that follows puts the number itself into $b of the 538 field.

```
        071.20$aB. & H. 8801$bBreitkopf & Härtel
```

becomes:

```
        538.00$aPl. no. Breitkopf & Härtel$bB. & H. 8801
```

- **If (…) translation instruction**
- **If (…) translation instruction else translation instruction**

These work like If-Then and If-Then-Else with the important difference in that the CDOut is created regardless of whether the condition was true.

It is possible to nest If … [Then] … Else statements and blocks:
If (*boolean instruction***) {**
    If (*boolean instruction***) I***translation instruction* **Else** *translation instruction*
}
Else *translation instruction*

- **While (…) translation instruction**

While can be used to execute the instruction as long as the condition remains true. There's a hard-coded safety limit of 1000 executions, but care must be taken not to cause conversion to slow down due to unnecessary looping.

- **For (PLACEHOLDER From num$_1$ To num$_2$ [Condition (…)]) translation instruction**

For can be used to execute an instruction or a group of instructions grouped with curly brackets a certain number of times. PLACEHOLDER is a string whose all occurrences in the instruction are replaced with the current loop number counting from **num$_1$** to **num$_2$** during each execution of the instruction. If **num$_1$** is larger than **num$_2$** the counter is decremented with each loop. **Condition** is optional and specifies a Boolean condition that must be true for the loop to continue.

Example:

```
      For (X1 From 1 To 3) While (PreviousSubIn(S, $n, '=X1') = 'c') {
ReplaceOcc('$n' By '. ', '=X1', Strict); ReplaceOcc('$p' By ', ', '=X1',
Strict) };
```

Curly brackets { } group together multiple instructions to be carried out as a result of a conditional statement. A semicolon is required after a closing curly bracket if there are further instructions.

Example:

```
      If (…){ Replace('$c' By ' = '); Replace('$d' By ' ; '); };
```

Boolean instructions

There are two kinds of boolean instructions: simple boolean instructions and boolean functions. Boolean instructions can be grouped with parentheses, and combined with **And, Or** and **Not** operators.

- *boolean$_1$* **And** *boolean$_2$*: is TRUE if each boolean expression is TRUE, otherwise it is FALSE
- *boolean$_1$* **Or** *boolean$_2$*: is TRUE if one of both boolean expressions are TRUE, otherwise it is FALSE
- **Not** *boolean*: is TRUE if boolean expression is FALSE. It is FALSE if boolean expression is TRUE.

Note: Not affects the complete following expression. If this is not desired, **Not** and the expression must be enclosed in parenthesis (e.g. If ((Not $c Precedes $d) And …))**.**

- **(***boolean***)**: is a valid boolean expression

- **Simple boolean instructions**

A simple boolean instruction can be:

- *t_instruction$_1$* = *t_instruction$_2$*: The result is TRUE if the result of both instructions is the same string, otherwise it is FALSE. If one instruction is a number and the other a string, the number is converted into a string before comparing them.

Example :

`12 = 12, 12 = '12'` are True but

`13 = 12, 13 = 'ƀ13'` are False, because of the blank before 13.

- *t_instruction*$_1$ **!=** *t_instruction*$_2$: The result is TRUE if the instructions are not equal, otherwise it is FALSE. If one instruction is a number and the other is a string, the number is converted into a string before comparing them.
- *t_instruction*$_1$ **IN** *t_instruction*$_2$: The result is TRUE if the result of *t_instruction*$_1$ is a substring of result of *t_instruction*$_2$, otherwise it is FALSE. If the instruction is a number it is converted into a string.

Example :

