Kofax Customer Communications Manager

Batch & Output Management Scripting Language Developer's Guide

Version: 5.2

Date: 2018-11-21

© 2018 Kofax. All rights reserved.

Kofax is a trademark of Kofax, Inc., registered in the U.S. and/or other countries. All other trademarks are the property of their respective owners. No part of this publication may be reproduced, stored, or transmitted in any form without the prior written permission of Kofax.

Table of Contents

Preface **5**
Related documentation 5
Getting help for Kofax products 6
Chapter 1: Standard processes: scripts and functions **8**
Structure of the standard processes 8
Electronic output 8
Printed output 10
CCM_CleanUpAll 17
Processing component functions 17
RegisterRequest function 17
ProcessRequest function 17
CommunicateCorrespondence function 18
DeliverCommunication function (for electronic output) 19
DeliverCommunication function (for printed output) 19
CreateRenditionForSelectedProcess function 20
Bundling component functions 21
Stacking component functions 25
Streaming component function 28
Conversion component functions 28
Distribution component functions 30
GarbageCollection component functions 31
Chapter 2: Custom scripting **33**
Configure the accounting functionality 33
Configure a custom database for accounting 33
Initiate request accounting 34
Initiate communication accounting 35
Add and modify document pages in streaming 37
Include additional pages for stack, envelope, or document 37
Modify a document page 38
Insert a bar code on a document page 40
Configure the distribution functionality 41
Chapter 3: Batch & Output Management scripting language **43**
Create scripts and conditions 43
Create a script 43

Create a condition ... 43
View context information ... 44
Create a script parameter ... 44
Assign a script to an event ... 44
Script and condition editor tools ... 45
Useful key combinations ... 46
Debug a script or condition ... 46
Scripting functions ... 48
Data types ... 48
Control structures ... 49
Exception handling ... 51
Standard functions ... 53
Date functions ... 87
File processing functions ... 91
XML processing functions ... 105
Array functions ... 107
Object functions ... 108
Scripting contexts ... 110
Process ... 110
OnProcessError ... 111
StreamingUnit ... 112
StreamingPage ... 117
StackDistribution ... 119
Appendix A: Encoding parameters ... 121
Appendix B: HTML color names ... 126

Preface

This guide provides detailed information on the scripts used in the standard processes in CCM Studio and describes the scripting language of CCM Batch & Output Management (also known as CCM B&OM).

Related documentation

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

https://docshield.kofax.com/Portal/Products/CCM/520-nz7r6s9geq/CCM.htm

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

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

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

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

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

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

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

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

Kofax Customer Communications Manager Template Scripting Language Developer's Guide
Describes the CCM Template Script used in Master Templates.

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

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

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

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

Getting help for Kofax products

Kofax regularly updates the Kofax Support site with the latest information about Kofax products.

To access some resources, you must have a valid Support Agreement with an authorized Kofax Reseller/Partner or with Kofax directly.

Use the tools that Kofax provides for researching and identifying issues. For example, use the Kofax Support site to search for answers about messages, keywords, and product issues. To access the Kofax Support page, go to www.kofax.com/support.

The Kofax Support page provides:

- Product information and release news
  Click a product family, select a product, and select a version number.
- Downloadable product documentation
  Click a product family, select a product, and click **Documentation**.
- Access to product knowledge bases
  Click **Knowledge Base**.
- Access to the Kofax Customer Portal (for eligible customers)
  Click **Account Management** and log in.

To optimize your use of the portal, go to the Kofax Customer Portal login page and click the link to open the *Guide to the Kofax Support Portal*. This guide describes how to access the support site, what to do before contacting the support team, how to open a new case or view an open case, and what information to collect before opening a case.

- Access to support tools

  Click **Tools** and select the tool to use.
- Information about the support commitment for Kofax products

  Click **Support Details** and select **Kofax Support Commitment**.

Use these tools to find answers to questions that you have, to learn about new functionality, and to research possible solutions to current issues.

Chapter 1

Standard processes: scripts and functions

The CCM B&OM functionality is implemented in a set of standard processes that are installed with the product. Running the set of standard processes enables you to transform input request files into distribution output.

Structure of the standard processes

Each process consists of a script that is used to implement functionality.

Electronic output

1: CCM_Registration

The process has this structure:

1. Loop body script, or the CCM_DoRegistration script, which relies on the RegisterRequest function of the Registration component for registering a single request file from the watch folder. The order to process the request files is not determined. For more information on RegisterRequest, see RegisterRequest function.
2. Surrounding loop, which ensures that the CCM_DoRegistration script is called repeatedly.

Loop body script

The CCM_DoRegistration script calls the RegisterRequest function and monitors the result:

- If a request file is processed successfully, the script ends.
- If no request file is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as request files are available, they are picked up one by one without delay. If no request file is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoRegistration script throws an exception. As long as the CCM_Registration process is running, request files are continuously picked up from the watch folder.

Note It is only safe to run multiple registration processes simultaneously if they are monitoring different watch folders. When the same watch folder is monitored, a single request file may be registered in the database more than once, leading to duplicate output.

2: CCM_Application

The process has this structure:

1. Loop body script, or the CCM_DoApplication script, which relies on the ProcessRequest function of the Application component for processing a single request record. The order to process the request records is not determined. For more information on ProcessRequest, see ProcessRequest function.
2. Surrounding loop, which ensures that the CCM_Application script is called repeatedly.

Loop body script

The CCM_DoApplication script calls the ProcessRequest function and monitors the result:

- If a request record is processed successfully, the script ends.
- If no request record is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as request records are available, they are picked up one by one without delay. If no request record is available, the next check is delayed for five seconds.

3: CCM_Communication

The process has this structure:

1. Loop body script, or the CCM_DoCommunication script, which relies on the CommunicateCorrespondence function of the Communication component for processing a single correspondence. The order to process the correspondences is not determined. For more information on CommunicateCorrespondence, see CommunicateCorrespondence function.
2. Surrounding loop, which ensures that the CCM_DoCommunication script is called repeatedly.

Loop body script

The CCM_DoCommunication script calls the CommunicateCorrespondence function and monitors the result:

- If a correspondence is processed successfully, the script ends.
- If no correspondence is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as correspondences are available, they are picked up one by one without delay. If no correspondence is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoCommunication script throws an exception. As long as the CCM_Communication process is running, correspondences are continuously processed into electronic communications.

It is safe to run multiple communication processes simultaneously, even if they process correspondences from the same run-time database.

4: CCM_ElectronicDelivery

The process has this structure:

1. Loop body script, or the CCM_DoElectronicDelivery script, which relies on the DeliverCommunication function of the ElectronicDelivery component for processing a single electronic communication. The order to process the communications is not determined. For more information on DeliverCommunication, see DeliverCommunication function (for electronic output).
2. Surrounding loop, which ensures that the CCM_DoElectronicDelivery script is called repeatedly.

Loop body script

The CCM_DoElectronicDelivery script calls the DeliverCommunication function and monitors the result:

- If an electronic communication is processed successfully, the script ends.
- If no electronic communication is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as electronic communications are available, they are picked up one by one without delay. If no electronic communication is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoElectronicDelivery script throws an exception. As long as the CCM_ElectronicDelivery process is running, electronic communications are continuously distributed.

It is safe to run multiple electronic delivery processes simultaneously, even if they process communications from the same run-time database.

Printed output

1: CCM_Registration

The process has this structure:

1. Loop body script, or the CCM_DoRegistration script, which relies on the RegisterRequest function of the Registration component for registering a single request file from the watch folder. The order to process the request files is not determined. For more information on RegisterRequest, see RegisterRequest function.
2. Surrounding loop, which ensures that the CCM_DoRegistration script is called repeatedly.

Loop body script

The CCM_DoRegistration script calls the RegisterRequest function and monitors the result:

- If a request file is processed successfully, the script ends.
- If no request file is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as request files are available, they are picked up one by one without delay. If no request file is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoRegistration script throws an exception. As long as the CCM_Registration process is running, request files are continuously picked up from the watch folder.

Note It is only safe to run multiple registration processes simultaneously if they are monitoring different watch folders. When the same watch folder is monitored, a single request file may be registered in the database more than once, leading to duplicate output.

2: CCM_Application

The process has this structure:

1. Loop body script, or the CCM_DoApplication script, which relies on the ProcessRequest function of the Application component for processing a single request record. The order to process the request records is not determined. For more information on ProcessRequest, see ProcessRequest function.
2. Surrounding loop, which ensures that the CCM_Application script is called repeatedly.

Loop body script

The CCM_DoApplication script calls the ProcessRequest function and monitors the result:

- If a request record is processed successfully, the script ends.
- If no request record is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as request records are available, they are picked up one by one without delay. If no request record is available, the next check is delayed for five seconds.

3: CCM_Communication

The process has this structure:

1. Loop body script, or the CCM_DoCommunication script, which relies on the CommunicateCorrespondence function of the Communication component for processing a single correspondence. The order to process the correspondences is not determined. For more information on CommunicateCorrespondence, see CommunicateCorrespondence function.

