Kofax Communications Manager

Template Scripting Language Developer's Guide

Version: 5.3.0

Date: 2019-06-05

KOFAX

Table of Contents

Preface

This guide provides a detailed information on the Template scripting language structure, its keywords, formulas, operators, and functions.

Related documentation

The documentation set for Kofax Communications Manager is available here:[1]

https://docshield.kofax.com/Portal/Products/CCM/530-1h4cs6680a/CCM.htm

In addition to this guide, the documentation set includes the following items:

Kofax Communications Manager Release Notes
Contains late-breaking details and other information that is not available in your other Kofax Communications Manager documentation.

Kofax Communications Manager Getting Started Guide
Describes how to use Contract Manager to manage instances of Kofax Communications Manager.

Help for Kofax Communications Manager Designer
Contains general information and instructions on using Kofax Communications Manager Designer, which is an authoring tool and content management system for Kofax Communications Manager.

Kofax Communications Manager Repository Administrator's Guide
Describes administrative and management tasks in Kofax Communications Manager Repository and Kofax Communications Manager Designer for Windows.

Kofax Communications Manager Repository User's Guide
Includes user instructions for Kofax Communications Manager Repository and Kofax Communications Manager Designer for Windows.

Kofax Communications Manager Repository Developer's Guide
Describes various features and APIs to integrate with Kofax Communications Manager Repository and Kofax Communications Manager Designer for Windows.

Kofax Communications Manager Core Developer's Guide
Provides a general overview and integration information for Kofax Communications Manager Core.

[1] You must be connected to the Internet to access the full documentation set online. For access without an Internet connection, see "Offline documentation" in the Installation Guide.

Kofax Communications Manager Core Scripting Language Developer's Guide
Describes the KCM Core Script.

Kofax Communications Manager API Guide
Describes Contract Manager, which is the main entry point to Kofax Communications Manager.

Kofax Communications Manager ComposerUI for HTML5 JavaScript API Web Developer's Guide
Describes integration of ComposerUI for HTML5 into an application, using its JavaScript API.

Kofax Communications Manager Batch & Output Management Getting Started Guide
Describes how to start working with Batch & Output Management.

Kofax Communications Manager Batch & Output Management Developer's Guide
Describes the Batch & Output Management scripting language used in KCM Studio related scripts.

Kofax Communications Manager DID Developer's Guide
Provides information on the Database Interface Definitions (referred to as DIDs), which is an alternative method retrieve data from a database and send it to Kofax Communications Manager.

Kofax Communications Manager ComposerUI for ASP.NET and J2EE Customization Guide
Describes the customization options for KCM ComposerUI for ASP.NET and J2EE.

Kofax Communications Manager ComposerUI for ASP.NET Developer's Guide
Describes the structure and configuration of KCM ComposerUI for ASP.NET.

Kofax Communications Manager ComposerUI for J2EE Developer's Guide
Describes JSP pages and lists custom tugs defined by KCM ComposerUI for J2EE.

Getting help with Kofax products

The Kofax Knowledge Base repository contains articles that are updated on a regular basis to keep you informed about Kofax products. We encourage you to use the Knowledge Base to obtain answers to your product questions.

To access the Kofax Knowledge Base, go to the Kofax website and select **Support** on the home page.

Note The Kofax Knowledge Base is optimized for use with Google Chrome, Mozilla Firefox or Microsoft Edge.

The Kofax Knowledge Base provides:

- Powerful search capabilities to help you quickly locate the information you need.
  Type your search terms or phrase into the **Search** box, and then click the search icon.

- Product information, configuration details and documentation, including release news.

  Scroll through the Kofax Knowledge Base home page to locate a product family. Then click a product family name to view a list of related articles. Please note that some product families require a valid Kofax Portal login to view related articles.
- Access to the Kofax Customer Portal (for eligible customers).

  Click the **Customer Support** link at the top of the page, and then click **Log in to the Customer Portal**.
- Access to the Kofax Partner Portal (for eligible partners).

  Click the **Partner Support** link at the top of the page, and then click **Log in to the Partner Portal**.
- Access to Kofax support commitments, lifecycle policies, electronic fulfillment details, and self-service tools.

  Scroll to the **General Support** section, click **Support Details**, and then select the appropriate tab.

Chapter 1

Basic structure of the Template scripting language

Template scripting language is divided into the following parts:

1. Instructions to create text
2. Instructions to retrieve data
3. Instructions to manipulate data

To distinguish between text that has to appear in the result document and Template scripts, the hash symbol is used. Each time the hash symbol is encountered, KCM switches from instruction mode to text mode, or vice versa. A Master Template is started in the text mode, and you should place the hash symbol before the first Template script.

The following is an example of a simple Master Template.

```
#
BEGIN
#
This is a very simple Master Template document.
#
END
```

`BEGIN` and `END` indicate the beginning and the end of the Template script sequence.

To clarify coding in a Master Template, you can add comments. Comments begin with `(*` and end with `*)`.

Integrate text and data

If data has to be merged into the text, use the @ construct:

```
...
#
Dear Ms. or Mr. @(Cust.Surname),
In reference to you writing...
#
...
```

In the example, KCM merges the Field `Surname` from the record `Cust` into the document.

Data Backbone

The Data Backbone describes the structure of the data and the data fields that can be used in template scripts. Each KCM project has a single data backbone definition that is available to all templates in that project. For more information on defining a Data Backbone, see "Work with Data" in the *Repository User's Guide*.

In the template scripting language the Data Backbone is represented by the _data variable. This variable shows a regular Data Structure and can be read and manipulated as any regular Data.

```
ASSIGN item := _data.Invoices[4].Line[5].Item
```

Most KCM contract manager interfaces supply a Data Backbone XML when composing a template.

Data retrieval

Alternatively, to offer compatibility with older KCM releases, it is possible to define *data retrieval* in the Data Backbone or templates to provide data to the templates. This is an older method that uses DID definitions to specify a data/database abstraction. Templates that use data retrieval retrieve the data directly from a data source: a database or an XML file. For more information on defining data retrieval and DIDs, see "Work with Data" in the *KCM Repository User's Guide* and *KCM DID Developer's Guide*.

In the following example, the entry `Customer` retrieves data from the database. KCM groups the retrieved fields in a record that can be referred to as `Cust`. This record is available between the Template scripts `DO` and `OD`. The fields that the record contains have been defined in the entry definition in the DID. As the method to retrieve the data is also defined in the DID, no more specifications are needed in a template script.

```
...
WITH Cust IN EXP.Customer DO
# Dear Ms. or Mr. @(Cust.Surname),
In reference to your writing...
#
OD
...
```

The construct `WITH` retrieves one record from the database. If KCM should retrieve multiple records, use the construct `FORALL`. The part `DO ... OD` of the `FORALL` construct is executed for each retrieved record.

```
...
WITH Cust IN EXP.Customer DO
#
Dear Ms. or Mr. @(Cust.Surname),
In reference to your writing....
A list of the ordered articles follows:
#
FORALL Art IN Cust.Ordered_articles DO
#
@(Art.Number_of_articles) @(Art.Article_description)
#
OD (* FORALL Art IN Cust.Ordered_articles *)
OD (* WITH Cust IN EXP.Customer *)
...
```

The relation between `Customer` and `Ordered_articles` is defined in the DID.

Be aware of some limitations that Data Retrieval has:

- In the document pack templates data retrieval can only be applied to the Data preparation template. All other templates must use a Data Backbone XML. The Data preparation template results in a Data Backbone XML that operates as input for all other templates in the document pack template.
- Data retrieval may require KEY parameters. These KEY parameters can only be passed with specific interfaces to the composition run. For this purpose, the contract manager offers the CCMCompatibility contract type, which is not added to any of the standard contracts by default.
- Next to the CCMCompatibility contract type the ITPRun core scripting command allows passing keys and executes data retrieval. The data retrieval will be executed only if no DBB_XMLInput parameter is passed.

Note Keys should be supplied in the same order in which the Data Retrieval statements are entered into the template.

Variables and control structures

The contents of a document may differ, based on parameters such as gender.

```
...
Dear #
IF Cust.Sex = "M" THEN # Mr. #
ELIF Cust.Sex = "F" THEN # Ms. #
ELSE # Ms. or Mr. #
FI
# @(Cust.Surname),
In reference to your writing...
```

Use of variables facilitates the process of composing the document.

```
...
TEXT ms_mr
IF Cust.Sex = "M" THEN
ASSIGN ms_mr := "Mr."
ELIF Cust.Sex = "F" THEN
ASSIGN ms_mr := "Ms."
ELSE
ASSIGN ms_mr := "Ms. or Mr."
FI
#
Dear @(ms_mr) @(Cust.Surname),
In reference to your writing...
```

In the second example, a variable of type `TEXT` is declared. KCM assigns a value to the variable based on the contents of the field. The contents of the variable is merged into the text using the @ construct.

Built-in functions

A number of built-in functions are available to format and convert numerical and text values.

```
...
#
@(Cust.Name)
@(Cust.Street) @(Cust.Housenumber)
@(uppercases(Cust.City))
                                Nijmegen, @(date(today))
...
#
...
```

This example produces the following results.

```
Aia Software b.v.
Kerkenbos 10 -129
NIJMEGEN
                                Nijmegen, 12 April 2016
...
```

In this example, the following built-in functions are used: `uppercases`, `date`, and `today`.

`uppercases` converts text to all uppercase.

`today` results in a numerical representation of the current date.

`date` prints a date with the name of the month. The result depends on the language used.

Chapter 2

Includes

This chapter contains information on how to use, define, and manage Includes.

Use Includes in KCM

KCM is able to integrate Includes in a Template script and a result document.

Define Includes

Pre-includes

KCM recognizes the text `__INC(filename)` in a Template script and replaces it with the contents of the Include file. The Include files are stored in the Includes folder in a KCM Designer project.

Post-includes

You can use the `@(inc(...))` function to include documents into a result document. These documents are included from the file system by the KCM Server during document composition.

Specify the directories for KCM to search for Includes in the KCM Core environment.

Manage styles in Includes

Style documents are used to manage styles in documents.

If styles used in Includes are not defined in the style documents, the following applies:

- All styles used in Template scripts and Includes are replaced by the style as defined in the Master Template. Styles defined in more than one Include are replaced by the style defined in the first Include that contained this style. All styles defined in a single Include are left as defined.
- ASCII documents do not contain styles.

Errors in Includes

Pre-includes

An error document is generated if Includes could not be found during the include phase. The error document contains all successfully included Includes and error messages for the Includes that could not be found.

Pre-includes are processed before the script translation, which means that `__INC` statements are still processed. To disable an `__INC` statement, break the `__INC` text with spaces or other characters.

- The inc standard function can be used to generate `__INC` statements in the result document.
- In result documents, `__INC` statements can be also resolved by the `put_in_document`, `put_in_text_document`, and `add_to_output` functions with their fifth parameter set to Y.

Post-includes

The post-include phase takes place during Master Template execution after Template script execution. If KCM cannot find Includes, the user is presented with a PST0005 error message and no output is produced. To generate an error document, you should do the following:

Add the setting `ITPALLOWMISSINGPOSTINC=Y` to the configuration file.

KCM removes the `__INC` statement and continues Master Template execution without throwing an error message if the subdocument cannot be included.

Restrictions on ASCII documents

ASCII documents must conform to the following restrictions:

- The documents must be made in codepage 1252 (Microsoft Windows ANSI).
- The documents can only contain the following control codes: Tab (ASCII 9, Hex: `0x09`); Line Feed (= Paragraph break) (ASCII 11, Hex: `0x0a`); Carriage Return (ASCII 10, Hex `0x0d`); Form Feed (= Page break) (ASCII 12, `0x0c`). Conventional ASCII editors can only produce these four control codes.
- KCM matches these tabs, line feeds, and Form feeds to their appropriate equivalent within Microsoft Word. Carriage returns are ignored.

ASCII documents and styles

ASCII documents cannot contain styles or Microsoft Word instructions. KCM assigns included documents the same style as the paragraph that contains the Include statement in a Microsoft Word document.

Chapter 3

Keywords

To compile documents, you need to use different keywords. This chapter contains descriptions of the keywords and usage instructions.

General

The basic keywords of the Template Scripting language allow you to open and close a Master Template or a section of it.

The Template Scripting language can be used to build an entire document or sections of a document. In either case, the keywords `BEGIN` and `END` define where the Template Scripting language begins and ends and where the document is connected to the database.

Keywords to start and end a Template script

Each Template script must start with `#BEGIN` and end with `END#`.

Any content before the `#BEGIN` or after `END#` is copied into the result document.

Note Do not put `#BEGIN` in the header or footer.

Blocks of code

Variables are restricted to blocks of code in a Template script. These blocks are denoted by the `DO...OD`, `REPEAT … UNTIL`, and `IF … FI` keywords.

Variables are not recognized unless they are placed within a code block.

Scope

You can declare variables, arrays and procedures within any of the blocks. The scope for these variables, arrays or procedures is the block they are declared in. The names of variables, arrays and procedures must be unique in one block (on one level).

Nested blocks

`DO...OD`, `REPEAT … UNTIL` and `IF … FI` blocks can be nested. In a nested block you can declare variables, arrays and procedures with the same names as those on a higher level.

```
#BEGIN
DO
```

```
NUMBER i := 3
    DO
        TEXT i := "xxx"

            first: @( i )
            #
        OD
              #
    second: @( i )
    #
OD
END#
```

```
Result:
first: xxx
second: 3.00
```

The first and second `i` are not the same because they are at the different levels of the instruction.

Naming conventions for variables

Use descriptive variable names that are unique for the Master Template. When variable names are unique, scope issues are detected during the creation of the Master Template and error documents are produced.

```
#BEGIN
IF TRUE THEN
NUMBER my_number := 3
FI
#
The value of my_number is @(my_number).
#
END#
```

Result: None. User does not get an error document.

The variable `my_number` is unknown after the `FI` as it is defined in the `IF` and closed with `FI`. The Master Template cannot be created.

Entries

The following section contains a description of the entry keywords and usage instructions.

FORALL

The data retrieval statement `FORALL` is used to retrieve the fields or subentries of a plural entry to the database. When you browse a DID in the list of entries, you find which entries can be used and are plural.

The following is the definition statement `FORALL`.

```
FORALL <Entry_reference_name> IN <Database_entry_reference> (* See below *)
DO
  ITP-model part with declarations
OD
```

In case there is a main entry, the database subentry reference has the following structure.

```
<DID_three_letter_code>.<DID_defined_Main_Entry>
```

If there are subentries, the database subentry has the following structure.

```
<Existing-Entry_reference_name>.<DID-defined_Sub_Entry>
```

`Entry_reference_name` is the name a script developer assigns to the entry. It has to begin with an uppercase character followed by uppercase or lowercase characters, numbers, and/or underscores.

`Entry_reference_name` can be used in the `DO...OD` part of the Template script to approach fields or sub-entries from the entry.

Note The `DO ... OD` part is executed once for each record found in the database. If no data is found, the `DO ... OD` part is skipped.

The following is an example of a main entry field value retrieval that results in a list of customers.

```
#BEGIN
FORALL Cust IN EXP.Customer
DO
   #
   Customer: @(Cust.Surname)
   #
OD


END#
```

The following is an example of nesting a singular entry access in a plural entry access.

```
#BEGIN
FORALL Cust IN EXP.Customer
DO
    #
    Customer: @(Cust.Initials) @(Cust.Surname)
    #
    WITH Part IN Cust.Partner
        DO
            #
            Partner: @(Part.Initials) @(Part.Surname)
            #
        OD
OD
END#
```

WITH

The data retrieval statement `WITH` is used to retrieve fields or subentry values of a singular entry to the database.

The following is a definition of the statement `WITH`.

```
WITH <Entry_reference_name> IN <Database entry reference> (* see below *)
DO
   <ITP-model part with declarations>
OD
```

The database entry reference has either of the following structures.

In case of a main entry:

```
<DID_three_letter_code>.<DID_defined_Main_Entry>
```

In case of a subentry:

```
<Entry_reference_name>.<DID_defined_Sub_Entry>
```

`Entry_reference_name` is the name for the entry. It must start with an uppercase character followed by uppercase or lowercase characters, numbers, and/or underscores.

This name can be used to approach fields or subentries from the entry.

Note The `DO ... OD` part is executed once for each record found in the database. If no data is found, the `DO ... OD` part is skipped.

The following is an example of the main entry field value retrieval.

```
#BEGIN
WITH Cust IN EXP.Customer
DO
#
Dear Mr. @(Cust.Surname)
#
OD
END#
```

The following is an example of an access to a subentry and field value retrieval.

```
#BEGIN
WITH Cust IN EXP.Customer
DO
WITH Part IN Cust.Partner
    DO
#
@(Part.Surname)
#
    OD
OD
END#
```

PATH

The `PATH` keyword modifies a `FORALL` or `WITH` statement to retrieve records directly from the database. The Template must explicitly provide all keys for the record using the `PAR` keyword.

```
FORALL <Entry_reference_name> PATH <Xxx>.<Entry_sequence> WHERE
 PAR ( 1 ) = <expr>
 PAR ( 2 ) = <expr>
 …
DO
 …
OD
```

`Xxx` is the three letter code of the DID.

`Entry_sequence` identifies a path of entries through the DID, separated by dots. The first entry must be a main entry, the last entry specifies the entry retrieved. Each pair of entries must be defined in the DID as an entry-subentry relation.

`PAR ( i ) = <expr>` defines the expression that provides the value for the i^{th} parameter of the entry. All formal parameters of the entry must be defined using `PAR` keywords. The type of each expression (`TEXT` or `NUMBER`) must match the type of the formal parameter.

An example is provided here.

```
FORALL Child PATH XXX.Customers.Children WHERE
 PAR ( 1 ) = 34162548 (* Customer number *)
 PAR ( 2 ) = "Child" (* Type of relationship *)
DO
 …
OD
```

This example retrieves all records for customer `34162548` with the `Child` attribute. The intermediary `.Customers.` part in the path only serves to identify the `Children` entry. No data is retrieved from the `Customers` entry.

WHERE-PAR

The `WHERE-PAR` keyword modifies a `FORALL` or `WITH` statement to override one or more parameters when retrieving records using an entry-subentry relationship. All parameters that are not explicitly overridden by `PAR` keywords are filled as defined in the DID.

```
FORALL <Entry_reference_name> IN <Entry_reference_name>.<DID-defined_sub_entry> WHERE
 PAR ( <i> ) = <expr>
 …
DO
 …
OD
```

An example is provided here.

```
FORALL Customer IN XXX.Customers
DO
 FORALL Partner IN Customer.Children WHERE
  PAR ( 2 ) = "Partner" (* Type of relationship *)
 DO
  …
 OD
OD
```

This example loops over all Customers, retrieving Children records for each Customer.

The first parameter for the `Children` entry is derived from the Customer record as defined in the DID. The second parameter is overridden by the Template, modifying the request to retrieve records with the `Partner` attribute instead.

Functions and procedures

The following section contains a description of some of the function and procedure keywords and usage instructions.

PROC

A procedure is a script fragment that can be called throughout the Master Template. Create a procedure when you need to use the same piece of code in more than one place.

The following is an example of the procedure declaration.

```
PROC proc_name ( parameter_1; parameter_2; ..., optional parameter list )
DO
   ITP model part with declarations
OD
```

The requirements for declaring a procedure are as follows:

- Procedures must be declared before they are used.
- The name of the `PROC` must start with a lowercase character and is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- The parameter list is optional.

The requirements for the parameter list are the following:

- Parameters must be separated with a semicolon.
- A parameter can be one of four types: `CONST TYPE` name (a parameter cannot be changed in the `PROC`); `TYPE` name (a variable as parameter); `ARRAY TYPE` name (an array as parameter); `MAP TYPE` name (a map as parameter).
- The parameter name (`parameter_1`) must start with a lowercase character and is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- `TYPE` must be used to indicate the type pf parameter. The following types can be used: `TEXT`, `NUMBER`, `BOOL`, and `FIELDSET`. `TYPE` can also be a previously defined `DATASTRUCTURE`. For more information on the types of parameters, see Types.

  Examples are provided here.

```
PROC myproc1(TEXT par; CONST TEXT cpar; ARRAY TEXT apar; MAP TEXT mpar)
DO
OD
```

```
PROC myproc2(BOOL par; CONST BOOL cpar; ARRAY BOOL apar; MAP BOOL mpar)
DO
OD
```

```
PROC myproc3(NUMBER par ; CONST NUMBER cpar; ARRAY NUMBER apar; MAP NUMBER mpar)
DO
OD
```

```
DATASTRUCTURE MyStruct
BEGIN
END
```

```
PROC myproc4(FIELDSET FPar; MyStruct spar)
DO
OD
```

`CONST` parameters values cannot be changed in the procedure. The values of variable and `ARRAY` parameters can be changed in the procedure.

Note ARRAY and MAP cannot be passed as CONST parameters.

FIELDSET and Data Structures cannot be used as an ARRAY or MAP parameter or passed as a CONST parameter.

You cannot use recursive procedures.

The following is an example of a procedure call.

```
APROC proc_name ( parameter_1; parameter_2; ..., the actual parameter list)
```

The number of parameters and their types in the actual parameter list must be the same as the parameter list in the declaration of the procedure.

The following is an example of the definition.

```
PROC amount2 ( CONST NUMBER money ; TEXT result)
DO
    NUMBER positive  :=  money
    IF money < 0
        THEN
        ASSIGN positive  :=  - money
    FI
    ASSIGN result  := number( positive ; 2)
    IF money = 0
        THEN
        ASSIGN result  :=  "-.-"

        ELIF money < 0
            THEN
            ASSIGN result  :=  result + " negative"
    FI
OD
```

Use the procedure as follows.

```
TEXT amount_text
APROC amount2 ( 23.14 ; amount_text)
#
€ @( amount_text)
#
APROC amount2 ( 0.0 ; amount_text)
#
€ @( amount_text)
#
```

In this example, 23.14 is assigned to money and the value of amount_text is assigned to result.

FUNC

Functions are code fragments that can perform a parameterized calculation and return a single value. You can use functions anywhere where a value is expected, increasing the maintainability of Master Templates.

The following is an example of the function declaration.

```
FUNC TYPE function_name ( parameter_1;parameter_2;..., optional parameter
```

```
list)
DO
   ITP part of a model with declarations
   ASSIGN function_name := result_of_function
OD
```

The requirements for creating a function are as follows:

- The `function_name` must start with a lowercase character and is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- The `TYPE` the result of the function; `NUMBER`, `TEXT` of `BOOL`.
- The result of a function is returned by assigning it to `function_name`.
- The parameter list is optional.

The requirements for the parameter list are the following:

- Parameters must be separated with a semicolon.
- A parameter can be one of four types: `CONST TYPE` name (a parameter cannot be changed in the `PROC`); `TYPE` name (a variable as parameter); `ARRAY TYPE` name (an array as parameter); `MAP TYPE` name (a map as parameter).
- The parameter name (`parameter_1`) must start with a lowercase character and is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- `TYPE`, the following types are possible: `TEXT`, `NUMBER`, `BOOL, and FIELDSET`. `TYPE` can also be a previously defined `DATASTRUCTURE`. For more information on the types of parameters, see Types.

Constant parameters values cannot be changed in the function. The values of variable and `ARRAY` parameters can be changed in the function. Functions are always used in the context of a formula.

Note `ARRAY` and `MAP` cannot be passed as `CONST` parameters.

`FIELDSET` and Data Structures cannot be used as an `ARRAY` or `MAP` parameter or passed as a `CONST` parameter.

You cannot use recursive procedures.

The following is an example of a function call.

```
function_name ( parameter_1; parameter_2; ... )
```

The number of parameters and their types in the actual parameter list must be the same as the parameter list in the declaration of the function. Parameters can only be passed to a function when a parameter list is declared in the function declaration.

Types

The following sections contains a description of the rest of the function and procedure keywords and usage instructions.

KCM distinguishes three basic variable types:

- `BOOL` contains `TRUE` or `FALSE`.
- `NUMBER` contains a number or a date with or without decimals.

- TEXT contains a string of characters and word processor instructions.

BOOL

A variable can be declared as BOOL variable and contain either the value TRUE or FALSE. BOOL variables are typically used in control statements like IF, WHILE, and REPEAT. The default value of BOOL is FALSE.

NUMBER

A variable can be declared as NUMBER and contain numbers with or without decimals. The default value of NUMBER is 0. NUMBER is implemented as a floating point number with a precision of 15 digits.

In the result document, NUMBER is by shown with two decimal positions and a thousand separator by default. This can be changed with the format function (see FORMAT functions).

Note You cannot assign a number to a TEXT or BOOL variable directly. Use the numerals and number_in_words functions to convert a number to a text value (see numerals and number_in_words, respectively).

TEXT

A variable can be declared as TEXT. Text variables can contain a mix of text and word processor instructions. The default value of TEXT is an empty text. A TEXT value should be placed between double quotes as shown in the following example.

```
TEXT mytext
ASSIGN mytext := "This is my text"
```

Special characters in a TEXT value

There is a number of characters that you need to escape using a slash. The following is an example of using a slash to escape double quotes.

```
TEXT example := "This is /"only/" an example"
```

If a literal slash is required, you need to escape it with a second slash, as shown in the following example.

```
TEXT example := "mr//mrs"
```

ARRAY

ARRAY defines an array list of variables. Elements of an ARRAY are accessed by their index number.

The first element of the ARRAY has index 1.

The following restrictions apply to ARRAY:

- The maximum index allowed is 2^30.
- The total size of all ARRAYs in a Template script grows as required and is limited by the amount of memory available.

The following is an example of the ARRAY declarations.

```
ARRAY TYPE array_name
ARRAY TYPE array_name [ formula ]
```

The restrictions list is as follows:

- TYPE can be BOOL, NUMBER, TEXT, FIELDSET, or any data structure TYPE.
- The ARRAY name is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- The name must be unique in the scope it is declared in.
- The formula must be of TYPE NUMBER. This determines the initial number of elements in the ARRAY. If the size is omitted, ARRAY is initialized with 0 elements.
- ARRAY variables must be declared before they are used.
- ARRAY variables can be used wherever regular variables of the same TYPE can be used.

```
ARRAY BOOL valid_numbers [ 1000 ]
ARRAY NUMBER hour_rates [ 5 ]
ARRAY TEXT customer_names
```

ASSIGN values to elements in ARRAY

ARRAY elements get a value by assignment.

In the declaration of ARRAY, the number between the brackets represents the initial amount of the elements in the ARRAY. If elements are accessed in the ARRAY above the initial amount, it is extended automatically. The number between the brackets and the brackets themselves are optional. The declared size of ARRAY is used to reserve the initial memory for the ARRAY.

Also, there can be a formula between the brackets that can be calculated when a Master Template is running.

```
ASSIGN valid_numbers [ 1 ] := FALSE
ASSIGN hour_rates [ 5 ] := 234,30
ASSIGN customer_names [ 5 + 9 ] := P.Initials + " " + P.Name
```

In the preceding example, KCM data retrieval is used to retrieve the customer's initials and name from the database to fill this ARRAY.

You can place ARRAY elements in the result document by using the @ construct, as shown in the following example.

```
#

@( valid_numbers [ 1 ] )

@( hour_rates [ 5 ] )

@( customer_names [ 5 + 9 ] )
#
```

This example may results in the following.

```
FALSE
234,30
A Johnson (possible answer)
```

Another example is provided here.

```
ARRAY TEXT list
ASSIGN list[5] := “My value”
```

This assigns the value "My value" to the fifth element in the array.

MAP

MAP is an array that uses a TEXT value as an index.

MAPs are an extension to regular ARRAYs as they allow any text to be used as an index. This gives you the ability to use complex information (such as social security numbers, policy numbers, car license numbers, and so on) as index entries.

MAPs need to be declared before they are used.

The following requirements apply to MAP:

- TYPE can be BOOL, NUMBER, TEXT, FIELDSET, or any data structure TYPE.
- The MAP name must start with a lowercase character and is limited to lowercase characters, digits, and underscores. Spaces are not allowed.
- The name must be unique in the scope where it is declared.
- MAP variables can be used wherever regular variables of the same TYPE can be used.

The number of elements in MAP in a Template script grows as required and is limited by the amount of memory available.

ASSIGN values to elements in MAP

When assigning values to elements in MAP, consider the following:

- The default value of an element in the MAP is an empty text (TEXT), 0 (NUMBER), or FALSE (BOOL).
- If a Template script tries to use an element that is not present in the MAP, the element is added to the MAP with the default value. This new element is then operated on.
- The ASSIGN statement must be used to add an element to MAP or to assign a value to an element.
- Indices are case-sensitive and word processor instructions are ignored. If the setting EnhancedUnicodeMaps is disabled in the KCM Core Administrator, indices are also limited to the Latin-1 character set and all characters not in this set are ignored as well.

To test the presence of an element without creating it, use the functions as shown here.

```
key_used_in_text_map
key_used_in_number_map
key_used_in_bool_map
key_used_in_fieldset_map
key_used_in_struct_map
```

To access an element with a known key or index value, use the functions as shown here.

```
ASSIGN variable_name := map_name[ "key" ]
```

An example of how to add an element and assign a value to that element in MAP is provided here. The MAP element with the text_expression index now contains the value value_expression.

```
ASSIGN map_name [ "text_expression" ] := value_expression
```

An example of how to declare MAP as parameter of a Procedure or a Function is provided here.

```
FUNC TEXT example_func ( MAP TEXT map_name )
```

An example of how to pass MAP to a procedure or a function is provided here.

```
APROC example_proc ( map_name )
```

An example of how to use MAP as a VALUES list in INTERACT is provided here.

```
INTERACT "Test"
   QUESTION "Question"
   VALUES ( map_name )
   ANSWER variable
```

You can assign values of TEXT MAP and FIELDSET to each other, or merge them.

FIELDSET

FIELDSETs are variables that contain a number of named fields. Script developers can work with a FIELDSET as a whole or manipulate fields in FIELDSET as separate TEXT variables.

FIELDSET and KCM Designer

KCM Designer contains Field Sets that can be used in Text Blocks and Forms. To use these Field Sets in dynamic content, they must be exposed in the Data Backbone. Field Sets that are not added to the Data Backbone cannot be selected in the Text Block or Form Editor. When a Template script uses a Text Block or Form that refers to a Field Set that is not in the Data Backbone, it must declare a FIELDSET variable with the same name. The preferred method is to use Field Sets in Text Blocks and Forms that are in the Data Backbone. For more information on using Text Blocks in the Template scripting language, see insert_text_block.

You can declare FIELDSET variables that do not have a corresponding Field Set in KCM Designer. Also, you can manipulate fields to FIELDSETs that are not defined in the corresponding Field Set.

Declare FIELDSET

FIELDSET variables can be declared anywhere in a Template script. FIELDSET names are case-sensitive and must match the same naming conventions as entry names. The names of fields in FIELDSET must match the same conventions as entry field names.

The following declares FIELDSET FieldSet_x.

```
FIELDSET FieldSet_x
```

When working with Text Blocks, the contents of the _data variable is used to locate the Field Sets when the Text Block refers to a Field Set. Wherever a Field of a Field Set is used in the Text Block, KCM substitutes the value of the Field of that Field Set in the Data Backbone instead in the result document.

For dynamic Forms, the Template must explicitly indicate which part of the Data Backbone to use, using the DATA keyword. By default, a dynamic Form looks for Field Sets in the Template itself and ignores the Data Backbone.

Initialize FIELDSET

You can initialize FIELDSET during declaration. The possible assignments are described here.

```
FIELDSET NewFieldSet := ExistingFieldSet   (* Another FIELDSET *)
FIELDSET NewFieldSet := text_map                        (* MAP TEXT *)
FIELDSET NewFieldSet := X0                              (* An entry (within a
FORALL/WITH loop) *)
FIELDSET NewFieldSet := EXP.Customer                    (* A specific record *)
                        WHERE PAR(1) = "1002"
```

ASSIGN FIELDSET to another FIELDSET

Assigning FIELDSET to another FIELDSET clears the content of the destination FIELDSET before copying all fields from the source FIELDSET.

```
ASSIGN DestinationFieldSet := SourceFieldSet
```

Merge FIELDSET into another FIELDSET

Use the +:= operator to merge the contents of FIELDSET into another FIELDSET.

```
ASSIGN DestinationFieldSet +:= SourceFieldSet
```

If a Field exists in both the source and destination FIELDSETs, it is overwritten with the value from the source FIELDSET.

Add values to FIELDSET

Before using Fields from a Field Set in a Text Block, you should assign them values. Use <name of the FIELDSET>.<name of the field> to refer to a field and to assign a value to a field. For example, FieldSet_x.Field1 refers to Field1 in FIELDSET FieldSet_x. Assigning a value to a field in a FIELDSET is done in the same way as assigning a value to a variable.

The values assigned to a field in a FIELDSET must be of the type TEXT. Field Sets that are part of the Data Backbone can be accessed using through the _data variable. You can assign values directly to the Fields in a Field Set, but you can also use Input XMLs or Data Retrieval to assign values to Fields of the Field Set in the Data Backbone

```
ASSIGN FieldSet_x.Field1 := “This is the value of Field1”
```

In this example, the new value is assigned to the Field1 in the FIELDSET FieldSet_x. Every instance of the FieldSet_x.Field1 in a Text Block is replaced by "This is the value of Field1" when that Text Block is inserted in the result document.

ASSIGN values from an entry to FIELDSET

You can fill FIELDSET with all database fields from an entry in a single statement by assigning this entry to the FIELDSET. This clears the FIELDSET and then creates a field in the FIELDSET for each database field in the entry. Numerical database fields are converted to text using the function number with two decimal positions (see number).

There are two ways to assign an entry to FIELDSET:

- Use the identification in a FORALL/WITH loop.

```
FORALL X0 IN EXP.Customers DO
  FIELDSET Customer := X0
OD
```

  In this example, the program loops through the records in the EXP.Customers entry and assigns the fields from each EXP.Customer record to the FIELDSET Customer during the loop. This construction can be used with both singular and plural entries.

- Assign a specific record directly to the FIELDSET using a WHERE clause.

```
ASSIGN FieldSet_x := EXP.Customer WHERE PAR(1) = 1002
```

  In this example, the command retrieves the entry Customer for the customer 1002. If a record is retrieved successfully, it creates a Field for each database Field in this entry in the FIELDSET FieldSet_x and fills it with the value from the record. Only singular entries can be assigned to a FIELDSET using the WHERE clause.

  FIELDSET assignments are generated during the compilation of the Master Template. If the entry changes in the DID, the Master Template has to be recompiled to reflect these changes.

ASSIGN values from TEXT MAP to FIELDSET

You can assign a TEXT MAP directly to a FIELDSET. First, this clears the FIELDSET and then creates a field in the FIELDSET for each key in the TEXT MAP.

In the following example, the command clears the FIELDSET FieldSet and then creates the field FieldSet.A with the value B.

```
MAP TEXT text_map
ASSIGN text_map [ "A" ] := "B"
ASSIGN FieldSet := text_map
```

Note Copying a MAP TEXT into a FIELDSET can introduce elements that have an index that does not meet the naming conventions for FIELDSET fields. Even though these elements cannot be accessed through the Template scripting language or used in dynamic content, these elements remain part of the FIELDSET and are included if the FIELDSET is saved or copied back into a MAP TEXT variable.

ASSIGN values from FIELDSET to TEXT MAP

You can assign FIELDSET directly to TEXT MAP. First, this clears the TEXT MAP and then creates an entry in the TEXT MAP for each field in the FIELDSET. The field name in the FIELDSET is used as the key for this entry.

In the following example, the command clears the TEXT MAP text_map and then creates the value text_map ["A"] with the value "B".

```
FIELDSET FieldSet
ASSIGN FieldSet.A := "B"
ASSIGN text_map := FieldSet
```

Use FIELDSET contents

A Field from `FIELDSET` can be used anywhere in a Template script where KCM expects a value with type `TEXT`.

```
ASSIGN FieldSet.Field := "42"
ASSIGN address_line[3] := FieldSet.ZIPCode + " " + FieldSet.City
ASSIGN variable := lookup_map [ FieldSet.Key ]
#@(FieldSet.Field)#
```

If a Field that is not in the `FIELDSET` is used, it is treated as if the Field was assigned an empty `TEXT` value.

Remove all assigned Fields from FIELDSET

The function `clear_fieldset` can be used to clear all Fields from `FIELDSET`. The result of this function is of type `BOOL`. If `FIELDSET` is cleared successfully or unsuccessfully, the function returns `TRUE` or `FALSE` accordingly.

```
BOOL success := clear_fieldset(FieldSet)
```

Forms and QForms

KCM Designer features Form Editor for Forms. These Forms are not included into a Master Template during compilation but retrieved by KCM Core from the KCM Designer database once a Master Template is run.

`FIELDSET` variables are used to dynamically change the content of the Forms and to receive the answers provided by the user. For more information on dynamic Forms, see FORM. References to Fields in the Forms are inserted from the `FIELDSET`s.

The following table shows the format used to pass defaults and answers between a Template script and the dynamic Forms. As `FIELDSET` values always have type `TEXT`, some values must be encoded.

	Content	Example of content
Text Question	Any text	The Quick Brown Fox Jumps
Date Question	A date in the YYYYMMDD format	20080229
Choice Question	One or more values in comma-separated format. For more details, see create_csv and split_csv	first value, second value, third value
Checkbox Question	Y for checked, N for cleared	Y
Number Question	A number formatted in the active output language	3.1415

Special Field Sets

The following section describes reserved Field Sets of the Template scripting language.

_Template

The `_Template` Field Set is automatically created in any Master Template. This Field Set contains fields representing the Document Template properties defined for the current Master Template.

A metadata XML file can only be generated for Document Template runs that do not have `OutputMode` set to "pack".

The following command retrieves the `Category` property from the Master Template.

```
TEXT category := _Template.Category
```

The `_Template` Field Set is read only. If the property does not exist, the value of the Field is an empty text.

The contents of the `_Template` Field Set are written in the `<Template>` section in the metadata XML file. A typical section is shown here.

```
<ccm:Template>
  <ccm:label name="Wizard">A Content Wizard</ccm:label>
  <ccm:label name="Category">Letter</ccm:label>
</ccm:Template>
```

KCM Core scripts can access the contents through the `template_property()` function.

Note _Template is a reserved name. You cannot declare any object with this name.

_Document

The `_Document` Field Set is automatically created in any Master Template. This Field Set can be used by the Master Template to define additional properties based on retrieved or derived data. The contents of the `_Document` Field Set are written in the metadata XML file and exposed to the calling KCM Core scripts.

A metadata XML file can only be generated for Document Template runs that do not have `OutputMode` set to "pack".

The following command defines the properties CustomerID and OutputFormat.

```
ASSIGN _Document.CustomerID := _data.Customer.SSN
ASSIGN _Document.OutputFormat := "PDF//A"
```

The contents of the `_Document` Field Set are written in the `<Document>` section in the metadata XML file. A typical section is shown here.

```
<ccm:Document>
  <ccm:label name="CustomerId">000-00-0000</ccm:label>
  <ccm:label name="OutputFormat">PDF/A</ccm:label>
</ccm:Document>
```

KCM Core scripts can access the contents through the `document_metadata()` function.

Note _Document is a reserved name. You cannot declare any object with this name.

DATASTRUCTURE

Data Structures are an extension to the Template scripting language with types that define a collection of member variables. Each member of such a collection is treated as a variable with its own type. Script developers can work with the Data Structure as a whole or manage members as variables of the specified type.

In contrast to the other types, Data Structure variables are treated as references. If a Data Structure variable is assigned to another variable, both variables share the same content. Changes made to either variable apply to the other variable as well. If contents should not be shared, script developers must use the statement `NEW` when assigning Data Structure variables. The statement `NEW` copies all contents to the new variable.

Define a Data Structure

Data Structures types must be defined before they are used. Each Data Structure has a name and a list of members with their types. The name of the Data Structure and its members are case-sensitive and must match the same naming conventions as entry names.

You can declare Data Structure types using the keyword `DATASTRUCTURE`.

```
DATASTRUCTURE Name
  BEGIN
    members
  END
```

The following requirements apply to Data Structures:

- The name of a Data Structure must begin with an uppercase character and is limited to uppercase characters, lower-case characters, digits, and underscores. Spaces are not allowed.
- The name must be unique in the scope where it is declared.
- The name cannot be a keyword in the Template scripting language. To avoid conflicts, including with future extensions to the language, do not use all capitals for the name.
- Members are a list of zero or more member declarations.

The `DATASTRUCTURE` statement defines a Data Structure and assigns it a name.