‘every’ In ‘hello everybody’ is TRUE

‘19’ In ‘1994’ is TRUE

- *t_instruction*$_1$ > *t_instruction*$_2$: The result is TRUE if *t_instruction*$_1$ is greater than *t_instruction*$_2$, otherwise it is FALSE. If both instructions are numbers, the comparison is made between the numbers. If one is a string, the comparison is made between the strings. One string is classified as greater than another if it is sorted alphabetically after another.
- *t_instruction*$_1$ < *t_instruction*$_2$: The result is TRUE if *t_instruction*$_1$ is less than *t_instruction*$_2$, otherwise it is FALSE. If both instructions are numbers, the comparison is made between numbers. If one is a string, the comparison is made between strings. A string is less than another, if the other is sorted alphabetically before it.
- *t_instruction*$_1$ ≥ *t_instruction*$_2$: The result is TRUE if *t_instruction*$_1$ is greater than or equal to *t_instruction*$_2$, otherwise it is FALSE. If both instructions are numbers, the comparison is made between numbers. If one is a string, the comparison is made between strings. A string is greater than another if it is sorted alphabetically after it. A string is equal to another if it is sorted alphabetically in the same order.
- *t_instruction*$_1$ ≤ *t_instruction*$_2$: The result is TRUE if *t_instruction*$_1$ is less than or equal to *t_instruction*$_2$, otherwise it will be FALSE. If both instructions are numbers, the comparison is made between the numbers. If one is a string, the comparison is made between strings. A string is less than or equal to another (respectively), if it is sorted alphabetically before it or if it is in the same order.
- *subfield*$_1$ **Precedes** *subfield*$_2$: The result is TRUE if *subfield*$_1$ appears before *subfield*$_2$, otherwise it is FALSE. *Subfield* is a CD in which SS is mandatory. If the Tag of each subfield is not the same, the result will be FALSE.

Example:

```
700$a Precedes 700$b,
$a Precedes $c,
```

700$a Precedes $c - **WARNING**: Here $c means $c of the current CDIn. If the current CDIn is in Tag 700, the comparison will work, if it is not, the result will be FALSE.

- *subfield*$_1$ **Follows** *subfield*$_2$: The result is TRUE if *subfield*$_1$ appears after *subfield*$_2$, otherwise it is FALSE. If the Tag of each subfield is not the same, the result is FALSE.

- **Boolean functions**

There are three boolean functions in the software at present.

- **Exists(** *CD* **)**: This is TRUE if *CD* exists in the input record (@CD) or output record(%CD), or FALSE if it does not exist. It can be used in combination with ‘Not’.

Example:

```
710$e | 712$c | If (($e Precedes $c) Or (Not Exists($c)) Then S
```

- **ExistsIn(***'string', CD* **)**: This is TRUE if *CD* exists in the given string such as S. It can be used in combination with ‘Not’.

Example:

```
710 | 712 | If (($e Precedes $c) Or (Not Exists(S, $c)) Then S
```

- **InTable(***'string'***, '***file_name'* **)**: Returns TRUE if *string* is found in table file *file_name*. Can be used for example to check if S is one of the values listed in the table. InTable will ignore any #DEFAULT value in the table.

Example:

```
008/7/ | 008/7/ | If (InTable(S, 'valid_codes.tbl')) Then S Else '|'
```

Loop instruction

There is only one loop instruction.

- **Redo**: This resolves the problem of (1 to n) conversion. It creates (or modifies) the current CDOut with the result of the preceding instructions. It increments **no** and **nso** or **nto**. Then the following instructions create a pseudo-CDIn. If that is not empty, the rule will be re-evaluated to create (or modify) CDOut.

Example :

```
041$a    |  101$a(no)      |  Sto(0); To(3); Redo; Mem(0); From(4)
```

Suppose an input record contains the following 041 fields:

```
041.00 $afreengspa
041.00 $ager
```

When the software reaches the first 041, it creates a 101$a (**no** is 1). In 101$a, it copies the first 3 characters of 041$a (**To(3)**). When it reaches **Redo**, **no** is incremented, and the output record contains:

```
101. $afre
```

After **Redo**, the software gets 041$a back (it has been stored with **Sto(0)**, and restored with **Mem(0)**), skips the first 3 characters in it, and, returning to the beginning of the command line, stores the result by **Sto(0)**. The pseudo-CDIn contains ‘engspa’: it is not empty. The rule is then re-evaluated with ‘engspa’ as the content of CDIn.

The software then creates a new occurrence of $a (**no** is 2), with the first three characters of of the pseudo-CDIn:

```
    101. $afre$aeng
```

After Redo, the process is repeated with just ‘spa’ in the pseudo-CDIn (**no** is 3).

```
    101. $afre$aeng$aspa
```

By the end of the command line the software skips 3 characters and the pseudo-CDIn is empty. So, the next loop does nothing and the evaluation of the rule is finished.