2. Surrounding loop, which ensures that the CCM_DoCommunication script is called repeatedly.

Loop body script

The CCM_DoCommunication script calls the CommunicateCorrespondence function and monitors the result:

- If a correspondence is processed successfully, the script ends.
- If no correspondence is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as correspondences are available, they are picked up one by one without delay. If no correspondence is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoCommunication script throws an exception. As long as the CCM_Communication process is running, correspondences are continuously processed into electronic communications.

It is safe to run multiple communication processes simultaneously, even if they process correspondences from the same run-time database.

4: CCM_PrintDelivery

The process has this structure:

1. Loop body script, or the CCM_DoPrintDelivery script, which relies on the DeliverCommunication function of the PrintDelivery component for processing a single print communication. The order to process print communications is not determined. For information on DeliverCommunication, see DeliverCommunication function (DeliverCommunication function (for printed output)).
2. Surrounding loop, which ensures that the CCM_DoPrintDelivery script is called repeatedly.

Loop body script

The CCM_DoPrintDelivery script calls the DeliverCommunication function and monitors the result:

- If a print communication is processed successfully, the script ends.
- If no print communication is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as print communications are available, they are picked up one by one without delay. If no print communication is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoPrintDelivery script throws an exception. As long as the CCM_DoPrintDelivery process is running, print communications are continuously processed into process records.

It is safe to run multiple print delivery processes simultaneously, even if they process print communications from the same run-time database.

5: CCM_XpsConversion

The process has this structure:

1. Loop body script, or the CCM_DoXpsConversion script, which relies on the CreateRenditionForSelectedProcess function of the XpsConversion component for processing a single process record. The order to implement processing of the process records is not determined. For more information on CreateRenditionForSelectedProcess, CreateRenditionForSelectedProcess function.
2. Surrounding loop, which ensures that the CCM_DoXpsConversion script is called repeatedly.

Loop body script

The CCM_DoXpsConversion script calls the CreateRenditionForSelectedProcess function and monitors the result:

- If a process record is processed successfully, the script ends.
- If no process record is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as process records are available for conversion, they are picked up one by one without delay. If no process record is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoXpsConversion script throws an exception. As long as the CCM_XpsConversion process is running, process records are continuously transformed by XPS conversion.

It is safe to run multiple XPS conversion processes simultaneously, even if they transform process records from the same run-time database.

Note Before CCM 5.1.1.1, running CCM_XpsConversion in parallel with CCM_Bundling could lead to a deadlock in the database, which set process records in the error state. If you encounter this issue, do not resubmit such process records, but abandon them instead. Please contact Technical Support if you have already resubmitted process records that were put in error due to a database deadlock.

6: CCM_Bundling

1. Timer body script, or the CCM_DoBundling script, which selects a customizable set of process records for a specific channel from the database, and bundles them into envelopes in one go by calling the Run function from the Bundling component. For more information on the Run function and the available configuration functions, see Run function in bundling and Bundling component functions.
2. Surrounding timer, which ensures that the CCM_DoBundling script is called at a particular time.

Loop body script

By default, CCM_DoBundling behaves as follows:

- Selects all available process records for the Print channel at once.
- Does not group process records; each process record gets its own envelope.
- Does not restrict the size of the created envelopes.
- Does not compose a new cover letter.
- Uses Default_PostageDefinition to determine the postage class of each created envelope.
- Sends all created envelopes to the Default_OutsourcingPrinter printer.

You can customize this behavior by editing the CCM_DoBundling script. In the script, add or change the calls to the configuration functions that are provided by the Bundling component.

Surrounding timer

The surrounding timer runs the CCM_DoBundling script at a set interval. By default, the script is run once a day at 2 AM. The running stops if the process receives a Stop or Cancel signal, or if the CCM_DoBundling script throws an exception.

Error handling

After the CCM_DoBundling script ends, the surrounding timer calls it again at the next scheduled run.

Note It is only safe to run multiple bundling processes simultaneously if the process records that they select are disjointed; that is, they are guaranteed to select different process records from the database.

7: CCM_Stacking

The process has this structure:

- Timer body script, or the CCM_DoStacking script, which selects a customizable set of envelopes from the database and batches them into stacks by calling the Run function from the Stacking component. For more information on the Run function and the available configuration functions, see Run function in stacking and Stacking component functions.
- Surrounding timer, which ensures that the CCM_DoStacking script is called at a particular time.

Loop body script

By default, CCM_DoStacking behaves as follows:

- Selects all available envelopes at once
- Batches all envelopes for the same printer in one stack
- Does not sort the envelopes within a stack
- Does not restrict the size of the created stacks

You can customize this behavior by editing the CCM_DoStacking script. To do so, add or change calls to the configuration functions provided by the Stacking component.

Surrounding timer

The surrounding timer runs the CCM_DoStacking script at a set interval. By default, the script is run once a day at 3 AM. The run stops if the process receives a Stop or Cancel signal, or if the CCM_DoStacking script throws an exception.

Error handling

After the CCM_DoStacking script ends, the surrounding timer calls it again at the next scheduled run.

Note It is only safe to run multiple stacking processes simultaneously if the envelopes that they select are disjointed; that is, they are guaranteed to select different envelopes from the database.

8: CCM_Streaming

The process has this structure:

1. Loop body script, or the CCM_DoStreaming script, which relies on the Run function of the Streaming component for processing a single stack. The order to process stacks is not determined. For more information on the Run function, see Run function in streaming.
2. Surrounding loop, which ensures that the CCM_DoStreaming script is called repeatedly.

Loop body script

The CCM_DoStreaming script calls the Run function and monitors the result:

- If a stack is processed successfully, the script ends.
- If no stack is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as stacks are available for streaming, they are picked up one by one without delay. If no stack is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoStreaming script throws an exception. As long as the CCM_Streaming process is running, stacks are continuously streamed.

It is safe to run multiple streaming processes simultaneously, even if they transform stacks from the same run-time database.

9: CCM_Conversion

The process has this structure:

1. Loop body script, or the CCM_DoConversion script, which relies on the ConvertStack function of the Conversion component for processing a single stack. The order to process stacks is not determined. For more information on the ConvertStack function, see ConvertStack function .
2. Surrounding loop, which ensures that the CCM_DoConversion script is called repeatedly.

Loop body script

The CCM_DoConversion script calls the ConvertStack function and monitors the result:

- If a stack is processed successfully, the script ends.
- If no stack is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as stacks are available for conversion, they are picked up one by one without delay. If no stack is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoConversion script throws an exception. As long as the CCM_Conversion process is running, stacks are continuously converted.

It is safe to run multiple conversion processes simultaneously, even if they transform stacks from the same run-time database.

10: CCM_Distribution

The process has this structure:

1. Loop body script, or the CCM_DoDistribution script, which relies on the DistributeStack function of the Distribution component for processing a single stack. The order to process stacks is not determined. For more information on the DistributeStack function, see DistributeStack function.
2. Surrounding loop, which ensures that the CCM_DoDistribution script is called repeatedly.

Loop body script

The CCM_DoDistribution script calls the DistributeStack function and monitors the result:

- If a stack is processed successfully, the script ends.
- If no stack is found, the script waits five seconds, and then ends.
- If an exception marked as Handled is thrown, the script ends.
- If an exception is thrown that is not marked as Handled, the script throws an exception.

When the script ends normally, the surrounding loop immediately calls the script again. Therefore, as long as stacks are available for distribution, they are picked up one by one without delay. If no stack is available, the next check is delayed for five seconds.

Surrounding loop

The surrounding loop continues indefinitely until the process receives a Stop or Cancel signal, or the CCM_DoDistribution script throws an exception. As long as the CCM_Distribution process is running, stacks are continuously distributed.

It is safe to run multiple distribution processes simultaneously, even if they distribute stacks from the same run-time database.

CCM_CleanUpAll

The CCM_CleanUpAll process consists of a CCM_DoCleanUp script, which is called regularly by the timer.

You can customize the behavior of this process by editing the CCM_DoCleanUp script. To do so, you can change the functions used in the GarbageCollection component (see GarbageCollection component functions).

Processing component functions

This section describes the configuration functions of the processing components.

RegisterRequest function

The `RegisterRequest` function registers a request file from the watch folder by creating a request record for it in the run-time database and moving the request file to the storage folder, thus removing it from the watch folder. It does not parse the request file itself.

Parameters

The `RegisterRequest` function does not have any parameters. It searches the configured watch folder for XML files:

- If an XML file is found, it is assumed to be a request file, and it is processed.
- If multiple XML files are found, they are picked up one be one, but the order is not determined.

Result

If an XML file is processed successfully, `RegisterRequest` returns true. If no XML file is found in the watch folder, `RegisterRequest` returns false.

When an error is encountered, the erroneous request file is moved to the error folder, and an error information file is written next to it.

Note You cannot call `RegisterRequest` from different processes simultaneously, if the same watch folder is used.

ProcessRequest function

The `ProcessRequest` function performs these actions in the following order:

1. Moves the input request record to the Busy status and sets its start time.
2. Parses the request file that belongs to the request record:
   - If the request file is a correspondence or import request, parsing results in a set of correspondences.