Define members of the Data Structure

Every member of the Data Structure type has a type and a name. The type can be either another KCM type (including other Data Structure types) or an `ARRAY` or a `MAP` of variables of a KCM type.

```
type Name
MAP type Name
ARRAY type Name
ARRAY type Name [number] (* Optional number of elements -- is ignored *)
FIELDSET Name
FIELDSET Name DEFINED_AS OtherName
```

You can define `FIELDSET` members using the `DEFINED_AS` keyword. This keyword indicates that the member should be treated as a Field Set with the alternative name when constructing Content Wizards or inserting Text Blocks.

The following requirements apply to the members:

- The name of the members must begin with an uppercase character and is limited to uppercase characters, lower-case characters, digits, and underscores. Spaces are not allowed.
- The name must be unique in the Data Structure.
- The name cannot be a keyword in the Template scripting language. To avoid conflicts, including with future extensions to the language, do not use all capitals for the name.
- `TYPE` can be any accepted KCM type including previously declared Data Structure types.
- The size of the `ARRAY` members is optional. Any value specified here is ignored when a Data Structure variable is declared and the initial size is always 0.

You can define Data Structure members inline in the definition. For more information, see Inline DATASTRUCTURE definitions.

The following example defines the Data Structure type `Relation` with five members.

```
DATASTRUCTURE Relation
BEGIN
  NUMBER CustomerNumber
  NUMBER Age
  FIELDSET Person
  FIELDSET Partner DEFINED_AS Person
  ARRAY FIELDSET Children DEFINED_AS Person (* Array of 'Person' Field Sets
*)
END
```

The `Partner` member and each element in the `Children ARRAY` is treated as if it is a `Person` Field Set.

The following command defines the Data Structure type `Policy`.

```
DATASTRUCTURE Policy
BEGIN
  TEXT PolicyNumber
  NUMBER InsuredAmount
  FIELDSET PolicyData
  Relation Insured             (* Data Structure *)
  ARRAY Relation Benificiaries (* Array of Data Structures *)
END
```

The `Insured` and `Benificiaries` members have the previously declared `Relation` Data Structure type.

Define a Data Structure variable

Data Structure variables can be declared anywhere in a Master Template document after a Data Set has been defined using the `DATASTRUCTURE` statement. The name of the variable must match the same naming conventions as other variables.

Variables are declared using the `DECLARE` keyword.

```
DECLARE variable
DECLARE variable DEFINED_AS type
```

KCM automatically determine the type of the variable the first time a value is assigned to this variable. All other assignments to that variable must have the same type. The `DEFINED_AS` keyword can be used to explicitly assign a type. Use of the `DEFINED_AS` keyword is necessary when an empty variable is created.

A Data Set variable cannot be initialized during declaration. All members are automatically assigned their default empty value.

An example is provided here.

```
DECLARE life_insurance
DECLARE intermediary
DECLARE customer DEFINED_AS Relation
```

Access members

Members of a Data Set variable can be used where a variable of the corresponding type is allowed.

Use `<Data Set variable>.<name of the member>` to refer to a member.

If the member is `MAP`, `ARRAY`, `FIELDSET`, or another Data Structure variable, you can also directly refer to an element in the member.

```
ASSIGN life_insurance.PolicyNumber := "652365X4564" (*
Assigning a member *)
ASSIGN policy := text_fragment (life_insurance.PolicyNumber; 1; 5) (* Using
a member *)
```

Field Set members can be accessed either directly as a Field Set or on a Field-by-Field basis by referencing specific Fields.

```
ASSIGN customer.Person := Customer (* Copy the content of Customer to
the Person member *)
ASSIGN customer.Person.Name := "Bundy"  (* Only change the Name field *)
ASSIGN fullname := customer.Person.Name + "-" + customer.Partner.Name
```

`MAP` and `ARRAY` members can be accessed as normal `MAP`s and `ARRAY`s.

```
ASSIGN customer.Children[1] := Child (* Copies
FIELDSETs *)
ASSIGN intermediary.Partner.Name := customer.Children[2].Name (* Copies one
field *)
```

If the members are Data Set types, the expressions can be combined into longer expressions.

```
ASSIGN customer.Children[1].Name := "Bundy"
ASSIGN name := life_insurance.Benificiaries[4].Children[1].Name
```

In the preceding example, the program accesses the fourth element from the `Benificiaries` member, which is defined as a `Relation` Data Set. In the `Relation` Data Set, the first element of the `Children` member is accessed, which is itself a Field Set. In this Field Set, the `Name` field is retrieved and assigned to the name variable.

ASSIGN a Data Structure variable to another Data Set variable

You can assign Data Structure variables using the `SET` statement.

```
SET destination -> source     (* Creates a reference *)
SET destination -> NEW source (* Creates a copy *)
```

Assignment of a Data Structure variable to another Data Set variable creates a reference to the source Data Structure. No data is copied. All instances of a reference access the same set of content members,

and changes affect both variables. Both the source variable Data Structure and destination variable Data Structure must have the same type.

The optional keyword `NEW` forces KCM to copy all content from the source Data Structure instead of creating a reference. Any subsequent change to the source Data Structure variable or the destination Data Structure variable affects that variable only.

```
DECLARE pol DEFINED_AS Policy
DECLARE customer DEFINED_AS Relation
DECLARE someone_else DEFINED_AS Relation
...
ASSIGN customer.CustomerNumber := 42
ASSIGN someone_else.CustomerNumber := 1
SET pol.Insured -> customer
```

The preceding example creates a reference between the customer variable and the insured member of the `pol` variable. Both `customer.CustomerNumber` and `pol.Insured.CustomerNumber` share the same content and have the value 42.

```
ASSIGN pol.Insured.CustomerNumber := 100
```

Because of the reference both `customer.CustomerNumber` and `pol.Insured.CustomerNumber` now have the value 100.

```
SET pol.Insured -> NEW someone_else
```

Copy the content of the variable `someone_else` to `pol.Insured`. The existing reference is overwritten by this copy and does not affect the variable customer.

```
ASSIGN pol.Insured.CustomerNumber := 123456
```

Since there is no reference anymore the assignment to `pol.Insured` only affects its own content. `CustomerNumber` still has the value 100. Because of the copy `someone_else.CustomerNumber` is also unaffected and retains the value 1.

You can use the statement `SET` to create references to Data Structure members within a larger variable Data Structure and access these structures directly.

```
FOR i FROM 1 UPTO length_struct_array (life_insurance.Beneficiaries) DO
  DECLARE ben
  SET ben -> life_insurance.Beneficiaries[i]
  ...
  ASSIGN ... := ... ben.CustomerNumber ...
  ASSIGN ben.Person := ...
  ...
OD
```

The following code loops through all elements of the member `ARRAY Beneficiaries` and assigns them to the variable `ben`. Since `ben` is assigned as a reference, it functions as an alias to the current element in the `Benificiaries` member. Any change to `ben` also affects the current element.

The declaration of the variable `ben` does not need an explicit type because this can be derived from the subsequent statement `SET`.

```
DECLARE senior
FOR i FROM 1 UPTO length_struct_array (life_insurance.Benificiaries) DO
  IF life_insurance.Beneficiaries[i].Age > senior.Age THEN
   SET senior -> life_insurance.Beneficiaries[i]
  FI
```

```
OD
...
```

The preceding loop selects the most senior element from the `Benificiaries ARRAY` for further processing.

Inline DATASTRUCTURE definitions

Nested Data Structure members can also be defined inline in a Data Structure. These inline definitions define a member but cannot be used to define Data Structure variables. To declare an inline Data Structure, the definition can be placed in the statement `DATASTRUCTURE`.

```
DATASTRUCTURE Member BEGIN ... END
ARRAY DATASTRUCTURE Member [0] BEGIN ... END
MAP DATASTRUCTURE Member BEGIN ... END
```

Inline Data Structure member definitions can also be nested in another inline definition. The following example defines a Data Structure type `Array2d` that functions as a 2-dimensional array.

```
DATASTRUCTURE Array2d
BEGIN
  ARRAY DATASTRUCTURE D [0]
   BEGIN
    ARRAY DATASTRUCTURE D [0]
     BEGIN
      TEXT T
    END
  END
END
```

The dimensions are implemented using the D-member arrays, the content is accessed in the T-member.

```
DECLARE my_grid DEFINED_AS Array2d
ASSIGN my_grid.D[4].D[8].T := "This is element[4][8]"
```

The inline Data Structure members function as Data Structure variables do but they can only be copied between variables of their parent Data Structure type.

```
SET my_grid.D[1] -> NEW my_grid.D[18]      (* OK - copy *)
SET my_grid.D[2] -> reference_grid.D[1024] (* OK - reference *)
SET my_grid.D[3] -> my_grid.D[11].D[4]     (* Different definitions: not
allowed *)
```

You can use Inline `DATASTRUCTURE` definitions as types by specifying the path to the definition.

In the following example, the member `AnotherOne` has the same structure as the member `One` that includes the member `Two`. The member `AnotherTwo` is a copy of the member's structure `Two`.

```
DATASTRUCTURE Example
BEGIN
  DATASTRUCTURE One
    BEGIN
      DATASTRUCTURE Two
       BEGIN
         TEXT T
       END
    END
  Example.One AnotherOne
  Example.One.Two AnotherTwo
END
```

```
DECLARE full_struct DEFINED_AS Example
DECLARE two_sub_struct DEFINED_AS Example.One.Two

SET full_struct.One.Two -> two_sub_struct
```

PARAMETER

Parameters specify relevant information required to produce a document based on a Master Template using a Document Template. Parameters are available starting from KCM version 4.4 and requires KCM Core and KCM Designer (for Web) 4.4.

The `PARAMETER` keywords allows a Master Template to be parameterized with fixed values and dynamic objects. Whenever a Document Template is defined in KCM Designer the designer provides fixed choices for all parameters. Parameters allow the use of single generic Master Templates to create a range of Document Templates.

The values of the parameters can be accessed in the Master Template through the `_Template` Field Set.

Define PARAMETER

The following code defines `PARAMETER`.

```
PARAMETER TYPE Name_of_the_parameter
```

The restrictions are as follows:

- `TYPE` specifies the type of data passed through the parameter. The following types are available: `TEXT` `NUMBER` (an unformatted integral number), `BOOL` (A boolean value, either Y for true or N for false), `DATE` (a date in YYYYMMDD format), `WIZARD` (a reference to a Content Wizard in the project), `FORM` (a reference to a Form in the project), `TEXTBLOCK` (a reference to a Text Block in the project).
- Parameter names are case-sensitive and must start with an uppercase character. Parameter names follow the same conventions as entry names.
- You must define parameters after the `#BEGIN` instruction.

```
PARAMETER TEXT Sender
PARAMETER WIZARD Document
```

The following is a list of limitations:

- A maximum of ten parameters can be defined in a Master Template.
- Parameter definitions are not supported in combination with auto-includes in KCM Designer for Windows.
- When parameters are added to a Master Template that is already used by one or more Document Templates, the parameters should be added as optional.

Description for parameters

You can give a description to parameters. The description should be put between quotes.

```
PARAMETER TEXT Name_of_parameter
    DESCRIPTION "description_of_parameter"
```

```
PARAMETER TEXT Sender
DESCRIPTION "The name of the sender of this document"
```

```
PARAMETER WIZARD Document
DESCRIPTION "Select a document type to produce. For instance Invoice."
```

Default values for parameters

You can add default values to the parameters in a Document Template with the `DEFAULT` keyword. The system uses the default value when the Document Template does not provide a value for the parameter. This happens if the list of parameters is extended after the Document Template was created or when the Document Template developer does not provide values for optional parameters.

Default values are only allowed for parameters of the types `TEXT`, `NUMBER`, `BOOL`, and `DATE`.

```
PARAMETER TYPE Name_of_parameter DEFAULT "default_value"
```

An example is provided here.

```
PARAMETER TEXT Sender
    DESCRIPTION "The name of the sender of this document"
    EXTENSION
    DEFAULT "Director "J. Doe"
PARAMETER NUMBER Identifier
    DEFAULT "123456789"
```

Note Invalid `DEFAULT` values are not verified and appear in the result document as they are.

Optional parameters

You can define parameters as optional with the `OPTIONAL` keyword. When the keyword is used, the Document Template developer can omit providing values for these parameters when defining a Document Template. If an optional parameter is not explicitly assigned a value, the default value is used. If no default value is specified or if the parameter is of a type that does not allow default values, the value is set to an empty string during composition.

```
PARAMETER TYPE Name_of_parameter OPTIONAL
```

Examples are provided here.

```
PARAMETER TEXT Sender
    OPTIONAL DEFAULT "J. Doe"
```

```
PARAMETER FORM Questions
    DESCRIPTION "Additional questions to provide data for this document."
    OPTIONAL
```

```
IF _Template.Questions <> "" THEN
    FORM VAR _Template.Questions
    DATA _data
FI
```

New parameters

For all mandatory parameters defined in a Master Template, a value must be provided in each Document Template that is based on the Master Template. If the list of parameters must be extended, this also requires an update on all existing Document Templates.

The EXTENSION keyword allows the Master Template to add new parameters without updating existing Document Templates. If a Document Template does not define the parameter, it uses the default value provided by the Master Template.

If no default value is provided, the value will be empty. For the TEXTBLOCK, FORM, and WIZARD parameters that allow no default to be provided, you can use this to check whether a value was provided by the Document Template.

```
PARAMETER TYPE Name_of_parameter EXTENSION
```

The EXTENSION keyword implies that the question also has the OPTIONAL keyword.

Examples are provided here.

```
PARAMETER TEXT Sender
    DESCRIPTION "The name of sender of this document"
    EXTENSION DEFAULT "J. Doe"
```

```
PARAMETER FORM Questions
DESCRIPTION "Some extra questions to retrieve data for this document"
EXTENSION
IF _Template.Questions <> "" THEN
    FORM VAR _Template.Questions
    DATA _data
FI
```

Initial values for parameters

You can use the PREPOPULATE keyword to allow the Master Template developer to provide initial values for parameters in a Document Template. These values are used by the Document Template Editor when a new Document Template is created based on this Master Template. When the Document Template developer is creating the Document Template, these values are filled in.

This value has no effect during composition.

You cannot use PREPOPULATE with the parameters TEXTBLOCK, FORM, and WIZARD.

```
 PARAMETER TYPE Name_of_parameter PREPOPULATE "value"
```

An example is provided here.

```
 PARAMETER TEXT Sender
     DESCRIPTION "The name of sender of this document"
     PREPOPULATE "J. Doe"
```

Note Invalid PREPOPULATE values are silently ignored and do not appear in the Document Template Editor.

Use parameters

The values entered for parameters defined in the Master Template are available in the Master Template through the _Template Field Set (for more information, see _Template).

Examples are provided here.

Using a parameter of the type WIZARD.

```
WIZARD "WizardId"
NAME _Template.Document
```

Using a parameter of the type TEXT.

```
 @(_Template.Sender)
```

I/O operations

FORM

With the statement FORM, the script developer can create QUESTION and ANSWER Forms that are shown when the Master Template is run with KCM ComposerUI for HTML5. The answers provided by the user can be used in a Template script.

The statement FORM is similar to the statement INTERACT (see INTERACT). FORM is used with KCM ComposerUI for HTML5.

KCM ComposerUI for HTML5 application determines how the Form questions and constructs such as BEGINGROUP…ENDGROUP are shown to the user.

Add a FORM

1. To start a Form, add the following instruction to the Master Template.

```
FORM "Form title"
GROUP_STATE
```

The title of the Form is of type TEXT. This text may contain HTML instructions that are passed to KCM ComposerUI for HTML5. This allows parts of the title text to appear in bold or to insert a linefeed. A slash is the KCM escape character. Therefore, an HTML closing tag may look as follows: <//b>. Using incorrect HTML tags may result in damaged layout.

A FORM may contain the following objects:

- one or more QUESTIONs
- optional TEXTBLOCKs
- optional GROUPs
- optional TABLEs (deprecated functionality)
- optional BUTTONs

For information on how to add them to FORM, see the next sections.

You can use the keyword GROUP_STATE to suppress validation and assignment of questions in groups that are not shown due to the conditions SHOW/SHOWNOT for that group. If this keyword is not present, the setting IgnoreUnseenGroups in the KCM Core Environment determines whether or not these questions are validated.

Add identification to a FORM

A Form can have an optional identification. The keyword `ID` allows for assigning identification to a Form and must appear after the keyword `FORM`. The use of the Form `ID`s also makes the use of the Question `ID`s mandatory.

```
FORM "Form title"
ID ("string that determines the ID of the Form")
```

Add a QUESTION to a FORM

You can add `QUESTION`s within the `FORM` statement. A combination of up to 65,000 questions and Text Blocks are allowed in each `FORM`.

```
QUESTION "Question text"
ANSWER variable
```

The `QUESTION` text is of type `TEXT`. This text may contain HTML instructions that are passed to KCM ComposerUI Server. This allows parts of the question text to appear in bold or to insert a linefeed. A slash is the escape character. Therefore, an HTML closing tag may look as follows: <//b>. Using incorrect HTML tags may result in damaged layout.

The answer filled out by the user can be stored in a variable, an `ARRAY`, or an element of an `ARRAY` depending on the type of question. This variable or an `ARRAY` has to be declared in the Master Template before it is used in a `FORM`.

Create a FORM with a file question

While composing a document, the user can upload a file using a file question. The `ANSWER` to the question refers to the uploaded file on the server. For example, the answer can be used to include the file in the document.

File questions are not supported in the dynamic Forms that can be maintained as part of the content in KCM Designer. They have to be part of a static Form in a Master Template instead. The following is an example of a simple static Form that contains a single file question.

```
TEXT f
FORM "File upload"
  QUESTION "Select a file"
    ID("filequestion1")
    FILE
    ERRORCONDITION f=""
    MESSAGE "Please provide a file"
  ANSWER f
```

Note KCM ComposerUI does not enforce any restrictions on file type. The maximum file size is currently 20 MB.

QUESTION

The script developer can use a number of optional elements between the keywords `QUESTION` and `ANSWER`.

```
QUESTION "Question text"
  ID ("string that determines the ID of the question")
```

```
  LEN (maximum length of answer)
  DFT default answer
  DATE
  TIME
  VIEW name of view
  FILTER name of filter
  AUTOINSERT
  EDITABLE_TEXTBLOCK
  VALUES (list with possible answers)
  MULTISELECT
  ORDER
  RADIOBUTTONS
  HELPTEXT text shown by ITP/OnLine as help
  ERRORCONDITION TRUE|FALSE
  MESSAGE message shown when ERRORCONDITION is TRUE
  HIDECONDITION TRUE|FALSE
  READONLY TRUE|FALSE
  FILE
  FILTER "filter"
  ANSWER variable, ARRAY or ARRAY element
```

Keywords that can be used between QUESTION...ANSWER

You can use the following keywords between `QUESTION…ANSWER`.

`ID` keyword

```
ID ("string, determining the ID")
```

The keyword `ID` gives you the possibility to assign identification to a question. The identification is used by the keywords `SHOW` and `SHOWNOT` to determine the question that shows or hides the group and to identify individual questions. The use of the keyword `ID` is mandatory when the `ID` keyword is used on the `FORM` statement.

`LEN` keyword

```
LEN (maximum length of answer)
```

The keyword `LEN` gives you the possibility to set a maximum length for an answer. If the answer that the user has to provide is of type `TEXT`, the argument to `LEN` is a number between 1 and 65535. The value determines the total amount of positions for the answer. If the answer is of type `NUMBER`, the argument to `LEN` consists of two numbers where the first is the maximum number of digits and the second is the number of decimals. These two numbers are separated by a space. For example, `LEN (7 3)` indicates a number with three decimal places and four integral digits.

Note If the user does not enter decimal digits, they are still considered to be present. For example, when `LEN (7 3)` is specified, such answer as "1234" is interpreted as "1234.000," consisting of seven digits with three decimal positions. The total number of digits should be between 1 and 15.

For answers of type `TEXT`, the default length is `LEN (125)`. For answers of type `NUMBER`, the default length is `LEN (15 3)`.

`LEN` is ignored if either of the keywords `DATE` or `TIME` is used.

`DFT` keyword

```
DFT default answer
```

The keyword DFT allows to provide a default answer to a question. This default answer needs to be of the same TYPE as the variable, ARRAY, or an ARRAY element. The default answer is displayed when the Form is loaded. The default answer must be a valid answer to the question and must therefore appear in the list with possible answers when the keyword VALUES is used.

DFT should be an ARRAY if the keyword VALUES or VIEW has been used to provide possible answers (or Text Blocks) and the keyword MULTISELECT has been used to indicate that the user can select more than one answer (or Text Block) from the list. For an EDITABLE_TEXTBLOCK question, DFT should be either a reference to a KCM Designer Text Block or a Text Block reference that was returned by an earlier EDITABLE_TEXTBLOCK question or by a call to the function.

Note If DFT is used in combination with FILE in a static Form, it is ignored.

DATE keyword

When the keyword DATE is specified, the user is asked to select a date. The values that are passed to DFT, VALUES and ANSWER are treated as NUMBER variables and are interpreted as YYYYMMDD. The keyword DATE also changes the way the question is displayed. In the KCM ComposerUI for HTML5 example implementation, the user can select a date from a calendar.

TIME keyword

When the keyword TIME is specified, the user is asked to select time. The values that are passed to DFT, VALUES and ANSWER are treated as NUMBER variables and are interpreted as HHMM in the 24-hour notation. In KCM ComposerUI for HTML the user can select a time from a (localized) time control.

VIEW keyword

```
VIEW name of view
```

Use the keyword VIEW to indicate that the user should select a Text Block from a Text Block List in KCM Designer. The name of the List is indicated by a TEXT value. The answer to the question is a TEXT value containing a reference to the Text Block that the user selected. Combining the VIEW keyword with the MULTISELECT keyword gives the user the possibility to select more than one Text Block.

FILTER keyword

```
FILTER name of filter
```

Use the keyword FILTER to apply the Filter function to the Text Block List. The Filter function is used to select matching Text Blocks from the list before the selection is presented to the user.

AUTOINSERT keyword

This keyword is deprecated, and you should not use it. It was used in combination with the VIEW keyword to automatically insert Text Blocks to be selected by the user in the result document.

To add the selected Text Blocks, use the TEXTBLOCK statement instead, as shown in the following example.

```
(* Declare the answers that will contain the selected Text Blocks *)
TEXT selected_tb
ARRAY TEXT selected_tbs

(* Form to select Text Blocks *)
```

```
FORM "Example"
QUESTION "Make a selection, select one"
VIEW "TBList"
ANSWER selected_tb
QUESTION "Make a selection, more than one is possible"
MULTISELECT
VIEW "TBList"
ANSWER selected_tbs

(* Add the selected Text Blocks to the result document. *)
TEXTBLOCK VAR selected_tb
TEXTBLOCK VAR selected_tbs
```

If the user selects multiple Text Blocks, they are separated by paragraph breaks. This behavior can be influenced by the `AutoInsertSeparator` and `AutoInsertTerminator` pragma functions.

`EDITABLE_TEXTBLOCK` **keyword**

Use the keyword `EDITABLE_TEXTBLOCK` to indicate that the user is asked to edit text content with some layout. The user may be asked to edit existing Text Blocks or to enter new text content with some layout.

```
TEXT newtext_reference
FORM "Enter text"
  QUESTION "Enter new text content"
    EDITABLE_TEXTBLOCK
    ANSWER newtext_reference
(* You can use an EDITABLE_TEXTBLOCK to insert the edited text into the result document
 *)
TEXTBLOCK
VAR newtext_reference
```

When the keyword `EDITABLE_TEXTBLOCK` is specified, the value passed to `DFT` should be a reference to a Text Block in KCM Designer. Also, you can specify the answer to an earlier `EDITABLE_TEXTBLOCK` question in the same Master Template.

```
TEXT edited_textblock1
TEXT edited_textblock2

FORM "Change text"
(* On this Form the question: "Edit the text content", allows you to edit the
 content of textblock1. The edited textblock is accessible through the variable
 editted_textblock1. The content of textblock1 stored in the CCM Repository is not
 changed. *)
  QUESTION "Edit the text content"
    EDITABLE_TEXTBLOCK
    DFT "textblock1"
  ANSWER edited_textblock1
FORM "Check earlier entered text"
(* A edited Text Block can be used as a default for another TEXTBLOCK question *)
  QUESTION "Check the text"
    EDITABLE_TEXTBLOCK
    DFT edited_textblock1
  ANSWER edited_textblock2
(* You can used an EDITABLE_TEXTBLOCK to insert the edited text into the result
 document. *)
TEXTBLOCK
VAR edited_textblock1
TEXTBLOCK
VAR edited_textblock2
```

The answer to an `Editable_TEXTBLOCK` question is a value of type `TEXT`. This value contains a reference to the Text Block content entered by the user. To insert the content provided by the user into the document, use the reference with the statement `TEXTBLOCK` (see TEXTBLOCK statement). You can also use it as the `DFT` value of a subsequent `EDITABLE_TEXTBLOCK` Form question. The reference cannot be stored.

`VALUES` **keyword**

```
VALUES (list with possible answers)
```

Use the keyword `VALUES` to create a list with possible answers. The user can select one of these values from a list to answer the question. The combination of the keywords `VALUES` and `MULTISELECT` gives the user the possibility to select more than one answer from the list.

A semicolon is used to separate items in the list. The items can be of type `TEXT` or type `NUMBER`, depending on the variable the answer is stored in.

```
TEXT answer_day
FORM "Select the answer"
QUESTION "Select a day"
VALUES ("Monday"; "Tuesday"; "Wednesday"; "Thursday"; "Friday"; "Saturday"; "Sunday")
ANSWER answer_day
```

Also, you can pass an `ARRAY` with possible answers.

`MULTISELECT` **keyword**

The `MULTISELECT` keyword is only used in combination with the keyword `VALUES` or `VIEW`. The keyword gives the user the possibility to select more than one answer or Text Block from the `VALUES` list or more than one Text Block from a `VIEW`. To store multiple answers, declare an `ARRAY` to save the answer to.

`MULTISELECT` cannot be used in combination with `RADIOBUTTONS`.

`ORDER` **keyword**

The keyword `ORDER` can only be used in combination with `MULTISELECT`.

When the keyword `ORDER` is defined, the user can select more than one answer or Text Block from the `VALUES` list or the `VIEW` and put this selection in the desired order.

`RADIOBUTTONS` **keyword**

This keyword is only used in combination with the keyword `VALUES`. It shows the possible answers in a radio button list.

`HELPTEXT` **keyword**

Use `HELPTEXT` to provide additional information for a question by means of a `TEXT` value. The help text may contain HTML instructions that are passed to KCM ComposerUI for HTML5. For example, this allows part of the help text to appear in bold or to insert a linefeed. The slash is used as the escape character in quoted text. Therefore HTML closing tags must be written as follows: <//b>. Using incorrect HTML tags may result in incorrect layout.

KCM ComposerUI for HTML5 shows a question mark when `HELPTEXT` is defined. When the mouse moves over this question mark, the help text shows up.

ERRORCONDITION **keyword**

```
ERRORCONDITION TRUE|FALSE
```

ERRORCONDITION gives the script developer the ability to check the answer that the user provides and to show a message. ERRORCONDITION is followed by a value type BOOL. This value is calculated when the user ends the form clicking OK or ENTER. If the value is TRUE, the message specified using the keyword MESSAGE is shown, and the user has to change the answer to the question before proceeding with the form.

MESSAGE **keyword**

When ERRORCONDITION is TRUE, the MESSAGE text is shown to the user. This text may contain HTML instructions that are passed to KCM ComposerUI for HTML5. This allows parts of the message text to appear in bold or to insert a linefeed. The slash is used as the escape character in quoted text. Therefore HTML closing tags must be written as follows: <//b>. Using incorrect HTML tags may result in damaged layout.

HIDECONDITION **keyword**

```
HIDECONDITION TRUE|FALSE
```

HIDECONDITION is followed by a value type BOOL. This value determines if a question is visible or not. The value of the BOOL is calculated when the form is loaded. If the value yields TRUE, the question is not shown. In this case, the DFT value is stored in a variable or an ARRAY element. If no DFT answer has been provided, the variable or the ARRAY element is assigned an empty value (in case of a TEXT), zero (in case of a NUMBER), or FALSE (in case of a BOOL).

By default, a question is shown.

READONLY **keyword**

```
READONLY TRUE|FALSE
```

READONLY is followed by a value type BOOL. This value determines if a question is read only or not. The value of the BOOL is calculated when the form is loaded.

By default, a question is editable.

FILE **keyword**

This keyword gives the user the ability to select a file with a file selection window. A variable or an ARRAY element where the answer to the question is stored must be of type TEXT.

Group the FORM content

QUESTIONs and TEXTBLOCKs that belong together can be grouped in groups.

```
BEGINGROUP "Group title"
  ...
ENDGROUP
```

The group title is of type TEXT. This text may contain HTML instructions that are passed to KCM ComposerUI for HTML5. This allows parts of the text to appear in bold or to insert a linefeed. The slash is used as the escape character in quoted text. Therefore HTML closing tags must be written as follows: <// b>. Using incorrect HTML tags may result in a damaged layout.

The content of the group is defined by QUESTIONs and TEXTBLOCKs that are between the BEGINGROUP and ENDGROUP instruction. In addition to QUESTIONs and TEXTBLOCKs, groups can contain other groups and tables (see Group the FORM content in TABLE).

```
TEXT answer1
TEXT answer2
TEXT answer3

FORM "This is an example"
  TEXTBLOCK "Answer the following questions"

  BEGINGROUP "Group 1"
    QUESTION "This is the first question"
    ANSWER answer1
    QUESTION "This is the second question"
    ANSWER answer2
  ENDGROUP

  QUESTION "This is the third question"
  ANSWER answer3
```

In the example, a form contains a group titled Group 1 that contains two questions. In addition, the form contains a TEXTBLOCK and a third question that are not in the group.

Group the FORM content in TABLE

Use the BEGINTABLE…ENDTABLE instruction to show QUESTIONs and TEXTBLOCKs next to each other in a table.

```
BEGINTABLE "Table title"
  ...
ENDTABLE
```

The keyword BEGINTABLE is followed by the title of a table. The table title is of type TEXT. This text may contain HTML instructions that are passed to KCM ComposerUI Server. This allows parts of the table title text to appear in bold or to insert a linefeed. A slash is the KCM escape character. Therefore, an HTML closing tag may look as follows: . Using incorrect HTML tags may result in damaged layout.

You can specify the table content using one or more BEGINROW…ENDROW instructions between the keywords BEGINTABLE and ENDTABLE.

```
BEGINROW
  ...
ENDROW
```

Each BEGINROW…ENDROW instruction defines a single row in a table. You can specify the row contents using QUESTIONs, TEXTBLOCKs, and groups between the keywords BEGINROW and ENDROW. The title text of QUESTIONs is ignored by the table construct. As an alternative, TEXTBLOCKs can be used to define headers for the table columns or rows.

Including the same number of elements (QUESTIONs, TEXTBLOCKs, and groups) within each BEGINROW… ENDROW instruction.

```
TEXT answer1
TEXT answer2
TEXT answer3
TEXT answer4
FORM "This is an example of a form"
  BEGINTABLE "An example of a table"
```

```
   BEGINROW
      TEXTBLOCK "Question 1"
      TEXTBLOCK "Question 2"
   ENDROW
   BEGINROW
      QUESTION ""
      ANSWER answer1
      QUESTION ""
      ANSWER answer2
   ENDROW
   BEGINROW
      QUESTION ""
      ANSWER answer3
      QUESTION ""
      ANSWER answer4
   ENDROW
 ENDTABLE
```

In the example, the instructions result in a table with three rows and two columns. The first row only contains text and provides the table header.

Keywords that can be used between BEGINGROUP…ENDGROUP

There is a number of optional elements that can be included between the keywords `BEGINGROUP` and `ENDGROUP`.

You can use the following keywords after the keyword `BEGINGROUP`.

`EXPANDABLE` **keyword**

Use `EXPANDABLE` to show or hide a group. `EXPANDABLE` cannot be used in combination with `SHOW` or `SHOWNOT` in a single group.

`EXPANDED` **keyword**

```
EXPANDED TRUE|FALSE
```

The keyword `EXPANDED` is followed by a value type `BOOL`. This value decides whether a group is expanded or collapsed. The value is calculated once KCM ComposerUI for HTML5 questions screen is loaded. By default, a group is expanded.

`HIDECONDITION` **keyword**

```
HIDECONDITION TRUE|FALSE
```

The keyword `HIDECONDITION` is followed by a value type `BOOL`. This value decides whether a group is visible or invisible for the user. The value is calculated once KCM ComposerUI for HTML5 questions screen is loaded. If the value yields `TRUE`, the group is not shown and all questions are assigned the `DFT` value. If no `DFT` answer has been provided, a variable or an `ARRAY` element is assigned an empty value (in case of a `TEXT`), zero (in case of a `NUMBER`), or `FALSE` (in case of a `BOOL`). By default, a group is visible.

`SHOW` **keyword**

You can use the keyword `SHOW` to show a group dynamically, depending on the answer given to a question that precedes the group. The keyword `SHOWNOT` is used to hide a group dynamically. The question needs to have an ID that the keywords `SHOW` or `SHOWNOT` can refer to. When using the keyword `SHOW`, the group is shown when the value of the question that has an ID satisfies the condition.

When the group is currently visible and the user changes the answer to something that does not satisfy the condition, the group turns hidden. When using the keyword SHOWNOT, the behavior is reversed.

```
SHOW ID("id of the question")
SHOW ID("id of the question") = value
SHOW ID("id of the question") </<=/>=/> value
SHOW ID("id of the question") CONTAINS value
SHOW ID("id of the question") IN value
SHOWNOT ID("id of the question")
SHOWNOT ID("id of the question") = value
SHOWNOT ID("id of the question") </<=/>=/> value
SHOWNOT ID("id of the question") CONTAINS value
SHOWNOT ID("id of the question") IN value
```

SHOW and SHOWNOT cannot be used together in the same group.

The following expressions are currently supported:

- ID ("id") requires that the question with ID ("id") is a BOOL question. The group is shown if the answer to the question is TRUE.
- ID ("id") = value requires that the question with ID ("id") is either a TEXT or NUMBER question. The group is shown if the answer to the question is equal to the value.
- ID ("id") </<=/>=/> value requires that the question with ID ("id") is a NUMBER question. The group is shown if the answer to the question is respectively smaller (<), smaller or equal (<=), greater or equal (>=), or greater (>) than the specified value.
- ID ("id") CONTAINS value requires that the question with ID ("id") is a MULTISELECT TEXT question. The group is shown if the answer to the question contains the value of one of the selected items.
- ID ("id") IN value requires that the question with ID ("id") is a TEXT question and the value must be an ARRAY TEXT variable. The group is shown if the answer to the question is present in the array.

The same expressions are supported for the keyword SHOWNOT, but the groups are hidden when the condition is met.

The keywords DATE and TIME are supported for NUMBER answers when performing a test.

The result of an expression can be negated using the NOT operator.

SHOWNOT **keyword**

```
SHOWNOT ID("id of the question") = "test"
```

Multiple expressions can be combined using the logical operators AND and OR. The result of an expression can be negated using the NOT operators. Enclose expressions in brackets to disambiguate the use of AND and OR.

```
SHOW ( (ID("id question 1") = 4) AND NOT (ID("id question 2") = "test") ) OR (ID("id
 question 1") = 42)
```

Note the following limitations when using the keywords listed in this section:

- The ID keyword cannot reference questions that specify the keywords EDITABLE_TEXTBLOCK or VIEW.
- The SHOW and SHOWNOT keywords cannot be used in a group marked EXPANDABLE.

- The CONTAINS keyword cannot be used on an ordered MULTISELECT question, such as a question that has both the keywords MULTISELECT and ORDER.
- The FILE question cannot be used on SHOW and SHOWNOT conditions.

Add a static text to a FORM

A FORM can also contain one or more sections of static text that can be used for messages or additional help text.

```
TEXTBLOCK "Static text content"
```

The type of the text content is TEXT. This text may contain HTML instructions that are passed to KCM ComposerUI for HTML5 This allows parts of the static text content to appear in bold or to insert a linefeed. The slash is used as the escape character in quoted text. Therefore HTML closing tags must be written as follows: <//b>. Using incorrect HTML tags can result in damaged layout.

```
TEXT answer1
FORM "Example form"
  TEXTBLOCK "Please fill out the following question"
  QUESTION "This is the question"
  ANSWER answer1
```

You can specify static text using the keyword TEXTBLOCK.

Available Field Sets

When a default Text Block is set, the user can enter Fields from the list of available Field Sets. They are the same Field Sets that are available when the Text Block is edited in KCM Designer. When no default Text Block is set, no Field Sets are available. To have Field Sets available for an empty Text Block, create an empty Text Block in KCM Designer and use it as a default.

Add a BUTTON to a FORM

You can add a self-defined button to a FORM using the BUTTON ID statement. To determine the behavior of the button, use the ON BUTTON ID statement.

To close a FORM, use the ON ENTER or ON EXIT statements (see Close a FORM). Once all ON BUTTON ID, ON ENTER, and ON EXIT statements are processed, the Form is closed.

```
BUTTON "Name of the button" ID ("id of the button")
ON BUTTON ID("id of the button")
DO
  ASSIGN _Document.ButtonClicked := "id of the button"
OD
```

The part of the script between ON BUTTON ID ("id of the button") DO and OD is executed when the user clicks the button. In the preceding example, a value is passed to the KCM Core script.

You should consider the following:

- Buttons are declared at the FORM level. You cannot use BUTTON within nested FORM statements, such as in questions, groups, or tables.
- Button IDs are case-sensitive.

Close a FORM

You can close a FORM using the buttons NEXT and CANCEL. Also, you can specify actions that should be taken when the FORM is closed using the NEXT button or when the FORM is closed using the CANCEL button.

When the user has completed the Form and presses Enter, Next, or a self-defined button, the answers are stored in corresponding ANSWER variables or ARRAY elements.

If ON ENTER is specified, the actions between DO and OD are executed after the user clicks the NEXT button.

If ON EXIT is specified, the actions between DO and OD are executed after the user clicks the CANCEL button.

Also, KCM continues executing the script code that follows the FORM statement.

```
ON ENTER
DO
 Part of the Master Template that will be executed when the FORM is closed using the OK
 button
OD
```

Default answers and validation

All Forms created in a Master Template or as a QFORM in the KCM/Model Development Kit are subject to validity checks. Value lists and lengths are enforced whenever possible.

The validation uses the following principles:

- default values that violate constraints are presented to the user who has to provide a valid answer.
- any question that presents a list of choices to the user must have valid defaults from that list. KCM Core automatically updates the defaults as described in this section.
- any question that is hidden or read only cannot be changed by the user.

Length attributes are only enforced on the user entered content. Answers from a choice list are excluded from validation.

Dates and times are only validated on the user entered content. Invalid answers selected from a choice list, such as selecting February, 29 in a non leap-year, are still accepted.

Questions that are unconditionally hidden or are in a group that is unconditionally hidden are not validated. Questions in a group that are conditionally hidden based on other questions are validated. If the answer is invalid, the question is shown to the user for correction. You can disable this validation with the keyword GROUP_STATE on the statement FORM or through the global setting IgnoreUnseenGroups in the KCM Core Environment.

All choice lists must have a valid default even if the questions are read-only or hidden. If the default in a single select list is invalid, KCM Core selects the first entry from the list. If the default in a multi-select list is invalid, KCM Core strips all invalid items from the list. This change provides consistent behavior for read-only questions which are visible to the user but cannot be changed.

Note You can configure KCM Core at the level Environments to disable answer validation.

EXTRA

You can use EXTRA variables to pass information from an application to KCM Templates. A list of values for EXTRA variables is passed to KCM Core at runtime. If no values are provided, the user is prompted interactively to provide the information.

Note Interaction requires that the Template is used from KCM ComposerUI for HTML5.

There are three types of EXTRA variables: TEXT, NUMBER, and BOOL.

```
EXTRA TEXT name_variable
   LEN ( length )
   DFT ( "standard value" )
   PROMPT ( "prompt parameter" )
   VALUES ( "value list" )
   CHOICE ( "choice parameter" )
```

```
EXTRA NUMBER name_variable
   LEN ( length )
   DFT ( "standard value" )
   PROMPT ( "prompt parameter" )
   VALUES ( "value list" )
   CHOICE ( "choice parameter" )
```

```
EXTRA BOOL name_variable
   PROMPT ( "prompt parameter" )
   DFT ( "standard value" )
```

The variable declared with the EXTRA keyword is treated as a regular variable after declaration. It must match the naming restraints that apply for these variables.

The EXTRA declaration can have several parameters. The LEN parameter is required, the other parameters only have an effect when the user is interactively queried to provide a value. The parameters are used to provide an interactive query to the user. They are not used to validate input provided by an application.