When the software reaches the second 041$a, it works the same way and creates a third occurrence of $a in 101.

```
    101. $afre$aeng$aspa$ager
```

Memory instructions

It is necessary to be able to store intermediate results in complex rules.

In the USEMARCON software, 100 memories are available. Four functions enable the manipulation of those memories :

- **Sto(*m*)**: *m* is in 0 to 99. This function stores the current value of **S** in the *m*th memory. The old content of this memory is lost. Content of **S** is not changed.
- **Mem(*m*)**: This function returns the content of *m*th memory. If this memory has not been initialized, the function returns an empty string.
- **Exc(*m*)**: This function stores the current value of **S** in the *m*th memory, and returns the old content of this memory.
- **Clr(*m*)**: This function clears the content of *m*th memory. Contents of **S** are not changed.

The contents of the memory are not cleared automatically, therefore it is possible to exchange information between rules and even between records.

Conversion instructions

There are three types of instructions in USEMARCON: string, number and boolean. A string can be converted to a number and a number to a string. Sometimes the conversion is made automatically, but sometimes it can be useful to have a conversion facility.

- **Str(*t_instruction*)**: The result of this instruction is a string. If *t_instruction* is a number, it is converted to a string. If *t_instruction* is a string, it stays a string.
- **Val(*t_instruction*)**: The result is a number. If *t_instruction* is a string containing only a number (like '123'), it is converted to a number. If the string contains anything else, it is converted to number 0. If *t_instruction* is a number it stays a number.
- **Ordinal(*t_instruction)***: The result is the ordinal number of the currently converted record. Its value is generally 1 for the first converted record, 2 for the second one, and so on. The first number is set in the Initialization parameters of the .INI file. **NB! The OrdinalNumber setting is only updated in the .INI file if UpdateOrdinalNumber setting in the same section is TRUE**. The update is unsafe in most environments where multiple conversions can be running simultaneously. If 20 records have been converted earlier, the first number can be set to 21 so that the current session correctly numbers the next batch of records. The command:

```
001 | 001 | 'UNI' + Ordinal(7)
```

then creates a record with record control number 'UNI0000021' i.e. 21 right-justified with zeroes, to 7 characters.

Translation instructions

Translation instructions can be grouped with parentheses.

Simple translation instructions

A simple translation instruction can be:

- *t_instruction$_1$* **+** *t_instruction$_2$*: If both t_instructions are numbers, the result is the sum of those numbers. If at least one is a string, the result is a string containing *t_instruction$_1$* merged with *t_instruction$_2$*.
- *t_instruction$_1$* **-** *t_instruction$_2$*: Both t_instructions must be numeric (if at least one is a string, an error appears), the result is the difference between *t_instruction$_1$* and *t_instruction$_2$*.
- *t_instruction$_1$* ***** *t_instruction$_2$*: Both t_instructions must be numeric (if at least one is a string, an error appears), the result is the product of *t_instruction$_1$* and *t_instruction$_2$*.
- *t_instruction$_1$* **/** *t_instruction$_2$:* Both t_instructions must be numeric (if at least one is a string, an error appears), the result is the integer division of *t_instruction$_1$* and *t_instruction$_2$*.

String functions

String functions are functions that manipulate or return a string.

- **From(***t_instruction*[,**Strict**]**)**: If *t_instruction* is a number *p*, the result of the function is the substring of **S** from *p*th position (first position is 1). If *t_instruction* is not a number, or is a number less than or equal to 0, or greater than **S** length, this function returns an empty string. If **Strict** is not specified and blank characters appear at the beginning of the result, they are skipped. If **Strict** is specified, blank characters at the beginning of the result are kept.
- **To(***t_instruction*[,**Strict**]**)**: If *t_instruction* is a number *p*, the result of the function is the substring of **S** truncated in *p*th position (first position is 1). If *t_instruction* is not a number, or is a number less than or equal to 0, or greater than **S** length, this function returns an empty string. If **Strict** is not specified and blank characters appear at the end of the result, they are truncated. If **Strict** is specified, blank characters at the end of the result are kept.
- **Len**(*t_instruction*): Returns the length of the given string.
- **Between(***t_instruction$_1$* , *t_instruction$_2$* [,**Strict**]**)**: Both *t_instructions* must be a number. If at least one is not, the result is an empty string. If both are a number p_1 and p_2 the result is a substring of **S** from p_1th position to p_2th position. If **Strict** is not specified any blank characters appearing at the boundaries of the result are deleted. If **Strict** is specified, any blank characters are kept.
- **Next(***subfield$_1$* [, *subfield$_2$*][,**Strict**]**)**: *subfield* is a CD in which SS is mandatory. Each subfield is assumed to be in CDIn. If only *subfield$_1$* is specified, this function will return the first subfield following current CDIn, identified by *subfield$_1$*, until the end of the field. If both *subfield$_1$* and *subfield$_2$* are specified, the function will return the first subfield following current CDIn, identified by *subfield$_1$*, until the subfield identified by *subfield$_2$*. If **Strict** is not