- If the request file is an application request, parsing results in a set of application events, which are transformed into correspondences by applying the application rules that are configured on the system.

3. For import requests, verifies that an accompanying Document Pack is also submitted.
4. Stores the correspondences in the run-time database.
5. Moves the input request record to the Finished status and sets its finish time.

Parameters

The `ProcessRequest` function does not have any input parameters. It searches the run-time database for a request record in the Waiting status. If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a request record is processed successfully, `ProcessRequest` returns true. If no request record is available, `ProcessRequest` returns false.

If an error is encountered, `ProcessRequest` moves the input request record to the Error status and sets the error time and error message.

It is safe to call `ProcessRequest` from different processes simultaneously, even if they process request records from the same run-time database.

CommunicateCorrespondence function

The `CommunicateCorrespondence` function performs these actions in the following order:

1. Moves the input correspondence to the Busy status and sets its start time.
2. Executes the document composition and internally stores the resulting Document Pack Template of the correspondence.
3. Applies the communication rules configured on the system to the correspondence, which results in a set of communications.
4. For each communication that will be printed, selects the representation that is best suited for printing, using PDF > DOCX > XPS > TIFF as the preferred order. If none of these formats exist, an error is reported.
5. Stores the communications in the run-time database.
6. Moves the input correspondence to the Finished status and sets its finish time.

Parameters

The `CommunicateCorrespondence` function does not have any input parameters. It searches the run-time database for a correspondence record in the Waiting status. If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a correspondence is processed successfully, `CommunicateCorrespondence` returns true. If no correspondence is available, `CommunicateCorrespondence` returns false.

If an error is encountered, `CommunicateCorrespondence` moves the input correspondence to the Error status and sets the error time and error message.

It is safe to call `CommunicateCorrespondence` from different processes simultaneously, even if they process correspondences from the same run-time database.

DeliverCommunication function (for electronic output)

The `DeliverCommunication` function in electronic delivery performs these functions in the following order:

1. Moves the input electronic communication to the Busy status and sets its start time.
2. Creates a single document pack for the communication from the composed Document Pack for the correspondence by doing the following:
   a. Removing slots not selected by the communication
   b. Optionally adding the cover letter if it was created during the communication
   c. Restricting the Document Pack to the channel selected by the communication
3. Creates a metadata file for the communication
4. Invokes the Electronic Delivery exit point on CCM Core
5. Moves the input electronic communication to the Finished status and sets its finish time.

Parameters

The process does not have any input parameters. It searches the run-time database for an electronic communication record in the Waiting status. If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If an electronic communication is processed successfully, `DeliverCommunication` returns true. Otherwise, it returns false.

If an error is encountered, `DeliverCommunication` moves the input electronic communication to the Error status and sets the error time and error message.

It is safe to call `DeliverCommunication` from different processes simultaneously, even if they process electronic communications from the same run-time database.

DeliverCommunication function (for printed output)

The `DeliverCommunication` function in print delivery performs these actions in the following order:

1. Moves the input print communication to the Busy status and sets its start time.
2. Converts all DOC files of the communication to XPS. When MS Word is configured for converting DOCX to XPS, it also converts all DOCX files of the communication to XPS.

   Note If Microsoft Word is not installed on the server, CCM B&OM cannot create print output for Document Packs that contain DOC files.

3. Creates a process record for the input print communication in the run-time database.
4. Creates job records in the run-time dataset for all documents of the print communication.
5. Moves the input print communication to the Finished status and sets its finish time.

Parameters

The process does not have any input parameters. It searches the run-time database for a print communication record in the Waiting status. If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a print communication is processed successfully, `DeliverCommunication` returns true. Otherwise, it returns false.

If an error is encountered, `DeliverCommunication` moves the input print communication to the Error status and sets the error time and error message.

It is safe to call `DeliverCommunication` from different processes simultaneously, even if they process print communications from the same run-time database.

CreateRenditionForSelectedProcess function

The `CreateRenditionForSelectedProcess` function performs these actions in the following order:

1. Moves the input process record to the Busy status.
2. For all job records that belong to the process record and have state Imported:
   a. The associated file is converted to XPS, either by using the built-in Rendition functionality, if the file is a Word or Tiff file, or by invoking DocBridge, if the file is a PDF file.

      Note If Microsoft Word is not installed on the server, CCM B&OM cannot create print output for Document Packs that contain DOC files.

   b. The state of the job record is changed to "Rendition created," and a link is added from the job record to the created XPS file.
3. When all jobs are converted, the input process record is moved back to the Waiting status, and the process state is changed to "Rendition created."

Parameters

The `CreateRenditionForSelectedProcess` does not have any input parameters. It searches the run-time database for a process record in the Waiting status and that has process state Imported. If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a process record is processed successfully, `CreateRenditionForSelectedProcess` returns its process identifier represented as a string. Otherwise, it returns the empty string.

If an error is encountered, `CreateRenditionForSelectedProcess` moves the input process record to the Error status and sets its message property to the error message.

It is safe to call `CreateRenditionForSelectedProcess` from different processes simultaneously, even if they transform process records from the same run-time database.

Bundling component functions

The Bundling component provides the following configuration functions.

Functions	Description
`SelectByChannelName`	*Required*. This function configures the channel for which process records are bundled.
`SetOutsourcingPrinterOId` `SetPostageDefinitionOId`	*Required*. These functions set the printer and postage definition to be used for the created envelopes.

`SelectByOrganisationalMetadata`
`SelectByPrintCommunicationId`
`SelectByRequestReference`
`SelectByCorrespondenceReference`
`SelectByCommunicationLabel`
`SelectAtMost`
	Optional. These functions configure which process records are selected for bundling. By default, all available process records for the configured channel are selected.

`SortByXPath` `SortByXPathDesc`	*Optional*. These functions define criteria for sorting process records when they are combined into an envelope. By default, process records are not sorted, and the order of process records in an envelope is undefined.
`AddChannelCoverLetter`	*Optional*. This function configures the composition of a new cover letter for created envelopes. By default, a new cover letter is not created.
`GroupByXPath`	*Optional*. This function configures when process records must be combined into the same envelope, and when they must get their own envelopes. By default, each process record is put in its own envelope. You can call the functions multiple times.
`ClearConfiguration`	*Optional*. This function clears all settings configured by the preceding functions.

Note You can only call a "SelectBy" function of the same type once in a single script. To override the settings configured with the previously called functions, use `ClearConfiguration`.

- `SelectByChannelName`
  Parameter: `object name` (String, case-insensitive). Set to a specific object name so bundling only selects process records distributed over the channel with this name.
- `SelectByOrganisationalMetadata`

Set the following parameters to configure bundling to only select the communications that have the given organisational metadata field with the given value.

1. `key` (String, runtime database collation[2]). Set to a specific organisational metadata field.
2. `value` (String, runtime database collation). Set to a specific value of this organisational metadata field.

- `SelectByPrintCommunicationId`

  Parameter: `identifier` (BigInteger). Set to a specific print communication identifier so bundling only selects process records that originate from this print communication.

- `SelectByRequestReference`

  Parameter: `reference` (String, runtime database collation). Set to a specific request reference so bundling only selects process records that originate from this request.

- `SelectByCorrespondenceReference`

  Parameter: `reference` (String, runtime database collation). Set to a specific correspondence reference so bundling only selects process records that originate from this correspondence.

- `SelectByCommunicationLabel`

  Parameter: `label` (String, runtime database collation). Set to a specific communication label so bundling only selects process records that originate from this communication.

- `SelectAtMost`

  Parameter: `count` (Integer). Set to a specific number so bundling only selects at most this number of process records. If there are more process records available, it is not determined which ones are selected.

- `SortByXPath`

  Parameter: `xpath` (String, case-sensitive). Set to a specific XPath expression so bundling sorts process records when they are combined into an envelope. The process records are sorted based on values computed by evaluating the XPath expression on the print communications from which these process records originate. The input for the XPath expression is an XML serialization according to the CcmBomBundlingInput XSD.

  The computed values are interpreted as text and compared using case-sensitive ordinal sorting rules. Sorting criteria specified by `SortByXPath` result in a sort in ascending order.

- `SortByXPathDesc`

  Parameter: `xpath` (String, case-sensitive). Set to a specific XPath expression so bundling sorts process records when they are combined into an envelope. The process records are sorted based on values computed by evaluating the XPath expression on the print communications from which the process records originate. The input for the XPath expression is an XML serialization according to the CcmBomBundlingInput XSD.

  The computed values are interpreted as text and compared using case-sensitive ordinal sorting rules. Sorting criteria specified by `SortByXPathDesc` result in a sort in descending order.

  You can call `SortByXPath` and `SortByXPathDesc` more than once to specify multiple sorting criteria. When sorting, the criteria is evaluated in the order in which they are specified.

[2] For all parameters with this remark, the collation is determined by the Batch & Output Management runtime database.

Note A case-sensitive ordinal comparison method indicates that string comparison must use successive Unicode UTF-16 encoded values of the string (code unit by code unit comparison). This sorting method is culture invariant.