Note For Document Pack Templates the Extras variables are only supported in the Data preparation template. The Data preparation template results in a Data Backbone XML that is used as input for other templates in the document pack template.

LEN

LEN is always required for EXTRAs of type TEXT or NUMBER.

When EXTRA is of type TEXT, LEN expresses the maximum number of characters the user is allowed to enter interactively. If an application provides a value for the EXTRA variable, the LEN property is ignored.

When EXTRA is of type NUMBER, LEN expresses the number of digits (between 1 and 15) and decimal positions in the value. These values must be separated by a space.

The behaviour depends on how the value for the EXTRA NUMBER is provided:

- When the application provides the value, the number of digits is ignored and the number is interpreted as an integral number, and the last digits are interpreted as decimal positions.

- When the user is interactively queried for a value the LEN keyword restricts the range of input values.

Example is provided here.

```
EXTRA NUMBER number_variable
   LEN (5 3)
```

This example specifies a variable with 5 digits and 3 decimals.

If an application provides the value 12345678, this is interpreted as 12345.678.

If the user provides a value interactively, the form restricts the answer to a value between -99.999 and 99.999.

DFT

DFT is the default value of EXTRAas presented to the user. This keyword is ignored when the application provides the answer to this EXTRA question. The default value is determined by the type of EXTRA and by the LEN setting.

- TEXT EXTRA. The default value has to be TEXT. The default value has to comply with the LEN setting.
- NUMBER EXTRA. The default value has to be TEXT that represents a number. The default value must comply with the LEN setting.
- BOOL EXTRA. The default value has to be TEXT and must be "Y" or "N".

```
EXTRA TEXT name_variable_text
   LEN ( 10 )
   DFT ( "Johnson Al" )
EXTRA NUMBER name_variable_number
   LEN ( 5 3 )
   DFT ( "12.345" )
 EXTRA BOOL name_variable_bool
   DFT ("N")
```

PROMPT

The PROMPT parameter is of type TEXT. Enclose its value in double quotes.

PROMPT is the question presented to the end-user if no value is assigned to the variable before the execution of the Master Template. If the parameter is not set, the name of the EXTRA is used as the question.

```
EXTRA TEXT name_variable
   LEN ( 10 )
   DFT ( "Johnson Al" )
   PROMPT ("Select a customer:")
```

VALUES

The VALUES parameter is of type TEXT. Enclose its value in double quotes.

When the parameter is set, the end-user is presented with a list of possible values to choose from.

VALUES must be set in different ways for the different types of EXTRAs:

- TEXT EXTRA. Values set must be enclosed in single quotes and separated by spaces. The complete list must be enclosed in double quotes.
- NUMBER EXTRA. Values must be separated by spaces. The complete list must be enclosed in double quotes.

```
EXTRA TEXT name_variable
   LEN ( 10 )
   DFT ( "Johnson Al" )
   PROMPT ("Select a customer:")
   VALUES ("'Johnson Al' 'Franklin A' 'Parker C' 'Biafra J'")
```

Note The value set at DFT must be part of the list of values set with VALUES.

CHOICE

CHOICE is a parameter of type TEXT.

The CHOICE parameter presents the user with a descriptive list of choices. In KCM ComposerUI for HTML5 this option sets the help text for the question.

In a KCM Core environment, you can use an EXTRA variable to pass run-time information to a Master Template.

You also use an EXTRA variable during Master Template development to pass dummy data to parts of a Master Template. This gives you the ability to debug parts of the Master Template.

INTERACT

INTERACT is a legacy alias for the FORM keyword.

The INTERACT keyword does not support dynamic Forms.

WIZARD

You can use the statement WIZARD to present the end-user with the Content Wizard where Text Blocks can be selected.

Note To run a Master Template containing this feature, use KCM ComposerUI for HTML5.

The following is the definition.

```
WIZARD id
  NAME path
  FOLDER path
  SELECTION_INPUT map
  SELECTION_OUTPUT map
  SUPPRESS_EDITABLE_TEXTBLOCKS
  SUPPRESS_QFORMS
  SHOW_ALL
  NON_INTERACTIVE
  DATA_DEFINITION NONE
  VIEWS_USE_ORDERING
```

The label `id` identifies the Content Wizard for later use with the statement `FOREACH WIZARD` (see FOREACH WIZARD/NODE). If multiple Content Wizards use the same label `id`, the statement `FOREACH WIZARD` used the data from the last occurrence.

Note If the following descriptions do not state that a keyword is optional, it is mandatory.

`NAME` specifies the `Content Wizard` path presented to the end user.

`FOLDER` specifies that the user can choose a Content Wizard from the Content Wizard path or its subfolders. If there is only a single choice possible, the selection screen is suppressed and the choice is made automatically. If a Title property is configured in KCM Designer for the Content Wizard path, it is shown as the title of the selection screen.

`NAME` and `FOLDER` are mutually exclusive. One of the keywords must be specified.

The path parameters are paths to the appropriate objects in KCM Designer. This path must include all containing folders separated by backslashes.

`SELECTION_INPUT` specifies a map that contains a set of overrides that are applied to the Content Wizard before it is presented to the user. This keyword is optional. If no map is specified, the defaults as specified by the editor of the Content Wizard are used.

`SELECTION_OUTPUT` specifies a map that receives a list of the state of each item that was presented in the Content Wizard. This keyword is optional.

`SUPPRESS_EDITABLE_TEXTBLOCKS` prevents the Content Wizard from showing any Editable Text Blocks that are otherwise presented to the user based on the selected Text Blocks. This keyword is optional.

`SUPPRESS_QFORMS` prevents the Content Wizard from showing any QForms that are otherwise presented to the user based on the selected Text Blocks. This keyword is optional.

`SHOW_ALL` specifies that all mandatory items in the Content Wizard should be presented to the end-user. This keyword is optional and can be ignored depending on the definition of the Content Wizard in KCM Designer.

Note The keywords `SUPPRESS_QFORMS` and `SHOW_ALL` only work for legacy ComposerUI for ASP.NET and ComposerUI for J2EE and have no effect in ComposerUI for HTML5.

`NON_INTERACTIVE` specifies that the Content Wizard should be run in a non-interactive modus. This keyword allows Content Wizards to be used in batch environments. The Content Wizard selects all default values for the Content Wizard, suppresses the editing of Editable Text Blocks, and selects all default values in the associated QForms. If the defaults do not satisfy constraints on the QForm or on the Content Wizard, such as the number of selected Text Blocks, an error is reported, and the Master Template stops.

`DATA_DEFINITION` specifies a variable that can be used as Data Definition when representing repeating structures in the Content Wizard. Currently, only the value `NONE` is allowed.

`VIEWS_USE_ORDERING` indicates that Text Block multi-selects that are based on a Text Block List should be presented as an ordered list. This feature gives the user the ability to reorder the Text Blocks. This keyword is optional.

Mandatory items in the Content Wizard

By default, all mandatory items in the Content Wizard are hidden from the end user. You can configure this in the Content Wizard Editor in KCM Designer. If the Content Wizard developer selects the option "a Master Template decides" in the Content Wizard Editor, the keyword `SHOW_ALL` can be used in a Master Template to change the behavior.

QForms

The Content Wizard can define one or more QForms. The Sections and Text Blocks that the user selects can be accessed with the statement `FOREACH WIZARD` (see FOREACH WIZARD/NODE).

Data Backbone context

The Data Backbone is implicitly used to control the Content Wizard, and QForms, and Text Blocks it contains. You can disable this implicit use by specifying `DATA_DEFINITION NONE` on the `WIZARD` statement. If the Data Backbone is disabled, all Field Sets used in the Content Wizard, QForm, and Text Blocks must be declared and in scope before the `WIZARD` statement is executed. If a Field Set cannot be resolved, a runtime error is reported.

SELECTION_INPUT and SELECTION_OUTPUT maps

You can use the keyword `SELECTION_OUTPUT` as input to the keyword `SELECTION_INPUT` in another `WIZARD` to reproduce a default state for the Content Wizard (for more information, see MAP).

The map contains keys that consist of a sequence of Section titles separated by backslashes. The sequence can optionally be followed by a Text Block title. For a Content Wizard containing Section "a" and Subsection "b" with Text Block "tb1," Subsection "b" may be presented by "a\b" and Text Block "tb1" may be presented by "a\b\tb1." Each key has a value indicating the selection state. These values are:

- `MS` indicates Mandatory Selected.
- `OS` indicates Optional Selected.
- `OD` indicates Optional Deselected.

Input that may alter Mandatory Sections or Text Blocks as defined in the Content Wizard Editor is ignored.

Input that sets Optional Sections or Text Blocks to `MS` is handled as if they are set to `OS`.

If no input is provided for a Section or Text Block, the definition as entered in the Content Wizard Editor is maintained.

Incorrect input is ignored:

- Sections or Text Blocks that do not exist in the Content Wizard Editor;
- Sections or Text Blocks that may result in multiple selections in a single select situation.

The `SELECTION_INPUT` and `SELECTION_OUTPUT` maps cannot differentiate between multiple identical items in the same section. A Content Wizard always writes the `MS` value into the `SELECTION_OUTPUT` map if one of the objects was `MS`. Otherwise, if at least one of the objects was selected, it writes the `OS` value. If none of the objects is selected, the `OD` value is used.

```
# BEGIN
```

```
WIZARD "myID"
  NAME "my wizard"

FOREACH WIZARD Element IN WIZARD "myID"
DO
  IF Element.Type = "section" THEN

#
Wizard @(Element.Wizard) [@(Element.WizardQForm)]
Section @(Element.Name) [@(Element.QForm)]
#
  ELIF Element.Type = "textblock" THEN
    TEXTBLOCK VAR Element.TextBlock
  FI

    FOREACH WIZARD SubElement IN Element
    DO
      IF SubElement.Type = "section" THEN

#
SubSection @(SubElement.Name) [@(SubElement.QForm)]
#
      ELIF SubElement.Type = "textblock" THEN
        TEXTBLOCK VAR SubElement.TextBlock
      FI
    OD
OD

END #
```

The `VIEWS_USE_ORDERING` keyword is introduced in KCM version 4.4 and requires KCM Core and KCM ComposerUI Server 4.4 or later with a KCM ComposerUI Server application that supports ordered Text Blocks.

STOP

The `STOP` statement is deprecated. Use `ERROR` instead (see ERROR).

When the statement `STOP` is encountered, Master Template execution stops and all text that has been produced so far is placed in the result document.

The following is the definition of the `STOP` statement.

```
STOP (return code) (* TEXT, optional, up to 7 characters *)
```

You can place a return code of type `TEXT` in the `STOP` statement. The first seven characters of the return code are available in a calling KCM Core Script in the _itpstop variable.

```
STOP ( "code 1")
```

WRITE

You can use the statement `WRITE` to save the contents of the `DATABACKBONE` into an XML file on the KCM Core server.

The following is a definition of the `WRITE` statement.

```
WRITE datastructure
```

```
  TO file
```

The `datastructure` is a Data Structure variable that is written to the XML file.

In the following example, the program writes the Data Backbone to the file named saved_data.xml. The keyword `TO` specifies the file in which the data is written.

```
WRITE _data TO session_path + "\saved_data.xml"
```

ERROR

When the `ERROR` statement is encountered, Master Template execution stops with an specified error message. No result document is produced.

Definition of the statement is as follows.

```
ERROR message
```

The text of the message is logged with an error code USR1000. This can be read from the _itplog KCM Core variable.

```
ERROR "Inconsistent data found. Unable to produce correspondence."
```

Use of the `ERROR` statement causes the KCM Core command `ITPRun` to fail as if a run-time error occurs. Scripts can detect this condition by testing for a _returncode value of 23.

WARNING

The `WARNING` statement reports a custom warning message from a Master Template and has no effect on the execution.

Definition of the statement `WARNING` is as follows.

```
WARNING message
```

The text of the message is logged with a message code USR1001. This can be read from the _itplog KCM Core variable.

```
WARNING "Processing customer " + _data.Customer.CustomerNr
```

The `WARNING` statement can be combined with the `ERROR` statement (see ERROR) to provide additional information.

Control Structures

IF

The `IF` statement conditionally execute parts of the Template. The test, a formula of type `BOOL`, is evaluated. If it is `TRUE`, the `THEN` part is executed. If it is `FALSE`, the next part of the statement is executed.

The following is a definition of the statement.

```
IF The Test (* formula of type BOOL *)
THEN
   ITP-model part with declarations
ELIF The Test2 (* formula of type BOOL *)
THEN
   ITP-model part with declarations
ELSE
   ITP-model part with declarations
FI
```

If a condition is false, you can use `ELSE` or `ELIF` to specify the actions to perform. Use `ELSE` to indicate a single set of actions. Use `ELIF` when an alternate set of conditions is to be considered. Within the `ELIF`, there can be `ELSE` and `ELIF` statements if the condition is false. For more information on these statements, see

FOR

The `FOR` statement repeats a block of instructions a fixed number of times. The number of times that the script part is executed depends on the `FROM` and `UPTO` formulas. When reaching the `FOR FROM UPTO` statement, KCM calculates arguments and truncates the values to an integer value. KCM executes the script part and with every pass 1 is added to the `FROM` value until the `UPTO` value is reached.

The following is a definition of the statement.

```
FOR loop_counter
      FROM formula
      UPTO formula
DO
   ITP-model part with declarations
OD
```

The loop counter is a `NUMBER` variable that must be declared before the `FOR` statement. The loop counter can be used in the `DO...OD` statement. Any changes to this variable are discarded at the end of the `DO...OD` part and do not affect the number of loops.

Note If the formula `UPTO` produces a smaller number then the formula `FOR`, the `DO - OD` is not executed.

```
#BEGIN
NUMBER counter
FOR counter
FROM 2
UPTO 6
DO
#
The counter is @(number(counter; 0)).
#
OD
END#
```

```
Result:
The counter is 2.
The counter is 3.
The counter is 4.
The counter is 5.
The counter is 6.
```

FOREACH

The statement FOREACH enumerates a set of objects and runs a block of instructions once for each object. The enumerated object is available in a variable for use in the script part.

You can use specific FOREACH statements for each type of enumeration. All FOREACH statements use the following format.

```
FOREACH <kind> <variable> IN <object>
DO
    ITP-model part
OD
```

FOREACH KEY

The statement FOREACH KEY enumerates all keys in a MAP variable and runs the script part for each key.

The following is a definition of the statement.

```
FOREACH KEY key IN map
DO
    ITP-model part
OD
```

```
FOREACH KEY key IN SORTED map
DO
    ITP-model part
OD
```

The enumeration variable (key) of type TEXT must be declared before the FOREACH loop. The map can be any type of MAP. You can use the keyword SORTED to enumerate the keys in a alphabetically sorted order. If this keyword is omitted, the ordering of the enumeration is undefined. Changes to the map in the DO … OD section do not affect the enumeration.

The following is an example of the statement.

```
# BEGIN
MAP NUMBER example
ASSIGN example["one"] := 1
ASSIGN example["two"] := 2
ASSIGN example["three"] := 3
ASSIGN example["four"] := 4
TEXT key

FOREACH KEY key IN SORTED example
DO

#
@(key)      @(number(example[key];0))
#

    ASSIGN example["magic"] := 42
OD

#
Show changes: Element 'magic' = @(number(example["magic"];0))
#
```

```
END #
```

```
Result:
four      4
one       1
three     3
two       2
Show changes: Element 'magic' = 42
```

The `'magic'` element is not part of the enumeration because it is introduced in the `DO...OD` part.

FOREACH WIZARD/NODE

The statements `FOREACH WIZARD` and `FOREACH NODE` enumerate selected Sections and Text Blocks in a Content Wizard.

The following is a definition of the statement.

```
FOREACH WIZARD Element IN WIZARD id
DO
  ITP-model part
OD
```

The statement `FOREACH WIZARD Element IN WIZARD id` enumerates the top level Sections and Text Blocks the end-user selected in the last presented Content Wizard window with the label id. If no such Content Wizard was presented to the end user, a fatal error is reported.

```
FOREACH WIZARD SubElement IN Element
DO
    ITP-model part
OD
```

The statement `FOREACH WIZARD SubElement IN Element` enumerates all Subsections and Text Blocks the end-user selected in the Element. You can use this statement only in the part `DO … OD` of another `FOREACH WIZARD` statement, and the Element must be the loop variable of such a statement.

```
FOREACH NODE Element IN WIZARD id
DO
  ITP-model part
OD
```

The statement `FOREACH NODE` resources through the Content Wizard and performs the same functionality as a nested set of `FOREACH WIZARD` loops.

`FOREACH NODE` and `FOREACH WIZARD` loops can be nested but such nesting starts a new enumeration from the start of the Content Wizard.

The loop variables `Element` and `SubElement` are automatically declared and available in the loop `DO … OD`. You can choose the names but they must match the rules for valid `FIELDSET` names. These loop variables behave for all intents and purposes as read-only `FIELDSET` variables. You can copy `Element` and `SubElement` to regular `FIELDSET` variables.

The following fields are defined.

`Type`. The `Type` field indicates the object type of the current element. This field can have the following values:

- `"section"`. The object is a Section.

- "textblock". The object is a Text Block or a Text Blocks List.
- "group". The object is a Group.
- "list". The objects is a Data Definition group representing a DATASTRUCTURE, an ARRAY of DATASTRUCTUREs, or an ARRAY of FIELDSETs.
- "item". The object is an iteration in the Data Definition group. These objects only occur in "list" elements. Elements hidden by conditions in the Content Wizard are represented by an empty "item" element.

Name. If the current element is a Text Block, the Name field contains the name and the path to the Text Block as specified in the Content Wizard. If the element allows multiple selections, this field contains a comma-separated list (CSV format) of Text Blocks. For other elements this field contains the name of the element as shown in the Content Wizard.

TextBlock. The TextBlock field contains the name and the path to the Text Block containing the effective content of the element. If the element allows multiple selections, this field contains a comma-separated list (CSV) of Text Blocks. If the element is editable, this field contains a reference to the edited Text Blocks. The content of this field can be passed directly to the statement TEXTBLOCK to produce the content of the Text Blocks. This field is empty if the current element is not a Text Block.

MinElements and MaxElements. These fields contain respectively minimum and maximum elements that can be chosen in the group. If MaxElements contains 0, the maximum number of elements is unlimited. These fields are empty if the current element is not a group.

Wizard. This field contains the name of the Content Wizard.

QForm. This field contains the name of the QForm that was defined in the Section or subsection. This field is empty if the current element is not a Section.

WizardQForm. This field contains the name of the QForm that was defined at the top level of the Content Wizard. This field is empty except for top-level Sections and groups.

Editable. This field contains Y if the Text Block could be edited during a Content Wizard run. It contains N if the Editable option was not selected for the Text Block in the Content Wizard Editor.

Count.This field contains the total number of elements in a Data Definition group. This field is present in both "list" and "item" elements.

Current. This field contains the sequence number of the current iteration in a Data Definition group. This field is present in "item" elements.

Member. This field contains the DATASTRUCTURE member over which the Data Definition group is enumerating. This field is present in "list" elements.

Path. Internal path in the Data Definition that indicates which FIELDSETs in the Data Definition are currently visible. This value can be passed to the TEXTBLOCK statement to apply the correct data to the Fields in the Text Blocks and associated QForm.

Note The content of the Path field is intended for internal use only and remains valid only until the next time a Content Wizard with the same ID is presented or the Master Template ends. Its value can be copied to variables but cannot be stored externally. Any changes or manipulations are rejected.

Level. This field contains the nesting level of the element in the Content Wizard.

Note The content of this field is of type `TEXT`. Relative comparisons should use `text_to_number()` to convert the value to a `NUMBER` first. For more information on the `text_to_number()` function, see text_to_number.

Example

```
# BEGIN
WIZARD "myID"
NAME "my wizard"

FOREACH WIZARD Element IN WIZARD "myID"
DO
  IF Element.Type = "section" THEN

#
Wizard @(Element.Wizard) [@(Element.WizardQForm)]
Section @(Element.Name) [@(Element.QForm)] at level: @(Element.Level)
#
  ELIF Element.Type = "textblock" THEN
    TEXTBLOCK VAR Element.TextBlock
  FI

    FOREACH WIZARD SubElement IN Element
    DO
      IF SubElement.Type = "section" THEN

#
SubSection @(SubElement.Name) [@(SubElement.QForm)] at level: @(SubElement.Level)
#
      ELIF SubElement.Type = "textblock" THEN
        TEXTBLOCK VAR SubElement.TextBlock
      FI
    OD
OD

END #
```

```
Result (for a sample Content Wizard):

Wizard my wizard []
Section one [] at level: 1
--text block 1.1--

SubSection one.two [qform 1] at level: 2
--text block 1.2.1--
--text block 1.2.2--
--text block 1.2.3--

SubSection one.three [] at level: 2

Wizard my wizard  []
Section two [qform 2] at level: 1
--text block 2.1--
```

The following is the same loop written using `FOREACH NODE`.

```
# BEGIN
WIZARD "myID"
NAME "my wizard"

FOREACH NODE Element IN WIZARD "myID"
```

```
DO
  IF Element.Type = "section" THEN
    IF Element.Level = "1" THEN
#
Wizard @(Element.Wizard) [@(Element.WizardQForm)]
Section @(Element.Name) [@(Element.QForm)] at level: @(Element.Level)
#
    ELSE
#
SubSection @(SubElement.Name) [@(SubElement.QForm)] at level: @(Element.Level)
#
    FI
  ELIF Element.Type = "textblock" THEN
    TEXTBLOCK VAR Element.TextBlock
  FI
OD

END #
```

The `FOREACH WIZARD` example only processes two levels of elements in the Content Wizard. The `FOREACH NODE` example processes **all** elements in the Content Wizard.

Data Definition groups and Text Blocks. Any use of the statement `TEXTBLOCK` to insert a Text Block in a `FOREACH WIZARD/NODE` loop will by default apply the context of the `FOREACH` loop to the Text Blocks and associated QForm. Any Field Sets used in the Text Block and QForm are resolved in the context of the Data Definition first. If no Data Definition is defined, Field Sets are searched outside the Data Definition.

Text Blocks in a Data Definition group always refers to the Field Sets in the current iteration of the Data Definition group. This handles repeating elements automatically.

You can use the keywords `DATA_DEFINITION` and `PATH` on the statement `TEXTBLOCK` to either disable the context handing in a `FOREACH WIZARD/NODE` loop, or to simulate this behavior outside the loop `FOREACH WIZARD/NODE` by keeping track of the `Path` field associated with the Text Block.

FOREACH SLOT

The statement `FOREACH SLOT` enumerates all documents that are produced when running a Document Pack Template.

The following is a definition of the statement.

```
FOREACH SLOT Document IN DOCUMENTPACK
DO
  ITP-model part
OD
```

The loop variable `Document` is automatically declared and available in the `DO ... OD` loop. The name of the variable can be chosen but must match the rules for valid `FIELDSET` names. This loop variable behaves for all intents and purposes as a read-only `FIELDSET` variable. The contents can be copied to a regular `FIELDSET` variable.

If the `FOREACH SLOT` statement is used in a Template run as the Data Preparation Template or run outside the context of a Document Pack, the `DO ... OD` part is not executed. As an Data Preparation Template cannot produce a document, this Template is excluded from the enumeration.

The following fields are defined.

- `Type`. The `Type` field indicates the type of the current slot. This field can have the following values:
  1. `"Template"`. This slot contains a Document Template.
  2. `"Import"`. This slot contains an Import Document.
- `Slot`. This field contains the name of the slot as entered in the Document Pack Template Editor. If the slot is not assigned a name, this field remains empty.
- `Name`. This field contains the slot identifier of the slot as entered in the Document Pack Template Editor.
- `Template`. This field contains the full path of the Document Pack Template.
- `Description`. This field contains the description of the slot in the Document Pack Template. If the slot is empty, the field contains the name of the Document Template.
- `Optional`. This field indicates whether or not the slot is defined as optional in the Document Pack Template. If the slot is defined as optional, this field contains the value Y, otherwise N.

Note Optional slots must be selected by the user or application to appear in the `FOREACH SLOT` enumeration.

- `GUID`. This field contains the `GUID` that identifies the slot.

The following is an example of the statement.

```
# BEGIN

  ...

#
Please find the following documents attached:
#
FOREACH SLOT Doc IN DOCUMENTPACK
DO
  IF Doc.Slot <> "Coverletter" THEN
#
- @(Doc.Description)
#
  FI
OD

END #
```

The result for this example Document Pack is as follows.

```
Please find the following documents attached:
- Policy
- Disclaimer
- Invoice
```

WHILE

The `WHILE` statement repeats a block of instructions while the test condition evaluates to TRUE.

The following is a definition of the statement.

```
WHILE the test condition
DO
   ITP-model part with declarations
```

```
OD
```

```
#BEGIN
NUMBER counter := 3
WHILE counter > 0
DO
#
The counter is @(number(counter;0))
#
    ASSIGN counter := counter - 1
OD
END#
```

```
Result:
 The counter is 3
 The counter is 2
 The counter is 1
```

REPEAT

The `REPEAT` statement executes a block of instructions and then checks the `UNTIL` condition. While the `UNTIL` condition evaluates to `FALSE` the block of instructions is repeated once more.

The following is a definition of the statement.

```
REPEAT
   ITP-model part with declarations
UNTIL stop_test (* formula of the type BOOL *)
```

The code between `REPEAT` and `UNTIL` always runs once. If you want to optionally not run it, use the statement `WHILE` (see WHILE).

```
#BEGIN
NUMBER counter := 3
REPEAT
#
The counter is @(number(counter;0))
#
    ASSIGN counter := counter - 1
    UNTIL counter = 0
END#
```

```
Result:
The counter is 3
The counter is 2
The counter is 1
```

Dynamic building blocks

Dynamic FORM statement

KCM Designer provides a Form Editor that can design Dynamic Forms interactively.

To use a Dynamic Form in a Template, a special form of the `FORM` statement is used.

Note Dynamic Forms are not inserted, as they do not generate any text themselves.

Insert a dynamic FORM

To insert a dynamic Form, add the following lines to the Template script.

To use a Dynamic Form, add the following statement to the Template.

```
FORM
 NAME "<folder>\<form>"
```

The `NAME` keyword specifies the name of the Form as it is defined in KCM Designer. KCM Designer can recognize this statement and include the relation in its dependency analysis.

The `FORM` statement also accepts an expression with the `VAR` keyword.

```
FORM
 VAR reference
```

This variant should be used if the name of the Form is passed as a `PARAMETER` to the Master Template.

```
PARAMETER FORM DynamicForm
FORM
 VAR _Template.DynamicForm
```

Dynamic Forms and the Data Backbone

The Dynamic Forms refer to Fields within Field Sets to present questions and store answers.

The `DATA` keyword indicates the Data Structure variable that should be used scope in the Data Backbone where Field Sets are located.

```
FORM
 NAME "<name>"
 DATA <data structure>
```

If the `<data structure>` is a path into a subnode, the Template searches for Field Sets in reverse order through this path. The path can refer to elements within arrays to indicate specific nodes in the Data Structure.

```
FORM
 NAME "Invoices\CustomerData"
 DATA _data
```

The preceding example uses the top-level Field Sets in the Data Backbone.

```
FORM
 NAME "Invoices\CustomerData"
 DATA _data.Company.Relations[4]
```

The receding example searches through the Data Backbone in the following order:

1. Field Sets in the 4th Data Structure in the `_data.Company.Relations` array.
2. Field Sets in the `_data.Company` Data Structure.
3. Top-level Field Sets of the Data Backbone.

If a Field Set could not be located a run-time error is reported.

If the DATA keyword is omitted Field Sets will be located in the scope of the FORM statement and the scopes of callers.

```
FIELDSET Customer
FORM
 NAME "Invoices\CustomerData"
```

This statement looks for the Field Set Customer in the scope of the FORM statement and does not refer to data from the Data Backbone.

TEXTBLOCK statement

Use the statement TEXTBLOCK to insert regular or Rich Text Blocks in the result document.

The following is a definition of the statement.

```
TEXTBLOCK
  NAME "name"
  VAR expression
  PARAGRAPH_SYMBOL expression
  STYLE_PREFIX expression
  TABLE_STYLE_PREFIX expression
  ASSIGN_TO variable
  QFORM expression
  SUPPRESS_FINAL_PARAGRAPH
  DATA_DEFINITION NONE
  PATH expression
```

Description of the keywords.

- NAME "name"

  Use NAME to specify a fixed Text Block name or list of Text Blocks.

  ```
  TEXTBLOCK NAME "Salutation"
  TEXTBLOCK NAME "\Customers\Correspondence\Clause14"
  TEXTBLOCK NAME "Salutation, Invoice, Signature"
  ```

  The Text Blocks specified with the NAME keyword are used by KCM Designer in its dependency analysis.

- VAR expression

  Use VAR to specify an expression that evaluates to the name of a Text Block or list of Text Blocks.

  The expression can either evaluate to a TEXT value, for a single Text Block, or an ARRAY TEXT value for a list of Text Blocks.

  You can use the VAR keyword in combination with the PARAMETER keyword to parameterize Master Templates.

  ```
  PARAMETER TEXTBLOCK Clause
  ```

```
TEXTBLOCK VAR _Template.Clause
```

The NAME and VAR keywords are mutually exclusive.

- PARAGRAPH_SYMBOL expression

  This optional keyword specifies a paragraph break that defines the base layout for paragraphs in the Text Block. The expression must evaluate to a string that contains a single paragraph break.

  Because of limitations in the Microsoft Word DOC file format, paragraphs can only be inserted into a table if each paragraph break is also derived from a table. To enforce this, use PARAGRAPH_SYMBOL to pass a paragraph break from within a table.

Description of the TEXTBLOCK options

TEXTBLOCK

The TEXTBLOCK keyword indicates that a Text Block is inserted. It requires at least NAME or VAR.

```
NAME "folder\name of the (rich) text block"
NAME "\folder\name of the (rich) text block"
NAME "MyProject.name of the (rich) text block"
NAME "Myproject.\folder\name of the (rich) text block"
```

If a Text Block specification does not start with a backslash, KCM Repository searches for the object in a number of folders that were specified during project configuration. Text Blocks have their own search paths that can be configured on the Runtime tab of the project configuration window.

You can specify an absolute location when inserting an object. A backslash in front of the name or path of the object signifies an absolute location. The Text Blocks specified are searched for in that exact location. A backslash comes after a possible project specification.

If the Master Template refers to the Text Block with a name that includes the project, it searches for the object in that project. You can include the project to the reference by adding the project name and a dot in front of the actual name.

```
VAR variable
```

Use VAR if the name or folder and name of the Text Block is stored in a variable or a Field from a Field Set. If the variable or the Field contains more than one Text Block selected with a Text Block MULTISELECT question, the statement TEXTBLOCK handles all Text Blocks in the Field Set. As the Text Blocks are saved to the Field or variable as a CSV, you can also split the different Text Blocks in separate values using the split_csv function (for more information, see split_csv).

PARAGRAPH_SYMBOL "paragraph symbol"

This optional keyword is mainly used when a Text Block needs to be inserted in a table cell. PARAGRAPH_SYMBOL contains the paragraph symbol that holds the style applied to the Text Block. To insert this Text Block in a paragraph cell, declare the paragraph symbol in a table cell as well. A corrupted result document could be produced if an incorrect paragraph symbol is used.

STYLE_PREFIX "name of the prefix"

This optional keyword contains the prefix of the set Text Block styles that needs to be applied to this particular Text Block.

If this parameter is left empty, KCM searches for the Text Block styles with the ITP prefix.

`TABLE_STYLE_PREFIX "name of the prefix"`

This keyword is optional. It contains the prefix of the Text Block table styles to apply to this particular Text Block. This keyword has no effect on tables in a Rich Text Block.

If this parameter is left empty, KCM searches for the Text Block table styles with the prefix set by the `STYLE_PREFIX` parameter. If the `STYLE_PREFIX` parameter is also left empty, KCM searches for the Text Block table styles with the ITP prefix.

`ASSIGN_TO variable`

This keyword is optional. You can use it if the Text Block should be stored in a variable and not printed at the location where the statement `TEXTBLOCK` is called.

`QFORM qform`

This keyword is optional. You can use a QForm to ask the end-user questions to provide answer for Fields that do not have values yet. The end-user is only asked questions for the Fields that are also used in inserted Text Blocks. You can only use a QForm when you insert a Text Block.

`SUPPRESS_FINAL_PARAGRAPH`

This keyword is optional. If this keyword is used, the paragraph marker at the end of the final paragraph of the Text Block is omitted. This can be used to treat Text Blocks as text fragments.

Note If the Text Block ends with a list element, the final paragraph marker cannot be omitted.

`DATA_DEFINITION NONE`

This keyword is optional. The `TEXTBLOCK` statement uses the Data Backbone to replace any fields in the Text Block with the corresponding values from the Field Sets. If the keyword `DATA_DEFINITION NONE` is specified the Field Sets are located in the scope of the `TEXTBLOCK` statement instead of in the Data Backbone.

`PATH variable`

`PATH NONE`

This keyword is optional. If this keyword is omitted, the `TEXTBLOCK` statement determines the location of the fields based on the context within the current nested `FOREACH WIZARD` / `FOREACH NODE` statements.

The `PATH` keyword overrides this specifies an explicit context. The value for the `PATH` keyword must be a value retrieved from the `.Path` field in a `FOREACH WIZARD` / `FOREACH NODE` variable.

The special value `PATH NONE` disables the automatic derivation without providing an explicit context.

Manage Text Blocks styles

The style of Text Blocks is translated into styles in the Microsoft Word style sheet. The style names are generated on the basis of the attributes defined for the paragraphs, lists, and tables.

Available ITP_ styles

The following built-in `ITP_` styles are used to lay out a Text Block:

- `ITP_normal`. Basic style. This style is used when no layout has been applied to the Text Block.
- `ITP_normal_x`. Basic style for indentation. This style is used for Text Blocks that use indentation. Replace `x` with the level of indentation.
- `ITP_header`. Basis style for the header.
- `ITP_header_x`. Additional style for the header. Replace `x` with the level of indentation.
- `ITP_numbered_list`. Basic style for a numbered list.
- `ITP_numbered_list_x`. Additional style for a numbered list. Replace `x` with the level of nesting.
- `ITP_bullet_list`. Basic style for a bulleted list.
- `ITP_bullet_list_x`. Additional style for a bulleted list. Replace `x` with the level of nesting.
- `ITP_table`. Basic style for a table.
- `ITP_customclass1`, `ITP_customclass2`, and `ITP_customclass3`. Custom styles. See Define ITP_customclass styles.

You can find the sample Text Block Style document that contains these `ITP_` styles in the Style Documents folder of a new KCM Repository DOC or DOCX project. By default, this document is configured as the Style Document (Style Sheet) on the project. You can use the styles on your Text Blocks in KCM Repository or copy them to a Style Sheet in KCM Designer.

Note Whenever you manually add these `ITP_` styles to a Style Sheet, verify that the names correspond exactly to those on the list.

Styles structure and elements

All styles in a set are grouped under a common prefix. Once you insert a Text Block, all appropriate styles are looked up in a set that begins with a specified prefix. The default prefix is `ITP` (see the previous section).

You can override the default prefix with the `TEXTBLOCK` command using the `STYLE_PREFIX` keyword for regular text and the `TABLE_STYLE_PREFIX` keyword for tables in a Text Block and their contents. The `TABLE_STYLE_PREFIX` keyword has not effect on tables in a Rich Text Block.

You can use different prefixes to define different layout for the body of a Text Block and the content of tables within this Text Block (see Define an alternate set of styles). When the `TABLE_STYLE_PREFIX` is not set but the `STYLE_PREFIX` is set, the `STYLE_PREFIX` prefix is also used for and within tables.

Paragraph styles

```
<prefix>_<style><level>
```

- `prefix` is the style prefix.
- `style` is the style attribute of the `<par>` element.
- `level` is the indentation level of the text, prefixed with an underscore. Level 0 is not included in the naming.

These names are mapped to paragraph styles in Microsoft Word.

The exact definition of these styles depends on the style sheets defined in KCM Repository and the run-time configuration of the template.

Examples

`ITP_normal` is used for normal text with no indentation.

`ITP_header_2` is used for a header indented twice.

`OUR_header` is used for highlighted text with no indentation in a Text Block for which `STYLE_PREFIX` is set to `OUR`.

List styles

```
<prefix>_<style>_list<level>
```

- `prefix` is the style prefix.
- `style` is the type of list. Numbered lists use the style numbered; bulleted lists use the style bulleted.
- `level` is the nesting level of the list prefixed with an underscore. Nesting level 0 is not included in the naming.

These names are mapped to list styles in Microsoft Word.

The exact definition of these styles depends on the style sheets defined in KCM Repository and the run-time configuration of the template.

Examples

`ITP_bulleted_list` is used for a bulleted list not enclosed within another list.

`ITP_numbered_list_3` is used for a numbered list enclosed within three other lists, regardless of the type of list.

Table styles

```
<prefix>_table
```

`prefix` is a prefix differentiating style sets.

These table style names are mapped to table styles in Microsoft Word.

Best practice is to define a table style for every style set in use.

The content of table cells is mapped according to the rules, matching their context. When you only use the `ITP_` style set, *all* content is formatted according to the definition of the `ITP_` style set. If you need to format the content of the table differently from text outside the table, you should define a second style set to apply to tables (see Define an alternate set of styles).

Use the `TABLE_STYLE_PREFIX` keyword on the `TEXTBLOCK` statement to use an alternate style for the tables in your Text Block (for more information on this keyword, see TEXTBLOCK statement).

Examples

`ITP_table` is the typically used table style.

`OUR_TABLE_normal` is used for normal text with no indentation for contents within a table in a Text Block for which the `TABLE_STYLE_PREFIX` is set to `OUR_TABLE`.

Define an alternate set of styles

You can define an alternate set of styles. This may be useful in the following cases:

- To lay out Text Blocks in several different styles with one document
- To lay out the content of table cells differently from the text style used in the Text Block

To do so, you may create an alternate style set with a new prefix, define each style in this set, and apply them to your Text Blocks.

Example

`XXX_normal`. Alternate style for a regular text

`XXX_header`. Alternate style for the header

`XXX_table`. Alternate style for a regular table

Use the `STYLE_PREFIX` keyword on the statement `TEXTBLOCK` to insert the Text Block with an alternate style in the result document (for more information on this keyword, see TEXTBLOCK statement).

Define ITP_customclass styles

There are three built-in custom styles that you can apply to a regular Text Block in the Text Block Editor:

- `ITP_customclass1` (Custom 1)
- `ITP_customclass2` (Custom 2)
- `ITP_customclass3` (Custom 3)

You can customize these styles if you need a special formatting for the regular Text Blocks. To do so, add these styles to the appropriate Style Sheet, define them, and then apply them to the Text Blocks in the Text Block Editor.

Note When adding the styles to a Style Sheet, verify that the names correspond exactly to those on the list.

Insert Text Blocks in a table

Use the option `STYLE_PREFIX` prior to inserting a Text Block in a table cell. This option allows you to pass the paragraph sign required while inserting the Text Block. When a paragraph sign created within a table cell is passed, you can insert the Text Block in the table cell (for more information on `STYLE_PREFIX`, see TEXTBLOCK statement).

Insert multiple Text Blocks

Use the `TEXTBLOCK` statement to insert multiple Text Blocks at once. To do so, use either the `VAR` keyword with an `ARRAY TEXT` argument, or the `NAME/VAR` keyword with a `TEXT` argument that contains a comma-separated list of Text Blocks.

```
TEXTBLOCK
  NAME "First,Folder\Second,Third,Other Folder\Last"
ARRAY TEXT clauses
TEXTBLOCK
  VAR clauses
```

When multiple Text Blocks are inserted, the `SUPPRESS_FINAL_PARAGRAPH` keyword is only applied to the final Text Block. The individual Text Blocks are always separated by at least one paragraph break.

You can use the AutoInsertSeparator and AutoInsertTerminator pragmas to add additional text between the Text Blocks. For more information, see pragma.

Text Blocks and Data Definitions

If the statement `TEXTBLOCK` is used in `FOREACH WIZARD` and `FOREACH NODE` loops, the context of the `FOREACH WIZARD` and `FOREACH NODE` statements is used to automatically determine the location of the Text Block in the Content Wizard. If the Content Wizard uses Data Backbone to support repeating groups and the `FOREACH WIZARD` loop is in a Data Definition Group, the current iteration of the Data Definition Group elements is determined to pick Field Sets in that loop. This ensures that Text Blocks in a repeating group refer to the correct iteration.

If the statement `TEXTBLOCK` is used outside any `FOREACH WIZARD` loops, or Field Sets cannot be located in the Data Backbone, the Field Sets are additionally searched for in the current scope in the Master Template.