specified any blank characters appearing at the boundaries of the result are deleted. If **Strict** is specified, any blank characters are kept. If the subfield is not found, the function returns an empty string.

- **Last(***subfield$_1$* [, *subfield$_2$*][,**Strict**]**)**: Each subfield is assumed to be in CDIn. If only *subfield$_1$* is specified, this function will return the first subfield preceding current CDIn, identified by *subfield$_1$*, until the beginning of the field. If both *subfield$_1$* and *subfield$_2$* are specified, the function will return the first subfield preceding current CDIn, identified by *subfield$_1$*, until the subfield identified by *subfield$_2$* . If **Strict** is not specified any blank characters appearing at boundaries of the result are deleted. If **Strict** is specified, any blank characters are kept. If the subfield is not found, the function returns an empty string.
- **NextSub**: Returns the next subfield code (without the dollar sign) relative to the current subfield or '' if current subfield is the last one.
- **PreviousSub**: Returns the previous subfield code (without the dollar sign) relative to the current subfield or '' if current subfield is the first one.
- **NextSub(***subfield* [, *'occurrence'*]**)**: Returns the next subfield code (without the dollar sign) relative to the specified subfield or '' if none is found. 'occurrence' can be specified and is used to compare the occurrence of the specified subfield.

Example:

```
// Add comma if next subfield of second or later $n is $p
245$n | 245$n | S; If (NextSub($n, '>1') = 'p') Then S + ' ,'
```

- **NextSubIn(**'string', subfield [, *'occurrence'*]**)**: Returns the next subfield code (without the dollar sign) relative to the specified subfield in the given string or '' if none is found. 'occurrence' can be specified and is used to compare the occurrence of the specified subfield.

Example:

```
// Move all $n subfields after $a if next subfield after $a
// is not $p
If (NextSubIn(S, $a) != 'p') MoveAfter('n', $a);
```

- **PreviousSub(***subfield* [, *'occurrence'*]**)**: Returns the next subfield code (without the dollar sign) relative to the specified subfield or '' if none is found. 'occurrence' can be specified and is used to compare the occurrence of the specified subfield.

Example:

```
// Add [text] if previous subfield is $h and this subfield is not the first one
245$n | 245$n | If (PreviousSub($n(ns), '>1') = 'h') Then S + ' [text]'
```

- **PreviousSubIn(***'string'*, *subfield* [, *'occurrence'*]**)**: Returns the next subfield code (without the dollar sign) relative to the specified subfield in the given string or '' if none is found. 'occurrence' can be specified and is used to compare the occurrence of the specified subfield.

Example:

```
// Convert $n to ' : ' if if comes after $a in S
If (PreviousSubIn(S, $n) = 'a') Then Replace('$n' By ' : ', Strict);
```

- **MoveBefore(***'source subfields', subfield, 'target subfields'*, [*'prefix', 'suffix', 'preserved* *prefixes'*]**)**: Moves 'source subfields' in the current field before *subfield* changing their codes to 'target subfields' at the same time. Additionally a prefix can be added to any preceding subfield and suffix in the end of the given subfields. Can only be used when the source or target is a complete field.