- `AddChannelCoverLetter`
  Set the following parameter to configure bundling to compose a new cover letter for each envelope that it creates:
  `alsoForSingletons` (Boolean). Set to True to add a cover letter for all created envelopes, regardless of their size. Set to False to only add a cover letter for envelopes that consist of more than one process record.
  The Document Pack Template that composes this cover letter must be configured on the channel object for which envelopes are created. Otherwise, calling `AddChannelCoverLetter` results in an error. The Data Backbone defined by the CcmBomCoverLetterBackbone XSD is used for the cover letter composition.
- `GroupByXPath`
  Parameter: `xpath` (String, case-sensitive). Set to a specific XPath expression so bundling finds the process records with values evaluating to this XPath expression and then combines them into the same envelope. The XPath expressions are evaluated on XML serializations of the print communications from which the process records originate, according to the CcmBomBundlingInput XSD.
  You can call `GroupByXPath` more than once to specify multiple grouping criteria. All XPath expressions must evaluate to the same value in order for process records to be combined. If you omit calling `GroupByXPath`, all process records are put into their own envelopes.

  Note Only process records for the same channel can be combined into the same envelope.

- `SetOutsourcingPrinterOId`
  Parameter: `object identifier` (String, case-sensitive). Set to a specific object identifier so bundling marks all created envelopes with the outsourcing printer that has this identifier. The envelopes are distributed to this printer.
- `SetPostageDefinitionOId`
  Parameter: `object identifier` (String, case-sensitive). Set to a specific object identifier so bundling classifies the created envelopes according to the postage classes defined in the postage definition with this identifier.

  Note The postage definition set by this function may not have any enclosure specified.

- `ClearConfguration`
  Removes the settings configured by `SelectByChannelName`, `SelectByOrganisationalMetadata`, `SelectByPrintCommunicationId`, `SelectByRequestReference`, `SelectByCorrespondenceReference`, `SelectByCommunicationLabel`, `SelectAtMost`, `GroupByXPath`, `AddChannelCoverLetter`, `SetOutsourcingPrinterOId`, and `SetPostageDefinitionOId`. To reconfigure bundling, you need to call these functions anew.

Example

The following example shows a customization of the CCM_DoBundling script.

```
; Initialize the Bundling component.
Bundling = ModusProcess.GetComponent("Bundling")
Bundling.ClearConfiguration()

; Only select the CustomerOriginal communications for the request
RQ-ADF for channel Channel-Name with EMEA for the AccountingGroup
in the organisational metadata.
Bundling.SelectByChannelName("Channel-Name")
Bundling.SelectByRequestReference("RQ-ADF")
Bundling.SelectByCommunicationLabel("CustomerOriginal")
Bundling.SelectByOrganisationalMetadata("AccountingGroup", "EMEA")

; Select at most 1000 communications.
Bundling.SelectAtMost(1000)

; Make sure that (only) communications for the same recipient end up in the same
 envelope.
Bundling.GroupByXPath("ccm:Recipient/ct:Name")

;Make sure that processes are sorted by CaseIdentifier.
Bundling.SortByXPath("ccm:OrganisationalMetadata/data:CaseIdentifier")


; Configure the printer that will be set on each created envelope, and the postage
 definition that will be used.
Bundling.SetOutsourcingPrinterOId("04032016-64-opr-ccmebatch")
Bundling.SetPostageDefinitionOId("23102008-160-pod-dm")

; Run bundling.
Protocol("Running bundling.", 5)
EnvelopeCount = Bundling.Run()
Protocol("Bundling created " + EnvelopeCount + " envelope(s).", 5)
```

Run function in bundling

The `Run` function performs these actions for the bundling component:

1. Moves the input process records to the Busy status.
2. In each created envelope, it sorts the process records, according to the configured sorting criteria.
3. Combines the process records into envelopes, according to the specified grouping criteria and paper count and weight limits.
   Only process records for the same channel can be combined into the same envelope. The organisational metadata values shared by all process records are taken over in the envelope.
4. Assigns all created envelopes to the configured outsourcing printer.
5. Computes the paper count and weight of each envelope, using the tray information from the assigned printer, and the forms of the documents (jobs) in the envelopes.
6. Assigns a postage class to each envelope, according to the configured postage definition.
7. Writes the created envelopes to the run-time database. In addition, it performs these steps:
   a. Moves the process records to the Finished status.
   b. Updates the process records with the envelope.
   c. Updates the job records with the envelope and with the printer to which the envelope is assigned.

Parameters

The `Run` function does not have any input parameters. It selects process records from the run-time database, according to the configured selection criteria. Only process records, which are in the Waiting status, have process state "Rendition created," and are not yet assigned to an envelope, are considered for selection.

Result

The `Run` function returns the number of created envelopes.

If an error is encountered, the `Run` function moves the input process records to the Error status and sets the error message.

Stacking component functions

The Stacking component provides the following configuration functions.

Functions	Description

`SelectByOrganisationalMetadata`
`SelectByEnvelopeId`
`SelectByPostageName`
`SelectByChannelName`
`SelectByOutsourcingPrinterOId`
`SelectAtMost`
	Optional. These functions configure which envelopes are selected for stacking. By default, all available envelopes are selected.

`SortByXPath` `SortByXPathDesc`	*Optional*. These functions define criteria used for sorting envelopes when they are combined into a stack. By default, envelopes are not sorted, and the order of envelopes in a stack is undefined.

Note When calling a particular "SelectBy" function more than once in one script, only the last call is taken into account, and the previous calls are silently ignored.

- `SelectByOrganisationalMetadata`
  Set the following parameters to configure stacking to only select the communications that have the given organisational metadata field with the given value.
  1. `key` (String, runtime database collation[3]). Set to a specific organisational metadata field.
  2. `value` (String, runtime database collation). Set to a specific value of this organisational metadata field.
- `SelectByEnvelopeId`
  Parameter: `identifier` (BigInteger). Set to a specific envelope identifier so stacking only selects the envelope with this identifier.
- `SelectByPostageName`

[3] For all parameters with this remark, the collation is determined by the Batch & Output Management runtime database.

Parameter: `name` (String, runtime database collation). Set to a specific postage class name so stacking only selects envelopes assigned to the postage class with this name.

- `SelectByChannelName`
  Parameter: `object name` (String, case-insensitive). Set to a specific channel object name so stacking only selects envelopes assigned to the channel with this object name.
- `SelectByOutsourcingPrinterOId`
  Parameter: `object identifier` (String, case-sensitive). Set to a specific outsourcing printer object identifier so conversion only selects stacks assigned to the outsourcing printer with this object identifier.
- `SelectAtMost`
  Parameter: `count` (Integer). Set to a specific number so stacking selects at most this number of envelopes. If there are more envelopes available, it is not determined which ones are selected. If there are remaining envelopes, you need to call the CCM_Stacking process again.
- `SortByXPath`
  Parameter: `xpath` (String, case-sensitive). Set to a specific XPath expression so stacking sorts envelopes when they are combined into a stack. The envelopes are sorted based on values computed by evaluating the XPath expression on the XML serialization of the envelopes according to the CcmBomStackingInput XSD. The sender and recipient in the XML serialization are taken from the first communication that ended in the envelope.

  The computed values are interpreted as text and are compared using case-sensitive ordinal sorting rules. Sorting criteria specified by `SortByXPath` result in a sort in ascending order.
- `SortByXPathDesc`
  Parameter: `xpath` (String, case-sensitive). Set to a specific XPath expression so stacking sorts envelopes when they are combined into a stack. Sorting criteria specified by `SortByXPathDesc` result in a sort in descending order.

  The envelopes are sorted based on values computed by evaluating the XPath expression on the XML serialization of the envelopes according to the CcmBomStackingInput XSD. The sender and recipient in the XML serialization are taken from the first communication that ended in the envelope.

  The computed values are interpreted as text and compared using case-sensitive ordinal sorting rules.

  You can call `SortByXPath` and `SortByXPathDesc` more than once to specify multiple sorting criteria. When sorting, the criteria is evaluated in the order in which they are specified.

  Note A case-sensitive ordinal comparison method indicates that string comparison must use successive Unicode UTF-16 encoded values of the string (code unit by code unit comparison). This sorting method is culture invariant.

Example

The following is an example of a customization of CCM_DoStacking, which consists of two scripts that must be called separately.

Script 1

```
; Initialize the Stacking component.
Stacking = ModusProcess.GetComponent("Stacking")

; Create stacks for all envelopes with high priority.
; Repeatedly stack at most 10 envelopes at the time, until all have been stacked.
Done = false
While (not Done)
```

```
                Stacking.SelectByOrganisationalMetadata("Priority", "High")
                Stacking.SelectAtMost(10)
                Stacking.SortByXPath("ccm:Recipient/ct:Zipcode")
                StackCount = Stacking.Run()
                Protocol("Stacking created " + StackCount + " high priority stack(s).",
 5)
                If (StackCount = 0)
                                Done = true
                End-If
End-While
```

Script 2

```
; Initialize the Stacking component.
Stacking = ModusProcess.GetComponent("Stacking")

; Create stacks for all envelopes with low priority.
; Repeatedly stack at most 1000 envelopes at the time, until all have been stacked.
Done = false
While (not Done)
                Stacking.SelectByOrganisationalMetadata("Priority", "Low")
                Stacking.SelectAtMost(1000)
                Stacking.SortByXPath("ccm:Recipient/ct:Zipcode")
                StackCount = Stacking.Run()
                Protocol("Stacking created " + StackCount + " low priority stack(s).",
 5)
                If (StackCount = 0)
                                Done = true
                End-If
End-While
```

As the Stacking instance is shared over two calls, the second `Stacking.Run` call uses the metadata filtering that was set in the first call.