You can use the keywords `DATA_DEFINITION` and `PATH` to change the default behavior. You can use the keyword `DATA_DEFINITION` to disable the use of the _data variable and limit the use of Field Sets to local variables.

The keyword `PATH` indicates a location in the Data Backbone where the Text Block tries to locate Field Sets. Only values copied from the Field Path in the statement `FOREACH WIZARD` can be used with the keyword `PATH`. Use `PATH NONE` to ignore any repeating or nested structures in the Data Backbone.

In the following example, the Text Block using the currently active `FOREACH WIZARD` context is inserted.

```
TEXTBLOCK
  NAME "..."
```

The following example demonstrates that any applicable `FOREACH WIZARD` context is ignored and Field Sets that are currently in scope are used.

```
TEXTBLOCK
  NAME "..."
  DATA_DEFINITION NONE
  PATH NONE
```

The following example shows that the variable `my_path` is used to determine the active Data Definition context. This can be used outside `FOREACH WIZARD` loops to control the context in which the Text Block is produced. The variable _data is used to retrieve the content.

```
TEXTBLOCK
  NAME text_block
  PATH my_path
```

Master Template Defined VIEWs

With Master Template Defined Views you can combine and filter Text Blocks Lists from KCM Designer to use them with the `FORM` statement.

Master Template Defined Views are introduced in KCM version 4.4 and require KCM Core version 4.4 onwards.

The following is a definition of the statement.

```
DEFINE VIEW Name AS
  FILTER "filter"
  VIEW "view1"
  ...
  VIEW "view_n"
```

You cannot use variables in the definition of the Master Template Defined View.

The following statement defines a Master Template Defined View and specifies its name. You cannot define Master Template Defined Views.

```
DEFINE VIEW Name AS ... VIEW "..."
```

The `FILTER` keyword is optional and specifies a Filter function that should be used to filter Text Blocks in the Master Template Defined View.

```
FILTER "filter"
```

The following statement is required and specifies a View included in the Master Template Defined View. You can repeat this option to create a Model Defined View composed of multiple Views. It is allowed to use previously defined Master Template Defined Views in a Master Template Defined View. The `VIEW` keyword terminates the definition of the Master Template Defined View.

The `FILTER` keyword specifies a Filter function used to select Text Blocks from the parent Views. The Filter function is applied whenever the Master Template Defined View is used in a `FORM` statement, using the actual state of the Data Backbone. If a Master Template Defined View includes one or more other Master Template Defined Views, Text Blocks in these Views are filtered before the filter on the resulting Text Blocks is applied.

The following example command creates a View filtered based on the `state` Filter function.

```
DEFINE VIEW StateClauses AS
  FILTER "state"
  VIEW "Clauses"
```

The following example command combines Views `ClauseSet1`, `Folder\ClauseSet2` and `ClauseSet3` into one View. Duplicate Text Blocks are not removed from the resulting Master Template Defined View.

```
DEFINE VIEW AllClauses AS
  VIEW "ClauseSet1"
  VIEW "Folder\ClauseSet2"
  VIEW "ClauseSet3"
```

The following example command creates a View that is filtered on the `texas` Filter function using the `StateClauses` Master Template Defined View in the preceding example. The resulting Master Template Defined View will contain all Text Blocks from the `Clauses` View that pass both the `state` and `texas` Filter functions.

```
DEFINE VIEW TexasStateClauses AS
  FILTER "texas"
  VIEW "StateClauses"
```

Functions and procedures in Libraries

Libraries are a means to create reusable instructions when Master Templates are executed. While Includes are compiled in Master Templates, Libraries are self-contained, dynamic components.

Libraries can contain different kinds of functions and procedures:

- Formatting functions used to format Fields in Text Blocks.
  You can extend the list of available functions by defining your own format functions in a Library.
- Filter functions used to filter Text Blocks in Text Block Lists.
- Export functions and procedures can be used in Master Templates.
  These functions and procedures cannot produce word processor instructions.
- Procedures that will be used as dynamic exit points.

Note The following restrictions apply when removing or renaming Libraries:

- Library functions cannot be renamed or removed after the Code Library has been unlocked.
- A Library cannot be renamed or removed if an object refers to one of the functions it contains.

For more information on how to create and edit a Library, see the section "Create and edit a Library" in the *KCM Designer online Help*.

FORMAT functions

With format functions, you can expand the Field formatting functionality in Text Blocks and Quick Documents in KCM Designer. Format functions are introduced in KCM version 4.4.

Note Currently, custom format functions cannot be applied to Editable Text Blocks in KCM ComposerUI, except for KCM ComposerUI if it is integrated using the `tbeformatfunctions` parameter.

Format functions are specified using the `FORMAT` keyword prefix. Format functions can only be defined in Code Libraries, and they cannot be defined in conditional statements or other nested constructs.

```
FORMAT FUNC TEXT function (CONST TEXT input)
DO
  ...
OD

FORMAT FUNC TEXT function (CONST TEXT input; CONST TEXT par1; ...)
DO
  ...
OD
```

A format function must have at least one `CONST TEXT` parameter that receives the data to be formatted. The function must return the formatted data as `TEXT`.

Format functions can have additional `CONST TEXT` parameters that can be used to pass additional information. These parameters are optional. If a Text Block or a Quick Document omits parameters, they are passed as empty texts. Excess parameters in a call are ignored.

Format functions have access to the Data Backbone through the _data variable and can use any Template scripting language functionality.

To use Microsoft Word instructions in Code Libraries, you must assign content containing Microsoft Word instructions to Data Backbone Fields in a Master Template. Use these Fields in Code Libraries when manipulating layout. Microsoft Word instructions typed directly in Code Libraries are stripped.

```
FORMAT
FUNC TEXT dotteddate (CONST TEXT date)
DO
  ASSIGN dotteddate := fragment_of_characters (date; 7; 2) + "."
                     + fragment_of_characters (date; 5; 2) + "."
                     + fragment_of_characters (date; 1; 4)
OD
```

Here is an example of the Function used in a Text Block or a Quick Template to print a date in DD.MM.YYYY format.

```
... «[Policy.Date:dotteddate]» ...
```

```
FORMAT
FUNC TEXT logo (CONST TEXT blob)
DO
  ASSIGN logo := insert_image ("Corporate Logo"; x:="15cm"; y:="1.25cm"; width:="4cm";
 height:="2cm"; base64data:=blob)
OD
```

Here is an example of the Function used in a Text Block to insert a base64 encoded image from a Field in the Data Backbone in the upper right corner of the page.

```
... «[Corporate.Logo:logo]» ...
```

```
FORMAT
FUNC TEXT verb (CONST TEXT gender; CONST TEXT male; CONST TEXT female; CONST TEXT
 other)
DO
  IF   gender = "M" THEN ASSIGN verb := male
  ELIF gender = "F" THEN ASSIGN verb := female
  ELSE                   ASSIGN verb := other
  FI
OD
```

The Function used in a Text Block or a Quick Template implements conditional text in a Text Block.

```
... If the «[Customer.Gender:verb|customer|customer|customers]»
 «[Customer.Gender:verb|prefers|prefers|prefer]»,
 «[Customer.Gender:verb|he|she|they]» can pay in ...
```

Note You cannot use word processing layout and instructions in Quick Templates. Word processor instructions produced with the `FORMAT` functions, including the use of `insert_image` and `insert_signature`, are removed from the result document.

FILTER functions

Filter functions are used to implement custom filters. These filters can be used to select Text Blocks in a Text Block List.

Filter functions are introduced in KCM version 4.4.

Filter functions are specified using the `FILTER` keyword. Filter functions can only be defined in Code Libraries, and they cannot be defined in conditional statements or other nested constructs.

```
FILTER FUNC BOOL filter (FIELDSET Properties)
DO
  ...
OD
```

A filter function must return TRUE if the Text Block should be included in the View, or FALSE if it should be excluded. The filter function can use the function `metadata_contains` to test for metadata attributes that are set on the Text Block.

Filter functions have access to the Data Backbone through the _data variable and can use any KCM language functionality. Attributes of the specific Text Block can be accessed through the FIELDSET parameter.

Text Block properties

The `Properties` Field Set contains some properties of the Text Block that is being filtered:

- `Properties._Name` is a name of the Text Block.
- `Properties._Rich` contains `"Y"` if the Text Block is a Rich Text Block, `"N"` otherwise.

The following command excludes all Rich Text Blocks from a Text Block List.

```
FILTER
FUNC BOOL text (FIELDSET Set)
DO
  ASSIGN text := (Set._Rich <> "Y")
OD
```

The following command includes all Text Blocks where the `"State"` metadata attribute contains the value from the `Customer.State` Field in the Data Backbone.

```
FILTER
FUNC BOOL state (FIELDSET Ignored)
DO
  ASSIGN state := metadata_contains ("State"; _data.Customer.State)
OD
```

EXPORT functions and procedures

With export functions you can write functions and procedures in Code Libraries available to all Master Templates in the project you are working on.

Export functions are introduced in KCM version 4.4.

Export functions are specified using the `EXPORT` keyword prefix. Export functions can only be defined in Code Libraries, and they cannot be defined in conditional statements or other nested constructs. An exported function can be used in Master Templates and in higher-numbered Code Libraries.

To use Microsoft Word instructions in Code Libraries, you must assign content containing Microsoft Word instructions to Data Backbone Fields in a Master Template. Use these fields in Code Libraries when manipulating layout. Microsoft Word instructions typed directly in Code Libraries are stripped.

```
EXPORT FUNC returntype function (parameters)
DO
  ...
```

```
OD

EXPORT PROC procedure (parameters)
DO
  ...
OD
```

Any function or procedure can be exported.

```
FUNC BOOL is_div (CONST NUMBER a; CONST NUMBER b)
DO
  ASSIGN is_div := ((a/b) = truncate (a/b; 0))
OD

EXPORT
FUNC BOOL is_leap_year (CONST NUMBER year)
DO
  ASSIGN is_leap_year := (is_div (year; 4) AND NOT is_div (year; 100)) OR is_div (year;
 400)
OD
```

The preceding code tests if a year is a leap years. The main function `is_leap_year` is available in other Code Libraries and Master Templates. The helper function `is_div` is not exported and can only be used in the Code Library it is defined in.

Exit point procedures

The following sections describe dynamic exit point procedures that can be implemented in Code Libraries. Once such a procedure is defined, it is automatically called during a Master Template run.

Note We recommend that you do not define a single exit point procedure more than once in your Code Libraries. For example, if you define DBB_PRE_LOAD twice in one Code Library, it is unknown which one will be used when this exit point is reached.

DBB_PRE_LOAD

You can execute the `DBB_PRE_LOAD` exit point to provide a custom routine that loads XML data passed with the `DBB_XMLInput(...)` parameter on the `ITPRun` command.

In the following example, `XML` is a `DATASTRUCTURE` that describes the XML data structure.

```
DBB_PRE_LOAD PROC load_gax (XML xml)
DO
  ...
OD
```

If the `DBB_PRE_LOAD` exit point is not present, the `DBB_XMLInput(...)` parameter is used to fill `_data`.

```
DATASTRUCTURE GAX
  BEGIN
    TEXT   PolicyNumber
    NUMBER ExpirationDate
  END

DBB_PRE_LOAD
PROC load_gax (GAX gax)
DO
```

```
  ASSIGN _data.PolicyData.Policy := gax.PolicyNumber
  IF gax.ExpirationDate <> "" THEN
    ASSIGN _data.PolicyData.Expires := gax.ExpirationDate
  FI
OD
```

The preceding example reads an XML file formatted as shown in the following example and uses its contents to fill `_data`.

```
<GAX>
  <PolicyNumber>123.456.7890</PolicyNumber>
  <ExpirationDate>20151231</ExpirationDate>
</GAX>
```

The `DBB_PRE_LOAD` exit point is introduced in KCM version 4.4.

DBB_POST_LOAD

You can execute the `DBB_POST_LOAD` exit point to provide a custom routine that performs calculations and transformations on the Data Backbone before a Master Template or Quick Template is produced.

Note When producing a Document Pack, you can only execute this exit point on the Data Preparation Template.

```
DBB_POST_LOAD PROC transform_data
DO
  ...
OD
```

The `DBB_POST_LOAD` exit point has no parameter and can manipulate the Data Backbone through the `_data` global variable.

If the `DBB_POST_LOAD` exit point is not present, a Master Template or a Quick Template is started with the Data Backbone as initialized.

The following example performs a custom transformation. The `format_policy_number` function must be defined in the Code Library.

```
DBB_POST_LOAD
PROC transform
DO
  (* Format policy number from 1234567890 to 123.456.7890 *)
  ASSIGN _data.PolicyData.PolicyNumber := format_policy_number
 (_data.PolicyData.PolicyNumber)
OD
```

When the Master Template or the Quick Template is started, the following actions are performed:

- In case the `DBB_XMLInput(...)` parameter is used, if the `DBB_PRE_LOAD` exit point is present, this is executed with the XML data as parameter. Otherwise, `_data` is filled with the XML data.
- Otherwise, the Data Retrieval part of the Data Backbone is executed.
- If the `DBB_POST_LOAD` exit point is present, it is executed.
- The Master Template or the Quick Document is executed.

The `DBB_POST_LOAD` exit point is introduced in KCM version 4.4.

DBB_POST_EDIT

You can execute the `DBB_POST_EDIT` exit point to provide a custom routine that performs calculations and transformations on the Data Backbone content after the Data Backbone content is edited interactively as part of the handling of a Content Wizard.

```
DBB_POST_EDIT PROC transform_data (CONST TEXT wizard)
DO
  ...
OD
```

The `wizard` parameter of the `DBB_POST_EDIT` exit point contains the name of the Content Wizard that has been processed.

The `DBB_POST_EDIT` exit point can use the `_data` variable to manipulate the Data Backbone content.

The following example command performs a custom transformation. The `format_policy_number` function must be defined in a Code Library.

```
DBB_POST_EDIT
PROC transform (CONST TEXT wizard)
DO
  (* Format policy number from 1,234,567,890 to 123.456.7890 *)
  ASSIGN _data.PolicyData.PolicyNumber := format_policy_number
 (_data.PolicyData.PolicyNumber)
OD
```

The `DBB_POST_EDIT` exit point is introduced in KCM version 4.4.

DBB_PRE_PACK

You can execute the `DBB_PRE_PACK` exit point to provide a custom routine that performs calculations and transformations on the Data Backbone content before each document in a Document Pack Template is produced.

The `DBB_PRE_PACK` exit point is introduced in KCM version 5.0.

```
DBB_PRE_PACK PROC transform_data ()
DO
  ...
OD
```

The `DBB_PRE_PACK` exit point can use the `_data` variable to manipulate the Data Backbone content.

The following example command fills a Field with the current date next year.

```
DBB_PRE_PACK
PROC transform ()
DO
  (* Valid for one year -- do not use on the 29th of february! *)
  ASSIGN _data.Notice.ExpirationDate := date (today + 10000)
OD
```

The `DBB_PRE_PACK` exit point is introduced in KCM version 5.0.

Variables

Variables are named references to a value or set of values. The content of a variable can be manipulated and used in expressions.

Declare variables

Variables must be declared before they can be used. The declaration defines the type of the variable, which restricts the kind of content that can be assigned and an initial value.

A variable declaration has the following form.

```
<type> <name>

<type> <name> := <initial value>
```

The `<type>` specifies the type of values that can be assigned to the variable. For a description of available data types, see Types.

The `<name>` must meet the following rules for all variable types except `FIELDSET` and `DATASTRUCTURE`:

- The first character must be a lowercase letter.
- The following characters must be lowercase letters, digits, or underscores.
- Only letters from the Latin-1 character set are allowed.

For `FIELDSET` and `DATASTRUCTURE` variables, the `<name>` must meet the following rules:

- The first character must be an uppercase letter.
- The following characters must be letters, digits, or underscores. Both uppercase and lowercase letters are allowed.
- Only letters from the Latin-1 character set are allowed.

An example is provided here.

```
TEXT saluation
NUMBER next_year
BOOL single_customer
FIELDSET PartnerData
MAP NUMBER stock_portfolio
```

The variable can be initialized with a value as part of the declaration. This initial value must have the appropriate type for the variable. If the initial value is omitted, a default value is assigned:

`TEXT` variables are initialized with an empty value.

`NUMBER` variables are initialized with the value 0.0.

`BOOL` variables are initialized with the value FALSE.

`MAP` and `FIELDSET` variables are initialized without any values.

`DATASTRUCTURE` variables use the default values for their members as described earlier in this section. Members that are a `DATASTRUCTURE` themselves are undefined.

An example is provided here.

```
TEXT saluation := "Dear"
NUMBER next_year := today + 10000
BOOL preferred_customer := (_data.NumberOfOrders >= 10)
FIELDSET Partner := _data.Partner
MAP NUMBER stock_portfolio := customer_previous_portfolio
```

Variables have a scope that is limited to the code block in which they are declared. When this block ends the variable is not accessible anymore. Variable names must be unique within the block in which they are defined. It is allowed to define variables with the same name in different blocks; when blocks are nested a variable in the inner block hide the variable in the outer block.

The following keyword pairs define blocks in the Template scripting language:

- `BEGIN` until `END`
- `DO` until `OD`
- `REPEAT` until `UNTIL`
- `IF` until the next `ELIF`, `ELSE`, or `FI`
- `ELIF` until the next `ELIF`, `ELSE`, or `FI`
- `ELSE` until `FI`

An example is provided here.

```
NUMBER var
IF var > 0 THEN
 #
 First Inner scope: @(var)
 #
 ASSIGN var := 42 (* Declared outside the IF..FI *)

 TEXT var := "Hello World" (* Hides the NUMBER var from here. *)

 #
 Second Inner scope: @(var)
 #
FI (* End of scope *)
#
Outer scope: @(var)
#
```

This example will produce the following result.

```
First Inner scope: 0.00
Second Inner scope: Hello World
Outer scope: 42.00
```

Assign variables

The `ASSIGN` statement assigns a new value to a variable. The new value must have the appropriate type for the type of the variable.

```
 ASSIGN variable := expression
```

An example is provided here.

```
ASSIGN saluation := "Dear"
ASSIGN next_year := today + 10000
```

```
ASSIGN preferred_customer := (_data.NumberOfOrders >= 10)
ASSIGN Partner := _data.Partner
```

Assign word processor instructions

Variables and members with the `TEXT` type can also contain Microsoft Word instructions in their content. In a Master Template, the fixed string notation "…" includes all layout and word processor instructions (tables, paragraph breaks, tabs) with their properties in the value. These properties include positioning and styling information.

Note During the Document Pack composition, any word processor instructions assigned to the Data Backbone of a Data Preparation Template are removed once the Data Preparation Template is executed.

An example is provided here.

```
TEXT break := "¶"
#
First line@(break)Next line
#
```

This example produces the following result.

```
First line¶
Next line
```

The `First line¶` paragraph will be produced in the paragraph style and layout from the `TEXT break := "¶` paragraph in the Master Template.

It is not possible to assign word processor instructions to variables using the fixed string notation "…" in Code Libraries. Code Libraries ignore all word processor instructions in their content.

Statements within Code Libraries can manipulate variables containing word processor instructions.

Operators

You can use operators to manipulate values in an expression. The following operators are available in the Template scripting language.

Operator	Operand	Result	Operation	Example
(...)	(...)	...	Grouping	((i + 1) * 3)
+	+NUMBER	NUMBER	Positive (monadic)	+4
-	-NUMBER	NUMBER	Negative (monadic)	-4
+	NUMBER + NUMBER	NUMBER	Addition	i + 1
	TEXT+TEXT	TEXT	Concatenation	line + "."
-	NUMBER - NUMBER	NUMBER	Subtraction	amount - fee
*	NUMBER * NUMBER	NUMBER	Multiplication	years * 365
/	NUMBER / NUMBER	NUMBER	Division	bottles / 12

Operator	Operand	Result	Operation	Example
%	NUMBER % NUMBER	NUMBER	Percentage	price % 21
<	NUMBER < NUMBER	BOOL	Less than	age < 21
	TEXT < TEXT	BOOL	Sorts before	class < "D"
<=	NUMBER <= NUMBER	BOOL	Less than or equal	age <= 21
	TEXT <= TEXT	BOOL	Equal or sorts before	class <= "D"
>	NUMBER > NUMBER	BOOL	Greater than	age > 21
	TEXT > TEXT	BOOL	Sorts after	class > "D"
>=	NUMBER >= NUMBER	BOOL	Greater than or equal	age >= 21
	TEXT >= TEXT	BOOL	Equal or sorts after	class >= "D"
=	NUMBER = NUMBER	BOOL	Equals	age = 21
	BOOL = BOOL	BOOL	Equals	paid = FALSE
	TEXT = TEXT	BOOL	Equals	class = "D"
<>	NUMBER <> NUMBER	BOOL	Not equal	age <> 18
	BOOL <> BOOL	BOOL	Not equal	paid <> shipped
	TEXT <> TEXT	BOOL	Not equal	class <> "D"
NOT	NOT BOOL	BOOL	Logical negation	NOT paid
AND	BOOL AND BOOL	BOOL	Logical AND	trained AND licensed
OR	BOOL OR BOOL	BOOL	Logical OR	shipped OR cancelled

Operators in an expression are evaluated based on their priority from top to bottom:

1. ( )
2. Monadic +, monadic -, NOT
3. *, /, %
4. +, -
5. <=, <, >, >=
6. =, <>
7. AND
8. OR

Examples are provided here.

`1 + 2 * -3` equals to -5

`(1 + 2) * -3` equals to - 9

The operators +, -, *, /, and % can be combined with the `ASSIGN` statement to modify a variable:

```
ASSIGN <variable> <op>:= <expression>
```

This example is a shorthand for:

```
ASSIGN <variable> := <variable> <op> <expression>
```

Examples are provided here.

```
ASSIGN counter +:= 1
ASSIGN line +:= ", " + item
```

Note The `AND` operator always calculates both operands. The operator evaluates to TRUE if both operands are TRUE, and FALSE if either operand is FALSE.

The `OR` operator always calculates both operands. The operator evaluates to TRUE if either operand is TRUE, and FALSE if both operands are FALSE.

The EnhancedUnicodeSupport setting in KCM Core Administrator affects `TEXT` comparisons. Comparisons are based on the `compare_characters` function, which is Unicode aware and handles word processor layout intelligently. If this setting is disabled, a literal comparison is used that might give unexpected results. For more information on the function `compare_characters`, see compare_characters.

Automatic conversions

The assignment statement offers several implicit conversions.

Converting `FIELDSET`s to `MAP`s

`FIELDSET` variables and `MAP TEXT` variables can be assigned to each other. All keys in the `MAP TEXT` that match the rules for Field Set Field names are accessible in the `FIELDSET` variable. Keys that are not valid names are included into the `FIELDSET`, but cannot be accessed directly in expressions.

An example is provided here.

```
MAP TEXT map
ASSIGN map["ValidField"] := "Data"
ASSIGN map["Invalid Field"] := "Cannot access"

FIELDSET Set
ASSIGN Set := map

#
Set: @(Set.ValidField)
#
MAP TEXT other_map
ASSIGN other_map := Set
#
Map: @(other_map["ValidField"])
Map: @(other_map["Invalid Field"])
#
```

This example produces the following result.

```
Set: Data
Map: Data
Map: Cannot access
```

Merging FIELDSETs

The +:= assignment can be used with FIELDSET variables to merge fields from a FIELDSET into another FIELDSET.

```
ASSIGN Dest +:= Override
```

This example adds all Fields from the Override Field Set into the Dest Field Set, leaving the fields in Dest that do not overlap with the Override Field Set intact.

Chapter 4

Formulas and operators

The @ statement

The statement `@(formula or variable)` places the value stored in the variable or the result of the formula in the result document. It can only be used within a text part of a Master Template.

Template scripts consist of KCM code and text. The switch between these modes is made with the hash sign. To put the value of variables or retrieved data in the text, use `@(formula or variable).`

The statement `@(expression)` outputs the result of the expression into the result document. It can only be used within the text parts of a Master Template.

The output depends on the type of the expression:

- A `TEXT` value is produced word for word.
- A `NUMBER` value is formatted as a number with thousand-separators and 2 decimal positions.
- A `BOOL` value produces the text `TRUE` or `FALSE`, localized to the active output language.

Other types of expressions cannot be used directly with the `@(...)` statement and causes an error during Template compilation.

An example is provided here.

```
TEXT text := "The Quick Brown Fox..."
NUMBER number := 1234567
BOOL bool := (0 = 1)
#
Text value: @(text).
Number value: @(number) or @(numerals(number)).
Boolean value: @(bool).
#
```

This example produces:

Text value: The Quick Brown Fox....

Number value: 1,234,567.00 or 1234567.

Boolean value: FALSE.

Monadic operators

The following monadic operators are permitted.

Operator	Description
+	The formula behind the + operator has to be of type NUMBER; the result obtained is of type NUMBER.
-	The formula behind the - operator has to be of type NUMBER; the result obtained is of type NUMBER.
NOT	The formula behind the NOT operator has to be of type BOOL; the result obtained is of type BOOL.

The monadic operators have the following priorities, where 1 is the highest priority and 6 the lowest:

1. %, *, and /
2. + and -
3. <=, <, >=, and >
4. = and <>
5. AND
6. OR

Monadic operators have a higher priority than dyadic operators.

Dyadic operators

Formulas in the Template scripting language are calculated from left to right in case of operators with equal priority.

The Template scripting language knows the following dyadic operators.

Operato	Associativity	Operand Types	Result Type	Operation Performed
+	Left	NUMBER	NUMBER	Addition
+	Left	TEXT	TEXT	Concatenation of operand texts
-	Left	NUMBER	NUMBER	Subtraction
*	Left	NUMBER	NUMBER	Multiplication
/	Left	NUMBER	NUMBER	Division
%	Left	NUMBER	NUMBER	Percentage
<	Left	NUMBER; TEXT	NUMBER	Less than
>	Left	NUMBER; TEXT (*)	BOOL	Greater than
<=	Left	NUMBER; TEXT (*)	BOOL	Less than or equal
>=	Left	NUMBER; TEXT (*)	BOOL	Greater than or equal
=	Left	Left and right the same type; NUMBER or BOOL	BOOL	Equal to

Operato	Associativity	Operand Types	Result Type	Operation Performed
<>	Left	Left and right the same type; `NUMBER` or `BOOL`	`BOOL`	Not equal to
AND	Left	`BOOL`	`BOOL`	Left `AND` right operand must be `TRUE` for the result to be `TRUE`.
OR	Left	`BOOL`	`BOOL`	Left `OR` right or both operands must be `TRUE` for the result to be `TRUE`

Comparing two text values with dyadic operators could fail if you try to compare two text values that use a different encoding system. You should use the function `compare_characters` to avoid errors (for more information, see compare_characters).

Chapter 5

Functions

This chapter describes the built-in functions that are available in the Template scripting language.

All examples in this chapter use the ENG (English) statements and function names. Output is based on the ENG (UK English) output language settings, unless indicated otherwise.

Text functions

fragment_of_characters

Use the `fragment_of_characters` function to extract a fragment from a larger text.

The function is capable of handling Unicode characters and word processor instructions.

```
fragment_of_characters ( input; offset; length )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `input`, type TEXT. The text from which the fragment is taken.
2. `offset`, type NUMBER. The position of the first character of the fragment.
3. `length`, type NUMBER. The number of characters in the fragment.

`offset` starts at 1 for the first character in input. A negative or 0 value is interpreted as the beginning of the value.

A negative or 0 value for `length` results in an empty result. If the requested fragment exceeds the number of available characters in input, the result only contains the available characters from input.

The input is always converted to Unicode Normalization Form C (NFC). This can cause merge or reordering of characters in the text.

Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical object, is stripped. The stripped content is not counted when calculating the offset and length and not included in the resulting fragment.

The resulting output is also in Unicode Normalization Form C (NFC).

Examples

```
Examples:
fragment_of_characters ("abcdefgh";  2;  4)  results in "bcde"
```

```
fragment_of_characters ("abcdefgh"; 20;  4) results in ""
fragment_of_characters ("abcdefgh"; -2;  4) results in "abcd"
fragment_of_characters ("abcdefgh";  2; -4) results in ""
fragment_of_characters ("abcdefgh";  2; 20) results in "bcdefgh"
```

The `fragment_of_characters` function supercedes the `text_fragment` function.

number_of_characters

Use the `number_of_characters` function to determine the number of characters in a text.

The function is capable of handling Unicode characters and word processor instructions.

```
number_of_characters ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `input`, type TEXT. The text with length to be determined.

The input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is not counted when calculating the length.

Examples

```
number_of_characters ("abcdefgh")
```

Results in `8`

```
number_of_characters ("")
```

Results in `0`

The `number_of_characters` function supercedes the `length` function.

compare_characters

Use the `compare_characters` function to compare two texts.

The function is capable of handling Unicode characters and word processor instructions.

```
compare_characters ( first; second )
```

The result of this function is of type NUMBER.

This function has two parameters:

1. `first`, type TEXT. The first text to compare.
2. `second`, type TEXT. The second text to compare.

All input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is ignored in the comparison.

The comparison is case-sensitive. Characters are compared based on their Unicode character codes, which is for most locales not a correct lexicographical sort.

Paragraph breaks and line breaks are considered equal for comparison purposes.

The return value indicates how the parameters compare:

- 0 both first and second are identical
- < 0 first compares textually before second
- > 0 first compares textually after second

Examples

```
compare_characters ("abc" ; "abc")
```

Results in 0

```
compare_characters ("abc" ; "abcd")
```

Results in a value > 0

```
compare_characters ("Abc" ; "abc")
```

Results in a value < 0 ("A" < "a")

```
compare_characters ("10"; "9")
```

Results in a value > 0 ("1" < "9")

The `compare_characters` function is used to implement the text comparison operators (<, <=, =, <>, >=, >) when Enhanced Unicode Support is enabled.

lowercase_of_characters

Use the `lowercase_of_characters` function to convert a fragment from a larger text to lower case.

The function is capable of handling Unicode characters and word processor instructions.

```
lowercase_of_characters ( input; offset; length )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `input`, type TEXT. The text from which the fragment is taken.
2. `offset`, type NUMBER. The position of the first character to be converted.
3. `length`, type NUMBER. The number of characters to be converted.

`offset` starts at 1 for the first character in input. A negative or 0 value is interpreted as the beginning of the value.

A negative or 0 value for `length` has no effect.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, are stripped. Stripped content is not counted when calculating the offset and length and not included in the resulting text.

Example

```
 lowercase_of_characters ("THIS IS AN example";
6; 2)
```

Results in `"THIS is AN example"`

The `lowercase_of_characters` function supercedes the `lowercase2` function.

uppercase_of_characters

Use the `uppercase_of_characters` function to convert a fragment from a larger text to uppercase.

The function is capable of handling Unicode characters and word processor instructions.

```
 uppercase_of_characters ( input; offset; length )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `input`, type TEXT. The text from which the fragment is taken.
2. `offset`, type NUMBER. The position of the first character to be converted.
3. `length`, type NUMBER. The number of characters to be converted.

`offset` starts at 1 for the first character in input. A negative or 0 value is interpreted as the beginning of the value.

A negative or 0 value for `length` has no effect.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped. Stripped content is not counted when calculating the offset and length and not included in the resulting text.

Example

```
 uppercase_of_characters ("This is an EXAMPLE"; 6; 2)
```

Results in `"This IS an EXAMPLE"`

The `uppercase_of_characters` function supercedes the `uppercase2` function.

trim

Use the `trim` function to remove leading and trailing whitespace.

```
 trim ( text )
```

This function returns a value of type TEXT.

The function has one parameter:

- `text`, type TEXT. The text that is to be trimmed.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped and not included in the result.

All tabs, spaces, paragraph breaks, and line breaks are removed from the beginning and end of `text`.

Example

```
trim (" example ")
```

Results in `"example"`

ltrim

Use the `ltrim` function to remove leading whitespace.

```
ltrim ( text )
```

This function returns a value of type TEXT.

The function has one parameter:

- `text`, type TEXT. The text that is to be trimmed.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped and not included in the result.

All tabs, spaces, paragraph breaks, and line breaks are removed from the beginning of text.

Example

```
ltrim (" example ")
```

Results in `"example "`

rtrim

Use the `rtrim ( text )` function to remove trailing whitespace.

```
rtrim ( text )
```

This function returns a value of type TEXT.

The function has one parameter:

- `text`, type TEXT. The text that is to be trimmed.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped and not included in the result.

All tabs, spaces, paragraph breaks, and line breaks are removed from the end of `text`.

Example

```
rtrim (" example ")
```

Results in `" example"`

search

Use this function to search a text for the first occurrence of a fragment. The search can be performed either case-sensitive or case-insensitive.

```
search ( haystack; needle; start; sensitive )
```

This function returns a value of type NUMBER.

The function has four parameters:

1. `haystack`, type TEXT. The text that the search is performed on.
2. `needle`, type TEXT. The fragment that is searched for.
3. `start`, type NUMBER. The offset in the haystack text where the search should start. Use a value of 1 to start at the first character of `haystack`.
4. `sensitive`, type BOOL. When TRUE the search is case-sensitive, otherwise case-insensitive.

The result of the search function is the offset of the first occurrence of the `needle` text fragment in the `haystack` text at or past the `start` offset. If the needle fragment could not be found, the function returns 0.

If the `sensitive` parameter is FALSE, a case-insensitive is performed by converting both needle and haystack to lowercase before searching. Applications should account for this conversion in case characters do not have a stable conversion (for example, straße and STRASSE).

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped before the comparison. Paragraph breaks, and line breaks are considered equal for comparison purposes.

Example

```
TEXT haystack := "The Quick Brown Fox Jumps Over The Lazy Dog"
TEXT needle := "o"
```

`search (haystack; needle; 1; TRUE )` returns `13` (Br**o**wn)

`search (haystack; needle; 20; TRUE )` returns `42` (D**o**g)

`search (haystack; needle; 20; FALSE)` returns `27` (**O**ver)

`search (needle; haystack; 1; TRUE )` returns `0`

search_first

Use the `search_first` function to search a text for the first occurrence of a fragment. The search can be performed either case-sensitive or case-insensitive.

```
search_first ( haystack; needle; casing )
```

This function returns a value of type NUMBER.

The function has three parameters:

1. `haystack`, type TEXT. The text that the search is performed on.
2. `needle`, type TEXT. The fragment that is searched for.
3. `casing`, type BOOL. When TRUE the search is case-sensitive; otherwise, case-insensitive.

The result of the search function is the offset of the first occurrence of the `needle` text fragment in the haystack text. If the `needle` fragment could not be found, the function returns 0.

If the casing parameter is FALSE, a case-insensitive is performed by converting both `needle` and `haystack` to lowercase before searching. Applications should account for this conversion in case characters do not have a stable conversion (for example, straße and STRASSE).

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped before the comparison. Paragraph breaks, and line breaks are considered equal for comparison purposes.

Example

```
TEXT haystack := "The Quick Brown Fox Jumps Over The Lazy Dog"
TEXT needle := "o"
```

`search_first (haystack; needle; TRUE )` returns `13` (Br'o'wn)

`search_first (haystack; needle; FALSE)` returns `27` ('O'ver)

`search_first (needle; haystack; TRUE )` returns `0`

search_last

Use the `search_last` function to search a text for the final occurrence of a fragment. The search can be performed either case-sensitive or case-insensitive.

```
search_last ( haystack; needle; casing )
```

This function returns a value of type NUMBER.

The function has three parameters:

1. `haystack`, type TEXT. The text that the search is performed on.
2. `needle`, type TEXT. The fragment that is searched for.
3. `casing`, type BOOL. When TRUE the search is case-sensitive; otherwise, case-insensitive.

The result of the search function is the offset of the final occurrence of the `needle` text fragment in the `haystack` text. If the `needle` fragment could not be found, the function returns 0.

If the `casing` parameter is FALSE, both `needle` and `haystack` are converted to lowercase before searching. Applications should account for this conversion in case characters do not have a stable conversion (for example, straße and STRASSE).

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical

objects, is stripped before the comparison. Paragraph breaks, and line breaks are considered equal for comparison purposes.

Example

```
 TEXT haystack := "The Quick Brown Fox Jumps Over The Lazy Dog"
TEXT needle := "o"
```

`search_last (haystack; needle; TRUE )` returns `42` (D'o'g)

`search_last (haystack; needle; FALSE)` returns `27` ('O'ver)

`search_last (needle; haystack; TRUE )` returns `0`

replace

Use the `replace` function to replace all occurrences of a fragment in a text with alternative text. The search is performed case-sensitive.

```
 replace ( haystack; needle; replacement )
```

This function returns a value of type NUMBER.

The function has three parameters:

1. `haystack`, type TEXT. The text that the search is performed on.
2. `needle`, type TEXT. The fragment that is searched for.
3. `replacement`, type TEXT. The alternative text that is used to replace the `needle` text.

The result of the search function is the input with all occurrences of `needle` replaced by `replacement`. Overlapping results are not considered for replacement.

Input is always converted to Unicode Normalization Form C (NFC). Tabs, paragraph breaks, and line breaks are handled as characters. Other word processor content, such as layout switches and graphical objects, is stripped before the comparison. Paragraph breaks, and line breaks are considered equal for comparison purposes.

Example

`replace ("The Quick Brown Fox"; "o"; "O")` returns `"The Quick BrOwn FOx"`

`replace ("The Quick Brown Fox"; "O"; "o")` returns `"The Quick Brown Fox"`

`replace ("XXX"; "XX"; "aX")` returns `"aXX" (not "aaX" )`

lowercase

Use the `lowercase` function to convert a number into the corresponding lowercase character from the alphabet.

```
lowercase ( index )
```

The result of this function is of type TEXT.

This function has one parameter:

- `index`, type NUMBER. The index of the character.

The `lowercase` function returns the indicated character from the alphabet, depending on the active output language.

If a negative value is provided for `index`, the last character of the alphabet is returned. If `index` exceeds the number of characters in the alphabet, the number is wrapped around.

Examples

`lowercase (1)` results in `"a"`

`lowercase (4)` results in `"d"`

`lowercase (-1)` results in `"z"`

`lowercase (27)` results in `"a"`

`lowercase (27)` results in `"æ"` (output language DAN)

uppercase

Use the `uppercase` function to convert a number into the corresponding uppercase character from the alphabet.

```
uppercase ( index )
```

The result of this function is of type TEXT.

This function has one parameter:

- `index`, type NUMBER. The index of the character.

The uppercase function returns the indicated character from the alphabet, depending on the active output language.

If a negative value is provided for `index`, the last character of the alphabet is returned. If `index` exceeds the number of characters in the alphabet, the number is wrapped around.

Examples

`uppercase (1)` results in `"A"`

`uppercase (4)` results in `"D"`

`uppercase (-1)` results in `"Z"`

`uppercase (27)` results in `"A"`

`uppercase (27)` results in `"Æ"` (output language DAN)

text_to_number

Use the `text_to_number` function to convert a text to a number.

```
text_to_number ( text )
```

This function returns a value of type NUMBER.

The function has one parameter:

- `text`, type TEXT. The text to be converted to a number.

The interpretation of the number (decimal point, thousands separator) depends on the currently active output language. Conversion stops are the first character that is not a digit or separator.

Examples

`text_to_number ("123")` results in `123`

`text_to_number ("123.000")` results in `123`

`text_to_number ("123.000")` results in `123000` (output language NLD)

`text_to_number ("12a30")` results in `12`

text_fragment

The `text_fragmentfunction` has been superseded by the `fragment_of_charactersfunction`. Unless Enhanced Unicode Support is disabled, all calls to `text_fragment` are automatically mapped to the `fragment_of_characters` function.

Use the `text_fragmentfunction` function to extract a fragment from a larger text.

```
text_fragment ( input; offset; length )
```

The result of this function is of type TEXT.

This function has three parameters:

1. `input`, type TEXT. The text from which the fragment is taken.
2. `offset`, type NUMBER. The position of the first character of the fragment.
3. `length`, type NUMBER. The number of characters in the fragment.

`offset` starts at 1 for the first character in input. A negative or 0 value is interpreted as the beginning of the value.

A negative or 0 value for `length` results in an empty result.

If the requested fragment exceeds the number of available characters in the input, the result only contains the available characters from input.

Examples

`text_fragment ("abcdefgh"; 2; 4)` results in `"bcde"`

`text_fragment ("abcdefgh"; 20; 4)` results in `""`

`text_fragment ("abcdefgh"; -2; 4)` results in `"abcd"`

`text_fragment ("abcdefgh"; 2; -4)` results in `""`

`text_fragment ("abcdefgh"; 2; 20)` results in `"bcdefgh"`

Note The `text_fragment` function is limited to content that only consists of latin-1 text. This function does not support word processor instructions or Unicode content. Use of this function with non latin-1 content can result in run-time errors or broken result documents.

length

The `length` function has been superseded by the `number_of_characters` function. Unless Enhanced Unicode Support is disabled, all calls to length are automatically mapped to the `number_of_characters` function.