- **MoveAfter(***'source subfields', subfield, 'target subfields',* [*'prefix', 'suffix', 'preserved prefixes'*]**)**: like MoveBefore, but moves the source subfields after *subfield.*

source subfields	A list (string) of subfield codes to move (all subfields with the listed codes are moved). The order of subfield codes is insignificant – the subfields are moved in the order they are in the field.
Subfield	The CD describing the subfield before which the source subfields are moved.
target subfields	A list (string) of subfields codes that the existing fields are given after they are moved. If empty, the subfields keep their existing codes.
Prefix	Optional prefix that will be added to the end of the subfield right before the first of the moved subfields.
suffix	Optional suffix that will be added to the end of the last moved subfield.
preserved prefixes	A pipe-separated list of prefixes that are preserved at their correct position. As a subfield may have a prefix that is stored at the end of the previous subfield, moving other subfields between these two subfields causes problems with the positioning of the prefix. If such prefix is found before the position where the subfields are to be moved, its position will be updated so that it is still adjacent to the original subfield. e.g. There is field: $a Title / $c More : $b Add. If subfield $b was moved after $a without taking care of prefixes, the result would be: $a Title / $b Add.$c More : With preserved prefix list of ' / | : |. ' the result is: $a Title : $b Add / $c More.

Examples:

```
// Move subfields c, d and e right before b and change c to f:
   MoveBefore('cde', $b, 'fde', '', '', '/|:|.');


// Move subfields f and g right after a. Add suffix ' – ' to a:
   MoveAfter('fg', $a, '', ' – ');
```

Note: When adding a single subfield to a complete field and using MoveBefore or MoveAfter it will be necessary to use e.g. **D+S** as the first rule instead of just **+S**. Otherwise the full contents of the field will not be available for processing and MoveBefore or MoveAfter will fail.

- **MoveFirst(***'source subfields', 'target subfields',* [*'prefix', 'suffix', 'preserved prefixes'*]**)**: Moves given subfields to the beginning of the field. See MoveBefore and MoveAfter for more information.

- **MoveLast(***'source subfields', 'target subfields',* [*'prefix', 'suffix', 'preserved prefixes'*]**)**: Moves given subfields to the end of the field. See MoveBefore and MoveAfter for more information.

- **Delete(***t_instruction* [, **At Beginning|At Ending|At Boundaries**] [,**Strict**]**)**: This function deletes any occurrence of the string *t_instruction* from **S**. If *t_instruction* is a number, it is converted into a string. If **At Beginning** is specified, *t_instruction* is deleted in **S**, only if it is at the beginning of **S**. If **At Ending** is specified, it is only deleted if it appears at the end of **S**. If **At Boundaries** is specified, it is only deleted if it appears at the boundaries of **S**. If **Strict** is not specified any blank characters appearing after/before the deleted characters of the result are deleted. If **Strict** is specified, any blank characters are kept.

- **Replace**(*'x' by 'y'* [,**Strict**]): Replace may be used to replace a subfield code with another subfield code, a subfield code with a mark of punctuation, a mark of punctuation with a subfield code or one string of text with another. If Strict is not specified, any blank characters at the beginning or end of the string are removed.

- **ReplaceOcc(**(*'x' by 'y', '>1'* [,**Strict**]): ReplaceOcc is used to replace multiple occurrences of the same subfield with different things in output. This is typically used when converting repeated fields to a single field with punctuation. This usually means that occurrences of the subfield after the first must be replaced by marks of punctuation. If Strict is not specified, any blank characters at the beginning or end of the string are removed.

- **Table('***file_name'* **)**: *file_name* is the name of a file (in the rule file directory) enclosed within single quotation marks i.e.'filename.tbl', containing the description of a table of coded data. The name prefix must not exceed 8 characters in length (because of the DOS environment) and must have the .tbl extension. The function should return **S** converted as described in the table. If the conversion is not possible the function returns an empty string and an error is appended to the log file.