Run function in stacking

The `Run` function performs these actions for the stacking component:

1. Moves the input envelopes records to the Busy status.
2. Batches envelopes into stacks.
3. In each created stack, it sorts the envelopes according to the configured sorting criteria
4. Puts all envelopes with the same channel and the same printer in the same stack. The organisational metadata values shared by all envelopes are taken over in the stack.
5. Writes the created stacks to the run-time database with stack type "Streaming (20)." It also performs these steps:
   a. Moves the input envelopes to the Finished status.
   b. Updates the input envelopes with the stack.

Parameters

The `Run` function does not have any input parameters. It selects envelopes from the run-time database, according to the configured selection criteria. Only envelopes in the Waiting status are considered for selection.

Result

The `Run` function returns the number of created stacks.

If an error is encountered, the `Run` function moves the input envelopes to the Error status.

Streaming component function

Run function in streaming

The `Run` function performs these actions for the streaming component:

1. Moves the input stack to the Busy status.
2. Creates stream files for the stack that contain instructions for page modifications, and an information file. The stream files are internal at this step, and they are stored in the internal storage folder.
3. Moves the input stack back to the Waiting status and updates the stack type to "Convert (30)".

Parameters

The `Run` function does not have any input parameters. It searches the run-time database for a stack in the Waiting status and stack type "Streaming (20)." If multiple records are available, it is not determined which one is selected.

Result

If a stack is processed successfully, `Run` returns its stack identifier, represented as a string. If no stack is available, `Run` returns the empty string.

If an error is encountered, the `Run` function moves the input stack to the Error status and sets the error message.

It is safe to call the `Run` function from different processes simultaneously, even if they transform stacks from the same run-time database.

Conversion component functions

The Conversion component provides the following configuration functions.

Functions	Description
`SelectByStackId`, `SelectByChannelName`, `SelectByOutsourcingPrinterName`/ `SelectByOutsourcingPrinterOId`, `SelectByOrganisationalMetadata`	*Optional*. Each call of the Run function processes a single stack from a set of available stacks. These functions configure which stacks are available. By default, all streamed stacks are available.

Note When calling a particular "SelectBy" function more than once, only the last call is taken into account, and the previous calls are silently ignored.

- `SelectByStackId`
  Parameter: `identifier` (BigInteger). Set to a specific stack identifier so conversion only selects the stack with this identifier.

- `SelectByChannelName`
  Parameter: `name` (String, case-insensitive). Set to a specific channel object name so conversion only selects stacks assigned to the channel with this object name.
- `SelectByOutsourcingPrinterName` / `SelectByOutsourcingPrinterOId`
  You can call either of these functions to specify the outsourcing printer, but only the last call is taken into account.
  1. `SelectByOutsourcingPrinterName`
     Parameter: `name` (String, case-insensitive). Set to a specific outsourcing printer object name so conversion only selects stacks assigned to the outsourcing printer with this object name.
  2. `SelectByOutsourcingPrinterOId`
     Parameter: `object identifier` (String, case-sensitive). Set to a specific outsourcing printer object identifier so conversion only selects stacks assigned to the outsourcing printer with this object identifier.
- `SelectByOrganisationalMetadata`
  Set the following parameters to configure conversion to only select stacks that have the given organisational metadata field with the given value.
  1. `key` (String, runtime database collation[4]). Set to a specific organisational metadata field.
  2. `value` (String, runtime database collation). Set to a specific value of this organisational metadata field.

Example

The following is an example of a customization of the CCM_DoConversion script.

```
; Initialize the Conversion component.
 Conversion = ModusProcess.GetComponent("Conversion")

 ; Pick up the stacks for the channel 'Print', which are printed on 'ColorPrinter', and
 which do not contain a ContractDocument.
 Conversion.SelectByChannelName("Print")
 Conversion.SelectByOutsourcingPrinterName("ColorPrinter")
 Conversion.SelectByOrganisationalMetadata("HasContractDocument", "no")

 ; Run conversion.
 StackId = Conversion.ConvertStack()
 If (StackId <> "")
  ; Stack found and spool file created. Report success and allow next loop iteration to
 start.
  Protocol("Successfully created spool file for stack with id " + StackId + ".", 8)
 Else
  ; No stack record found. Report and wait before starting the next loop iteration.
  Protocol("No stack record found. Waiting for " + WaitTime + "ms before trying
 again.", 8)
  Sleep(WaitTime)
 End-If
OnError()
 If (Error.Handled)
  ; Stack record found, but error while processing it. Report failure, but still allow
 next loop iteration to start.
```

[4] The collation for this and the following parameter is determined by the Batch & Output Management runtime database.

```
  Protocol("Put stack record " + Error.InputItem + " in error state due to " +
 Error.Message, 8)
 Else
  ; Fatal error. Bail out.
  Raise("UnrecoverableConversionError", Error.Message)
 End-If
End-Try
```

ConvertStack function

The ConvertStack function performs these actions in the conversion component:

1. Moves the input stack to the Busy status.
2. DocBridge is used to perform the following actions:
   a. The input pages of all documents of all envelopes of the stack are read, one by one. If configured to do so by streaming, pages are inserted.
   b. For each page, the page modifications determined by streaming are carried out.
   c. An output file is written in the configured output format to the internal storage folder, by writing the modified pages to it.
3. Moves the input stack back to the Waiting status and updates the stack type to "Distribute (70)."

Parameters

The ConvertStack function does not have any input parameters. It searches the run-time database for a stack in the Waiting status and stack type "Convert (30)." If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a stack is processed successfully, Run returns its stack identifier, represented as a string. If no stack is available, ConvertStack returns the empty string.

If an error is encountered, the ConvertStack function moves the input stack to the Error status and sets the error message.

Note If you call the ConvertStack function from different CCM_Conversion processes simultaneously, we recommend that you run them in a separate Windows Service.

Distribution component functions

DistributeStack function

The `DistributeStack` function performs these actions in the distribution component:

1. Moves the input stack to the Busy status.
2. Creates a stack description .xml file in the internal storage folder next to the output file. The stack description file conforms to the CcmBomStackDescription XSD. You can find the B&OM XSD files here: `<deploy root>\CCM\Documentation\5.2\Resources\Schemas\Output Management`

3. Executes the exit point configured in the OnStackDistribute property for the Distribution component. If the default script called DistributeStack is used, it copies the print and stack description files from the internal storage folder to the output folder.
4. Moves the input stack to the Finished status, updates the stack type to "Finished (99)," and sets the finish time.
5. When configured, calls the communication exit point for each communication that is part of the stack.

Also, when configured, the function calls the request accounting exit point for each request completed by the distribution of this stack.

Parameters

The `DistributeStack` function does not have any input parameters. It searches the run-time database for a stack in the Waiting status and stack type "Distribute (70)." If multiple records are available, they are picked up one by one, but the order is not determined.

Result

If a stack is processed successfully, `DistributeStack` returns its stack identifier, represented as a string. If no stack is available, `DistributeStack` returns the empty string.

If an error is encountered, the `DistributeStack` function moves the input stack to the Error status and sets the error message.

Note If you call the `DistributeStack` function from different CCM_Distribution processes simultaneously, we recommend that you run them in a separate Windows Service.

GarbageCollection component functions

The GarbageCollection component provides the following configuration functions.

Function	Description
`CleanUpAndAccount`	*Optional*. Performs these actions: 1. For finished requests for which the cleanup delay is passed, it deletes all database entries and files associated with the request or with the correspondences and communications that belong to the request. 2. For finished stacks for which the cleanup delay is passed, it deletes all database entries and files associated with the stack or with the envelopes, processes and jobs that are part of the stack. 3. For abandoned requests, processes, envelopes and stacks, it deletes the entries and files associated with the request itself and with the correspondences and communications that belong to the request. 4. For finished requests that contain abandoned items, it calls the accounting exit points. Document Packs for cover letters composed by CCM_Bundling are not cleaned up from the storage folder with the stack to which they belong, but only removed with the request of the first communication in the envelope.

The function does not have any parameters.

Example

The following is an example of a customization of the CCM_DoCleanUp script.

```
Try
  ; Remove run-time records that belong to finished stacks and finished requests.
  ; Also remove abandoned items and account the completion of requests that contain
 abandoned items.
  GarbageCollection = ModusProcess.GetComponent("GarbageCollection")
  GarbageCollection.CleanUpAndAccount()
OnError()
  ; Garbage collection errors are never handled. Always bail out.
  Raise("UnrecoverableGarbageCollectionError", Error.Message)
End-Try
```

Chapter 2

Custom scripting

Configure the accounting functionality

In CCM B&OM, you can configure and automatically call accounting scripts once a communication is distributed, and/or once a request is completed. With these scripts, you can obtain information about the distributed communication and competed request and write this information to a database.