Use the `length` function to determine the number of characters in a text.

```
length ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `input`, type TEXT. The text with length to be determined.

Examples

`length ("abcdefgh")` results in `8`

`length ("")` results in `0`

Note The `length` function is limited to content that only consists of latin-1 text. This function does not support word processor instructions or Unicode content. Use of this function with non latin-1 content can result in run-time errors or broken result documents.

lowercase2

The `lowercase2` function has been superseded by the `lowercase_of_characters` function. Unless Enhanced Unicode Support is disabled, all calls to `lowercase2` are automatically mapped to the `lowercase_of_characters` function.

Use the `lowercase2` function to convert a text fragment to lowercase.

```
lowercase2 ( input; start; end )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `input`, type TEXT. The text which is converted.
2. `start`, type NUMBER. The position of the first character to be changed.
3. `end`, type NUMBER. The position of the last character to be changed.

`start` and `end` start at 1 for the first character in `input`. A negative or 0 value is interpreted as the beginning of the value. If `end` is smaller than `start` the value is returned unmodified.

Examples

`lowercase2 ("THIS IS AN example"; 6; 7)` results in `"THIS is AN example"`

`lowercase2 ("THIS IS AN example"; 7; 6)` results in `"THIS IS AN example"`

Note The `lowercase2` function is limited to content that only consists of latin-1 text. This function does not support word processor instructions or Unicode content. Use of this function with non latin-1 content can result in run-time errors or broken result documents.

uppercase2

The `uppercase2` function has been superseded by the `uppercase_of_characters` function. Unless Enhanced Unicode Support is disabled, all calls to `uppercase2` are automatically mapped to `uppercase_of_characters` function.

Use the `uppercase2` function to convert a fragment of a text to uppercase.

`uppercase_of_characters` function.

```
uppercases2 ( input; start; end )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `input`, type TEXT. The text which is converted.
2. `start`, type NUMBER. The position of the first character to be changed.
3. `end`, type NUMBER. The position of the last character to be changed.

`start` and `end` start at 1 for the first character in input. A negative or 0 value is interpreted as the beginning of the value. If `end` is smaller than `start`, the value is returned unmodified.

Examples

`uppercases2 ("This is an EXAMPLE"; 6; 7)` results in `"This IS an EXAMPLE"`

`uppercases2 ("This is an EXAMPLE"; 7; 6)` results in `"This is an EXAMPLE"`

Note The `uppercases2` function is limited to content that only consists of latin-1 text. This function does not support word processor instructions or Unicode content. Use of this function with non latin-1 content can result in run-time errors or broken result documents.

uppercases

Use this function to convert a text to uppercase.

Note The `uppercases` function has been superseded by the `uppercase_of_characters` function. Unless Enhanced Unicode Support is disabled, all calls to `uppercases` are automatically mapped to the `uppercase_of_characters` function.

```
uppercases ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type TEXT. The text which is converted.

Examples

```
uppercases ("This is an EXAMPLE") results in "THIS IS AN EXAMPLE"
```

Note The `uppercases` function is limited to content that only consists of latin-1 text. This function does not support word processor instructions or Unicode content. Use of this function with non latin-1 content can result in run-time errors or broken result documents.

Date and time functions

date

Use the `date` function to convert a number into its date notation.

The date produced by this function is localized based on the active output language.

```
date ( number )
```

This function returns a value of type TEXT.

The function has one parameter:

- `number`, type NUMBER. The date to be converted. Fractions are ignored.

`number` is interpreted as a calendar date in ISO 8601 YYYYMMDD format.

For compatibility with legacy data sources, years between 0 and 99 are interpreted as years in the 1900s (1900-1999). Years between 100 and 199 are interpreted as years in the 2000s (2000-2099). The text `"????"` is returned for invalid dates.

Examples

`date (20150302)` results in `"2 March 2015"`

`date (20150302)` results in `"March 2, 2015"` (output language ENU)

`date (20150302)` results in `"le 2 mars 2015"` (output language FRA)

`date (150302)` results in `"2 March 1915"`

`date (1150302)` results in `"2 March 2015"`

`date (0)` results in `"????"`

`date (20150229)` results in `"????"`

The date is formatted in the default language if an output language is not specified.

date_in_words

Use the `date_in_words` function to convert a number into its date notation.

The date produced by this function is localized based on the active output language.

```
date_in_words ( number )
```

This function returns a value of type TEXT.

The function has one parameter:

- `number`, type NUMBER. The date to be converted. Fractions are ignored.

`number` is interpreted as a calendar date in ISO 8601 YYYYMMDD format.

For compatibility with legacy data sources, years between 0 and 99 are interpreted as years in the 1900s (1900-1999). Years between 100 and 199 are interpreted as years in the 2000s (2000-2099). The text "`????`" is returned for invalid dates.

Examples

`date_in_words (20150302)` results in `"the second of March two thousand fifteen"`

`date_in_words (20150302)` results in `"March second, two thousand fifteen"` (output language ENU)

`date_in_words (20150302)` results in `"le deux mars deux mille quinze"` (output language FRA)

`date_in_words (150302)` results in `"the second of March nineteen hundred fifteen"`

`date_in_words (1150302)` results in `"the second of March two thousand fifteen"`

`date_in_words (0)` results in `"????"`

`date_in_words (20150229)` results in `"????"`

The date is formatted in the default language if an output language is not specified.

today

Use the `today` function to retrieve the system date.

```
today
```

This function returns a value of type NUMBER.

The function has no parameters.

The value returned by the today function is the local date as set on the server. The result is in ISO 8601 YYYYMMDD format.

Example

`today` results in `20150302` (when called on March 2, 2015)

now

Use the function to retrieve the system time.

```
now
```

This function returns a value of type NUMBER.

The function has no parameters.

The value returned by the `now` function is the local time as set on the server. The result is in ISO 8601 HHMMSS format and uses the 24-hour clock system.

Example

`now` results in 165213 (when called at 16:52:13 / 4:52:13pm)

Depending on configuration, the result of the `now` function could be cached, returning the same result for every call. This function cannot be used for timing or benchmarking purposes.

number_to_date

Use the `number_to_date` function to convert a numerical date value into the ISO 8601 YYYYMMDD format based on a custom template.

```
number_to_date ( date; template )
```

This function returns a value of type NUMBER.

The function has two parameters:

1. `date`, type NUMBER. The date to be converted. Fractions are ignored.
2. `template`, type TEXT. A template describing how to interpret date.

`template` is a mask describing how the digits in date must be interpreted. It must contain the following three components in the correct order:

1. `DD` for the day part
2. `MM` for the month part
3. `YY`, `YYY`, or `YYYY` for the year part

- `YY` represents a year in the 2000s (2000-2099).
- `YYY` represents a year in the 1900s (1900-1999) for values between 0 and 99, or a year in the 2000s (2000-2099) for values between 100 and 199.
- `YYYY` represents the year without further interpretation.

`date` is considered to be left-padded with `0`'s if the template requires more digits than the value of date provides. If a component is omitted from the template, the corresponding part in the result is set to 0. The result of the number_to_date function for invalid templates is undefined.

Examples

`number_to_date (20150203; "YYYYDDMM")` results in `20150302`

`number_to_date (150302; "YYMMDD")` results in `20150302`

`number_to_date (150302; "YYYMMDD")` results in `19150302`

`number_to_date (201503; "YYYYMM")` results in `20150300`

`number_to_date (2015; "DDMMYYYY")` results in `20150000`

format_date

Use the `format_date` function to format a date according to the specified mask and language code.

```
format_date ( date; template; language )
```

This function returns a value of type TEXT.

The function has three parameters:

1. `date`, type NUMBER. The date to be formatted.
2. `template`, type TEXT. The template specifying how the date should be formatted.
3. `language`, type TEXT. Optional. The language used to format textual components in the template.

The template must be composed of a sequence of format types. The format types are case-sensitive.

The following format types are supported:

Format Type	Description	Example
d	Day of the month	2
dd	Day of the month, padded to 2 positions	02
ddd	Abbreviated day of the week	Mon
dddd	Day of the week	Monday
M	Month	3
MM	Month, padded to two positions	03
MMM	Abbreviated month	Mar
MMMM	Month	March
y	Last two digits of the year, omitting a leading 0	15
yy	Last two digits of the year	15
yyyy	The year	2015
g	Period or era	A.D.
gg	Period or era	A.D.
*SHORT	Windows short date format	02/03/2015
*LONG	Windows long date format	02 March 2015

Examples are based on the date 20150302 and the ENG output language.

Spaces in the template appear in the same location in the output. A template can contain text that is not matching any of the format types. This text is mostly passed unchanged. To ensure that text is passed unchanged you should enclose it in single quotation marks. A single quotation mark as fixed text must be written as a sequence of two single quotation marks in a row.

The format types `*SHORT` and `*LONG` cannot be combined with other format types or layout.

The `language` parameter is optional. If it is omitted or empty, the current output language is used. The following values are support for this parameter:

- Any of the output language supported by KCM Core (see language_code)
- Any Language Culture Name installed on the Microsoft Windows Server hosting KCM Core

If the language is not supported or the date is invalid the function returns "????".

Examples

`format_date (20150302; "MM'//'dd'//'yyyy")` results in `"03/02/2015"`

`format_date (20150302; "dd'-'MM'-'''yy")` results in `"02-03-'15"`

`format_date (20150302; "dddd")` results in `"Monday"`

`format_date (20150302; "*LONG"; language:="SVE")` results in `"den 3 februari 2015"`

`format_date (20150302; "*LONG"; "nl-NL")` results in `"dinsdag 3 februari 2015"`

`format_date (20150229; "yyyy")` results in `"????"` (invalid date)

`format_date (20150302; "yyyy"; "tlh")` results in `"????"` (invalid language)

The earliest date supported by this function is Jan 1, 1601 (16010101).

The implementation of the `format_date` function depends on the Microsoft Windows NLS support. The output can depend on the version of Microsoft Windows and installed components. For a detailed description of the supported templates, see the Microsoft Windows documentation on Day, Month, Year, and Era Format Pictures available on the Internet.

The `format_date` function was introduced in KCM Core 4.4.

Number function

numerals

Use the `numerals` function to convert a number into an unformatted textual representation. The resulting value is rounded to the nearest integral value.

```
numerals ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The number to convert.

Examples

`numerals (1234567)` results in `"1234567"`

`numerals (1234567.8)` results in `"1234568"`

`numerals (100 / 3)` results in `"33"`

number

Use the `number` function to convert a number into a formatted textual representation. The number is formatted according to the current output language.

This function returns a value of type TEXT.

The function has two parameters:

1. `input`, type NUMBER. The number to convert.
2. `precision`, type NUMBER. The number of decimal positions to display.

If input has a fractional part, it is rounded to the nearest value with the specified precision. The precision is limited to 9 digits.

Examples

`number (1234567; 0)` results in `"1,234,567"`

`number (1234567; 2)` results in `"1,234,567.00"`

`number (1234567; 2)` results in `"1.234.567,00" (output language "NLD")`

`number (123.435; 4)` results in `"123.4350"`

`number (123.435; 2)` results in `"123.44"`

`number (123.435; 0)` results in `"123"`

number_in_words

Use the `number_in_words` function to convert a number into a notation representing an integral number, written out fully in words. The output is localized based on the current output language.

```
number_in_words ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

1. `input`, type NUMBER. The number to be converted. This function supports numbers for all output languages up to 1000 billion (exclusive). Results for values outside this range are undefined.

The input value is rounded to an integer number.

Examples

`number_in_words (12)` results in `"twelve"`

`number _in_words (-12)` results in `"minus twelve"`

`number _in_words (1234567.89)` results in `"one million two hundred and thirty-four thousand five hundred and sixty-eight"`

`number_in_words (1234567.89)` results in `"een miljoen tweehonderdvierendertigduizend vijfhonderdachtenzestig"` (output language NLD)

`number_in_words (12.4)` results in `"twelve"`

`number_in_words (12.5)` results in `"thirteen"`

`number_in_words (0)` results in `"zero"`

picture

This function is deprecated and only supported for legacy templates. Templates should use the format function instead (see format).

The `picture` function converts a number into a formatted textual representation according to the specified mask.

```
picture ( input; mask )
```

format

Use the `format` function to convert a number into a formatted textual representation according to the specified mask.

```
format ( input; mask )
```

This function returns a value of type TEXT.

The function has two parameters:

1. `input`, type NUMBER. The number to convert.
2. `mask`, type TEXT. The formatting mask.

If the mask includes fractional digits, input is rounded to the nearest value with the specified precision. The mask must be sized to hold all integral digits.

Examples are provided here.

The `'ø'` character has been used in the example to represent space characters. This character is not generated by the `format` function.

`format (123.456; "Z,ZZZ.ZZ+")` results in `"øø123.46+"`

`format (-12345.67; "$B*,***,***.**BBDB")` results in `"£ø***12,345.67øøDB"`

These symbols are explained in the next section.

The mask is language independent.

Output is formatted based on the current output language. The output has exactly the same number of characters as the mask (exception: "U").

Symbols allowed in the mask

The following table lists and describes symbols that are allowed in `mask`.

Symbol	Meaning
9	A numeric position
*	A leading numeric position in which an asterisk is placed if this position contains a 0.
Z	A leading numeric position in which a space is placed if this position contains a 0.
.	The decimal point. Output is aligned on this position. This symbol is replaced by the appropriate symbol for the output language.
,	A thousand-separator. This symbol is replaced by the appropriate symbol for the output language.
$	A single-character currency symbol. This symbol is replaced by the appropriate symbol for the output language.
U	A multi-character currency symbol. This symbol is replaced by the appropriate 3-character ISO 4217 code for the output language.
+	Editing sign control
-	Editing sign control
CR	Editing sign control. The CR symbol is only permitted at the end of the mask
DB	Editing sign control. The DB symbol is only permitted at the end of the mask.
B	A space
/	A slash. Note that the / must be escaped when written in a "…" expression.

All symbols are case-insensitive.

Repeating symbols can be abbreviated by specifying a repeat-count between parentheses.

The following two patterns are equivalent:

```
"$9(5).9(2)CR"
```

```
"$99999.99CR"
```

Editing sign controls

The editing sign controls produce a result based on the sign of the number. The following table describes the editing signs.

Value	**+**	**-**	**CR**	**DB**
Positive	+	space	2 spaces	CR
0	+	space	2 spaces	CR
Negative	-	-	CR	DB

Currency symbols used by the format function

The currency symbols printed by the `format` function depend on the currently selected language or any custom overrides made in the Master Template.

The following table lists and describes currency symbols used by the `format` function.

Run-time Language	**Single-character Currency Code ($)**	**Multi-character Currency Code (U)**
Any language with euro support enabled.	€	EUR
DAN	:	DKK
DES	F	CHF
DEU	$	DEM
ENG	£	GBP
ENU	$	USD
ESP	$	ESP
FRA	F	FRF
FRB	F	BEF
FRS	F	CHF
HUN	F	HUF
ITS	F	CHF
NLB	F	BEF
NLD	*ƒ*	NLG
POL	Z	PLN
PTG	$	PTE
SVE	F	SEK

Note The information in the table is limited to those single-character currency codes that are supported by the standard ANSI code page. For those currencies not in the ANSI code page, the IBM default currencies are used. To learn how to override the currency code into a code appropriate to your combination of fonts and Microsoft Word, see the following sections.

The euro symbol requires the use of a Microsoft Windows single-byte font containing the euro symbol at position 128. When using non-Microsoft Windows fonts or Unicode fonts, the euro symbol may be printed incorrectly.

Note The U symbol in the mask expands to three positions. Unless the U symbol is used as a floating insertion symbol with the appropriate amount of extra space provided in the mask, the format function expands its result to create space for any extra overhead. This may cause misalignment in the result document.

Leading symbols

You can use the `format` function to insert a sign (+ or -) or currency symbol ($ or U) directly in front of the first digit or decimal point in the result.

To specify a leading symbol, the mask must start with at least two instances of the symbol. The symbol is inserted directly in front of the first digit or the decimal point and unused positions are padded with spaces. If there is insufficient space in the mask to place the expansion of the U symbol, the returned string is extended to fit the expansion.

If input is 0 and the mask contains the same symbol in all numerical positions, the result is a series of spaces with the length of the mask.

Examples

`format (0.123; "$$$$.99")` results in `"øøø£.12"`

`format (0.12; "$$$9.99")` results in `"øø£0.12"`

`format (-1234.56; "$,$$$,999.99")` results in `"øøø£1,234.56"`

`format (-1234.56; "U,UUU,UU9.99-")` results in `"øGBP1,234.56-"`

`format (1234.56; "U,UUU,UU9.99")` results in `"øGBP1,234.56"`

`format (-123456.78; "+,+++,999.99")` results in `"ø-123,456.78"`

`format (-1234567; "$$,$$$,$$$.99CR")` results in `"£1,234,567.00CR"`

`format (20150302; "9999//99//99")` results in `"2015/03/02"`

`format (0; "++,+++,+++.++")` results in `"øøøøøøøøøøøøø"`

Suppress leading zero

You can use the `format` function to blank (Z) or pad (*) leading zeros in the result.

To suppress leading zeros, the mask must contain a padding symbol in the first numerical position and any applicable subsequent positions. Any of these positions in the mask that would be filled with a leading zero are filled with the padding symbol instead.

If input is 0 and the mask contains only blank (Z) symbols, the result is a series of spaces with the length of the mask.

If input is 0 and the mask contains only asterisk (*) symbols, the result is a series of asterisks with the length of the mask. If the mask contains a decimal point, the decimal point is included in the result.

Examples

`format (123.456; "Z,ZZZ.ZZ+")` results in `"øø123.46+"`

`format (-123.45; "*,***.**+")` results in `"**123.45-"`

`format (12345678.9; "**,***,***.**")` results in `"12,345,678.90"`

`format (12345.67; "$Z,ZZZ,ZZZ.ZZCR")` results in `"£øøø12,345.67øø"`

`format (-12345.67; "$B*,***,***.**BBDB")` results in `"£ø***12,345.67øøDB"`

`format (0; "*,***.**")` results in `"*****.**"`

`format (0; "Z,ZZZ.ZZ")` results in `"øøøøøøøø"`

`format (0; "ZZZZ.99")` results in `"øøøø.00"`

`format (0; "*,***.99")` results in `"*****.00"`

`format (0; "ZZ99.99")` results in `"øø00.00"`

Language customization

The output of the `format` function is determined by a language independent mask. Some of the symbols expand, depending on the active output language, and can be changed by the Template.

The following symbols are based on the active output language.

Symbol	Representation
$	Single-character currency symbol
U	Multi-character currency symbol
.	Decimal point
,	Thousand separator

The template can change these expansions at any time:

- Through the `language_code` function (see language_code). The default output language is set in KCM Core Administrator. Changing the output language affects all four symbols.
- Through the `euro` function (see euro). This function affects the currency symbols ($ and U).
- Using the `pragma` function to override specific values (see pragma):

Pragma key	Symbol	Length	Representation

LOCALE:LC_MONETARY:int_curr_symbol	U	3	Multi-character currency symbol
LOCALE:LC_MONETARY:currency_symbol	$	1	Single-character currency symbol
LOCALE:LC_NUMERIC:decimal_point	.	1	Decimal point
LOCALE:LC_NUMERIC:thousands_sep	,	1	Thousand separator

The `pragma` function should be used to replace symbols with alternatives that have the appropriate length for the symbol. Using alternatives with incorrect lengths is not supported and can result in values that are not aligned correctly.

The built-in language support defines currency symbols for the supported languages. For legacy reasons, Euro support is by default disabled.

Output language	**$**	**U**
Euro support enabled	€	EUR
DAN	:	DKK
DES	F	CHF
DEU	$	DEM
ENG	£	GBP
ENU	$	USD
ESP	$	ESP
FRA	F	FRF
FRB	F	BEF
FRS	F	CHF
HUN	F	HUF
ITS	F	CHF
NLB	F	BEF
NLD	*ƒ*	NLG
POL	Z	PLN
PTG	Z	PTE
SVE	F	SEK

Examples

`format (0.12; "$$$9.99")` results in `"øø£0.12"`

`format (0.12; "$$$9.99")` results in `"øøF0,12"` (output language DES / de-CH)

`format (1234.56; "U,UUU,UU9.99")` results in `"øGBP1,234.56"`

`format (1234.56; "U,UUU,UU9.99")` results in `"øCHF1.234,56"` (output language DES / de-CH)

`pragma ("LOCAL:LC_MONETARY:int_curr_symbol"; "CNY")` changes the U symbol to `CNY`

`format (1234.56; "U,UUU,UU9.99")` results in `"øCNY1,234.56"`

round

Use the `round` function to round a number to the nearest value with the indicated precision using the commercial rounding rule.

```
round (input; precision)
```

This function returns a value of type NUMBER.

The function has two parameters:

1. `input`, type NUMBER. The number to round.
2. `precision`, type NUMBER. The number of significant decimals (up to 9 decimals is supported).

This function treats positive and negative values symmetrically.

Examples

`round (12.324; 0)` results in `12`

`round (12; 2)` results in `12`

`round (12.324; 2)` results in `12.32`

`round (12.325; 2)` results in `12.33`

`round (-12.324; 2)` results in `-12.32`

`round (-12.325; 2)` results in `-12.33`

truncate

Use the `truncate` function to adjust a number to the indicated precision by discarding any remaining fractional part. This function always adjusts numbers toward zero.

```
truncate ( input; precision )
```

This function returns a value of type NUMBER.

The function has two parameters:

1. `input`, type NUMBER. The number to round.
2. `precision`, type NUMBER. The number of significant decimals (up to 9 decimals is supported).

This function treats positive and negative values symmetrically.

Examples

`truncate (12.324; 0)` results in `12`

`truncate (12 ; 2)` results in `12`

`truncate (12.324; 2)` results in `12.32`

`truncate (12.325; 2)` results in `12.32`

`truncate (-12.324; 2)` results in `-12.32`

`truncate (-12.325; 2)` results in `-12.32`

round_upwards

Use the `round_upwards` function to adjust a number to the indicated precision. If a remaining fractional part would be discarded, the number is adjusted to the next value. This function always adjusts numbers away from zero.

```
round_upwards ( input; precision )
```

This function returns a value of type NUMBER.

The function has two parameters

1. `input`, type NUMBER. The number to round.
2. `precision`, type NUMBER. The number of significant decimals (up to 9 decimals is supported).

This function treats positive and negative values symmetrically.

Examples

`round_upwards (12.324; 0)` results in `13`

`round_upwards (12 ; 2)` results in `12`

`round_upwards (12.324; 2)` results in `12.33`

`round_upwards (12.325; 2)` results in `12.33`

`round_upwards (-12.324; 2)` results in `-12.33`

`round_upwards (-12.325; 2)` results in `-12.33`

uppercase_roman_number

Use the `uppercase_roman_number` function to convert a number into a roman numeral. The output is in uppercase characters.

```
uppercase_roman_number ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The number to be converted.

This function only supports values between 1 and 3999, inclusive. Fractional parts are ignored.

The output for values outside the supported range is undefined.

Examples

`uppercase_roman_number (3)` results in `"III"`

`uppercase_roman_number (2016)` results in `"MMXVI"`

lowercase_roman_number

Use the `lowercase_roman_number` function to convert a number into a roman numeral. The output is in lowercase characters.

```
lowercase_roman_number ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The number to be converted.

This function only supports values between 1 and 3999, inclusive. Fractional parts are ignored.

The output for values outside the supported range is undefined.

Examples

`lowercase_roman_number (3)` results in `"iii"`

`lowercase_roman_number (2016)` results in `"mmxvi"`

ordinal

Use the `ordinal` function to convert a number into the corresponding ordinal. The output is localized based on the current output language.

```
ordinal ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The number to be converted.

This function only supports values between 0 and 100, inclusive. Fractional parts are ignored.

The output for values outside the supported range is undefined.

Examples

`ordinal (1)` results in `"first"`

`ordinal (1)` results in `"eerste"` (output language NLD)

amount

Use the `amount` function to convert a number into a notation representing an amount of currency. The output is localized based on the current output language.

```
amount ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The amount to be converted.

The input value is rounded to cents.

Examples

`amount (0.12)` results in `"0.12"`

`amount (-0.12)` results in `"-0.12"`

`amount (1234567.89)` results in `"1,234,567.89"`

`amount (1234567.89)` results in `"1.234.567,89"` (output language NLD)

`amount (12.344)` results in `"12.34"`

`amount (12.345)` results in `"12.35"`

amount_in_words

Use the `amount_in_words` function to convert a number into a notation representing an amount of currency, written out fully in words. The output is localized based on the current output language.

```
amount_in_words ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The amount to be converted. This function supports amounts for all output languages up to 1000 billion (exclusive). Results for values outside this range are undefined.

The `input` value is rounded to cents.

Currencies are represented by the pre-Euro currencies. Use the `amount_in_words_euro` function to produce an amount in the Euro currency, or the `euro` function to switch the currency representation. See amount_in_words_euro and euro, respectively.

Examples

`amount_in_words (0.12)` results in `"zero pounds and twelve pence"`

`amount_in_words (-0.12)` results in `"zero pounds and twelve pence negative"`

`amount_in_words (1234567.89)` results in `"one million two hundred and thirty-four thousand five hundred and sixty-seven pounds and eighty-nine pence"`

`amount_in_words (1234567.89)` results in `"een miljoen tweehonderdvierendertigduizend vijfhonderdzevenenzestig gulden en negenentachtig cent"` (output language NLD)

`amount_in_words (12.344)` results in `"twelve pounds and thirty-four pence"`

`amount_in_words (12.345)` results in `"twelve pounds and thirty-five pence"`

amount_in_words_euro

Use the `amount_in_words_euro` function to convert a number into a notation representing an amount of currency, written out in words. The output is localized based on the current output language. The currency used is always the Euro.

```
amount_in_words_euro ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The number to be converted. This function supports amounts for all output languages up to 1000 billion (exclusive). Results for values outside this range are undefined.

The currency used to produce the amount is always the Euro, regardless of the output language. The input value is rounded to cents.

Examples

`amount_in_words_euro (0.12)` results in `"zero euro and twelve cent"`

`amount_in_words_euro (-0.12)` results in `"zero euro and twelve cent negative"`

`amount_in_words_euro (1234567.89)` results in `"one million two hundred and thirty-four thousand five hundred and sixty-seven euro and eighty-nine cent"`

`amount_in_words_euro (1234567.89)` results in `"een miljoen tweehonderdvierendertigduizend vijfhonderdzevenenzestig euro en negenentachtig cent"` (output language NLD)

`amount_in_words_euro (12.344)` results in `"twelve euro and thirty-four cent"`

`amount_in_words_euro (12.345)` results in `"twelve euro and thirty-five cent"`

area

Use the `area` function to convert a number into a surface representation.

```
area ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The area to be converted, in square meters.

The output is always based on the metric system. Fractional parts are ignored.

Examples

`area (42)` results in `"0,00,42"`

`area (42)` results in `"0.00.42"` (output language NLD)

`area (123456)` results in `"12.34.56"` (output language NLD)

area_in_words

Use the `area_in_words` function to convert a number into a surface representation.

```
area_in_words ( input )
```

This function returns a value of type TEXT.

The function has one parameter:

- `input`, type NUMBER. The area to be converted, in square meters.

The output is always based on the metric system. Fractional parts are ignored.

Examples

`area (42)` results in `"forty-two centiare"`

`area (42)` results in `"tweeënveertig centiare"` (output language NLD)

`area (123456)` results in `"twaalf hectare, vierendertig are en zesenvijftig centiare"` (output language NLD)

Mathematical functions

square

Use the `square` function to calculate the square of a number.

```
square ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

Examples

`square (2)` results in `4`

`square (4)` results in `16`

`square (4.2)` results in `17.64`

`square (-2)` results in `4`

`square (square_root (10))` results in `10`

square_root

Use the `square_root` function to calculate the square root of a number.

```
square_root ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `input`, type NUMBER. The number for the calculation.

The `square_root` function returns 0 for negative values.

Examples

`square_root (2)` results in `1.414213562`

`square_root (4.2)` results in `2.0493`

`square_root (4)` results in `2`

`square_root (-2)` results in `0`

`square_root (square_root (10))` results in `10`

exponent

Use the `exponent` function to calculate the base-*e* exponential of a number.

```
exponent ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `input`, type NUMBER. The exponent for the calculation.

Examples

`exponent (1)` results in `2.718281828`

`exponent (2)` results in `7.389056099`

`exponent (logarithm (10))` results in `10`

logarithm

Use the `logarithm` function to calculate the natural logarithm of a number.

```
logarithm ( input )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `input`, type NUMBER. The exponent for the calculation.

Examples

`logarithm (2)` results in `0.693147181`

`logarithm (10)` results in `2.302585093`

`logarithm (exponent (10))` results in `10`

sine

Use the `sine` function to calculate the sine of an angle.

```
sine ( angle )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `angle`, type NUMBER. The angle expressed in radians.

Examples

`sine (0)` results in `0`

`sine (90)` results in `0.893996664`

`sine (3.141592654)` results in `0`

cosine

Use the `cosine` function to calculate the cosine of an angle.

```
cosine ( angle )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `angle`, type NUMBER. The angle expressed in radians.

Examples

`cosine (0)` results in `0`

`cosine (90)` results in `-0.448073616`

`cosine (3.141592654)` results in `-1`

tangens

Use the `tangens` function to calculate the tangent of an angle.

```
tangens ( angle )
```

The result of this function is of type NUMBER.

This function has one parameter

1. `angle`, type NUMBER. The angle expressed in radians.

Examples

`tangens (0)` results in `0`

`tangens (90)` results in `-1.99520041`

`tangens (3.141592654)` results in `0`

arctan

Use the `arctan` function to calculate the arc tangent of an angle.

```
arctan ( value )
```

The result of this function is of type NUMBER.

This function has one parameter:

- `value`, type NUMBER. The value with arc tangent to be calculated.

The result is in the range of -π/2 to π/2.

Examples

`arctan (0)` results in `0`

`arctan (90)` results in `1.559685673`

`arctan (3.141592654)` results in `1.262627256`

Control functions

The following section contains information on the control functions that apply to all system-related data.

euro

The `euro` function has been deprecated and is provided for compatibility purposes. Use the `amount_in_words_euro` function and explicit currency overrides for the format function instead (see amount_in_words_euro).

The `euro` function switches the `amount_in_words` and `format` functions between a localized currency and the Euro currency.

Note This function switches **all** currencies, including for countries that are not part of the eurozone.

```
euro ( flag )
```

This function returns a value of type BOOL.

The function has one parameter:

- `flag`, type BOOL. Set to TRUE to enable the Euro representation, FALSE to disable.

This function returns the previous state of the Euro representation.

Examples

`amount_in_words (0.12)` results in `"zero pounds and twelve pence"`

`amount_in_words (0.12)` results in `"nul gulden en twaalf cent"` (output language NLD)

`format (12; "U,UUU,UU9.99")` results in `"øøøøGBP12.00"`

`euro (TRUE)` enables the Euro representation

`amount_in_words (0.12)` results in `"zero euro and twelve cent"`

`amount_in_words (0.12)` results in `"nul euro en twaalf cent"` (output language NLD)

`format (12; "U,UUU,UU9.99")` results in `"øøøøEUR12.00"`

The `euro` function has been deprecated and is provided for compatibility purposes. Use the `amount_in_words_euro` function and explicit currency overrides for the format function instead.

status_message

This function is deprecated.

The `status_message` function enables you to send status messages during a Master Template execution. The message to send has to be entered in the parameter of the function. The maximum length of the message is 250 characters.

Do not use this function if the Master Template is to be used in a KCM Core environment.

```
status_message ( the_message )
```

The result of this function is of type `TEXT`.

The function has one parameter:

- `the_message`, type `TEXT`. This is the message that is sent to the user's screen when the Master Template is executed.

An example is provided here.

```
@( status_message("Processing invoice records..."))
```

The `@ ( )` construction means that the result of the function is displayed in the text. You must use this construction in a text part of a Template script. The switch from the KCM mode to the text mode in a Template script is made with the # characters. A Template script starts in the text mode, the `#BEGIN` makes the switch to the KCM mode.

inc

The `inc` function generates a Template script that post-includes a document in the result document. The post-included text appears as is in the result document.

This function must be called in the text mode of a Template script and be used inside the `@ ( )` construction.

```
@( inc (post_include_document; path_to_the_post_include_document))
```

The result of this function is of type `TEXT`: the post-include statement itself. The construction `@( )` places this text in the result document.

This function has two parameters.

- `post_include_document`, type `TEXT`. This is the name of the document to be post-included. Add the document extension. Also, you can set the document with its complete path in this parameter and leave the second parameter empty.
- `path_to_the_post_include_document`, type `TEXT`. This is the path to the document to be post-included. This is either an UNC path or a path relative to the path set with KCM Core Administrator. The path setting can be left empty if the path where the document can be found is exactly this configured path.

Include a document part

You can include a part of a document. To do so, the text that is to be included should be marked with a bookmark. The name of this bookmark has to be added to the document name (first parameter) when calling the function.

This functionality can only be used in Microsoft Word documents.

Set alternate documents or documents parts

You can set one or more alternate documents or documents parts that KCM will try to include if it fails including a document or part of a document. To do so, the different documents/parts of documents have to be divided with a vertical bar. KCM tries to include the first document or a document part. If it fails, KCM tries to include the next document or a document part. This process stops once KCM succeeds in including a document or a document part. The include process fails if none of the documents or documents parts can be included.

This functionality can only be used in Microsoft Word documents.

An example is provided here.

```
@( inc("document"; "folder" ))
@( inc("document<bookmark>"; "folder" ))
@( inc("document1|document2"; "folder"))
```

The result of the preceding example is as follows.

`__INC(folder\document)` in the result document. This statement is processed if post-including is enabled in the KCM configuration or when `LAZYPOSTINC` is set to Y.

For WordPerfect this function usually generates native WordPerfect subdocument instructions. If the post-include mechanism is enabled in the configuration file, the `inc( )` function generates `__INC` statements in the WordPerfect result documents.

Post-include statements in result documents are also resolved by the `put_in_document( )`, `put_in_text_file( )`, and `add_to_output( )` functions, when the fifth parameter of these functions is set to Y, regardless of the post-include configuration setting of KCM.

put_in_document

The function `put_in_document` places produced text in a document. The text placed in a document in this way does not appear in the result document. All result text produced up to the function call is placed in the document specified in the function call. You can call the function several times in a Master Template. Each time the text produced up to the call is placed in a new document.

```
put_in_document ("document_to_be_produced";
                 "path/folder_of_the_document";
                 "overwrite Y/N"; "pagination (ignored)";
                 "process_includes Y/N")
```

The function returns a `TEXT`.

This function has five parameters:

- `document_to_be_produced`, type `TEXT`. This is the name of the document.
- `path/folder_of_the_document`, type `TEXT`. This is the path and name of the folder or a folder that the document is written to.
- `overwrite`, type `TEXT`, Y or N. When you set this parameter to Y, the existing document is overwritten; when you set this parameter to N, the end user is prompted to specify an alternative file name or folder. Interactivity is not possible in a KCM Core environment.
- `pagination`, type `TEXT`, Y or N. The value of this parameter is ignored but must be present.
- `process_includes`, type `TEXT`, Y or N. When set to Y, KCM resolves `__INC` post-includes in the result document. In KCM Core, this setting is enabled by default.

The function `put_in_document` writes the KCM output to a document and does not create a copy. Once written to a document, the output is no longer available for a Template script and does not appear in the result document. The reason for this is that KCM stores produced text in an internal buffer. When you call the `put_in_document` function, KCM writes the content of the buffer to the document, which clears the internal buffer.

Tip To create multiple files from one output, you can use the `open buffer` mechanism (see open_buffer).

Documents that KCM produces through the `put_in_document( )` function are directly available for opening after the function has been successfully completed.

An example is provided here.

```
#
This line will be stored in the document 'document.doc' on c:\temp, and will not appear
 in the result document.
#
TEXT doc := put_in_document("document.doc"; "c:\temp"; "Y"; "N"; "N")
```

```
#
The document produced is @(doc).
These lines will appear in the result document because they are produced after the
 put_in_document call above.
You could also write these lines to a second document by calling put_in_document again.
#
```

Note If you call the function `put_in_document` in a Microsoft Word DOC Master Template, you should be aware that the text part you produce by calling the function must end with a paragraph ending.

You cannot place the text from repeated calls to `put_in_document( )` in the same document. To retrieve the text from several calls in one document, use the `add_to_output` function. To leave the result document intact, you can use the `open buffer` mechanism (see open_buffer).

put_in_text_file

The function `put_in_text_file` writes produced text to a file. The text saved in this way does not appear in the result document. You can call this function several times in a Template script so the text produced up to the call is placed in the file. This function produces an ASCII document regardless of the word processor a Master Template has been made with.

```
put_in_text_file ("textfile_to_be_produced";
                  "path/folder_of_the_text_file";
                  "overwrite Y/N";
                  "pagination (ignored)";
                  "process_includes Y/N")
```

The function returns a `TEXT`: the name of the file produced.

The `put_in_text_file` requires five parameters:

- `textfile_to_be_produced`, type `TEXT`. This is the name of the document.
- `path/folder_of_the_text_file`, type `TEXT`. This is the path and name of the folder or a folder that the text file is written to.
- `overwrite`, type `TEXT`, Y or N. When you set this parameter to Y, the existing file is overwritten; when you set this parameter to N, the end user is prompted to specify an alternative file name or folder. Interactivity is not possible in a KCM Core environment.
- `pagination`, type `TEXT`, Y or N. The value of this parameter is ignored.
- `process_includes`, type `TEXT`, Y or N. When set to Y, KCM resolves `__INC` post-includes in the result document. In KCM Core, this setting is enabled by default.

The `put_in_text_file` function writes the KCM output to a text file and does not create a copy. Once written to a file, the output is no longer available for a Template script and it does not appear in the result document. The reason for this is that KCM stores produced text in an internal buffer. When you call the `put_in_text_file` function, KCM writes the content of the buffer to the file, which clears the internal buffer.

Tip To create multiple files from one output, you can use the `open buffer` mechanism (see open_buffer).

Documents that KCM produces through the `put_in_text_file( )` function are directly available for opening after KCM successfully executes the `put_in_text_file( )` function.

Note The ASCII result document also contains word processor instructions such as text boxes, fields, and footnotes. This may cause unexpected results as no attempt is made to maintain the layout of the output document. Therefore, you should avoid using word processor instructions that will appear in result documents when the `put_in_text_file` function is used.

An example is provided here.

```
#
This line will be stored in the file 'file.txt' on c:\temp, and will not appear in the
 result document.
#
TEXT doc := put_in_text_file("file.txt"; "c:\temp"; "Y"; "N"; "N")
#
The file produced is @(doc).
These lines will appear in the result document because they are produced after the
 put_in_text_file call above.
#
```

put_in_text_file2

The function `put_in_text_file2` writes produced text to a file. Text saved in this way does not appear in the result document. You can use this function several times in a Template script so the text produced up to the call is placed in the file.

This function produces a plain text document regardless of the word processor the Master Template has been made with. The plain text document is encoded using the character set encoding specified in the encoding parameter encoding. If the document content cannot be represented using the specified character set encoding, an error is generated. Unlike the `put_in_text_file` function, the `put_in_text_file2` function does not support processing of Includes.

```
put_in_text_file2 ("textfile_to_be_produced";
                   "path/folder_of_the_text_file";
                   "overwrite Y/N";
                   "encoding")
```

The function `put_in_text_file2` requires four parameters:

- `textfile_to_be_produced`, type `TEXT`. This is the name of the document.
- `path/folder_of_the_text_file`, type `TEXT`. The path and name of the folder or a folder that the text file is written to.
- `overwrite`, type `TEXT`, Y or N. When you set this parameter to Y, the existing file is overwritten; when you set this parameter to N, the end user is prompted to specify an alternative file name and folder. Interactivity is not possible in a KCM Core environment.
- `encoding`, type `TEXT`. The value must be one of the following: UTF-8, UTF-8bom, UTF-16, ISO-8859-1, ISO-8859-15 and 1252. This parameter specifies the character set encoding of the plain text document that is to be written.

The `put_in_text_file2` function writes the KCM output to a text file and does not create a copy. Once written to a file, the output is no longer available for a Template script and it does not appear in the result document. The reason for this is that KCM stores produced text in an internal buffer. When you call the `put_in_text_file2` function, KCM writes the content of the buffer to the file, which clears the internal buffer.

Tip To create multiple files from one output, you can use the `open buffer` mechanism (see open_buffer).

Documents that KCM produces through the `put_in_text_file( )` function are directly available for opening after KCM successfully executes the `put_in_text_file2( )` function.

Note The plain text result document also contains word processor instructions such as text boxes, fields, and footnotes. This may cause unexpected results as KCM makes no attempt to maintain the layout of the output document. Therefore, you should avoid using word processor instructions that will appear in result documents when the `put_in_text_file2` function is used.

An example is provided here.