Example:

```
   008/16-17 | 102$a | Table('Uk3166b.tbl') // Country codes table
```

- **Sort(**'*string'***)**: *string* is a string enclosed within single quotation marks where each character is a subfield identifier. This function can only appear in a rule linked with a complete CDOut. It enables the subfields of CDOut to be sorted when the record is finished. **N.B. Sort must be specified alone without any other rules.**

Example:

```
   010      | 022(n)    | S
```

```
010      | 022(n)    | Sort('abcdz')
```

- **SortField()**: This function enables the field to be put into its place (sorted alphanumerically in relation to other fields) when the record is finished. Only useful when SortRecord option is false. **N.B. SortField must be specified alone without any other rules.**

Example:

```
010      | 022(n)    | S
010      | 022(n)    | SortField()
```

- **RegFindPos('***regular expression***')**: Searches **S** for the specified '*regular expression*'. Returns the start position of the match, if the expression was found, 0 if no match and -1 if an error occurred.
- **RegFindPos(***'string to search', 'regular expression'***)**: Searches '*string to search*' for the specified '*regular expression*'. Returns the start position of the match, if the expression was found, 0 if no match and -1 if an error occurred.
- **RegFindNum('***regular expression***')**: Searches **S** for the specified '*regular expression*'. Returns the number of matches found, 0 if no match and -1 if an error occurred.
- **RegFindNum(***'string to search', 'regular expression'***)**: Searches '*string to search*' for the specified '*regular expression*'. Returns the number of matches found, 0 if no match and -1 if an error occurred.
- **RegFind('***regular expression***')**: Searches **S** for the specified '*regular expression*'. Returns >= 0 if the expression was found.
- **RegFind(***'string to search', 'regular expression'***)**: Searches '*string to search*' for the specified '*regular expression*'. Returns >= 0 if the expression was found.
- **RegMatch(***index***)**: Returns the specified match from a previous **RegFind**. The *index* is a number from 0 to 9.
- **RegReplace(***'find expression', 'replace expression' [, 'parameters']***)**: replaces '*find expression*' with '*replace expression*'. \\1 \\2 etc. can be used to reference the matched parts. 'parameters' can contain 'g' to indicate a global change (replace all).

Note: All regular expressions use the PCRE (Perl-compatible regular expressions) library. Information about the features and usage can be found at http://www.pcre.org/pcre.txt and http://en.wikipedia.org/wiki/PCRE.

Examples:

Replace "Most recent [anything] not available" with "Not available: most recent [anything]":

RegReplace('Most recent (.*) not available', 'Not available: most recent \\1');

Empty anything that has only numbers in it (^ denotes beginning of string and $ end of string. \\d is any number of characters 0-9):*

RegReplace('^\\d*$', '');

One way to replace "Available from [anything]" with "Begins at [anything]":

If ((RegFindNum('^Available from (.*)') > 0)) Then 'Begins at ' + RegMatch(1);

Swap around two words:

RegReplace('(.*) (.*)', '\\2 \\1');

Knowing the syntax of regular expressions is required to be able to get the most out of them. All expressions are case sensitive.

- **RegReplaceTable(***'table name', [, 'parameters']***)**: Works like RegReplace but reads ‘find expression’ and ‘replace expression’ from a table file. The file follows the formatting of the rule file unlike other table files. Possible parameters:

  g = global, change all occurrences (as usual with regular expressions)
  f = first only, processes the lines until the pattern matched

Example table file contents:

```
Regularly required replacements

'auth'       | 'author'
'titl'       | 'title'
'(\\d\\d\\d\\d)' | '[\\1]’ // Adds square brackets around
                       // 4 digits (1999 -> [1999])
'(\\d\\d)-(\\d\\d)-(\\d\\d\\d\\d) | '\\3-\\2-\\1'
```

- **Upper**(*'string'*): Converts the given string to uppercase
- **Lower**(*'string'*): Converts the given string to lowercase
- **Mixed((***'string'*): Converts the given string to mixed case (first capital, rest

N.B. If you need literal $ in the rule, it needs to be escaped by prepending it with a backslash: **\$** (otherwise it is translated to a subfield delimiter). Backslash (\) must be escaped too as shown in the examples: \\.

Number functions

Number functions are functions returning a number.

- **BFirst(***t_instruction* **|Number|String)**: If *t_instruction* is specified, this function returns the position of the beginning of the first occurrence of *t_instruction* in **S**. If *t_instruction* is a number it is converted into a string. If **Number** is specified, the function returns the position of the beginning of the first number in **S**. If **String** is specified, the function returns the position of the beginning of the first string in **S**. Returns 0 if no match is found.
- **EFirst(***t_instruction* **|Number|String)**: If *t_instruction* is specified, this function returns the position of the end of the first occurrence of *t_instruction* in **S**. If *t_instruction* is a number it

is converted into a string. If **Number** is specified, the function returns the position of the end of the first number in **S**. If **String** is specified, the function returns the position of the end of the first string in **S**. Returns 0 if no match is found.