The accounting functionality is implemented by exit points that you configure on the system. To write the accounting information to a custom database, you need to create and configure this database.

Configure a custom database for accounting

1. Create an empty database to store the accounting information.

   Note You cannot store accounting information in a B&OM database as it can only be accessed by B&OM.

2. In CCM Studio, create a database alias for the empty database.
   To do so, on the **Administration** tab, click **DB Alias Administration** and then click the button to create a new DB alias.
3. Configure the connection settings and check the connection.

   In **SQL Field Template**, to use the standard SQL template, click **Standard Value**. You can leave the **SQL Template** field empty.
4. Save the changes.
5. On the **Administration** tab, click **System Administration** and select the system object.
6. In the **Accounting** section on the right, set the **Database** parameter to the database alias you created, and then save the changes.

   The accounting exit points can now access this database through the `AccountingDatabase` variable that is available in context.

Example accounting scripts

A B&OM installation includes sample accounting scripts that provide a default implementation of the accounting functionality. The example scripts write information about completed requests and

communications to an external database. To use these scripts, you must configure them on the system, and then create and configure an accounting database that must at least contain the following tables:

- An AccountedRequest table with columns Reference, RegistrationTime, and CompletionTime.
  - The Reference column must be able to hold strings of up to 100 characters.
  - The RegistrationTime and CompletionTime must be able to hold dates.
- An AccountedCommunication table with columns RequestReference, CorrespondenceReference, Label, Channel, and RecipientType.
  - All columns must be able to hold strings of up to 100 characters.

Also, to write extra information, you can add custom columns to the tables in your database and include these columns in the accounting scripts.

Note If you configure the accounting sample scripts as the main accounting exit points but no accounting database is configured, the scripts have no effect.

Initiate request accounting

1. On the **Administration** tab, click **System Administration**.
2. Click the system object and, in the **Accounting** section on the right, set the **OnRequestCompletion** parameter to a script that has the **RequestAccounting** context.
   When the parameter is set, the configured script is called automatically whenever a request is completed.

For requests that do not contain abandoned items, the request accounting exit point is called immediately after the last output for a request is distributed by CCM B&OM. This happens in either of the following cases:

- When the CCM_Conversion standard process produces a print file, which contains the remaining communications of a request.
- When the CCM_Communication standard process decides that no communications are distributed for a correspondence, and all other correspondences of the request are either already processed or do not produce output.
- When the CCM_Application standard process decides that all application events of the request are ignored.

For requests that do contain abandoned items, the request accounting exit point is called when the request is cleaned up by the CCM_CleanUpAll process.

The `RequestAccounting` context property makes the following information available to the configured exit point:

- `Request.Reference`. Reference of the completed request.
- `Request.RegistrationTime`. Time when the completed request was picked up by the CCM_Registration process. The UTC time standard is used.
- `Request.CompletionTime`. Time when the request was completed. The UTC time standard is used.
- `Request.DistributedCommunicationsCount`. The number of communications distributed for the completed request.

- `Request.Success`. Flag that indicates whether components of the request were abandoned. If the request does not contain abandoned correspondences or communications, it is set to True. If the request contains abandoned correspondences or communications, it is set to False.
- `Request.PrintedPageCount`. The number of pages printed for the completed request. This number does not include communications created for electronic delivery, disposed or envelope cover letters, or extra pages added during streaming.
- `AccountingDatabase`. Name of the database alias configured on the system for the accounting database. When no database is configured, this is an empty string.

Example request accounting script

```
; Store accounting information in database, but only if an accounting database has been
 configured on the system.
If (AccountingDatabase <> "")

  ; Retrieve information from distributed communication.
  Reference = Request.Reference
  RegistrationTime = Request.RegistrationTime.ToString()
  CompletionTime = Request.CompletionTime.ToString()

  ; Create INSERT query.
  StringListClear("Sql")
  StringListClear("SqlParameters")
  StringListAdd("Sql", "INSERT INTO AccountedRequest VALUES
 (:Reference, :RegistrationTime, :CompletionTime)")
  StringListSetValue("SqlParameters", "Reference", "is")
  StringListSetValue("SqlParameters", "RegistrationTime", "i@")
  StringListSetValue("SqlParameters", "CompletionTime", "i@")

  ; Execute INSERT query.
  InitDatabase(AccountingDatabase)
  InitDataSet("DataSet", AccountingDatabase)
  SetExecSQL("DataSet", "Sql", "SqlParameters")
  CloseDatabase(AccountingDatabase)

End-If
```

Note This sample script is included in a B&OM installation. To use this script, you still must configure it on the system.

Initiate communication accounting

1. On the **Administration** tab, click **System Administration**.
2. Click the system object and, in the **Accounting** section on the right, set the **OnCommunicationDistribution** parameter to a script that has the **CommunicationAccounting** context.

   When the parameter is set, the configured script is called automatically whenever a communication is distributed.

The communication accounting exit point is called by the CCM_Conversion standard process once a print file is created in the configured output folder. The exit point is called once for each communication that is part of the print file. If the created print file completes requests, the request accounting exit point is called after the communication accounting exit point. This exit point is **not** called for abandoned communications.

The `CommunicationAccounting` context property makes the following information available to the configured communication accounting exit point:

- `Communication.RequestReference`. Reference of the request with the distributed communication.
- `Communication.CorrespondenceReference`. Reference of the correspondence with the distributed communication.
- `Communication.Label`. Label of the distributed communication.
- `Communication.Channel`. Name of the channel used to distribute the communication.
- `Communication.Printer`. Name of the printer used to distribute the communication.
- `Communication.HasSender`. Flag that indicates whether the sender information is available for the distributed communication.
- `Communication.GetSenderData(field)`. Function that produces the value of a contact field in the sender information of the distributed communication. Throws an exception if no sender information is available, or the contact field does not exist.
- `Communication.RecipientType`. Recipient type of the distributed communication.
- `Communication.GetRecipientData(field)`. Function that produces the value of a contact field in the recipient information of the distributed communication. Throws an exception if the contact field does not exist.
- `Communication.GetOrganisationalMetadata(key)`. Function that produces the value of a key in the organisational metadata of the distributed communication. Throws an exception if the key does not exist.
- `Communication.DistributionTime`. Time when the communication was distributed. The UTC time standard is used.
- `AccountingDatabase`. Name of the database alias configured on the system for the accounting database. When no database is configured, this is the empty string.

Example communication accounting script

```
; Store accounting information in database, but only if an accounting database has been
 configured on the system.
If (AccountingDatabase <> "")

  ; Retrieve information from distributed communication.
  RequestReference = Communication.RequestReference
  CorrespondenceReference = Communication.CorrespondenceReference
  Label = Communication.Label
  Channel = Communication.Channel
  RecipientType = Communication.RecipientType

  ; Create INSERT query.
  StringListClear("Sql")
  StringListClear("SqlParameters")
  StringListAdd("Sql", "INSERT INTO AccountedCommunication VALUES
 (:RequestReference, :CorrespondenceReference, :Label, :Channel, :RecipientType)")
  StringListSetValue("SqlParameters", "RequestReference", "is")
  StringListSetValue("SqlParameters", "CorrespondenceReference", "is")
  StringListSetValue("SqlParameters", "Label", "is")
  StringListSetValue("SqlParameters", "Channel", "is")
  StringListSetValue("SqlParameters", "RecipientType", "is")

  ; Execute INSERT query.
  InitDatabase(AccountingDatabase)
  InitDataSet("DataSet", AccountingDatabase)
```

```
  SetExecSQL("DataSet", "Sql", "SqlParameters")
  CloseDatabase(AccountingDatabase)

End-If
```

Note This sample script is included in a B&OM installation. To use this script, you still must configure it on the system.

Add and modify document pages in streaming

In CCM B&OM, you can call scripts during the streaming to include additional, separator pages, make modifications on pages, and insert bar codes. This functionality is implemented by exit points scripts that you modify and configure on the system.

The exit points are called during the transformation of stacks to be processed by CCM_Conversion.

Include additional pages for stack, envelope, or document

The following exit points are available to add separator pages **before** a stack, envelope, or document:

- `OnStackStart`
- `OnEnvelopeStart`
- `OnDocumentStart`

The following exit points are available to add separator pages **after** a stack, envelope, or document:

- `OnStackEnd`
- `OnEnvelopeEnd`
- `OnDocumentEnd`

To configure the context for the exit points, follow these steps:

1. In the **Navigator** lower pane, find and double-click **CCM_Streaming**.
   The process appears in the central pane.
2. Click **Streaming** in the lower central pane.
3. In the **Action** section on the right, set the relevant parameters to a script that has the **StreamingUnit** context.
   When the parameters are set, the configured script is called automatically whenever a stack is processed.

To create and modify the scripts, use the functions and variables listed in Properties of the StreamingUnit context.