```
#
This line will be stored in the file 'file.txt' on c:\temp, and will not appear in the
 result document.
#
TEXT doc := put_in_text_file2("file.txt"; "c:\temp"; "Y"; "ISO-8859-1")
#
The file produced is @(doc).
These lines will appear in the result document because they are produced after the
 put_in_text_file2 call above.
#
```

The UTF-8bom (UTF-8 + byte-order marker) encoding is introduced in KCM Core version 4.2.3.

add_to_output

The `add_to_output` function puts output in a document. Multiple calls to `add_to_output( )` are concatenated into one output document.

```
add_to_output ( "outputdocument_to_be_produced";
                "path_folder_of_the_output_document";
                "Overwrite Y/N"; "pagination (ignored)";
                "process_includes Y/N" )
```

The result of this function is of type `TEXT`, it is the path and name of the document produced. A TXT template produces a TXT file; a Microsoft Word template produces a Microsoft Word document.

This function has five input parameters of type `TEXT`:

- `outputdocument_to_be_produced`, type `TEXT`. This is the name of the produced document.
- `path_folder_of_the_output_document`, type `TEXT`. The path and name of the folder or a folder that the output document is written to.
- `overwrite`, Y or N, type `TEXT`. When you set this parameter to Y, the existing document is overridden; when you set this parameter to N, the end user is prompted to specify an alternative file name or folder. Interactivity is not possible in a KCM Core environment.
- `pagination`, type `TEXT`. The value of this parameter is ignored.
- `process_includes`, Y or N, type `TEXT`. Resolve `__INC` post-includes in the result document. When you have enabled the check box Process includes on the Run tab of KCM Core Administrator, KCM always resolves post-includes.

The `add_to_output( )` function writes the KCM output to a text file and does not create a copy. Once written to a file, the output is no longer available for a Template script and it does not appear in the result document. The reason for this is that KCM stores produced text in an internal buffer. When you call the `put_in_text` function, KCM writes the content of the buffer to the file, which clears the internal buffer.

Tip To create multiple files from one output, you can use the `open_buffer` mechanism (see open_buffer).

Documents that KCM produces through the `add_to_output( )` function are directly available for opening after KCM successfully executes the `put_in_text_file2( )` function.

An example is provided here.

```
TEXT outputdocument_to_be_produced := "output.doc"


TEXT path_folder_of_the_output_document := "c:\temp\"
#
This is the first part of the result document that will be put in the output document.
#
TEXT doc := add_to_output(outputdocument_to_be_produced;
 path_folder_of_the_output_document; "Y"; "N"; "N")
#
This is the text that is produced after the first function call. It will also be placed
 in the output document by the next function call.
#
ASSIGN doc := add_to_output(outputdocument_to_be_produced;
 path_folder_of_the_output_document; "Y"; "N"; "N")

#
The file produced is @(doc).
#
```

Note If you call the function `add_to_output( )` in a Microsoft Word DOC model, you should be aware that the final output document must end with a paragraph ending.

open_buffer

This function is always used in combination with the `put_buffer_in_document` function (see put_buffer_in_document). These functions together save parts of a result document to a location on the file system while the saved part is also written to the result document, as opposed to the `put_in_document` function (for more information on the `put_in_document` function, see put_in_document). This function can only be used in KCM Core.

The function `open_buffer` indicates the position in a Template script from where the result text is saved. The marker is placed in the result document wherever this function is called.

```
 open_buffer ( marker_name )
```

The function yields a result type `BOOL`: `TRUE` if the marker is successfully placed, `FALSE` if it failed to set the marker or if the function is not supported.

The function `open_buffer` has one parameter:

- `marker_name`, type `TEXT`. The function places a marker with this name at the position in a Template script where it is called. The markers are case-sensitive.

Important If creation of the document is delayed, it may later cause a fatal error.

The function `open_buffer` fails if the name of the buffer is invalid (empty).

You can call this function repeatedly throughout a Template script and store several fragments of the result text if you use a unique `marker_name`.

Note If the function `open_buffer` is called more than once with the same value for the parameter, only the last position of the marker is saved.

An example is provided here.

```
BOOL ok := open_buffer( "example" )
```

The marker example is placed in the text where this line is in the Template script. The success of this action is stored in `ok`.

put_buffer_in_document

The `put_buffer_in_document` function is always used in combination with the `open_buffer` function (see open_buffer). This function can only be used in combination with KCM Core.

These functions together save parts of a result document to a location on the file system while the saved part is also written to the result document, as opposed to the `put_in_document` function (see put_in_document).

The `put_buffer_in_document` function saves all text from the location where the marker tag was registered up to the call to this function. Markers are registered using the function `open_buffer`.

```
put_buffer_in_document ( document_name;folder_name;overwrite_Y/N;
                         pagination_Y/N;process_includes_Y/N;marker_name )
```

The result of the function is of type `BOOL`. The function yields `TRUE` if the result document could be written. The function yields `FALSE` if the file could not be written or if the function is not supported.

The function `put_buffer_in_document` call has six parameters separated by semicolons:

- `document_name`, type `TEXT`. This is the name of the document that will contain the text fragment.
- `folder_name`, type `TEXT`. Path and name of the folder that the file is written to. This folder must exist.
- `overwrite`, Y or N, type `TEXT`. When you set this parameter to Y, the existing document is overwritten.
- `pagination`, Y or N, type `TEXT`. The value of the option is ignored, but the parameter must be provided. Enter N.
- `process_includes`, type `TEXT`, with accepted values Y or N. If this parameter is set to Y, KCM always includes post-includes. If this parameter is set to N, the global post-include configuration determines whether or not post-includes are included. KCM Core enables the lazy post-include feature by default. The Master Template should use the `@(inc(...))` construction to ensure that documents are included correctly. For more information, see inc.
- `marker_name`, type `TEXT`. The name of the marker from where the result text has to be saved. This marker must have been placed with the `open_buffer` function (see open_buffer).

The function `put_buffer_in_document` fails if one of the following conditions is met:

- The name of the buffer is invalid (empty).
- The buffer has not been opened with the function `open_buffer`.
- The document cannot be written.
- The document already exists and the overwrite parameter is not set to Y.

Calling the function `put_buffer_in_document` removes the marker specified.

An example is provided here.

```
 BOOL ok2 := put_buffer_in_document( "document";
"folder"; "Y"; "Y"; "N"; "example" )
```

Part of the result text starting with the marker `example` is written to the file `document` that resides in the folder `folder`. If the file already exists, it is overwritten. `pagination` is not yet implemented and post-includes set with `__INC` are not processed. After this, the marker `example` is removed from the result document.

pragma

The use of the `pragma` function allows the Master template to override selected features in the execution process and the creation of result documents.

```
 pragma ( feature_name;feature_value )
```

The result of this function is of type `TEXT` and contains the necessary word processor instructions. These instructions should be put into the result document with the construction `@(pragma (...))`, unless otherwise specified.

The `pragma` function requires two parameters of type `TEXT`:

- `feature_name`, type `TEXT`. This is the type of feature to be modified.
- `feature_value`, type `TEXT`. This is the new value of the feature.

The following table describes the currently supported features.

Note Incorrect use of this function can result in documents that cannot be opened by the word processor. If a feature is not supported by the word processor support, it is silently ignored.

Feature	Value	Effect
Template	path\template name	Replaces the template of the result document with the template given as `feature_value`. In Microsoft Word DOC Master Templates, the template must be specified as a file system path. In Microsoft Word DOCX Master Templates, the template must be specified as a valid URL. For example, *{path}:\templates \sample.dotm*.

UnicodeEuro	Y or N	Activates (Y) or deactivates (N) within the Template script the translation of single-byte euro characters to Unicode euro characters. This functionality is only supported for Microsoft Word DOC Master Templates.
CurrencyEuro	Y or N	Activates (Y) or deactivates (N) within the Template script the translation of the International Currency Symbol to Unicode euro characters. This functionality is only supported for Microsoft Word Master Templates (both DOC and DOCX).
FormField: bookmark:Default	Any text	Sets the default content for a Form Field. The formatting of the default text should match the formatting mask. This functionality is only supported for Microsoft Word DOC Master Templates.
FormField: bookmark:Format	A formatting mask	Sets the formatting mask for a Form Field. This functionality is only supported for Microsoft Word DOC Master Templates.
FormField: bookmark:Help	Any text	Sets the text of a Form Field shown if the end user presses F1. This functionality is only supported for Microsoft Word DOC Master Templates.
FormField: bookmark:Status	Any text	Sets the text of a Form field shown in the status bar. This functionality is only supported for Microsoft Word DOC Master Templates.
FormField: bookmark:Checked	Y or N	Y sets the check boxes of a Form Field in the Checked state. N removes the check boxes. This functionality is only supported for Microsoft Word DOC Master Templates.
FormPassword	0 or "" (an empty value)	0 protects the result document with Form Fields with an empty password. An empty value protects the result document with an unknown password. This functionality is only supported for Microsoft Word DOC Master Templates.
RevisionMarking	Y or N	Enables or disables revision marking. This functionality is only supported for OpenOffice.org.

RevisionMarkingActive	Y or N	Enables or disables the tracking of new changes in the document. This functionality is only supported for OpenOffice.org.
RevisionMarkingPassword	password	Changes the revision marking password. If the password is enabled, the end user cannot change the revision marking options. An empty password disables the password protection. This functionality is only supported for OpenOffice.org.
Password	password	Sets a password that write protects the section in which the function is called. This functionality is only supported for OpenOffice.org.
Reset	"textboxchains" or "lists" or "list"	Resets aspects of an Microsoft Word document (both DOC and DOCX). Value *textboxchains* resets the chains of the text boxes in the document. Should be called in the body of the document (not in the header, footer, or text box itself). Value *lists* resets all numbered lists so they restart numbering the next time they are produced as output. Value *list* resets only the first numbered list produced as output after the `pragma` function.
LOCALE:LC_MONETARY:int_curr_symbol	int_curr_symbol	The representation of the multi-character currency code (U).
LOCALE:LC_MONETARY:currency_symbol	currency_symbol	The representation of the single-character currency code ($).
LOCALE:LC_NUMERIC:decimal_point	decimal_point	The representation of the decimal point (.).
LOCALE:LC_NUMERIC:thousands_sep	thousands_sep	The representation of the thousands separator (,).
AutoInsertSeparator	text	Can be used with the keyword TEXTBLOCK (see TEXTBLOCK statement) or with the AUTOINSERT keyword in a FORM. It sets the text generated after each Text Block.
AutoInsertTerminator	text	Can be used with the keyword TEXTBLOCK (see TEXTBLOCK statement) or with the AUTOINSERT keyword in a FORM. It sets the text generated after the last Text Block.

AutoInsertPrefix	prefix	Can be used with automatically inserted Text Block questions in a FORM or in a dynamic FORM (see FORM). It contains the prefix of the set of Text Block styles that needs to be applied to the Text Blocks inserted by the FORM. By default, KCM searches for the Text Block styles with an ITP prefix.
AutoInsertParagraph	paragraph sign	Can be used with automatically inserted Text Block questions in a FORM or in a dynamic FORM (see FORM). It contains the paragraph sign that holds the style applied to the Text Block. It is mainly used when a Text Block needs to be inserted into a table cell. To insert this Text Block into a table cell, the paragraph sign needs to be declared within a table cell itself as well. A corrupted result document could be produced if an incorrect paragraph sign is used.
ChangePassword	0 or "" (an empty value)	0 protects the result document with an empty password. An empty value protects the result document with an unknown password. This function activates the same password as the function `FormPassword` but without enabling the Form Field protection feature. This functionality is only supported for Microsoft Word DOC Master Templates.
EnhancedUnicodeSupport	Y, N, or ?	Overrides the setting EnhancedUnicodeSupport in the current environment for all subsequent text-based comparisons that are performed by the Master Template. This `pragma` can be called more than once to switch behavior within the Master Template. A call to this `pragma` sets a new value for the setting EnhancedUnicodeSupport and returns the previous value. The output of the `pragma` function call does not have to be put into the result document. The value ? can be used to inquire the current value of the setting EnhancedUnicodeSupport without changing it.

EnhancedUnicodeMaps	Y, N, or ?	Overrides the setting EnhancedUnicodeMaps in the current environment. You can use this `pragma` only before you make the first MAP assignment. Any attempt to change this setting after the first assignment results in a fatal error. A call to this `pragma` sets a new value for the setting EnhancedUnicodeMaps and returns the previous value. The output of the pragma function call does not have to be put into the result document. You can use the value ? to inquire the current value of the setting EnhancedUnicodeMaps without changing it.
FacingPages	Y, N, M, or P	Determines how the "different odd or even" setting for the headers and footers is controlled. This option affects the whole document, it cannot be changed in different sections. The following values are supported: Y: Enables the setting; N: Disables the setting; M: Uses the setting from the Master Template; P: If the function `pagestyle` (see pagestyle) is used, enables the setting if any of the template documents has the setting enabled; otherwise, disables the setting. If the function `pagestyle` is not used, use the setting from the Master Template. The option P is the default behavior. This functionality is only supported for Microsoft Word Master Templates (both DOC and DOCX).

MergeIncludeStyles	Y or N	Determines how styles in documents that are included through the `inc` function are processed. The following values are supported: Y: Replaces styles in the Includes with the styles from the parent document. The layout of the Include might change to match the parent document. N: Renames styles and document defaults in the Include. The original layout of the include document is maintained as best as possible. The option Y is the default behavior. This functionality is only supported for Microsoft Word DOCX Master Templates.
OverrideSFP	Y, N, or "" (an empty value)	Overrides the SUPPRESS_FINAL_PARAGRAPH option for the current Text Block. The following values are supported: Y: Enables SUPPRESS_FINAL_PARAGRAPH. N: Disables SUPPRESS_FINAL_PARAGRAPH. "": Disables previous overrides and uses the option as used with the TEXTBLOCK statement. This function should be called from within a `FORMAT` function (see FORMAT functions called from within the Text Block. It returns Y if the option was applied correctly; N if the value is unsupported. Other values indicate that the function was not recognized.

TextblockTableStyle	A style name or "" (an empty value)	Overrides the table style used to format the current table in a Text Block. The following values are supported: table: Uses the default table style (ITP_table). "": Uses the default table style (ITP_table). xxx: Uses a specific table style (ITP_xxx). The style name is subject to the STYLE_PREFIX and TABLE_STYLE_PREFIX options. This function should be called from within a `FORMAT` function called within the table. It returns Y if the option was applied correctly. Any other value indicates that the function was not recognized. This function has no effect in Rich Text Blocks. The resulting table style must exist in the style sheet of the result document.

LOCALE settings

The settings LOCALE set the outcome of the function `Format`. The symbols set with LOCALE override the language specific settings when the function is used.

Examples are provided here.

```
@(pragma("Template"; "c:\Word97\Templates\custom.dot"))
@(pragma("UnicodeEuro"; "Y"))
@(pragma("UnicodeEuro"; "N"))
@(pragma("FormField:Amount:Default"; "12,00"))
```

```
TEXT old_state := pragma ("EnhancedUnicodeSupport"; "N")
ASSIGN first_10_bytes := text_fragment (something; 1; 10)
TEXT ignored := pragma ("EnhancedUnicodeSupport"; old_state)
```

Obtain the first ten bytes of something regardless of the setting EnhancedUnicodeSupport.

pragma_struct

The `pragma_struct` function allows the Master Template to override selected features in the execution process and the creation of result documents.

```
FUNC TEXT pragma_struct (CONST TEXT feature_name; CONST TEXT feature_value;
 DATASTRUCTURE data_structure)
```

The result of this function is of `TEXT`.

The `pragma` function requires three parameters:

- `feature_name`, type `TEXT`. The type of feature to be applied.
- `feature_value`, type `TEXT`. Depends on the feature to be applied.
- `data_structure`, any `DATASTRUCTURE` variable or _data.

The following table lists currently supported features.

Feature	Value	Effect	Result	Required KCM Core
`Output`	Header text	Writes `feature_value`, followed by the full content of `data_structure`, formatted as XML into the result document. This function is intended for testing purposes only. All word processor content is stripped. This function is ignored when AIADOCXML output is requested and returns N.	Y if the output was produced successfully; N otherwise.	4
`WriteDebugLog`	Header text	Writes `feature_value`,followed by the full content of `data_structure`, formatted as XML into the debug log file.	Y if debug logging was enabled; N otherwise.	2.3

Any unsupported feature values are silently ignored and return an empty TEXT.

The resulting value of the `pragma_struct` function may contain word processor instructions. You should put these instructions in the result document using the `@(...)` construction. Manipulation of such a result using `text_fragment`, `fragment_of_characters`, `replace` or similar functions is not supported and can result in invalid result documents.

system

The `system` function starts a Microsoft Windows command line and passes a command to it. You can also specify to wait for the result of the command line.

```
system ( command, wait Y/N )
```

The result of this function is of type `NUMBER`.

This function has two parameters:

1. `command`, type `TEXT`. This is the command line to be executed. This command line must be a valid Microsoft Windows command line.
2. `wait`, type `TEXT`, Y or N. When you set this parameter to Y, KCM waits until the started command terminates. When you set this parameter to N, KCM continues as soon as the command has been started.

Return codes

The result of the function is of type NUMBER and can be one of the values described in the following table.

<0:	The command could not be started. The values is equal to the negative error code. Waitparameter is N:
0:	The command was started. Waitparameter is Y:

>= 0:	The command was started. The value is equal to the return code of the command.

An example is provided here.

```
NUMBER return_code

ASSIGN return_code := system("c:\winnt\system32\notepad.exe" ; "N")
#
@(return_code)
#
```

Note The function only works if notepad is located in `c:\winnt\system32`.

itp_setting

The `itp_setting` function allows the Master Template to retrieve settings from the current KCM configuration file.

```
 itp_setting (setting_to_be_retrieved)
```

The result of this function is of type `TEXT`. It contains the value of the setting from the KCM configuration file or an empty string if the setting was not present.

This function requires one parameter:

- `setting_to_be_retrieved`, type `TEXT`. This is the setting to be retrieved.

You can add private settings to the configuration file if they do not start with KCM.

An example is provided here.

```
TEXT temp_dir := itp_setting("ITPTMPDIR")
#
@(temp_dir)
#
```

document_property

With the function `document_property`, you can set a predefined Microsoft Word document property.

The function result is a Microsoft Word instruction set with the specified document property. The Master Template must produce this instruction in the result document to have effect.

```
document_property (property_name; set_to_text)
```

This function yields a value of type `TEXT`.

The function `document_property` has two parameters divided by semicolons:

1. `property_name`, type `TEXT`. This is the name of the property that should be set. See Microsoft Word document property names.
2. `set_to_text`, type `TEXT`. This is the text the property must be set to.

The `document_property` function is only supported for Microsoft Word Master Templates.

Microsoft Word document property names

The `document_property` function recognizes the following predefined Microsoft Word document properties. These properties are supported for both DOC and DOCX Master Templates.

Property	Value
Title	Any text
Category	Any text
Subject	Any text
Author	Any text
Keywords	Any text
Comments	Any text
Manager	Any text
Company	Any text
Createtime	Date
Printtime	Date
Savetime	Date
Status (DOCX only)	Any text

The date for the CreateTime, PrintTme, and SaveTime properties must be specified as a text in either YYYYMMDD or YYYYMMDDhhmmss format. An optional DATE: prefix is allowed.

An example is provided here.

```
@(document_property("Title"; "This is the document title"))
@(document_property("Createtime"; "20041231124353"))
```

Custom properties

All properties that are not recognized as a predefined property are added to the custom property list of the document. The type of the property can be set by adding a prefix to the value.

Type	Prefix
Text property	None
Date property	DATE:
Number property	NUMBER:

When dates must be added to the Custom property list, they must be specified in either DATE:YYYYMMDD or DATE:YYYYMMDDhhmmss format. The prefix DATE: is required.

An example is provided here.

```
@(document_property("Archival ID"; "KR35A"))
@(document_property("Date Received"; "DATE:20010702"))
@(document_property("Document Number"; "NUMBER:53245"))
```

itpserver_parameter

This function is deprecated. Use the Field Set _Document instead (see _Document).

With the `itpserver_parameter` function, you can pass information back to KCM Core or KCM ComposerUI Server when the Master Template is run by KCM Core or KCM ComposerUI Server.

```
 itpserver_parameter ( parameter_name;
return_value )
```

The function result is of type `BOOL`. The function always returns `TRUE`.

The `itpserver_parameter` function has two parameters divided by semicolons:

1. `parameter_name`, type `TEXT`. This is the name of the parameter as it is known or used in KCM Core.
2. `return_value`, type `TEXT`. This is the value to be returned for the parameter.

This function is intended to be used with the `itp_parameter` function in KCM Core scripts. The data marked and passed with `itpserver_parameter` function in the Master Template can be read in a KCM Core script with the `itp_parameter` function. To retrieve the value, the `itp_parameter` function takes the identifier `parameter_name` set in `tpserver_parameter` as parameter.

Note To run a Master Template containing this feature, you need KCM Core or KCM ComposerUI Server.

An example is provided here.

In the Master Template.

```
 BOOL dummy := itpserver_parameter ("print"; "yes, please")
```

The variable dummy is used to contain the return value of the function in the KCM Core script.

```
do_we_print_it = itp_parameter
("print");
```

The function `itpserver_parameter` supersedes the function `itpds_parameter` that is obsolete now.

runmodel_setting

The `runmodel_setting` function can be used to retrieve information on the KCM context in which the Master Template is run. The function has one parameter of type `TEXT` that contains the setting that the function retrieves.

The function result is of type `TEXT` and holds the retrieved value.

The supported settings are the following:

- **ResultDocument**. Retrieves the path and name of the ResultDocument.
- **Environment**. Retrieves the name of the KCM Core environment in which the Master Template is run.
- **Session**. Retrieves the session folder of KCM ComposerUI Server.

- **Product**. Retrieves the KCM product used to run Master Templates.
- **User**. Retrieves the user account with which KCM Core is run.
- **Project**. Returns the default KCM Repository project in which KCM Core locates Text Block Lists and Text Blocks. The RepositoryProject configuration setting in the active KCM Core environment is used as the default project. If this setting is not specified, the project from the Master Template is used as the default project.
- **Model**. Retrieves the path and name of the Master Template on the local filesystem.
- **ModelRepositoryName**. Retrieves the name of the Master Template as it was known in the KCM Repository.
- **ModelRevision**. Retrieves the revision of the Master Template. The retrieved revision includes the KCM Repository identification.
- **RepositoryCreator**. Retrieves the KCM Repository user who compiled the Master Template.
- **RepositoryCompileDate**. Retrieves the date the Master Template was compiled.
- **RepositoryProjectType**. Retrieves the type of project the Master Template was created in. Possible return values are `Fragment` for Data Backbone Libraries, `DataBackbone` for Projects created with KCM Repository version 4.1.3 or higher, or `Classic` for Projects created with KCM Repository before version 4.1.3.
- **ITPVersion**. The version of KCM Core in VRM format.
- **ITPFullVersion**. The version of KCM Core in extended format.
- **OutputMode**. The type of output currently being produced by the Master Template. The Output Mode is specified when the Master Template is started but can be changed in KCM Enterprise for every alternative. Possible return values are: native, utf8, utf16, xml.

An example is provided here.

```
TEXT result := runmodel_setting("ResultDocument")
TEXT user := runmodel_setting("User")
```

The settings ModelRepositoryName, ModelRevision, RepositoryCreator, and RepositoryCompileDate are stored in the Master Template at the moment it is compiled in KCM Repository. If the Master Template is renamed later, moved, imported, or exported, this information is kept unmodified. Master Templates must be compiled with KCM Repository version 3.5.11 or higher to include this information.

itpserver_setting

The `itpserver_setting` function can be used to retrieve information on the KCM Core context in which the Master Template is run.

The function has one parameter of type `TEXT` that contains the setting that the function retrieves. The function result is of type `TEXT` and holds the retrieved value.

Supported settings are the following:

- _JobID retrieves the KCM Core Job Identifier of the current request.
- _User retrieves the user who submitted the request. For KCM Core requests, this returns the user profile of the user who submitted the request; for KCM ComposerUIrequests, this returns the user credentials used to authenticate with KCM ComposerUI.
- _ApplicationID retrieves the application id of the KCM ComposerUI IIS application or the application name of the KCM ComposerUI J2EE application.

- _Server retrieves the Microsoft Windows Service name of the KCM Document Processor running the script that is accessing this constant.
- _Service retrieves the text "Load Balancer Interface:" followed by the name of the KCM Core service accessing this constant.
- _ServerName retrieves the name of the KCM Core installation.
- TempDir retrieves the temp location created of the KCM Document Processor.
- ITPWorkDir retrieves the folder ITPWORK of a KCM Core setup.
- KCM Core Constants can be used to retrieve the value of KCM Core constants. KCM Core constants are defined on the Constants tab of the Services. Their value can be retrieved with the `itpserver_setting` function passing the name of the constant as parameter.

An example is provided here.

```
 TEXT server :=
itpserver_setting("_Server")
 TEXT temp :=
itpserver_setting("TempDir")
```

environment_setting

The `environment_setting` function can be used to retrieve information on the Microsoft Windows context in which the Master Template is run.

The function has one parameter of type `TEXT` that contains the setting that the function retrieves. The function result is of type `TEXT` and holds the retrieved value.

All Microsoft Windows environment settings can be retrieved by passing their name as parameter to the `environment_setting` function.

An example is provided here.

```
 TEXT domain :=
environment_setting("USERDOMAIN")
```

session_parameter

You can use the function `session_parameter` to retrieve session parameters from the current session in which the Master Template is run.

The function has one parameter of type `TEXT` containing the name of the session parameter that the function retrieves.

The function results in type `TEXT` and holds the retrieved value.

An example is provided here.

```
 TEXT value1 :=
session_parameter("parameter1")
```

create_csv

The `create_csv` function is used to create a text string with comma separated values from an `ARRAY TEXT`.

```
create_csv ( list; number of elements)
```

The function has two parameters:

1. `list`, type `ARRAY TEXT`; contains the elements that should be contained in the text string with comma separated values.
2. `number_of` elements, type `NUMBER`; contains the number of elements from the list that should be included in the result.

The result of the function is type `TEXT`.

Note Use straight quote (") for text strings.

Tip You can use the `create_csv` function to create a comma-separated list of Text Block names that can be used with the statement `TEXTBLOCK`.

split_csv

The `split_csv` function is used to split a text string containing comma-separated values into separate values. You can use this function to split up lines taken from a CSV file.

```
split_csv ( csv; list )
```

The function has two parameters:

1. `csv`, type `TEXT`; contains the comma-separated values text string that needs to be split.
2. `list`, type `ARRAY TEXT`; contains the `ARRAY` where the separate values are stored.

The result of the function is of type `NUMBER`. This is the number of elements that were extracted from the text string comma-separated values. If an empty text string is passed as the parameter `csv`, the result is1, and the parameter list contains one element that is empty.

add_user_xml

The `add_user_xml` function is deprecated. Use the Field Set instead (see _Document).

You can use this function for a Master Template to add user-defined data to the metadata XML file that also contains Master Template run information.

A metadata XML file can only be generated for Document Template runs that do not have `OutputMode` set to "pack".

```
TEXT ignore := add_user_xml ( k; v )
```

The preceding call adds the key `k` and the value `v` to the XML metadata file if that file is generated. It inserts an element that looks in the following way.

```
<itp:element>
  <itp:key>k</itp:key>
  <itp:value>v</itp:value>
</itp:element>
```

The return value of the function should be ignored. The parameters are both of type `CONST TEXT`.

stylesheet

The `stylesheet` function specifies a Style Sheet applied to the result document once the function has been called. This Style Sheet overrides the styles in the result document when the document is produced.

```
TEXT ignore := stylesheet ( brand; stylesheet)
```

The result of the function `stylesheet` is of type `TEXT`.

This function has two parameters:

1. `brand`, type `TEXT`; the brand the Style Sheet belongs to. You can use brands to group associated styles to implement multi-labeling.
2. `stylesheet`, type `TEXT`; the name of the Style Sheet. If this parameter is empty, the previous override is canceled.

The result of the function `stylesheet` is of type `TEXT`. It is currently undefined and should be ignored.

The function `stylesheet` has the following effect on result documents:

- The result document uses the Style Sheet that was specified when the Master Template was finished.
- Documents produced with the `put_in_document` function (see put_in_document) and `put_buffer_in_document` function (see put_buffer_in_document) use the Style Sheet that was last to be specified before the function was called.
- Documents produced with the `add_to_output` function (see add_to_output) use the Style Sheet that was last to be specified before the first `add_to_output` function was called to create the result document.

The functionality of the function `stylesheet` is only available for KCM Microsoft Word output (both DOC and DOCX) and is ignored for other document formats.

An example is provided here.

```
# BEGIN

TEXT ignore := stylesheet ("Insurance For Life"; "Invoice")
TEXT path := put_in_document ("invoice.doc"; path; "Y"; "N"; "N")
ASSIGN ignore := stylesheet ("Insurance For Life"; "Policy")

END #
```

pagestyle

The `pagestyle` function inserts a section break and applies the page settings from the specified page style document.

```
@(pagestyle (brand; template))
```

The result of the function `pagestyle` is of type `TEXT`.

This function has two parameters:

1. `brand`, type `TEXT`; the brand that the template belongs to. You can use brands to group associated styles to implement multi-labeling.
2. `template`, type `TEXT`; the name of the template document. The page settings of the first page of this document are applied.

The result of the `pagestyle` function must be put into the result document to have an effect.

The first time the `pagestyle` function is applied to a result document, it does not insert a section break. This gives the script developer the ability to use this function to set the page style for the initial section of the document. Any subsequent applications automatically insert a new section break. If the function `pagestyle` is applied for the first time after producing some initial pages, the script developer is responsible for inserting a section break manually immediately followed by the function `pagestyle`.

In KCM Enterprise, the function `pagestyle` does not insert a section break the first time it is used for any document or document alternative produced by the Master Template. Any subsequent applications within each document or document alternative automatically insert a new section break.

The attributes of the section break are derived from the first section of the template document. Headers and footers are not copied from the template document but inherited from the previous section (if any) of the result document. The functions headers and footers can be used to make changes to the section.

The function `pagestyle` also sets the "different odd and even" setting in the result document. If any of the template documents has this option enabled, it is also enabled in the result document. If none of the template documents has this option enabled, it is disabled in the result document. This feature can be disabled with the pragma function (see pragma).

The functionality of the function `pagestyle` is only available for the KCM Microsoft Word output (both DOC and DOCX) and ignored for other document formats.

You cannot use the function `pagestyle` to introduce a section break within a table in the result document. This results in a run-time error. If the new section must begin with a table, the function `pagestyle` must be called before the table is created.

An example is provided here.

```
# BEGIN

#
@(pagestyle ("Insurance For Life"; "Invoice"))
#

#
@(pagestyle ("Insurance For Life"; "Policy"))
#

END #
```

language_code

The `language_code` function enables you to change the Template scripting output language. The Template scripting output language is used to execute certain functions that convert values depending on the language such as `date` (see date), `date_in_words` (see date_in_words), `number` (see number), `number_in_words` (see number_in_words), `amount_in_words` (see amount_in_words), and

`format_date` (see format_date). The default output language can be set with KCM Core Administrator. Whenever you start a Master Template, KCM sets the output language to the value of this setting.

```
language_code ( language )
```

The result of this function is of type `TEXT`. This is the Template scripting output language selected.

This function has one parameter:

- `language`, required, type `TEXT`. One of the KCM or RFC 1766 language codes. For more information on the codes, see Supported KCM output languages.

Supported KCM output languages

The `language_code` function accepts both the KCM and RFC 1766 language codes presented in the following table. The `language_code` function always returns the KCM language code for the selected language.

The RFC 1766 language codes are case-insensitive.

ITP	RFC 1766	Language	Country
DAN	da-DK	Danish	Denmark
DES	de-CH	German	Switzerland
DEU	de-DE	German	Germany
ENG	en-GB	English	Great Britain
ENU	en-US	English	United States
ESP	es-ES	Spanish	Spain
FRA	fr-FR	French	France
FRB	fr-BE	French	Belgium
FRS	fr-CH	French	Switzerland
HUN	hu-HU	Hungarian	Hungary
ITS	it-CH	Italian	Italy
NLB	nl-BE	Dutch	Belgium
NLD	nl-NL	Dutch	The Netherlands
POL	pl-PL	Polish	Poland
PTG	pt-PT	Portuguese	Portugal
SVE	sv-SE	Swedish	Sweden

If support for the requested language could not be found, the current language is not changed, and the old language setting is returned.

Examples are provided here.

```
ASSIGN language := language_code( "ENG" )
ASSIGN language := language_code( "en-GB" )
```

Both calls to the `language_code` function select the English (Great Britain) language support. Both calls return the Template scripting language code ENG.

```
ASSIGN language := language_code ( "X-Query")
```

The below call retrieves the currently active language code by attempting to switch to a non-existing language.

The Polish and Hungarian languages use characters that are not present in the latin-1 character set. Therefore, the functions `uppercases`, `uppercase2`, and `lowercase2` may not always result in the desired output. If the `language_code` is set to HUN or POL, you should use the function `compare_characters` (see compare_characters) to compare two `TEXT`s, the function `fragment_of_characters` (see number_of_characters) to separate a fragment from a `TEXT`, and the function `number_of_characters` (see number_of_characters) to calculate the length of a string.

The function `date_in_words` (see date_in_words) is not supported for the Polish language.

The function `area_in_words` (see area_in_words) is not supported for the Hungarian language.

Support for the RFC 1766 language codes is introduced in KCM Core 4.4

headers

The `headers` function gives the script developer the ability to specify the headers for the current section of the document.

```
@(headers (first; odd; even))
```

The result of the `headers` function is of type `TEXT`.

This function has three parameters:

1. `first`, type `TEXT`. This is the content for the header on the first page. The content depends on the specific word processor you use to develop a Master Template.
2. `odd`, type `TEXT`. This is the content for the headers on odd numbered pages. The content depends on the specific word processor you use to develop a Master Template.
3. `even`, type `TEXT`. This is the content for the headers on even numbered pages. The content depends on the specific word processor you use to develop a Master Template.

The result of the function headers is of type `TEXT`. The output of this function has effect at the location where it is put into the result document.

If the result of multiple calls to the `headers` function is put in the same section, only the last result has impact on the section.

The functionality of the `headers` function is only available for KCM Microsoft Word output (both DOC and DOCX) and ignored for other document formats.

An example is provided here.

```
# BEGIN

#
@(headers ("First page header"; ""; "Even page header containing two lines of text.")
#
```

```
END #
```

The option "Different first page" in the Microsoft Word GUI controls whether the first page header is shown on the first page. If this option is disabled in a section, the first page header is not shown, though it is present in the result document.

The option "Different odd and even" in the Microsoft Word GUI applies to the complete document and is controlled by the setting from the Template script. It can be changed with the `pagestyle` function (see pagestyle) or the `pragma` function (see pagestyle).

All headers must end with a paragraph break. As Microsoft Word accept headers that do not end with a paragraph break, the behavior is undefined.

Tip You can disable a header by specifying an empty string for that header. This takes up the white space around the headers compared to a header that is empty. This feature is not available in the Microsoft Word GUI.

footers

The `footers` function gives the script developer the ability to specify the footers for the current section of the document.

```
@(footers (first; odd; even))
```

The result of the function footers is of type `TEXT`. This function has three parameters:

1. `first`, type `TEXT`. This is the content for the footer on the first page. The content depends on the specific word processor you use to develop a Master Template.
2. `odd`, type `TEXT`. This is the content for the footers on odd numbered pages. The content depends on the specific word processor you use to develop a Master Template
3. `even`, type `TEXT`. This is the content for the footers on even numbered pages. The content depends on the specific word processor you use to develop a Master Template.

The result of the function `footers` is of type `TEXT`. The output of this function has effect at the location where it is put into the result document.

If the result of multiple calls to the function `footers` is put in the same section, only the last result has impact on the section.

The functionality of the function footers is only available for KCM Microsoft Word output (both DOC and DOCX) and ignored for other document formats.

An example is provided here.

```
# BEGIN
#
@(footers ("First page footer"; ""; "Even page footer containing two lines of text.")
#
END #
```

The option "Different first page" in the Microsoft Word GUI controls whether the first page footer is shown on the first page. If this option is disabled in a section, the first page footer is not shown, even though it is present in the result document.

The option "Different odd and even" in the Microsoft Word GUI applies to the complete document and is controlled by the setting from the Template script. It can be changed with the `pagestyle` function (see pagestyle) or the `pragma` function (see pragma).

All footers must end with a paragraph break. As Microsoft Word accepts footers that do not end with a paragraph break, the behavior is undefined.

You can disable a footer by specifying an empty string for that footer. This takes up the white space around the footers compared to a footer that is empty. This feature is not available in the Microsoft Word GUI.

paper_types

The `paper_types` function gives the script developer the ability to specify paper settings for the current section of the document.

```
@(paper_types (first; other))
```

The result of the function `paper_types` is of type `TEXT`.

This function has two parameters.

1. `first`, type `TEXT`. This is the paper type for the first page of the section. The content depends on the specific word processor you use to develop a Master Template.
2. `other`, type `TEXT`. This is the paper type for the subsequent pages of the section. The content depends on the specific word processor you use to develop a Master Template.

The result of the function `paper_types` is of type `TEXT`. The output of this function takes effect at the location where it is put into the result document.

If the result of multiple calls to the function `paper_types` is put in the same section, only the last result has impact on the section.

The functionality of the function `paper_types` is only available for KCM Microsoft Word output (both DOC and DOCX) and ignored for other document formats.

An example is provided here.

```
# BEGIN
#
@(paper_types("2"; "3"))
#
END #
```

The function `paper_types` sets the paper tray selections for the current section. Both the first and other parameter must be decimal numbers. The exact values for these parameters depend on the target printer driver used to print the result document.

insert_image

The `insert_image` function inserts an image into the result document during Master Template execution. The image is read from the file system or passed as base-64 encoded data. Sizing and placement can be controlled through parameters.

```
@(insert_image (filename; x; y; width; height; ...))
```

The result of the function `insert_image` is of type `TEXT`.

This function has a variable number of parameters:

1. `filename`, type `TEXT`. This is the name of the file containing the image.
2. `x`, type `TEXT`; optional. This is the X-coordinate where the image should be placed on the page.
3. `y`, type `TEXT`; optional. This is the Y-coordinate where the image should be placed on the page.
4. `width`, type `TEXT`; optional; width of the image.
5. `height`, type `TEXT`; optional; height of the image.

The result of the function `insert_image` must be inserted in the result document to take effect.

In addition to the preceding parameters, the function `insert_image` has a number of optional parameters that can only be used if they are specified by name. The following is a list of the optional parameters:

1. `base64data`, type `TEXT`. It loads image data from a variable or Field.
2. `resource`, type `TEXT`. It loads image data from the Resources folder in KCM Designer.
3. `keepaspect`, type `TEXT`. It maintains aspect ratio when resizing the image. Allowed values are Y (default) or N.
4. `align`, type `TEXT`. It aligns the image relatively to the x and y coordinates.
5. `overtext`, type `TEXT`. It controls placement of the image relatively to the text. Allowed values are Y or N (default).
6. `wrap`, type `TEXT`. It determines if text should be wrapped around the image. Allowed values are none (default), square, and topbottom.
7. `dpi`, type `TEXT`. It overrides the DPI ratio of the image when calculating its size.
8. `resample`, type `TEXT`. It resamples the image before storing it in the Microsoft Word document.
9. `title`, type `TEXT`. Specifies a title for the image.
10. `description`, type `TEXT`. Specifies a description for the image. If this parameter is omitted, the file part of the `filename` parameter is used as the description.

For more information on the parameters, see Named parameters.

The functionality of the function `insert_image` is only available for Master Templates in the Microsoft Word DOCX output. In Quick Templates and Master Templates in other document formats, the use of this function is ignored.

The function `insert_image` is introduced in KCM version 4.4 and requires KCM Core 4.4 or later.

Named parameters

All parameters on the `insert_image` function can be specified by name using the notation `parameter_name:= value`. Named parameters must follow all positional parameters.

Examples are provided here.

The following four expressions, mixing positional, and named parameters are equivalent:

```
@(insert_image (image; "1cm"; "2cm"; w; h))
@(insert_image (filename:=image; x:="1cm"; y:="2cm"; width:=w; height:=h))
@(insert_image (image; "1cm"; "2cm"; width:=w; height:=h))
@(insert_image (height:=h; width:=w; y:="2cm"; x:="1cm"; filename:=image))
```

The following expressions are incorrect and cause compilation errors:

```
@(insert_image (filename:=image; "1cm"; "2cm"; w; h))
@(insert_image (image; "1cm"; "2cm"; w; h; filename:=image))
```

Supported image formats

The following image formats are supported by the `insert_image` function:

- BMP
- GIF
- JPEG
- PNG
- TIFF

Measurements and units

The function `insert_image` supports measurements in the following units:

- centimeters (cm)
- inches (in or "")
- points (pt)
- English Metrical Unit (emu) -- base unit internally used by Microsoft Word

The unit must always be explicitly specified. No spaces are allowed between number and unit. Fractions are always indicated with a decimal point. The following measurements are equivalent.