- **BLast(***t_instruction* **|Number|String)**: If *t_instruction* is specified, this function returns the position of the beginning of the last occurrence of *t_instruction* in **S**. If *t_instruction* is a number it is converted into a string. If **Number** is specified, the function returns the position of the beginning of the last number in **S**. If **String** is specified, the function returns the position of the beginning of the last string in **S**. Returns 0 if no match is found.
- **ELast(***t_instruction* **|Number|String)**: If *t_instruction* is specified, this function returns the position of the end of the last occurrence of *t_instruction* in **S**. If *t_instruction* is a number it is converted into a string. If **Number** is specified, the function returns the position of the end of the last number in **S**. If **String** is specified, the function returns the position of the end of the last string in **S**. Returns 0 if no match is found.

Comments and Multi-line Rules

Comments can appear on a separate line, or at the end of a line, and are preceded by a double slash //. It is recommended that general comments are placed before the rule they refer to.

Rules can be written on multiple lines: the first line contains CDIn, then space, pipe (|), space, then CDOut, then space, pipe (|), space, then the beginning of the rule. When rules are on multiple lines the last character of all but the last line must be a semicolon.

Examples:

```
// Bibliography & index note
504   | 320     I1      |
504   | 320     I2      |
504$a | 320     $a      |
// Contents note
505   | 327     I1      |
505   | 327     I2      |
505$a | 327     $a      |
// Accompanying material note
525   | 307     I1      |

008/16-17 | 102$a |Table('Uk3166b.tbl') // Country codes table

041$a | 101$a(no) | Sto(0); to(3); Redo;
                  Mem(0): From(4)
```

Example Rules

Examples are often the most powerful documentation. Here are some examples of techniques to achieve different goals.

A complete conversion that removes field 509, moves 508 to 501 and leaves everything else as is:

```
Simple conversion to remove field 509 and move field 508 to 501.
Just an example.

508        | 501         | S
509        | <DEL        | S
```

```
#if (TAG != 'DEL')
???I1      | ???I1        | S
???I2      | ???I2        | S
???        | ???          | S
#endif
```

For the above to work properly, ConvertInFieldOrder must be TRUE in the ini file.

Convert field 005 as is, but create it if it doesn’t exist:

```
005             | 005             | S
000             | 005             | If (Not Exists(@005)) Then
Year+Month+Day+Hour+Minute+Second+'.0'
```

Convert only subfields a, b and c in field 260 and remove anything in parenthesis from subfield c:

```
260I1  | 260I1  | S
260I2  | 260I2  | S
260$a  | 260$a  | S
260$b  | 260$b  | S
260$c  | 260$c  | S; RegReplace('.*\\((.*)\\).*', '\\1')
```

Scenario: Field 560$x might contain any identifier. It needs to be moved to the proper subfield of field 787. Define a macro for checking for an ISBN to avoid repeating long code (the #define is a single line) and use along with other rules to accomplish this:

```
#define macro ISBNRegExp  | (RegFind('^[\\dxX]{10}( \\(.*\\))?\$') >= 0 Or
RegFind('^[\\dxX\\-]{13}( \\(.*\\))?\$') >= 0 Or RegFind('^\\d{13}(
\\(.*\\))?\$') >= 0 Or RegFind('^[\\d\\-]{17}( \\(.*\\))?\$') >= 0)

// ISSN
560$x   | 787$x          | If (RegFind('\\d{4}-\\d{4}') >= 0) Then S
// ISBN
560$x   | 787$z          | If (ISBNRegExp) Then S
// Other identifiers
560$x   | 787$o          | If (Not (RegFind('\\d{4}-\\d{4}') >= 0 Or
ISBNRegExp)) Then S
```

6. ERROR MESSAGES

Format

System messages describing errors detected during format checking or data conversion are stored by default in an ASCII text file in the relevant sub-directory for the conversion; this is known as the Error Log File. In common with the other system files, the path and name of the Error Log File may be changed in the Initialization Parameters of the .INI File using a text editor such as MS Windows Notepad. Beginning from version 1.3 the file name can be left empty so that an error log is not created.