Example script

```
Try
  Label = GetOrganisationalMetadata ("Label")
OnError()
  Label = "Generic Corporate Brand"
End-Try
```

```
If (HasSender)
  Label = Label + " :: " + GetSenderData ("Department")
End-If

If (HasRecipient)
  Agent = GetRecipientData ("AgentName")
End-If

Page = stream.LoadDocument ("C:\CCM\Templates\Banner.xps", OdinDuplexSetting.Simplex,
 "blank", "blank")
Page.EditPage (0)

Page.SetFont ("Courier New", 12)
Page.TextOut (75, 51, Label, false, false)
Page.TextOut (75, 60, Agent, false, false)

Page.Post()
stream.AppendDocument (Page, false, false)
```

This example loads the `Banner` page stored in the XPS format from the `C` drive, inserts the retrieved fields, and then appends the page as a separator page to the output stream.

Modify a document page

Use the `OnPage` exit point to manipulate a particular page of the document. To do so, you can use the same functions and variables as in the StreamingUnit context. In addition, a set of calls is available to make changes to the page.

To configure the context for the `OnPage` exit point, follow these steps:

1. In the **Navigator** lower pane, find and double-click **CCM_Streaming**.
   The process appears in the central pane.
2. Click **Streaming** in the lower central pane.
3. In the **Action** section on the right, set the **OnPage** parameter to a script that has the **StreamingPage** context.
   When the parameter is set, the configured script is called automatically whenever a stack is processed.

To create and modify the script, use the functions and variables listed in Properties of the StreamingPage context.

In addition to the general functions and variables, to modify the content of the current page, you can also use the following block of calls and variables:

- `EditPage()`
- `Post()`
- `SetFont(string fontName, int fontSize)`
- `TextOut(double x, double y, string text, bool bold, bool italic)`
- `TextOut(double x, double y, string text, bool bold, bool italic, OdinPageLayer layer)`
- `Line(double x1, double y1, double x2, double y2)`
- `Line(double x1, double y1, double x2, double y2, OdinPageLayer layer)`

- `Rectangle(double x, double y, double width, double height, bool fill, string fillColor)`
- `Rectangle(double x, double y, double width, double height, bool fill, string fillColor, OdinPageLayer layer)`
- `TextOut(double x, double y, string text, bool bold, bool italic, string color, int angle)`
- `TextOut(double x, double y, string text, bool bold, bool italic, string color, int angle, OdinPageLayer layer)`
- `Line(double x1, double y1, double x2, double y2, double lineWidth, OdinLineStyle lineStyle, string color)`
- `Line(double x1, double y1, double x2, double y2, double lineWidth, OdinLineStyle lineStyle, string color, OdinPageLayer layer)`
- `Rectangle(double x, double y, double width, double height, double lineWidth, OdinLineStyle lineStyle, string lineColor, bool fill, string fillColor)`
- `Rectangle(double x, double y, double width, double height, double lineWidth, OdinLineStyle lineStyle, string lineColor, bool fill, string fillColor, OdinPageLayer layer)`
- `Image(double x, double y, string imageFileName)`
- `Image(double x, double y, string imageFileName, OdinPageLayer layer)`
- `AddBookmark(string id, string parentId, string title)`
- `AddBookmark(string id, string parentId, string title, double x, double y)`
- `AddComment(string value)`
- `AddPJLCommand(string value)`
- `SetAfpCopyGroup(string copyGroup)`

For more information on these calls, see the chapter Scripting functions.

Example script

```
Try
Label = GetOrganisationalMetadata ("Label")
OnError()
Label = "Generic Corporate Brand"
End-Try
If (HasSender)
Label = Label + " :: " + GetSenderData ("Department")
End-If
If (HasRecipient)
Agent = GetRecipientData ("AgentName")
End-If

EditPage ()
SetFont ("Courier New", 12)
TextOut (75, 51, Label, false, false)
TextOut (75, 60, Agent, false, false)
Post()
```

This example puts the marker on the current page (through the `OnPage` event).

Also, you can insert a bar code on the page with the `AddBarcode` function as shown here.

```
EditPage ()
Bar = AddBarcode()
Bar.Type = "EAN 13"
```

```
Bar.CheckDigits = 1
Bar.Data = "978159059389"
Bar.Readable = true
Bar.Angle = 90
Bar.X = 12
Bar.Y = 3
Bar.Height = 10
Bar.Width = 30
Bar.Post()
Post()
```

For a description of the function parameters and explanation of the example, see the next section.

Insert a bar code on a document page

Use the `AddBarcode` function to insert a bar code on a document page. This function is supported in all streaming exit points.

The function has the following parameters:

1. `BarCode.Type`. *Required*. Type of bar code.

   Note This functionality uses the DocBridge Mill toolkit to produce the actual bar codes. The following bar code types are supported: POSTNET, QR code, EAN 13, EAN 8, CODE 128, MSI, Code 39, and Planet. For the complete list of supported types, see the DocBridge Mill documentation.

2. `BarCode.CheckDigits`. *Optional*. Number of check digits. A check digit is the number located in the far right side of a bar code, and is used to verify the information entered on the bar code. Possible values depend on the bar code type; default is 0.
3. `BarCode.Data`. *Required*. Value of the bar code. Must also include check digits.
4. `BarCode.QuietZone`. *Optional*. Set to True to add a quiet zone to the bar code. Default is false (no quiet zone). A quiet zone is a margin on the left and right side of a bar code. Not supported by some bar code types. If not supported, this setting is ignored.
5. `BarCode.Readable`. *Optional*. Set to true to show the human readable interpretation (HRI) of the bar code. For example, HRI can be digits below a standard EAN bar code. Default is false (no HRI). Not supported by some bar code types.
6. `BarCode.Angle`. *Optional*. The angle in degrees to rotate the bar code. Possible values: 90, 180, and 270 degrees. Default is 0.
7. `BarCode.X`. *Optional*. The X-axis position of the bar code in mm. Default is 0.
8. `BarCode.Y`. *Optional*. The Y-axis position of the bar code in mm. Default is 0.
9. `BarCode.Ratio`. *Optional*. Ratio of height and width. Not supported by some bar code types. Default is 1.
10. `BarCode.Height`. *Optional*. Bar code height in mm. Default is 0.
11. `BarCode.Width`. *Optional*. Bar code width in mm. Default is 0.

Example script

```
BarCode = document.AddBarcode()
```

```
BarCode.Type = "EAN 13"
BarCode.CheckDigits = 1
BarCode.Data = "978159059389"
BarCode.Readable = true
BarCode.Angle = 90
BarCode.X = 12
BarCode.Y = 3
BarCode.Height = 10
BarCode.Width = 30
BarCode.Post()
```

This example adds a bar code with value `978159059389` of type `EAN 13` with height `10` mm and width `30` mm at `12` mm from the left side and `3` mm of the top of the document with `1` check digit and with no quiet zone. The bar code is rotated at an angle of `90` degrees and shows the human readable interpretation.

Configure the distribution functionality

In CCM B&OM, you can call scripts to adjust the distribution of CCM B&OM output print files. This functionality is implemented by exit points that you can modify on the Distribution component. The installation includes a sample exit point called DistributeStack used by default.

The exit point is called during the distribution of stacks to be processed by CCM_Distribution.

To configure the context for the exit points, follow these steps:

1. In the **Navigator** lower pane, find and double-click **CCM_Distribution**.
   The process appears in the central pane.
2. Click **Distribution** in the lower central pane.
3. In the **Action** section on the right, set the **OnStackDistribute** parameter to a script that has the **StackDistribution** context.
   When this parameter is set, the configured script is called automatically whenever a stack with type "Distribute (7)" is processed.

To create and modify the script, use the functions and variables listed in Properties of the StackDistribution context .

Example script

```
;The StackDistribution context supplies necessary information
;It is also possible to retrieve the content of the organisational metadata
Key1Value = Stack.GetOrganisationalMetadata("key1");
Key2Value = Stack.GetOrganisationalMetadata("key2");

;The output folder can be changed
GetObject("Dos","MLDos")
ChannelPrinter = Stack.Channel + Stack.Printer
Stack.NewOutputFolder = Dos.Combine(Stack.OutputFolder,ChannelPrinter)
if (Dos.DirectoryExists(NewOutputFolder) = False)
   Dos.CreateDirectory(NewOutputFolder)
end-if
Stack.OutputFolder = NewOutputFolder
Protocol("New Output Folder = " + Stack.OutputFolder,5)

;The names of the print file and the description file can be changed
```

```
Stack.OutputPrintFile = Stack.Channel+ "_" + Stack.Printer + "_" +
 Stack.OutputPrintFile
Stack.OutputDescriptionFile = Stack.Channel + "_" + Stack.Printer + "_" +
 Stack.OutputDescriptionFile
Protocol("New Output Print File Name = " + Stack.OutputPrintFile,5)
Protocol("New Output Description File Name= " + Stack.OutputDescriptionFile,5)

;Check if the files already exist before saving
DestinationPrintFile = Dos.Combine(Stack.OutputFolder,Stack.OutputPrintFile)
DestinationDescriptionFile = Dos.Combine(Stack.OutputFolder,Stack.OutputPrintFile)
if (Dos.FileExists(DestinationPrintFile))
   ErrorMessage = "Cannot distribute " + Stack.OutputPrintFile + " to OutputFolder " +
 Stack.OutputFolder + " because the files already exist."
   Raise("UnrecoverableDistributionError",ErrorMessage)
end-if

if (Dos.FileExists(DestinationDescriptionFile))
   ErrorMessage = "Cannot distribute " + Stack.OutputDescriptionFile + " to
 OutputFolder " + Stack.OutputFolder + "because the files already exist."
   Raise("UnrecoverableDistributionError",ErrorMessage)
end-if

try
  ;Copies the print file and the description file to the output folder
  Stack.Save()
OnError
  ;Clean up already distributed files to facilitate reruns
   Dos.DeleteFile(DestinationPrintFile)
   Dos.DeleteFile(DestinationDescriptionFile)
   Raise("UnrecoverableDistributionError",Error.Message)
End-try
```

Chapter 3

Batch & Output Management scripting language

The Batch & Output Management (B&OM) scripting language offers a wide range of possibilities to enhance the functional scope of CCM Studio. The scripting language includes function calls to various contexts and offers control structures such as loops and branching.