```
 2.54cm = 1.0in = 1" = 72pt =
914400emu
```

Note An empty string or a value of zero, such 0cm and 0in, is considered unspecified.

Image dimensions

KCM Core calculates the dimensions of an image automatically based on the properties of the image. This calculation can be modified by providing the width and/or height parameters to override the corresponding dimensions. The following requirements must be met:

- No dimensions are specified: the size is calculated based on the number of pixels and the DPI property of the image.
- One dimension is specified: the other dimension is calculated relative to the original dimensions of the image.
- Both dimensions are specified: the image is resized to fit within a box with the specified dimensions. Both dimensions are resized by the same scale to maintain the aspect ratio of the image.

The parameter `dpi` can be used to override the DPI property of the image if neither width nor height is specified.

The parameter `keepaspect` can be used to control the resizing of the image if both dimensions are specified. If the parameter is set to `N`, the image is resized to fill the box and may cause the image to stretch in one direction. If this parameter is set to `Y` or omitted, the aspect ratio of the image is maintained, and the image is resized to fill the bounding box optimally.

Examples are provided here.

```
 @(insert_image (logo))
```

This example uses the native dimensions of the image.

```
 @(insert_image (logo; width:="4cm";
height:="2cm"))
```

The preceding example resizes the image to fit within a 4cm x 2cm box. An image of 100x200 pixels would be resized to 1cm x 2cm.

```
 @(insert_image (logo; width:="4cm";
height:="2cm"; keepaspect:="N"))
```

The preceding example resizes and stretches the image to fill a 4cm x 2cm box. An image of 100x200 pixels would be stretched to 4cm x 2cm.

Positioning

If the `x` and `y` parameters are omitted or have empty values, the image is positioned inline with the text. If both `x` and `y` are specified, the image is positioned at the indicated position. If only `x` or `y` is specified, the image cannot be placed, and an error is reported.

By default, positioning is relative to the upper left corner of the page. This can be controlled by adding the following overrides to the parameter.

- *distance* or *distance;page* to position relative to the left/top of the page.
- *distance;margin* to position relative to the left/top margin of the text.

Examples are provided here.

```
[@(insert_image (image))]
```

The preceding example places the image inline in the paragraph between the brackets.

```
 @(insert_image (logo; x:="4cm";
y:="1cm;margin"))
```

The preceding example places the image `4 cm` from the left side of the page, 1 cm down from the top margin.

Image alignment

If an image has to be scaled to fit the provided dimensions, and the aspect ratio of the image does not match the dimensions, the image could end up being misaligned to the specified bounding box. The

`align` parameter can be used to dynamically calculate the placement of the image based on its effective dimensions and the specified coordinates. The following alignment options are available:

- `NW` (default) aligns the top-left corner of the image.
- `Center` aligns the center of the image.
- one of `N`, `NE`, `W`, `E`, `SW`, `S` or `SE` aligns to the corresponding cardinal corner of the image.

Examples are provided here.

```
 @(insert_image (logo; x:="2cm";
y:="2cm"))
```

This example places the image `2cm` from the top and left edges of the paper.

```
@(insert_image (logo; x:="2cm"; y:="2cm";
align:="center"))
```

The preceding example places the center of the image 2cm from the left and top margin of the paper.

```
@(insert_image (logo; x:="19cm"; y:="2cm";
align:="ne"))
```

The preceding example places the image `2cm` from the top and right edges of the paper, such as 21 cm wide, A4 sized.

Overlapping and text wrapping

Inline images are placed in line with the text and Microsoft Word resizes the line of text accordingly. Images that are explicitly positioned can overlap with other content on the page.

Images placed with the `insert_image` function are positioned in the order in which they are inserted. Every image potentially overlaps any image previously inserted through the `insert_image` function. Ordering relatively to images inserted through other means, such as manually inserted through the Microsoft Word GUI, fields, and so on, is undefined.

By default, images are placed below the text of the document. The `overtext` parameter can be set to `Y` to put the images on top of the text.

An example is provided here.

```
@(insert_image (watermark;
                width:="8.5in"; height:="11in";
                x:="4.25in"; y:="5.5in"; align:="center"
                overtext:="Y"))
```

The preceding example adds a centered image as a watermark overlay on top of a page, such as a US Letter sized page.

By default, text runs through the images. The wrap parameter can be set to determine how text should flow around the image.

- *none* (default) no wrapping -- text runs through the image.
- *topbottom* -- no text to the left and right of the image.
- *square* -- text wraps around the image. Space to the left and right of the image is filled with text.

An example is provided here.

```
@(insert_image (watermark;
                width:="2in"; height:="2in";
                x:="4.25in"; y:="5.5in"; align:="center"
                wrap:="square"))
```

The preceding example places a centered image on the page and wraps the text around it.

Images from the database or KCM Designer

The `insert_image` function can also be used to insert images from KCM Designer or base-64 encoded image data from variables that are obtained from the Data Backbone XML.

To insert an image from the Resources folder in KCM Designer, use the `resource` parameter. To insert base-64 encoded image data, use the `base64data` parameter. Using both parameters at the same time is not supported and results in an error.

An empty value for either parameter is ignored. If either the `resource` or the `base64data` parameter is used, and is not empty, the required `filename` parameter is ignored.

An example is provided here.

```
@(insert_image (filename:="signature";
resource:="CorporateLogo"))
```

This example inserts the Resource `"CorporateLogo"` from KCM Designer.

```
@(insert_image (filename:="signature";
 base64data:=_data.Corporate.Signature))
```

The preceding example inserts a signature from image data in a Field in the Data Backbone.

Title and description

The `title` parameter sets the title of the image. The `description` parameter sets the alternative text of the image. This text is used by assistive technologies, and it functions as a placeholder when the image is not displayed. If the `description` parameter is omitted or empty, the part of the `filename` parameter after the last slash or backslash is used.

An example is provided here.

```
 @(insert_image (logo; title := "Logo";
description := "The Northwind Corporate Logo"))
```

This example sets the title `"Logo"` and the description `"The Northwind Corporate Logo"` for the inserted image.

Image storage

KCM Core keeps track of repeatedly inserted images and stores each image only once in the result document. This is done regardless of the dimensions and placement parameters of the image.

If images are obtained from external sources or stored with excessively high resolutions, you can store a downsampled copy of the image with a resolution that better matches the purpose of the document. If the document is processed further, such printed or converted to PDF, you should include the original image.

The `resample` parameter controls how the image is stored.

- `""` / omitted : the image is stored without modifications.
- `number` or `number,png` : the image is downsampled to the indicated resolution and stored in the PNG format.
- `number;jpeg` : the image is downsampled to the indicated resolution and stored in the JPEG format.

The PNG image format uses lossless compression but is less efficient than the JPEG compression. You should not recompress JPEG images as this can result in recompression artifacts and significant quality loss.

Images are never upsampled. If the image is rendered at a resolution below the resample value, this option is ignored and the original image is stored.

An example is provided here.

```
 @(insert_image (watermark;
resample:="600;jpeg"))
```

The preceding example includes the image and downsamples it to a 600 DPI JPEG file.

insert_signature

The function `insert_signature` inserts a signature line object in the result document during the Master Template execution. The properties of this object, such as signer, title, and so on, can be set through parameters.

```
 @(insert_signature ("John Doe"; title :=
"Director"; email := "J.Doe@example.com"; ...))
```

The result of the function `insert_signature` is of type `TEXT`. This function has a variable number of parameters.

1. `signer`, type `TEXT`. This is the name of the person who is expected to sign.

The result of the function `insert_signature` must be inserted in the result document to have effect.

In addition to the preceding parameter, the function `insert_signature` has a number of optional parameters which can only be used if they are specified by name:

1. `title`, type `TEXT`. Sets the title of the signer.
2. `email`, type `TEXT`. Sets the email address of the signer.
3. `instructions`, type `TEXT`. Sets instructions for the signer.
4. `allow_comments`, type `TEXT`. Gives the signer the ability to add comments when signing.
5. `show_date`, type `TEXT`. Shows the sign date in the signature line.
6. `width`, type `TEXT`. Width of the signature line.
7. `height`, type `TEXT`. Height of the signature line.

Note The `email` parameter supports plain text values only. Hyperlinks are not supported.

The functionality of the function `insert_signature` is available only for Master Templates in the Microsoft Word DOCX output. In Quick Templates and Master Templates in other document formats, the use of this function is ignored.

The function `insert_signature` is introduced in KCM version 4.4 and requires KCM Core 4.4 or higher.

Measurements and units

The function `insert_signature` supports measurements in a number of units:

- centimeters (cm)
- inches (in or "")
- points (pt)
- English Metrical Unit (emu) -- base unit internally used by Microsoft Word

The unit must always be explicitly specified. No spaces are allowed between number and unit. Fractions are always indicated with a decimal point. The following measurements are equivalent.

```
 2.54cm = 1.0in = 1" = 72pt =914400emu
```

Note An empty string or a value of zero, such as 0cm, 0in, is considered unspecified.

Sizing

KCM Core adjusts the dimensions of the signature line automatically based on the following Microsoft Word default properties:

- No dimensions are specified: the size is set to 2.67in x 1.33in.
- One dimension is specified: the other dimension is calculated relative to the default dimensions.
- Both dimensions are specified: the image is resized to fit within a box with the specified dimensions.

Examples are provided here.

```
 @(insert_signature (signer))
```

The preceding example uses the native dimensions of in the image.

```
 @(insert_signature (signer; width:="4cm";
height:="2cm"))
```

The preceding example resizes the signature line to fit within a 4cm x 2cm box.

```
 @(insert_signature (signer;
width:="2in"))
```

The preceding example resizes the signature line to fit within a 2in x 1in box.

Arrays functions

The following section contains information on the arrays function that you can use to sort arrays and calculate the number of elements in these arrays.

length_text_array

The `length_text_array` function calculates the length of an array of type `TEXT`. The length of an array is either the array length in the initial declaration of the array or, if the array has more elements than specified in the initial declaration, the length is the actual number of elements in the array.

```
length_text_array ( input_array )
```

The result of this function is of type `NUMBER` and is the number of elements in `input_array`.

The function has one parameter:

- `input_array`, type `TEXT`. This is the array with length to be determined.

An example is provided here.

```
ARRAY TEXT test_array [1]
ASSIGN test_array [1] := "This is the first value"
ASSIGN test_array [2] := "This is the second value"
ASSIGN test_array [3] := "This is the third value"
ASSIGN test_array [4] := "This is the fourth value"
ASSIGN test_array [5] := "This is the fifth value"
NUMBER function_result

ASSIGN length_function_result := length_text_array (test_array)

#
@(function_result)
#
```

If you refer to an array element of an array that is filled beyond its initial length, it results in the creation of that element. The element is created with a default value: 0 for `NUMBER`, "" for `TEXT`, and `FALSE` for `BOOL`. For example, the test `IF testing[1] THEN FI` will result in the creation of element 1 in array testing.

length_number_array

The `length_number_array` function calculates the length of an array of type `NUMBER`. The length of an array is either the array length in the initial declaration of the array or, if the array has more elements than specified in the initial declaration, the length is the actual number of elements in the array.

```
 length_number_array ( input_array )
```

The result of this function is of type `NUMBER` and is the number of elements in `input_array`.

The function has one parameter:

- `input_array`, type `NUMBER`. This is the array with length to be determined.

An example is provided here.

```
ARRAY NUMBER test_array [1]

ASSIGN test_array [1] := 12
ASSIGN test_array [2] := 38
ASSIGN test_array [3] := 60
ASSIGN test_array [4] := 77
ASSIGN test_array [5] := 91
NUMBER function_result
```

```
ASSIGN function_result := length_number_array (test_array)

#
@(function_result)
#
```

If you refer to an array element of an array that is filled beyond its initial length, it results in the creation of that element. The element is created with a default value: 0 for `NUMBER`, "" for `TEXT`, and `FALSE` for `BOOL`. For example, the test `IF` testing[1] `THEN FI` will result in the creation of element 1 in array testing.

length_bool_array

The `length_bool_array` function calculates the length of an array of type `BOOL`. The length of an array is either the array length in the initial declaration of the array or, if the array has more elements than specified in the initial declaration, the length is the actual number of elements in the array.

```
 length_bool_array ( input_array )
```

The result of this function is of type `NUMBER`; the number of elements in `input_array`.

The function has one parameter:

- `input_array`, type `BOOL`. This is the array with length to be determined.

An example is provided here.

```
ARRAY BOOL test_array [1]
ASSIGN test_array [1] := TRUE
ASSIGN test_array [2] := FALSE
ASSIGN test_array [3] := TRUE
ASSIGN test_array [4] := TRUE
ASSIGN test_array [5] := FALSE

NUMBER function_result

ASSIGN function_result := length_bool_array (test_array)

#
@(function_result)
#
```

If you refer to an array element of an array that is filled beyond its initial length, it results in the creation of that element. The element is created with a default value: 0 for `NUMBER`, "" for `TEXT`, and `FALSE` for `BOOL`. For example, the test `IF` testing[1] `THEN FI` will result in the creation of element 1 in array testing.

sort_text_array

The `sort_text_array` function sorts the elements of a text array and stores the sorted elements in a second array.

```
 sort_text_array ( input;output;number of
elements to sort )
```

The function `sort_text_array` is of type `BOOL`.

The function has three parameters:

1. `input`; `ARRAY` of type `TEXT`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `TEXT`. This array receives the sorted elements.
3. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_text_array` returns `TRUE` if the array was sorted successfully, and `FALSE` in one of the following conditions:

- If number of elements to be sorted is negative or 0.
- If the function `sort_text_array` fails the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function `sort_text_array` performs an alphabetical sort on the text elements using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause unexplainable orderings.

An example is provided here.

```
ARRAY TEXT input [4]
ARRAY TEXT output[4]
NUMBER i
ASSIGN input[1] := "c"
ASSIGN input[2] := "d"
ASSIGN input[3] := "a"
ASSIGN input[4] := "b"
IF sort_text_array (input; output; 4) THEN
  FOR i FROM 1 UPTO 4
DO
#
@(i) @(output[i])
#
OD

ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_text_array` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

Note The setting EnhancedUnicodeSupport in KCM Core Administrator can be used to map the function `sort_text_array` automatically to the function `sort_text_array_characters` (see sort_text_array_characters). For more information, see Enhanced Unicode Support.

sort_text_array_characters

The function `sort_text_array_characters` sorts the elements of a text array and stores the sorted elements in a second array.

```
sort_text_array_characters ( input;output;number
of elements to sort )
```

The function `sort_text_array_characters` is of type `BOOL`.

The function has three parameters:

1. `input`; `ARRAY` of type `TEXT`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `TEXT`. This array receives the sorted elements.
3. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_text_array_characters` returns `TRUE` if the array was sorted successfully, and `FALSE` in one of the following cases:

- If number of elements to sort is negative or 0.
- If the function `sort_text_array_characters` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array.
- If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified. The function `sort_text_array_characters` performs a culture-neutral alphabetical sort on the text elements using Unicode codepoints.

Note All Microsoft Word instructions in the text elements are included in the sort and sort before regular characters.

An example is provided here.

```
ARRAY TEXT input [4]
ARRAY TEXT output[4]
NUMBER i

ASSIGN input[1] := "c"
ASSIGN input[2] := "d"
ASSIGN input[3] := "a"
ASSIGN input[4] := "b"

IF sort_text_array_characters (input; output; 4) THEN
  FOR i FROM 1 UPTO 4
DO
#
@(i) @(output[i])
#
OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type BOOL of the sort_text_array_characters function is used to implement error handling. If the sort succeeds, the return value of the function is TRUE, the IF statement becomes TRUE, and the THEN part is executed. If the return is FALSE, the ELSE part of the IF statement is executed.

sort_text_array_index

The function sort_text_array_index sorts the indices of a text array based on the elements in the array.

```
sort_text_array_index ( input;output;number of elements to sort )
```

The function sort_text_array_index is of type BOOL.

It has three parameters:

1. input; ARRAY of type TEXT. This array contains the elements that must be sorted.
2. output; ARRAY of type NUMBER. This array receives the sorted index.
3. number of elements to sort; NUMBER. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a NUMBER variable or expression.

The function sort_text_array_index returns TRUE if the array was sorted successfully, and FALSE if number of elements to be sorted is negative or 0.

If the function sort_text_array_index fails, the output array is unmodified.

The input array is not modified by the function. The output array receives the sorted index from the input array.

If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

The function sort_text_array_index performs an alphabetical sort on the text elements using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause unexplainable orderings.

An example is provided here.

```
ARRAY TEXT input[4]
ARRAY NUMBER index[4]
NUMBER i

ASSIGN input[1] := "c"
ASSIGN input[2] := "d"
ASSIGN input[3] := "a"
ASSIGN input[4] := "b"

IF sort_text_array_index (input; index; 4) THEN
  FOR i FROM 1 UPTO 4 DO
#
@(i) @(input [ index[i] ])
#
  OD
ELSE
#
```

```
Failed to sort the input.
#
  STOP
FI
```

This example produces the same result as the example included with the function `sort_text_array` (see sort_text_array). A sorted list is produced by using the sorted index stored in the output array to read the values from the input array.

The type `BOOL` of the `sort_text_array_index` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

Note The setting EnhancedUnicodeSupport in KCM Core Administrator can be used to map the function `sort_text_array` automatically to the function `sort_text_array__index_characters` (see sort_text_array_index_characters). For more information, see Enhanced Unicode Support.

sort_text_array_index_characters

The function `sort_text_array_index_characters` sorts the indices of a text array based on the elements in the array.

```
sort_text_array_index_characters (input;output;number of elements to sort )
```

The function `sort_text_array_index_characters` is of type `BOOL`.

It has three parameters:

1. `input`; `ARRAY` of type `TEXT`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `NUMBER`. This array receives the sorted index.
3. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_text_array_index_characters` returns `TRUE` if the array was sorted successfully, and `FALSE` if number of elements to sort is negative or 0.

If the `sort_text_array_index_characters` function fails, the output array is unmodified.

The input array is not modified by the function.

The output array receives the sorted index from the input array.

If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

The `sort_text_array_index_characters` function performs a culture-neutral alphabetical sort on the text elements using Unicode codepoints.

Note All Microsoft Word instructions in the text elements are included in the sort and sort before regular characters.

An example is provided here.

```
ARRAY TEXT input[4]
```

```
ARRAY NUMBER index[4]
NUMBER i

ASSIGN input[1] := "c"
ASSIGN input[2] := "d"
ASSIGN input[3] := "a"
ASSIGN input[4] := "b"

IF sort_text_array_index_characters (input; indices; 4) THEN
  FOR i FROM 1 UPTO 4 DO
#
@(i) @(input [ index[i] ])
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

This example produces the same result as the example included with the function `sort_text_array_characters` (see sort_text_array_index_characters). A sorted list is produced by using the sorted index stored in the output array to read the values from the input array.

The type `BOOL` of the `sort_text_array_index` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

sort_number_array

The function `sort_number_array` sorts a number array. The sorted elements are stored in a second array.

```
 sort_number_array ( input;output;number of elements to be sorted )
```

The `sort_number_array` function is of type `BOOL`.

The function has three parameters:

1. `input`; `ARRAY` of type `NUMBER`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `NUMBER`. This array receives the sorted elements.
3. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_number_array` returns `TRUE` if the array was sorted successfully, and `FALSE` if number of elements to be sorted is negative or zero.

If the `sort_number_array` function fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

An example is provided here.

```
ARRAY NUMBER input [4]
ARRAY NUMBER output[4]
NUMBER i
```

```
ASSIGN input[1] := 4
ASSIGN input[2] := 2
ASSIGN input[3] := 1
ASSIGN input[4] := 3
IF sort_number_array (input; output; 4) THEN
  FOR i FROM 1 UPTO 4

DO
#
@(i) @(output[i])
#
OD

ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_number_array` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

sort_number_array_index

The function `sort_number_array_index` sorts the index of a number array based on the elements in the array.

```
 sort_number_array_index ( input;output;number of elements to sort )
```

The `sort_number_array_index` function returns `TRUE` (success) or `FALSE` (failure). The function is of type `BOOL`.

The function has three parameters:

1. `input`; `ARRAY` of type `NUMBER`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `NUMBER`. This array receives the sorted index.
3. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_number_array_index` returns `TRUE` if the array was sorted successfully, and `FALSE` if number of elements to be sorted is negative or zero.

If the function `sort_number_array_index` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted index from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

An example is provided here.

```
ARRAY NUMBER input[4]
ARRAY NUMBER indices[4]
NUMBER i
ASSIGN input[1] := 4
ASSIGN input[2] := 2
```

```
ASSIGN input[3] := 1
ASSIGN input[4] := 3
IF sort_number_array_index (input; indices; 4) THEN
  FOR i FROM 1 UPTO 4 DO
#
@(i) @(input [ indices[i] ])
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

This example produces the same result as the example included with the function `sort_number_array` (see sort_number_array). A sorted list is produced by using the sorted index stored in the output array to read the values from the input array.

The type `BOOL` of the `sort_number_array` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

Maps functions

The following section contains information on the maps functions that you can use to get the length of a map, to get the keys of a map, and to retrieve the keys used in a map.

length_text_map

The `length_text_map` function yields the number of elements contained in a `TEXT MAP`.

```
 length_text_map ( MAP_to_be_counted )
```

This function returns a `NUMBER`, which is the number of elements stored in the map.

This function has one parameter:

- `MAP to be counted`, `MAP` of type `TEXT`. A `MAP` of which the number of elements is counted.

An example is provided here.

```
MAP TEXT test_map

ASSIGN test_map ["first"] := "This is the first value"
ASSIGN test_map ["second"] := "This is the second value"
ASSIGN test_map ["third"] := "This is the third value"
ASSIGN test_map ["fourth"] := "This is the fourth value"
ASSIGN test_map ["fifth"] := "This is the fifth value"
NUMBER length_function_result

ASSIGN length_function_result := length_text_map (test_map)

#
@(length_function_result)
#
```

If you refer to a MAP element, it results in the creation of that element. The element is created with a default value: 0 for NUMBER, "" for TEXT, and FALSE for BOOL. For example, the test IF testing[1] THEN FI will result in the creation of element 1 in MAP testing.

length_number_map

The length_number_map function yields the number of elements contained in a NUMBER MAP.

```
length_number_map ( MAP_to_be_counted)
```

The function returns a NUMBER, which is the number of elements stored in the map.

This function has one parameter:

- MAP_to_be_counted, MAP of type NUMBER. A MAP whose number of elements is to be counted.

An example is provided here.

```
MAP NUMBER test_map

ASSIGN test_map ["first"] := 12345
ASSIGN test_map ["second"] := 67890
ASSIGN test_map ["third"] := 9876
ASSIGN test_map ["fourth"] := 54321
ASSIGN test_map ["fifth"] := 65748
NUMBER length_function_result

ASSIGN length_function_result := length_number_map (test_map)

#
@(length_function_result)
#
```

If you refer to an MAP element, it results in the creation of that element. The element is created with a default value: 0 for NUMBER, "" for TEXT, and FALSE for BOOL. For example, the test IF testing[1] THEN FI will result in the creation of element 1 in MAP testing.

length_bool_map

The length_bool_map function yields the number of elements contained in a BOOL MAP.

```
length_bool_map ( MAP_to_be_counted )
```

The function returns NUMBER, which is the number of elements stored in the map.

This function has one parameter:

- MAP_to_be_counted, MAP of type BOOL. A MAP of which the number of elements is counted.

An example is provided here.

```
MAP BOOL test_map

ASSIGN test_map ["first"] := TRUE
ASSIGN test_map ["second"] := FALSE
ASSIGN test_map ["third"] := TRUE
ASSIGN test_map ["fourth"] := TRUE
ASSIGN test_map ["fifth"] := FALSE
NUMBER length_function_result
```

```
ASSIGN length_function_result := length_bool_map (test_map)

#
@(length_function_result)
#
```

If you refer to an MAP element, it results in the creation of that element. The element is created with a default value: 0 for NUMBER, "" for TEXT, and FALSE for BOOL. For example, the test IF testing[1] THEN FI will result in the creation of element 1 in MAP testing.

get_keys_text_map

The get_keys_text_map function retrieves the index values of all elements in a MAP and stores them in an array. The index values are stored in the array in random order.

```
 get_keys_text_map ( input_map;result_array)
```

This function returns a NUMBER, which is the number of elements stored in the array.

This function has two parameters:

1. input_map is a MAP of type TEXT. The MAP whose index values you need.
2. result_array is an array of type TEXT. The array with index values to be stored.

An example is provided here.

```
MAP TEXT input_map

ASSIGN input_map ["first"] := "This is the first value"
ASSIGN input_map ["second"] := "This is the second value"
ASSIGN input_map ["third"] := "This is the third value"
ASSIGN input_map ["fourth"] := "This is the fourth value"
ASSIGN input_map ["fifth"] := "This is the fifth value"

ARRAY TEXT result_array[1]
NUMBER number_of_elements

ASSIGN number_of_elements := get_keys_text_map (input_map;result_array)

NUMBER counter
FOR counter
 FROM 1
 UPTO number_of_elements
DO
#
The key @(result_array[counter]) contains @(input_map[result_array[counter]]) in the
 map.
#
OD
```

get_keys_number_map

The get_keys_number_map function retrieves the index values of all elements in a MAP and stores them in an array. The index values are stored in the array in random order.

```
 get_keys_number_map ( input_map;result_array)
```

This function returns a NUMBER, which is the number of elements stored in the array.

This function has two parameters:

1. input_map is a MAP of type NUMBER. The MAP with the index values you need.
2. result_array is an array of type TEXT. The array with index values to be stored.

An example is provided here.

```
MAP NUMBER input_map

ASSIGN input_map ["first"] := 12345
ASSIGN input_map ["second"] := 67890
ASSIGN input_map ["third"] := 9876
ASSIGN input_map ["fourth"] := 54321
ASSIGN input_map ["fifth"] := 65748
ARRAY TEXT result_array[1]

NUMBER number_of_elements

ASSIGN number_of_elements := get_keys_number_map (input_map;result_array)

NUMBER counter
FOR counter
 FROM 1
 UPTO number_of_elements
DO
#
The key @(result_array[counter]) contains @(input_map[result_array[counter]]) in the
 map.
#
OD
```

get_keys_bool_map

The function get_keys_bool_map retrieves the index values of all elements in a MAP and stores them in an array. The index values are stored in the array in random order.

```
get_keys_bool_map ( input_map;result_array)
```

This function returns a NUMBER, which is the number of elements stored in the array.

This function has two parameters:

1. input_map is a MAP of type BOOL. The MAP with the index values.
2. result_array is an array of type TEXT. The array with index values to be stored.

An example is provided here.

```
MAP BOOL input_map
ASSIGN input_map ["first"] := TRUE
ASSIGN input_map ["second"] := FALSE
ASSIGN input_map ["third"] := TRUE
ASSIGN input_map ["fourth"] := TRUE
ASSIGN input_map ["fifth"] := FALSE
ARRAY TEXT result_array[1]

NUMBER number_of_elements
```

```
ASSIGN number_of_elements := get_keys_bool_map (input_map;result_array)

NUMBER counter
FOR counter
 FROM 1
 UPTO number_of_elements
DO
#
The key @(result_array[counter]) contains @(input_map[result_array[counter]]) in the
 map.
#
OD
```

key_used_in_text_map

The function `key_used_in_text_map` tests if a certain key is present in a map of type `TEXT`, without creating the corresponding element.

```
key_used_in_text_map ( key_to_find; map_name)
```

The result of the function is type `BOOL`. The function yields `TRUE` if the map contains an element map[key].

The function `key_used_in_text_map` has two parameters:

1. `key_to_find` is of type `TEXT`. The key to be detected in the map.
2. `map_name` is of type `MAP TEXT`. The map to be tested for the presence of the `key_to_find`.

An example is provided here.

```
 BOOL key_found := key_used_in_text_map ( "a_key"; mymap )
```

The index of the map `mymap` will be searched for the key `a_key`. If it is found, the variable `key_found` will be set to `TRUE`.

key_used_in_number_map

The `key_ised_in_number_map` function tests if a certain key is present in a map of type `NUMBER`, without creating the corresponding element.

```
 key_used_in_number_map ( key_to_find; map_name)
```

The result of the function is type `BOOL`. The function yields `TRUE` if the map contains an element map[key].

The function `key_used_in_number_map` has two parameters:

- `key_to_find` is of type `TEXT`. The key to be detected in the map.
- `map_name` is of type `MAP NUMBER`. The map to be tested for the presence of the `key_to_find`.

An example is provided here.

```
BOOL key_found := key_used_in_number_map ("a_key" ; mymap )
```

The index of the map `mymap` index will be searched for the key `a_key`. If it is found, the variable `key_found` will be set to `TRUE`.

key_used_in_bool_map

The `key_used_in_bool_map` function tests if a certain key is present in a map of type `BOOL`, without creating the corresponding element.

```
key_used_in_bool_map ( key_to_find; map_name)
```

The result of the function is type `BOOL`. The function yields `TRUE` if the map contains an element map[key].

The function `key_used_in_bool_map` has two parameters:

1. `key_to_find` is of type `TEXT`. The key to be detected in the map.
2. `map_name` is of type `MAP BOOL`. The map to be tested for the presence of the `key_to_find`.

An example is provided here.

```
BOOL key_found := key_used_in_bool_map ( "a_key"; mymap )
```

The index of the map `mymap` will be searched for the key `a_key`. If it is found, the variable `key_found` will be set to `TRUE`.

Text Blocks functions

insert_text_block

The keyword `insert_text_block` is deprecated. Use the statement `TEXTBLOCK` instead (for more information, see TEXTBLOCK statement).

Use this function to insert predefined Text Blocks into a result document. See the *Kofax KCM Designer online Help* for more information on how to use Text Blocks.

The function `insert_text_block` has one parameter of type `TEXT`. This parameter contains the label of the Text Block that needs to be inserted. The result of the function is of type `TEXT` and is the text of the predefined Text Block.

If the Text Blocks set that the Text Block belongs to has an associated Field Set, you need to assign values to all Fields from this Field Set using a variable of type `FIELDSET`. Also, do so if the Text Block inserted into the result document does not use any of these Fields.

The layout of the Text Block is determined by Microsoft Word styles. You need to define the correct styles. For more information on how to define Text Blocks styles, see TEXTBLOCK statement.

An example is provided here.

```
@(insert_text_block("Clauses"))
```

This example shows how the Text Block with label `"Clauses"` is inserted into the result document.

Note To run a Master Template containing this feature, you need KCM Core or KCM ComposerUI Server.

insert_text_block_extended

The keyword `insert_text_block` is deprecated. Use the statement `TEXTBLOCK` instead (for more information, see TEXTBLOCK statement).

The function `insert_text_block_extended` is an extension to the `insert_text_block` function (see insert_text_block). The `insert_text_block` function is used if Text Blocks need to be inserted in the same result document using different styles and if a Text Block needs to be inserted in a table cell.

The result of the function is of type `TEXT` ans is the text of the predefined Text Block.

The function has three parameters of type `TEXT`:

1. The first parameter contains the label of the Text Block that needs to be inserted.
2. The second parameter contains the prefix of the styles set that needs to be used to layout the Text Block. If this parameter is left empty, KCM searches for a styles set with the KCM prefix. For more information on the layout of Text Blocks, see TEXTBLOCK statement.
3. The third parameter contains the paragraph sign assigned to the Text Block. You need to pass a paragraph sign declared within a table cell to insert a Text Block in a table cell. Using the wrong paragraph sign to insert a Text Block in a table cell may result in a corrupted Microsoft Word document. This paragraph can be left empty if the Text Block is not inserted in a table cell.

If the Text Blocks set the Text Block belongs to has an associated Field Set, you need to assign values to all Fields from this Field Set using a variable of type `FIELDSET` (see FIELDSET). Also, do so if the Text Block inserted into the result document does not use any of these Fields.

An example is provided here.

```
@(insert_text_block_extended("Clauses"; "TST";""))
```

This example shows how the Text Block with the label `"Clauses"` is inserted in the result document. The styles setused has the prefix `"TST"`.

Another example is provided here.

```
TEXT par := ""
@ (insert_text_block_extended("Clauses"; ""; par))
```

The above example shows how the Text Block with the label `"Clauses"` is inserted in the result document. The function needs to be called within a table cell as the paragraph sign passed in the third parameter is declared within a table cell.

Note To run a Master Template containing this feature, you need KCM Core or KCM Repository Server.

get_fields_from_text_block

The function `get_fields_from_text_block` retrieves a list of Fields that are used in a Text Block.

```
get_fields_from_text_block (Text block;list)
```

The function `get_fields_from_text_block` is of type `NUMBER`.

The function has two parameters:

1. The parameter `Text Block` is of type `TEXT`. This is the name of the Text Block with Fields to be retrieved.
2. The parameter list `ARRAY` of type `TEXT`. This array receives a list of Fields.

The function `get_fields_from_text_block` returns the number of unique Fields found in the Text Block. This can be 0 if a Text Block does not Fields. The output array receives the names of the Fields in the format FieldSet.Field. Each field is present only once. The order of these Fields is not specified. If the output array is larger than the number of Fields, those elements outside the range are left unmodified.

An example is provided here.

```
ARRAY TEXT list[0]
NUMBER i
NUMBER n := get_fields_from_text_block ("My Text Block"; list)
#
Found @(numerals(n)) fields:
#
FOR i FROM 1 UPTO n DO
#
@(i) @(list[i])
#
OD
```

text_block_exists

The function `text_block_exists` tests if a Text Block exists in the KCM Designer and has a revision that can be used in a Master Template.

```
text_block_exists (text_block;reserved)
```

The function `text_block_exists` is of type `BOOL`.

The function has two parameters:

1. The parameter `text_block` is of type `TEXT`. This is the name of the Text Block which is searched for in KCM Designer.
2. The parameter reserved is of type `NUMBER`. This parameter is reserved for future use and must always be set to 0.

The function `text_block_exists` returns `TRUE` if the Text Block exists in KCM Designer and has a suitable revision that can be used in a Master Template. The function returns `FALSE` if the Text Block does not exist or if there is no such revision.

Note The function `text_block_exists` only checks if a Text Block can be found in KCM Designer. It does not validate if Field Sets are available in the Master Template. The function `get_fileds_from_text_block` (see get_fields_from_text_block) can be used to query the Fields used in the Text Block to perform such validation in the Master Template.

An example is provided here.

```
TEXT block := "BuildingBlocks\" + Customer.Language + "\Salutation"
```

```
IF text_block_exists (block; 0) THEN
  TEXTBLOCK
      NAME block
ELSE
  TEXTBLOCK
      NAME "BuildingBlocks\ENG\Salutation"
FI
```

The preceding example shows how a Text Block `'Salutation'` based on the language contained in the `Customer.Language` Field. If no Text Block is available for this language, it defaults to the language `'ENG'`.

import_text_block

The function `import_text_block` imports a stored XML Text Block structure so that it can be used in the Master Template run.

```
import_text_block ( "<?xml ... >")
```

The result of this function is a value of type `TEXT` that contains a reference to a Text Block. This Text Block reference can be used in the parameter `VAR` of the statement `TEXTBLOCK` or as the default value of an `EDITABLE_TEXTBLOCK` question in a `FORM`. The Text Block reference is in an unspecified format, and it is valid only during the current Master Template run.

The parameter to the function `import_text_block` should be an XML representation of a Text Block that was retrieved from a KCM Core XML metadata file. When KCM Core completes a Master Template run, all answers to `FORM` questions are stored in an XML metadata file including the answers to `EDITABLE_TEXTBLOCK` questions. For more information on KCM Core XML metadata, see the *Kofax Communications Manager Core Developer's Guide*.

get_text_blocks_in_view

The function `get_text_blocks_in_view` retrieves a list of Text Blocks in a Text Block List.

```
get_text_blocks_in_view (view; list)
```

The function `get_text_blocks_in_view` is of type `NUMBER`.

The function has two parameters:

1. The parameter view is of type `TEXT`. This is the name of the Text Block List with Text Blocks to be retrieved.
2. The parameter list: `ARRAY` is of type `TEXT`. This array receives the list of Text Blocks.

The function `get_text_blocks_in_view` returns the number of Text Blocks found in the Text Block List. This can be 0 if the Text Block List is empty. The output array receives the names of the Text Blocks in the format Project.TextBlock. The order of the names matches the order in the Text Block List. If the output array is larger than the number of Text Blocks, the elements outside the range are left unmodified.

An example is provided here.

```
ARRAY TEXT list[0]
NUMBER i
NUMBER n := get_text_blocks_in_view ("My View"; list)
#
```

```
Found @(numerals(n)) Text Blocks:
#
FOR i FROM 1 UPTO n DO
#
@(i) @(list[i])
#
OD
```

read_text_block_from_file

The `read_text_block_from_file` function imports a Text Block from an XML file on the file system.

```
FUNC TEXT read_text_block_from_file (CONST TEXT file)
```

The function `read_text_block_from_file` is of type `TEXT`.

The function has one parameter:

- The parameter file is of type `TEXT`. This is the name of the file that contains the XML.

The function `read_text_block_from_file` returns a Text Block name that can be used by the Master Template to present it for editing and to produce the content in the output document. This name is only valid for the current Master Template run and should not be stored anywhere.

import_text_block_base64

The `import_text_block_base64` imports a base64 encoded Text Block XML from a variable. This function allows Text Blocks to be imported from external sources such as a database or XML resource.

```
FUNC TEXT import_text_block_base64 (CONST TEXT base64)
```

The function `import_text_block_base64` is of type `TEXT`.

The function has one parameter:

- The parameter file is of type `TEXT`. This is the base-64 encoded representation of a Text Block.

The function `import_text_block_base64` returns a Text Block name that can be used by the Master Template to present it for editing and to produce the content in the output document. This name is only valid for the current Master Template run and should not be stored anywhere.

Field Sets functions

clear_fieldset

You can use the `clear_fieldset` function to remove all assigned fields from a `FIELDSET`.

```
clear_fieldset ( fieldset )
```

The function has one parameter:

- `field set`, type `FIELDSET`. This is the Field Set to be cleared.

The result of the function is type `BOOL`. If the Field Set was cleared successfully, the function returns `TRUE`; otherwise, it returns `FALSE`.

Data Structures functions

The following topics provide a description of the functions that only work with data structures and their members.

Arrays functions

The following section describes the arrays functions used to sort arrays and calculate the number of elements in these arrays.

length_fieldset_array

The `length_fieldset_array` function calculates the length of an array of type `FIELDSET`. The length of an array is the actual number of elements in the array.

```
length_fieldset_array ( input_array )
```

The result of this function is of type `NUMBER` and is the number of elements in the `input_array`.

The function has one parameter:

- `input_array`, type `FIELDSET`. This is the array with length to be determined.

An example is provided here.

```
DATASTRUCTURE DS
  BEGIN
    ARRAY FIELDSET Test_array [0]
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Test_array [1].Field := "TRUE"
ASSIGN struct.Test_array [2].Field := "FALSE"
ASSIGN struct.Test_array [3].Field := "TRUE"
ASSIGN struct.Test_array [4].Field := "TRUE"
ASSIGN struct.Test_array [5].Field := "FALSE"

NUMBER function_result

ASSIGN function_result := length_fieldset_array (struct.Test_array)

#
@(function_result)
#
```

If you refer to an array element of an array that is beyond its initial length, it results in the creation of that element. The created element is empty.

length_struct_array

The `length_struct_array` function calculates the length of an array of type `DATASTRUCTURE`. The length of an array is the actual number of elements in the array.

```
length_struct_array ( input_array )
```

The result of this function is of type `NUMBER` and is the number of elements in `input_array`.