For each error, one line is logged in the report file; this enables easier filtering of the report file.

Each line begins with either WARNING, ERROR or FATAL.

- WARNING will not stop the program
- ERROR gives you the option to stop in Interactive mode but will not stop in Batch mode
- FATAL errors will not allow the software to proceed further.

The rest of the line consists of:

‘(‘, <coded error number>, ‘)’, <the error description>, <a description of the problem>

```
Line ::= <kind of error> (<coded number>) - <error message> : <parameters>
<kind of error> ::= ‘FATAL’   -     'ERROR'     -     'WARNING'
<coded number> ::= number
<error message> ::= words
<parameters> ::= words
```

Error Definitions

Errors are numbered 000-9999. The first digit describes the part of the process in which the error appears:

1xxx The error appears during reading of a MARC file, i.e. a format problem.
2xxx The error appears during MARC checking in input (in comparison with the input checking table).
3xxx The error appears during character translation.
5xxx The error appears during rule analysis or conversion.
7xxx The error appears during checking of the MARC output.
9xxx Other internal errors.

Examples:

```
WARNING(2001) - Invalid input format checking rule ( item expected ) …
ERROR(3001) - Character not transcoded (unable to find it in transco
table) : 0x90 in …
FATAL(7011) -....
```

6. USING USEMARCON IN ANOTHER APPLICATION

Usemarcon object

The Usemarcon object is used to control operations of the program. It is declared in usemarconlib.h. Include this file in another project and link with the appropriate library such as usemarcon_library.lib or libusemarcon.a.

For the best performance the object should be reused without calling SetInteractive, SetIniFileName, AddConfigOverride or ClearConfigOverrides. Recreating the object or changing these settings will cause USEMARCON to parse the configuration file, rule file, any tables etc. which will slow down the first call to Convert.

See program/usemarcon.cpp for an example (the part after `#else // api based run example)`.

Usemarcon.SetInteractive(value);

Setting value to true will cause USEMARCON to output messages to console. Should normally be set to false when embedding in another program.

Usemarcon.SetIniFileName(filename);

Defines the location of the .ini file where parameters are defined. The path can be an absolute path or relative to the current working directory.

Usemarcon.SetMarcRecord(pcRecord, iLength);

Used only in the on the fly conversion - pcRecord is a string containing a MARC record to be converted in ISO 2709 format. iLength contains the length of the string.

Usemarcon.SetForceVerbose(value);

Set value to true and the program will print out warning and error messages to the screen in addition to the error log when running the command line version.

Usemarcon.SetDisableCharacterConversion(value);

Set value to true and the program will not do character set conversion. Default is false.

Usemarcon.AddConfigOverride(section, setting, value);

Add a configuration override that overrides the corresponding ini file setting.

Usemarcon.ClearConfigOverrides();

Remove all configuration overrides previously added with *AddConfigOverride()*.

Usemarcon.Convert();

Instructs the conversion to convert the input record.

Usemarcon.GetMarcRecord(pcRecord, iLength);

Used only in the on the fly conversion - retrieves the converted record to pcRecord and its length in iLength. The output record is in ISO 2709 format. NB: the string is allocated with strdup and must be freed separately.

Usemarcon.GetWarningCount(msg);

Used to retrieve the number of warning reported during Convert().

Usemarcon.GetErrorCount(msg);

Used to retrieve the number of errors reported during Convert().

Usemarcon.GetLastWarningMessage(msg);

Used to retrieve the last warning message after Convert(). This can be used to display warning messages in the controlling program.

Usemarcon.GetLastErrorMessage(msg);

Used to retrieve the last error message after Convert(). This can be used to display error messages in the controlling program.

Upgrading to version 3 from older versions

The library interface has been changed in version 3. While it’s functionally equivalent to the old one, the naming and internal structure has been changed. The following changes are required to compile against version 3 compared to older versions:

Include usemarconlib.h instead of objectlist.h
Use Usemarcon class instead of CDetails
Call Convert() instead of Start()