The function has one parameter:

- `input_array`, type `DATASTRUCTURE`. This is array with length to be determined.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Field
  END

DATASTRUCTURE DS
  BEGIN
    ARRAY EL Test_array [0]
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Test_array [1].Field := "TRUE"
ASSIGN struct.Test_array [2].Field := "FALSE"
ASSIGN struct.Test_array [3].Field := "TRUE"
ASSIGN struct.Test_array [4].Field := "TRUE"
ASSIGN struct.Test_array [5].Field := "FALSE"

NUMBER function_result

ASSIGN function_result := length_struct_array (struct.Test_array)

#
@(function_result)
#
```

If you refer to an array element of an array that is beyond its initial length, it results in the creation of that element. The created element is empty.

sort_fieldset_array

The function `sort_fieldset_array` sorts the elements of a `FIELDSET` array and stores the sorted elements in a second array.

```
sort_fieldset_array ( input;output;field;number of elements to sort )
```

The function `sort_fieldset_array` is of type `BOOL` and has four parameters:

1. `input`; `ARRAY` of type `FIELDSET`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `FIELDSET`. This array receives the sorted elements.
3. `field`; `TEXT`. This is the `Field` in the `FIELDSET` that is used as the key to sort the array. If this `Field` is not present in the `FIELDSET`, it is considered empty.

4. number of elements to sort; NUMBER. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a NUMBER variable or expression.

The function sort_fieldset_array returns TRUE if the array was sorted successfully, and FALSE in one of the following conditions:

- If the number of elements parameter to be sorted is negative or 0.
- If the function sort_fieldset_array fails the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function sort_fieldset_array performs an alphabetical sort on the text elements using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause illogical orderings.

An example is provided here.

```
DDATASTRUCTURE DS
  BEGIN
    ARRAY FIELDSET Array[0]
  END

DECLARE input DEFINED_AS DS
DECLARE output DEFINED_AS DS

NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_fieldset_array (input.Array; output.Array; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[i].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type BOOL of the sort_filedset_array function is used to implement error handling. If the sort succeeds, the return value of the function is TRUE, the IF statement becomes TRUE, and the THEN part is executed. If the return is FALSE, the ELSE part of the IF statement is executed.

Note The setting EnhancedUnicodeSupport in the KCM Core Administrator can be used to map the function sort_fieldset_array automatically to the function sort_fieldset_array_characters (see sort_fieldset_array_characters). For more information, see Enhanced Unicode Support.

sort_fieldset_array_characters

The function `sort_fieldset_array_characters` sorts the elements of a `FIELDSET` array and stores the sorted elements in a second array.

```
sort_fieldset_array_characters (input;output;field;number of elements to sort )
```

The function `sort_fieldset_array_characters` is of type `BOOL` and has four parameters:

1. `input`; `ARRAY` of type `FIELDSET`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `FIELDSET`. This array receives the sorted elements.
3. `field`; `TEXT`. This is the `Field` in the `FIELDSET` used as the key to sort the array. If this `Field` is not present in the `FIELDSET`, it is considered empty.
4. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_fieldset_array_characters` returns `TRUE` if the array was sorted successfully, and `FALSE` in one of the following conditions:

- If the number of elements parameter to be sorted is negative or 0.
- If the function `sort_fieldset_array_characters` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

Any word processor instructions in the text elements are included in the sort and sort before regular characters.

```
DDATASTRUCTURE DS
  BEGIN
    ARRAY FIELDSET Array[0]
  END

DECLARE input DEFINED_AS DS
DECLARE output DEFINED_AS DS

NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_fieldset_array_characters (input.Array; output.Array; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[i].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type BOOL of the sort_filedset_array_characters function is used to implement error handling. If the sort succeeds, the return value of the function is TRUE, the IF statement becomes TRUE, and the THEN part is executed. If the return is FALSE, the ELSE part of the IF statement is executed.

sort_fieldset_array_index

The function sort_fieldset_array_index sorts the indices of a FIELDSET array based on the elements in the array.

```
sort_fieldset_array_index (input;output;field;number of elements to sort )
```

The function sort_fieldset_array_index is of type BOOL and has four parameters:

1. input; ARRAY of type FIELDSET. This array contains the elements that must be sorted.
2. output; ARRAY of type NUMBER. This array receives the sorted indices.
3. field; TEXT. This is the Field in the FIELDSET that is used as the key to sort the array. If this Field is not present in the FIELDSET, it is considered empty.
4. number of elements to sort; NUMBER. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a NUMBER variable or expression.

The function sort_fieldset_array_index returns TRUE if the array was sorted successfully, and FALSE in one of the following conditions:

- If the number of elements to be sorted parameter is negative or 0.
- If the function sort_fieldset_array_index fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted indices from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function sort_fieldset_array_index performs an alphabetical sort on the text elements using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause illogical orderings.

An example is provided here.

```
DDATASTRUCTURE DS
  BEGIN
    ARRAY FIELDSET Array[0]
  END

DECLARE input DEFINED_AS DS

ARRAY NUMBER output[0]
NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_fieldset_array_index (input.Array; output; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
```

```
#
@(i) @(input.Array[ output[i] ].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type BOOL of the sort_filedset_characters function is used to implement error handling. If the sort succeeds, the return value of the function is TRUE, the IF statement becomes TRUE, and the THEN part is executed. If the return is FALSE, the ELSE part of the IF statement is executed.

This example produces the same result as the example included with the function sort_fieldset_array (see sort_fieldset_array).

Note The setting EnhancedUnicodeSupport in KCM Core Administrator can be used to map the function sort_fieldset_array automatically to the function sort_fieldset_array_index_characters (see sort_fieldset_array_index_characters) . For more information, see Enhanced Unicode Support.

sort_fieldset_array_index_characters

The sort_fieldset_array_index_characters function sorts the indices of a FIELDSET array based on the elements in the array.

```
sort_fieldset_array_index_characters (input;output;field;number of elements to sort)
```

The function sort_fieldset_array_index is of type BOOL and has four parameters:

1. input; ARRAY of type FIELDSET. This array contains the elements that must be sorted.
2. output; ARRAY of type NUMBER. This array receives the sorted indices.
3. field; TEXT. This is the Field in the FIELDSET that is used as the key to sort the array. If this Field is not present in the FIELDSET, it is considered empty.
4. number of elements to sort; NUMBER. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a NUMBER variable or expression.

The function sort_fieldset_array_index_characters returns TRUE if the array was sorted successfully, and FALSE in one of the following conditions:

- If the number of elements to be sorted parameter is negative or 0.
- If the function sort_fieldset_array_index fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted indices from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

Note All Microsoft Word instructions in the text elements are included in the sort and sort before regular characters.

An example is provided here.

```
DDATASTRUCTURE DS
  BEGIN
```

```
    ARRAY FIELDSET Array[0]
  END

DECLARE input DEFINED_AS DS

ARRAY NUMBER output[0]
NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_fieldset_array_index_characters (input.Array; output; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(input.Array[ output[i] ].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_filedset_array_index_characters` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

This example produces the same result as the example included with the function `sort_fieldset_array_characters` (see sort_fieldset_array_characters).

sort_struct_array

The function `sort_struct_array` sorts the elements of a `DATASTRUCTURE` array and stores the sorted elements in a second array.

```
sort_struct_array ( input;output;member;number of elements to sort )
```

The function `sort_struct_array` is of type `BOOL`.

The function has four parameters:

1. `input`; `ARRAY` of type `DATASTRUCTURE`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `DATASTRUCTURE`. This array receives the sorted elements. Both `input` and `output` must be of the same `DATASTRUCTURE` type. If the types do not match, an error is reported during the execution of the Master Template.
3. `member`; `TEXT`. This is the member in the `DATASTRUCTURE` that is used as the key to sort the array. Both `TEXT` and `NUMBER` members can be used to sort. If a `NUMBER` member is used, the array is sorted in numerical order. If a `TEXT` member is used, the array is sorted lexicographically. If the member is not present in the `DATASTRUCTURE` or has a type other than `TEXT` or `NUMBER`, an error is reported during the Master Template execution.

4. number of elements to sort; NUMBER. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a NUMBER variable or expression.

The function sort_struct_array returns TRUE if the array is sorted successfully, and FALSE if one of the following conditions is met:

- If the number of elements to be sorted parameter is negative or 0.
- If the function sort_struct_array fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function sort_struct_array performs an alphabetical sort of the text members using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause unexpected orderings.

An example is provided here.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Key
  END

DATASTRUCTURE DS
  BEGIN
    ARRAY EL Array[0]
  END

DECLARE input DEFINED_AS DS
DECLARE output DEFINED_AS DS

NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_struct_array (input.Array; output.Array; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[i].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type BOOL of the sort_struct_array function is used to implement error handling. If the sort succeeds, the return value of the function is TRUE, the IF statement becomes TRUE, and the THEN part is executed. If the return is FALSE, the ELSE part of the IF statement is executed.

The setting EnhancedUnicodeSupport in KCM Core Administrator can be used to map the function `sort_struct_array_index` automatically to the function `sort_struct_array_index_characters` (see sort_struct_array_index_characters) . For more information, see Enhanced Unicode Support.

sort_struct_array_characters

The function `sort_struct_array_characters` sorts the elements of a `DATASTRUCTURE` array and stores the sorted elements in a second array.

```
sort_struct_array_characters (input;output;member;number of elements to sort )
```

The function `sort_struct_array` is of type `BOOL`.

The function has four parameters:

1. `input`; `ARRAY` of type `DATASTRUCTURE`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `DATASTRUCTURE`. This array receives the sorted elements. Both `input` and `output` must be of the same `DATASTRUCTURE` type. If the types do not match, an error is reported during execution of the Master Template.
3. `member`; `TEXT`. This is the member in the `DATASTRUCTURE` that is used as the key to sort the array. Both `TEXT` and `NUMBER` members can be used to sort. If a `NUMBER` member is used, the array is sorted in numerical order. If a `TEXT` member is used, the array is sorted lexicographically. If the member is not present in the `DATASTRUCTURE` or has a type other than `TEXT` or `NUMBER`, an error is reported during the Master Template execution.
4. number of elements to be sorted; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_struct_array_characters` returns `TRUE` if the array is sorted successfully, and `FALSE` if one of the following conditions is met:

- If the number of elements to be sorted parameter is negative or 0.
- If the function `sort_struct_array_characters` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.

Note All Microsoft Word instructions in text members are included in the sort and sort before regular characters.

An example is provided here.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Key
  END

DATASTRUCTURE DS
  BEGIN
    ARRAY EL Array[0]
  END
```

```
DECLARE input DEFINED_AS DS
DECLARE output DEFINED_AS DS

NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_struct_array_characters (input.Array; output.Array; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[i].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_struct_array_characters` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

sort_struct_array_index

The function `sort_struct_array_index` sorts the indices of a `DATASTRUCTURE` array based on the elements in the array.

```
sort_struct_array_index (input;output;member;number of elements to sort )
```

The function `sort_struct_array_index` is of type `BOOL`.

The function has four parameters:

1. `input`; `ARRAY` of type `DATASTRUCTURE`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `DATASTRUCTURE`. This array receives the sorted elements. Both `input` and `output` must be of the same `DATASTRUCTURE` type. If the types do not match, an error is reported during execution of the Master Template.
3. `member`; `TEXT`. This is the member in the `DATASTRUCTURE` that is used as the key to sort the array. Both `TEXT` and `NUMBER` members can be used to sort. If a `NUMBER` member is used, the array is sorted in numerical order. If a `TEXT` member is used, the array is sorted lexicographically. If the member is not present in the `DATASTRUCTURE` or has a type other than `TEXT` or `NUMBER`, an error is reported during the Master Template execution.
4. number of elements to sort; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_struct_array_index` returns `TRUE` if the array is sorted successfully, and `FALSE` if one of the following conditions is met:

- If the number of elements to be sorted parameter is negative or 0.

- If the function `sort_struct_array_index` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function `sort_struct_array_index` performs an alphabetical sort on the text members using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause unexpected orderings.

An example is provided here.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Key
  END

DATASTRUCTURE DS
  BEGIN
    ARRAY EL Array[0]
  END

DECLARE input DEFINED_AS DS

ARRAY NUMBER output[4]
NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_struct_array_index (input.Array; output; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[ output[i] ].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_struct_array_index` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

The below example produces the same result as the example included with the function `sort_struct_array` (see sort_struct_array).

sort_struct_array_index_characters

The function `sort_struct_array_index_characters` sorts the indices of a `DATASTRUCTURE` array based on the elements in the array.

```
sort_struct_array_index_characters (input;output;member;number of elements to sort)
```

The function `sort_struct_array_index_characters` is of type `BOOL`.

The function has four parameters:

1. `input`; `ARRAY` of type `DATASTRUCTURE`. This array contains the elements that must be sorted.
2. `output`; `ARRAY` of type `DATASTRUCTURE`. This array receives the sorted elements. Both `input` and `output` must be of the same `DATASTRUCTURE` type. If the types do not match, an error is reported during execution of the Master Template.
3. `member`; `TEXT`. This is the member in the `DATASTRUCTURE` that is used as the key to sort the array. Both `TEXT` and `NUMBER` members can be used to sort. If a `NUMBER` member is used, the array is sorted in numerical order. If a `TEXT` member is used, the array is sorted lexicographically. If the member is not present in the `DATASTRUCTURE` or has a type other than `TEXT` or `NUMBER`, an error is reported during the Master Template execution.
4. number of elements to be sorted; `NUMBER`. This is the number of elements in the array that must be sorted starting from the first element. This can either be a number or a `NUMBER` variable or expression.

The function `sort_struct_array_index_characters` returns `TRUE` if the array is sorted successfully, and `FALSE` if one of the following conditions is met:

- If the number of elements to be sorted parameter is negative or 0.
- If the function `sort_struct_array_index_characters` fails, the output array is unmodified. The input array is not modified by the function. The output array receives the sorted values from the input array. If the output array is larger than the number of sorted elements, those elements outside the sorted range are left unmodified.
- The function `sort_struct_array_index` performs an alphabetical sort on the text members using the ANSI code page.

Note All Microsoft Word instructions in the text elements are included in the sort and might cause unexpected orderings.

An example is provided here.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Key
  END

DATASTRUCTURE DS
  BEGIN
    ARRAY EL Array[0]
  END

DECLARE input DEFINED_AS DS
```

```
ARRAY NUMBER output[4]
NUMBER i

ASSIGN input.Array[1].Key := "c"
ASSIGN input.Array[2].Key := "d"
ASSIGN input.Array[3].Key := "a"
ASSIGN input.Array[4].Key := "b"

IF sort_struct_array_index_characters (input.Array; output; "Key"; 4) THEN
  FOR i FROM 1 UPTO 4
  DO
#
@(i) @(output.Array[ output[i] ].Key)
#
  OD
ELSE
#
Failed to sort the input.
#
  STOP
FI
```

The type `BOOL` of the `sort_struct_array_index_characters` function is used to implement error handling. If the sort succeeds, the return value of the function is `TRUE`, the `IF` statement becomes `TRUE`, and the `THEN` part is executed. If the return is `FALSE`, the `ELSE` part of the `IF` statement is executed.

The preceding example produces the same result as the example included with the function `sort_struct_array_characters` (see sort_struct_array_characters).

Maps functions

The following section contains a description of the maps functions. These functions are used to get the length of a map, to get the keys of a map, and to retrieve the keys used in a map.

length_fieldset_map

The `length_fieldset_map` function yields the number of elements contained in `FIELDSET MAP`.

```
 length_fieldset_map ( MAP_to_be_counted)
```

The function returns `NUMBER` that is the number of elements stored in the map.

This function has one parameter:

- `MAP_to_be_counted`, `MAP` of type `FIELDSET`. A `MAP` with number of elements to be counted.

An example is provided here.

```
DATASTRUCTURE DS
  BEGIN
    MAP FIELDSET Test_map
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Test_map ["first"].Field := "TRUE"
ASSIGN struct.Test_map ["second"].Field := "FALSE"
ASSIGN struct.Test_map ["third"].Field := "TRUE"
ASSIGN struct.Test_map ["fourth"].Field := "TRUE"
ASSIGN struct.Test_map ["fifth"].Field := "FALSE"
```

```
NUMBER length_function_result

ASSIGN length_function_result := length_fieldset_map (struct.Test_map)

#
@(length_function_result)
#
```

When you refer to a `MAP` element, it results in the creation of that element. The element is created empty.

length_struct_map

The `length_struct_map` function yields the number of elements contained in `FIELDSET MAP`.

```
 length_struct_map (MAP_to_be_counted)
```

The function returns `NUMBER` that is the number of elements stored in the map.

This function has one parameter:

- `MAP_to_be_counted`, `MAP` of type `FIELDSET`. A `MAP` with the number of elements to be counted.

An example is provided here.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Field
  END

DATASTRUCTURE DS
  BEGIN
    MAP EL Test_map
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Test_map ["first"].Field := "TRUE"
ASSIGN struct.Test_map ["second"].Field := "FALSE"
ASSIGN struct.Test_map ["third"].Field := "TRUE"
ASSIGN struct.Test_map ["fourth"].Field := "TRUE"
ASSIGN struct.Test_map ["fifth"].Field := "FALSE"
NUMBER length_function_result

ASSIGN length_function_result := length_struct_map (struct.Test_map)

#
@(length_function_result)
#
```

When you refer to a `MAP` element, it results in the creation of that element. The element is created empty.

get_keys_fieldset_map

The function `get_keys_fieldset_map` retrieves the index values of all elements in `MAP` and stores them in an array. The index values are stored in the array in random order.

```
 get_keys_fieldset_map ( input_map;result_array)
```

This function returns a `NUMBER`, which is the number of elements stored in the array.

This function has two parameters:

1. `input_map` is a `MAP` of type `FIELDSET`. The `MAP` whose index values you need.
2. `result_array` is an array of type `TEXT`. The array whose index values are stored.

An example is provided here.

```
DATASTRUCTURE DS
  BEGIN
    MAP FIELDSET Input_map
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Input_map ["first"].Field := "This is the first value"
ASSIGN struct.Input_map ["second"].Field := "This is the second value"
ASSIGN struct.Input_map ["third"].Field := "This is the third value"
ASSIGN struct.Input_map ["fourth"].Field := "This is the fourth value"
ASSIGN struct.Input_map ["fifth"].Field := "This is the fifth value"

ARRAY TEXT result_array[1]
NUMBER number_of_elements

ASSIGN number_of_elements := get_keys_fieldset_map (struct.Input_map;result_array)

NUMBER counter
FOR counter
 FROM 1
 UPTO number_of_elements
DO
#
The key @(result_array[counter]) contains
 @(struct.Input_map[result_array[counter]].Field) in the map.
#
OD
```

get_keys_struct_map

The function `get_keys_struct_map` retrieves the index values of all elements in `MAP` and stores them in an array. The index values are stored in the array in random order.

```
 get_keys_struct_map ( input_map;result_array)
```

This function returns `NUMBER`, which is the number of elements stored in the array.

This function has two parameters:

1. `input_map` is `MAP` of type `FIELDSET`. The `MAP` with the index values.
2. `result_array` is an array of type `TEXT`. The array with index values to be stored.

An example is provided below.

```
DATASTRUCTURE EL
  BEGIN
    TEXT Field
  END

DATASTRUCTURE DS
  BEGIN
    MAP EL Input_map
```

```
  END

DECLARE struct DEFINED_AS DS

ASSIGN struct.Input_map ["first"].Field := "This is the first value"
ASSIGN struct.Input_map ["second"].Field := "This is the second value"
ASSIGN struct.Input_map ["third"].Field := "This is the third value"
ASSIGN struct.Input_map ["fourth"].Field := "This is the fourth value"
ASSIGN struct.Input_map ["fifth"].Field := "This is the fifth value"

ARRAY TEXT result_array[1]
NUMBER number_of_elements

ASSIGN number_of_elements := get_keys_struct_map (struct.Input_map;result_array)

NUMBER counter
FOR counter
 FROM 1
 UPTO number_of_elements
DO
#
The key @(result_array[counter]) contains
 @(struct.Input_map[result_array[counter]].Field) in the map.
#
OD
```

key_used_in_fieldset_map

The function `key_used_in_fieldset_map` tests if a certain key is present in a map of type `FIELDSET`, without creating the corresponding element.

```
key_used_in_fieldset_map ( key_to_find; map_name)
```

The result of the function is type `BOOL`. The function yields `TRUE` if the map contains an element map[key].

The function `key_used_in_fieldset_map` has two parameters:

1. `key_to_find` is of type `TEXT`. The key to be detected in the map.
2. `map_name` is of type `MAP FIELDSET`. The map to be tested for the presence of the `key_to_find`.

An example is provided here.

```
BOOL key_found := key_used_in_fieldset_map ( "a_key" ;  mymap )
```

The map index of the `mymap` is searched for the key `a_key`. If it is found, the variable `key_found` is set to `TRUE`.

key_used_in_struct_map

The function `key_used_in_struct_map` tests if a certain key is present in a map of type `DATASTRUCTURE`, without creating the corresponding element.

```
 key_used_in_struct_map ( key_to_find; map_name)
```

The result of the function is type `BOOL`. The function yields `TRUE` if the map contains an element map[key].

The function `key_used_in_text_map` has two parameters:

1. The parameter `key_to_find` is of type `TEXT`. This is the key to be detected in the map.
2. The parameter `map_name` is of type `MAP DATASTRUCTURE`. The map to be tested for the presence of `key_to_find`.

An example is provided here.

```
BOOL key_found := key_used_in_struct_map ("a_key" ; mymap )
```

The map index of the `mymap` is searched for the key `a_key`. If it is found, the variable `key_found` is set to `TRUE`.

Data structures functions

The following sections contain information on all functions related to data structures.

struct_has_content

The `struct_has_content` function is used to test whether or not a `DATASTRUCTURE` variable has ever been used by either accessing or assigning content to one of its members.

```
struct_has_content ( datastructure )
```

The function has one parameter:

- `datastructure`, type `DATASTRUCTURE`; the data structure to be tested.

The result of the function is of type `BOOL`. If the data structure has never been used, the function returns `TRUE`; otherwise, it returns `FALSE`.

fieldsets_in_scope

The `fieldsets_in_scope` function is used to retrieve the Field Sets that are available within a `FOREACH WIZARD` loop (see FOREACH WIZARD/NODE) to be used in the Master Template. This function uses the same logic to select Field Sets from a Data Definition as used by the `TEXTBLOCK` statement (see TEXTBLOCK statement) with the `DATA_DEFINITION` and `PATH` keywords.

```
fieldsets_in_scope ( datainput; path; fieldsets)
```

The function has three parameters:

1. `datainput`, type `DATASTRUCTURE`. This is the Data Structure that is to be used as `DATA_DEFINITION` parameter to the `TEXTBLOCK` statement.
2. `path`, type `TEXT`. This is the path used as `PATH` parameter to the `TEXTBLOCK` statement. The content of the path parameter is subject to the same restrictions as documented for the `PATH` parameter on the `TEXTBLOCK` statement.
3. `fieldsets`, type `MAP FIELDSET`. This `MAP` receives a copy of the Field Sets, using the name of the Field Set as a key in the `MAP`.

The result of the function is type `BOOL`. If the function succeeds, it returns `TRUE`; otherwise, it returns `FALSE`.

```
DATASTRUCTURE FIS
  BEGIN
    MAP FIELDSET FieldSets
  END

DECLARE sets DEFINED_AS FIS

BOOL res := fieldsets_in_scope (data_definition; path; sets.FieldSets)
TEXT customer := sets.FieldSets["Customer"].FirstName + " " +
 sets.FieldSets["Customer"].LastName
```

The above example retrieves the active Field Sets, and then uses the Customer.FirstName and Customer.LastName fields from this set.

Dynamic building blocks

metadata_contains

The function `metadata_contains` tests if a metadata attribute is set on a dynamic object.

```
 metadata_contains (attribute; value)
```

The result of the function `metadata_contains` is of type `BOOL`.

This function has two parameters:

1. `attribute`. This is the metadata attribute to be tested.
2. `value`. This is the value that is tested for.

The function `metadata_contains` returns `TRUE` if the attribute is set on the dynamic object; otherwise, it returns `FALSE`. If value is not a valid attribute, the Master Template stops, and a run-time error is reported.

The `metadata_contains` function can only be used within the execution scope of a filter function. If you call the `metadata_contains` function directly, the Master Template stops, and a run-time error is reported.

Examples are provided here.

```
FILTER
FUNC BOOL texas (FIELDSET Set)
DO
  ASSIGN texas := metadata_contains ("State"; "TX")
OD
```

The preceding example filters all objects where the metadata attribute `"State"` has the value `"TX"` set.

```
FILTER
FUNC BOOL state (FIELDSET Set)
DO
  ASSIGN state := metadata_contains ("State"; _data.Customer.State)
OD
```

The second example filters all objects where the metadata attribute `"State"` contains the value from the `Customer.State` field in the Data Backbone.

The `metadata_contains` function is introduced in KCM version 4.4 and requires KCM Core version 4.4 or higher.

Chapter 6

Functions and keywords translations

The Template scripting language is available in three languages: Dutch, English, and German. The translation tables contain the functions and keywords in English and their equivalents in Dutch and German.

As of KCM 4.4, you can mix functions and keywords from the three languages in the same project.

Translation of the functions

Consult the following table if you need to translate the functions.

English	Dutch	German
add_to_output	voeg_toe_aan_uitvoer	ausgabe_erweitern
add_user_xml	add_user_xml	add_user_xml
amount	bedrag	betrag
amount_in_words	bedrag_voluit	betrag_in_worten
amount_in_words_euro	bedrag_voluit_euro	betrag_in_worten_euro
arctan	arctangens	atan
area	oppervlakte	fläche
area_in_words	oppervlakte_voluit	fläche_in_worten
clear_fieldset	maak_veldenset_leeg	lösche_fieldset
compare_characters	vergelijk_tekens	vergleich_zeichen
cosine	cosinus	cos
create_csv	maak_csv	erstelle_csv
date	datum	datum
date_in_words	datum_voluit	datum_in_worten
document_property	document_eigenschap	dokument_eigenschaft
environment_setting	environment_setting	environment_setting
euro	euro	euro
exponent	exponent	exp
fieldsets_in_scope	fieldsets_in_scope	fieldsets_in_scope
footers	voetteksten	footers

format	opmaak	formatiert
format_date	maak_datum_op	format_date
fragment_of_characters	deel_van_tekens	teil_der_zeichen
get_fields_from_text_block	haal_velden_uit_tekstblok	hole_fields_aus_textbaustein
get_keys_bool_map	haal_sleutels_bool_map	hole_schlüssel_der_bool_maptabelle
get_keys_fieldset_map	haal_sleutels_fieldset_map	hole_schlüssel_der_fieldset_maptabelle
get_keys_number_map	haal_sleutels_getal_map	hole_schlüssel_der_zahlen_maptabelle
get_keys_struct_map	haal_sleutels_struct_map	hole_schlüssel_der_struct_maptabelle
get_keys_text_map	haal_sleutels_tekst_map	hole_schlüssel_der_text_maptabelle
get_text_blocks_in_view	haal_tekstblokken_uit_view	hole_textbausteine_aus_view
headers	kopteksten	headers
import_text_block	importeer_tekstblok	importier_textbaustein
import_text_block_base64	importeer_tekstblok_base64	importier_textbaustein_base64
inc	inc	füge_ein
insert_image	insert_image	insert_image
insert_text_block	tekstblok_invoegen	textbaustein_einfügen
insert_text_block_extended	tekstblok_invoegen_uitgebreid	textbaustein_einfügen_erweitert
insert_text_blocks	insert_text_blocks	insert_text_blocks
insert_text_blocks_extended	insert_text_blocks_extended	insert_text_blocks_extended
itp_setting	itp_setting	itp_einstellung
itpserver_parameter	itpserver_parameter	itpserver_parameter
itpserver_setting	itpserver_setting	itpserver_setting
key_used_in_bool_map	sleutel_gebruikt_in_bool_map	schlüssel_verwendet_in_bool_maptabelle
key_used_in_fieldset_map	sleutel_gebruikt_in_fieldset_map	schlüssel_verwended_in_fieldset_maptabelle
key_used_in_number_map	sleutel_gebruikt_in_getal_map	schlüssel_verwendet_in_zahlen_maptabelle
key_used_in_struct_map	sleutel_gebruikt_in_struct_map	schlüssel_verwended_in_struct_maptabelle
key_used_in_text_map	sleutel_gebruikt_in_tekst_map	schlüssel_verwendet_in_text_maptabelle
language_code	taalcode	sprache
length	lengte	länge_der_var
length_bool_array	lengte_bool_rij	länge_der_bool_tabelle
length_bool_map	lengte_bool_map	länge_der_bool_maptabelle
length_fieldset_array	lengte_fieldset_rij	länge_der_fieldset_tabelle
length_fieldset_map	lengte_fieldset_map	länge_der_fieldset_maptabelle
length_number_array	lengte_getal_rij	länge_der_zahlen_tabelle
length_number_map	lengte_getal_map	länge_der_zahlen_maptabelle

length_struct_array	lengte_struct_rij	länge_der_struct_tabelle
length_struct_map	lengte_struct_map	länge_der_struct_maptabelle
length_text_array	lengte_tekst_rij	länge_der_text_tabelle
length_text_map	lengte_tekst_map	länge_der_text_maptabelle
logarithm	logarithme	log
lowercase	kleine_letter	kleinbs_von
lowercase_of_characters	kleineletters_van_tekens	kleinbs_der_zeichen
lowercase_roman_number	romeinse_cijfers_klein	röm_ziffer_klein
lowercase2	kleineletters2	kleinbs_an_stelle
ltrim	ltrim	ltrim
now	nu	jetzt
number	getal	als_zahl
number_in_words	getal_voluit	zahl_in_worten
number_of_characters	aantal_tekens	zahl_der_zeichen
number_to_date	getal_naar_datum	in_datum_umgesetzt
numerals	cijfers	in_ziffern
open_buffer	open_buffer	öffne_puffer
ordinal	rangtelwoord	ordinalzahl
pagestyle	paginastijl	pagestyle
paper_types	papier_soorten	paper_types
picture	picture	nach_schablone
pragma	pragma	pragma
pragma_struct	pragma_struct	pragma_struct
put_buffer_in_document	zet_buffer_in_dokument	speichere_puffer_in_dokument
put_in_document	zet_in_dokument	speichere_in_dokument
put_in_text_file	zet_in_tekst_bestand	als_textdatei_ausgeben
put_in_text_file2	put_in_text_file2	als_textdatei_ausgeben2
read_text_block_from_file	lees_tekstblok_uit_bestand	read_text_block_from_file
replace	vervang	replace
round	rond_af	gerundet
round_upwards	rond_af_naar_boven	aufgerundet
rtrim	rtrim	rtrim
runmodel_setting	runmodel_setting	runmodel_setting
search	zoek	search
search_first	zoek_eerste	search_first

search_last	zoek_laatste	search_last
session_parameter	sessie_parameter	session_parameter
sine	sinus	sin
sort_fieldset_array	sorteer_fieldset_rij	sortier_fieldset_tabelle
sort_fieldset_array_index	sorteer_fieldset_rij_index	sortier_fieldset_tabelle_index
sort_fieldset_array_characters	sorteer_fieldset_rij_tekens	sortier_fieldset_tabelle_zeichen
sort_fieldset_array_index_characters	sorteer_fieldset_rij_index_tekens	sortier_fieldset_tabelle_zeichen_index
sort_number_array	sorteer_getal_rij	sortier_zahl_tabelle
sort_number_array_index	sorteer_getal_rij_index	sortier_zahl_tabelle_index
sort_struct_array	sorteer_struct_rij	sortier_struct_tabelle
sort_struct_array_index	sorteer_struct_rij_index	sortier_struct_tabelle_index
sort_struct_array_characters	sorteer_struct_rij_tekens	sortier_struct_tabelle_zeichen
sort_struct_array_cindex_characters	sorteer_struct_rij_tekens_index	sortier_struct_tabelle_zeichen_index
sort_text_array	sorteer_tekst_rij	sortier_text_tabelle
sort_text_array_characters	sort_text_array_characters	sort_text_array_characters
sort_text_array_index	sorteer_tekst_rij_index	sortier_text_tabelle_index
sort_text_array_index_characters	sort_text_array_index_characters	sort_text_array_index_characters
split_csv	splits_csv	teile_csv
square	kwadraat	quadrat
square_root	wortel	wurzel
status_message	statusbericht	status_meldung
struct_has_content	struct_heeft_inhoud	struct_hat_inhalt
stylesheet	stylesheet	stylesheet
system	systeem	system
tangens	tangens	tangens
text_block_exists	tekstblok_bestaat	textblock_exists
text_fragment	deel_tekst	teilstring
text_to_number	tekst_naar_getal	zahl_aus_text
today	vandaag	heute
trim	trim	trim
truncate	kap_af	abgeschnitten
uppercase	hoofd_letter	großbs_von
uppercase_of_characters	hoofdletters_van_tekens	großbs_der_zeichen
uppercase_roman_number	romeinse_cijfers_groot	röm_ziffer_groß
uppercase2	hoofdletters2	großbs_an_stelle

uppercases	hoofdletters	alles_großbs

Translation of the keywords

Consult the following table if you need to translate the Template scripting language keywords.

English/German	**Dutch**
AIADOCXML	AIADOCXML
ALFA	ALFA
AND	EN
ANSWER	ANTWOORD
APROC	APROC
ARRAY	RIJ
AS	AS
ASCII	ASCII
ASSIGN	MAAK
ASSIGN_TO	WIJS_TOE
AUTOINSERT	AUTOINSERT
BEGIN	BEGIN
BEGIN_ALTERNATIVE	BEGIN_ALTERNATIVE
BEGIN_DOCUMENT	BEGIN_DOCUMENT
BEGIN_SET	BEGIN_SET
BEGINGROUP	BEGINGROEP
BEGINROW	BEGINRIJ
BEGINTABLE	BEGINTABEL
BINAIR	BINAIR
BOOL	BOOL
BUTTON	BUTTON
BY_DATE	BY_DATE
CHOICE	CHOICE
COMMA	COMMA
CONST	CONST
CONTAINS	BEVAT
DATA	DATA
DATE	DATUM

DATA_DEFINITION	DATA_DEFINITIE
DATABACKBONE	DATABACKBONE
DATASTRUCTURE	DATASTRUCTURE
DBB_POST_EDIT	DBB_POST_EDIT
DBB_POST_LOAD	DBB_POST_LOAD
DBB_PRE_LOAD	DBB_PRE_LOAD
DECLARE	DECLAREER
DEFAULT	DEFAULT
DEFINE	DEFINE
DEFINED_AS	GEDEFINIEERD_ALS
DEFAULTSET	DEFAULTSET
DESCRIPTION	BESCHRIJVING
DFT	DFT
DISCARD_OUTPUT	VERWERP_UITVOER
DO	DOE
DOCD	DOCD
EBCDIC	EBCDIC
EDITABLE_TEXTBLOCK	BEWERKBAAR_TEKSTBLOK
EDITBOX	COMBOBOX
ELIF	ALSANDERS
ELSE	ANDERS
END	EINDE
END_ALTERNATIVE	END_ALTERNATIVE
END_DOCUMENT	END_DOCUMENT
END_SET	END_SET
ENDGROUP	EINDEGROEP
ENDROW	EINDERIJ
ENDTABLE	EINDETABEL
ENTER	ENTER
ENTRY	INGANG
ERROR	ERROR
ERRORCONDITION	FOUTCONDITIE
EXIT	EXIT
EXPANDABLE	UITKLAPBAAR
EXPANDED	UITKLAPCONDITIE

EXPORT	EXPORTEER
EXTRA	EXTRA
FALSE	ONWAAR
FI	SLA
FIELD	VELD
FIELDSET	VELDENSET
FILE	BESTAND
FILTER	FILTER
FIXED	FIXED
FLDSET	VLDSET
FOLDER	FOLDER
FOR	VOOR
FORALL	VOORALLE
FOREACH	VOORIEDERE
FORM	FORMULIER
FORMAT	FORMAAT
FROM	VANAF
FUNC	FUNC
GROUP_STATE	GROEP_STATUS
HELPTEXT	HELPTEKST
HIDECONDITION	VERBERGCONDITIE
ID	ID
IF	ALS
IN	IN
INTERACT	INTERACT
KEY	SLEUTEL
KEYS	SLEUTELS
LAYOUT	OPMAAK
LEN	LEN
MAP	MAP
MAXSET	MAXSET
MESSAGE	MELDING
MULTISELECT	MULTISELECT
NAME	NAAM
NATIVE	NATIVE

NEW	NIEUW
NON-INTERACTIVE	NIET_INTERACTIEF
NONE	GEEN
NOT	NIET
NUMBER	GETAL
OD	EOD
ON	ON
OPTIONAL	OPTIONEEL
OR	OF
ORDER	VOLGORDE
PACKED	PACKED
PAR	PAR
PARAGRAPH_SYMBOL	PARAGRAAF_TEKEN
PARAMETER	PARAMETER
PATH	PAD
PROC	PROC
PROMPT	PROMPT
QFORM	QFORM
QUESTION	VRAAG
RADIOBUTTONS	RADIOBUTTONS
READONLY	READONLY
RECORDSET	RECORDSET
REPEAT	HERHAAL
SELECTION_INPUT	SELECTIE_INVOER
SELECTION_OUTPUT	SELECTIE_UITVOER
SET	ZET
SHOW	TOON
SHOW_ALL	SHOW_ALL
SHOWNOT	TOONNIET
SORT	SORTEER
SORTED	GESORTEERDE
STOP	STOP
STYLE_PREFIX	STIJL_PREFIX
SUPPRESS_EDITABLE_TEXTBLOCKS	ONDERDRUK_BEWERKBARE_TEKSTBLOKKEN
SUPPRESS_FINAL_PARAGRAPH	ONDERDRUK_LAATSTE_PARAGRAAF

SUPPRESS_QFORMS	ONDERDRUK_QFORMS
TABLE_STYLE_PREFIX	TABEL_STIJL_PREFIX
TEXT	TEKST
TEXT_BLOCK	TEKST_BLOK
TEXTBLOCK	TEKSTBLOK
THEN	DAN
TIME	TIJD
TO	TO
TRUE	WAAR
UNTIL	TOTDAT
UPTO	TOTAAN
UTF8	UTF8
UTF16	UTF16
VALUES	VALUES
VAR	VAR
VIEW	VIEW
VIEWS_USE_ORDERING	VIEWS_ZIJN_GEORDEND
WARNING	WARNING
WHERE	WAARBIJ
WHILE	ZOLANG
WITH	MET
WIZARD	WIZARD
WRITE	WRITE
XML	XML
ZONED	ZONED

As of KCM version 4.4, the Template scripting language setting is only used to parse numbers. English/German and Dutch keywords can be mixed within the same document and/or Includes.

Chapter 7

Enhanced Unicode Support

Use the setting EnhancedUnicodeSupport in KCM Core Administrator to control how text comparisons and other text related functions are handled. If this setting is enabled, all comparisons and functions that perform a bytewise comparison are automatically mapped to their character based Unicode aware equivalent when the Master Template is executed.

Note Enabling these options may cause Master Templates to behave differently if comparisons are performed between C_CHAR (byte) and W_CHAR (Unicode) data, or if there are Microsoft Word instructions mixed with text.

The following table contains description of the mapping applied.

Used in the Master Template	When EnhancedUnicodeSupport is Enabled, Comparisons and Functions Map to the Following Characters
<	compare_characters
<=	compare_characters
=	compare_characters
<>	compare_characters
>=	compare_characters
>	compare_characters
text_fragment	fragment_of_characters
length	number_of_characters
uppercases	uppercase_of_characters
uppercase2	uppercase_of_characters
lowercase2	lowercase_of_characters
sort_text_array	sort_text_array_characters
sort_text_array_index	sort_text_array_index_characters

You can use the pragma `"EnhancedUnicodeSupport"` to override the setting EnhancedUnicodeSupport locally in the Master Template.

The EnhancedUnicodeMaps setting

Use the setting EnhancedUnicodeMaps in KCM Core Administrator to control how `MAP` indices are handled. If this setting is enabled, all `MAP` indices are stored as Unicode and compared using the function `compare_characters` (see compare_characters). Without this setting the characters used in `MAP` indices are limited to the character set Latin-1.

You can use the pragma `"EnhancedUnicodeMap"` to override the setting EnhancedUnicodeMaps in the Master Template.

Note The EnhancedUnicodeMaps setting can be changed only before the first `MAP` element has been assigned. Any further changes result in a fatal error.