Create scripts and conditions

Scripts give you the ability to solve specific tasks and implement requirements that the use of standard objects alone cannot fully cover.

Conditions are scripts that check the validity of a statement in connection with conditions in a process definition.

For more information on the syntax of the scripting language, see Scripting functions.

Create a script

1. In the lower **Navigator** pane, right-click and then click **New** > **Script**.
2. Enter a name for the new script and click **OK**.
3. In the **Object Inspector** pane, click the **Context** property and then select the context for your script.
   The context determines which objects you can assign to the script and which properties and functions are available. For more information, see Scripting contexts.
4. In the editor window, type the script.
5. On the **Script Note Editor Tools** tab, in the **Control** section, click **Compile** .
   Information about the compilation result appears in the **Output** pane.

Create a condition

1. In the lower **Navigator** pane, right-click and then click **New** > **Condition**.
2. Enter a name for the new condition and click **OK**.
3. In the **Object Inspector** pane, click the **Context** property and then select the context for your condition.
   The context determines which objects can be assigned to a condition and which properties and functions are available. For more information, see Scripting contexts.

4. Assign `true` or `false` to the variable `Result` as the result of the conditional statement.
5. On the **Script Note Editor Tools** tab, in the **Control** section, click **Compile**.
   Information about the compilation result appears in the **Output** pane.

Operators

You can use operators to link a number of logical expressions together in a condition. Enclose combined expressions in brackets to maintain their logical intent. You can nest logical expressions.

The following operators are available.

Operator	Meaning
=	The attribute must be identical to the value.
<>	The attribute must not be identical to the value.
<	The attribute must be less than the value.
>	The attribute must be greater than the value.
<=	The attribute must be less than or equal to the value.
>=	The attribute must be greater than or equal to the value.
<>""	The attribute cannot be empty.
AND()	Logical AND.
OR()	Logical OR.

View context information

The context information shows the available variable pools and process components for the context you specified.

To view the context information, below the editor window, select the **Context Information** tab.

Create a script parameter

You can specify script parameters and standard values when another script calls the script with the parameter passing function.

1. Below the editor window, select the **Parameter** tab.
2. Click **New Parameter** and then type a name for the parameter.
   - Optional. In the **Standard Value** field, type a standard value. For more information, see CallSystem.

Assign a script to an event

You can assign scripts to events generated by the Streaming and Distribution components and to accounting events that can be generated by several components.

1. For component events, open the process with the component and select the component.

For accounting events, on the **Administration** tab, click **System Administration** and click the system object.

2. In the **Object Inspector** pane, in the **Action** section, click the event property and click the ellipsis button.
3. In the **Select an object** dialog, select the preferred script and click **Select**.

   Note Only scripts with the appropriate context are available for selection.

4. In the **Object Inspector** pane, double-click an event to open the assigned script. If you double-click an event with an empty event property, the **Create New Script** dialog appears. The script automatically receives the context that fits.

You can also assign scripts to process designer blocks and components.

Script and condition editor tools

This table describes the available editor tools for scripts and conditions.

Icon	Name	Description
	Export Script	Saves a script to a file.
	Debug Service	Adds the opened script to, or removes it from, the debug service.
	Compile	Compiles scripts and conditions.
	Execute	Runs a script or condition. Code is executed in the background.
	Debug	Scripts that only contain the B&OM standard functions can be executed directly in debug mode, if the context `None` is assigned. For more information, see Debug a script or condition.
	Selection Browser	Opens a selection browser in the editor window.
	Edit	Provides standard functions for editing the script code, including Cut, Copy, Paste, Undo and Redo.

Useful key combinations

You can use the following shortcut key combinations to facilitate the code writing.

Ctrl+Spacebar	Opens a list of all B&OM standard functions. This list also shows information about the function of a statement and its parameters. To insert a command into the script editor, press Enter or double-click the command.
Ctrl+Alt+Spacebar	Opens a list of properties, functions and pools available in the current context.
Ctrl+K+X	Opens control structures. Alternatively, you can press the Tab key for one of the listed keywords to display the corresponding command structure. For more information, see Control structures.
Alt+Right Arrow or Ctrl+Spacebar	Displays word completion.
Ctrl+K+C	Marks the current code line as a comment.
Ctrl+K+U	Removes the comment syntax from the current code line.
Ctrl+K+F	Indents the code line selected like the surrounding code lines.
Shift+Alt+T	Moves a line with the cursor below the next line.
Ctrl+U	Converts selected text to lowercase.
Ctrl+Shift+U	Converts selected text to uppercase.
Ctrl+Shift+Spacebar	Shows information about the current parameter.
Ctrl+Delete	Deletes the word to the right of the cursor.
Ctrl+Backspace	Deletes the word to the left of the cursor.
Ctrl+F	Displays the search dialog.
Ctrl+H	Displays the search and replace dialog.
F3	Continues the search down the text.
Shift+F3	Continues the search up the text.

Debug a script or condition

Scripts and conditions that do not need a context for execution

In debug mode, you can run scripts and conditions that do not use any specific context. To debug a script or a condition, complete the following steps:

1. Open the script or condition.
2. In the **Object Inspector** pane, set the **Context** property to the value **None**.
3. On the **Script Note Editor Tools** tab, in the **Control** section, click **Execute in Debug Mode** .
   The script appears in a new debug mode window.

4. Run the script using the buttons on the **Debugger Tools** tab.

Scripts and conditions that need a context for execution

In debug mode, you cannot run scripts and conditions that use a specific context. Instead, run these scripts and conditions in the corresponding context. To debug a script or a condition, follow these steps:

1. In the **Navigator** pane, right-click the script or condition and click **Add to Debug Service**.
2. Open the document or the process that uses the script and start the document creation or process. Once the script is executed, the debugger window opens.
3. Run the script using the buttons on the **Debugger Tools** tab.

To disable execution in the debug mode for a script or condition, in the **Navigator** pane, right-click the script or condition and click **Remove from Debug Service**.

Debugger tools

Icon	Name	Description
	Run (CTRL+F9)	Executes the script and closes the debugger. Does not take into account any breakpoints.
	Run (F9)	Executes the script but does not close the debugger. If breakpoints are set, the script is stopped at the corresponding code line and can be continued from this position.
	Set / Remove Breakpoint (F5)	Sets or removes a breakpoint at the current position.
	Step Into (F7)	Executes the current code line. If this statement is `CallSystem`, the debugger branches into the script being called.
	Procedure Step (F8)	Executes the current code line.
	Stop (CTRL+F2)	Ends the script and closes the debugger.
	Quick Monitoring	Shows the current variable values.

Scripting functions

The following sections describe all available functions. Your installation includes a few examples of scripts written in the CCM B&OM scripting language:

```
<deploy root>\CCM\Documentation\5.2\Resources\Examples\Output Management
Scripting
```

Data types

The B&OM scripting language works with the data types Integer, BigInteger, Float, and String.

Data Type	Min	Max
I: Integer (int32)	-2147483648	2147483647
L: BigInteger (int64/long)	-9223372036854775808	9223372036854775807
F: Float (double)	-1.7976931348623157E+308	1.7976931348623157E+308
S: String		

This table lists the result data types according to data operations.

Operands		Result data type			
Operand 1	**Operand 2**	**Operator +**	**Operator -**	**Operator ***	**Operator /**
S	S	S	Error	Error	Error
S	I	S	Error	Error	Error
S	L	S	Error	Error	Error
S	F	S	Error	Error	Error
I	S	S	Error	Error	Error
I	I	I/L	I/L	I/L	F
I	L	L	L	L	F
I	F	F	F	F	F
L	S	S	Error	Error	Error
L	I	L	L	L	F
L	L	L	L	L	F
L	F	F	F	F	F
F	S	S	Error	Error	Error
F	I	F	F	F	F
F	L	F	F	F	F
F	F	F	F	F	F

Control structures

With the following control structures, you can control the flow of scripts and conditions:

- `CheckEvery`
  Checks validity of a variable. If `EVERY` is passed as the first parameter, more than one agreement is possible.
  - The variable to check is passed as the second parameter.
  - `value` checks the value passed for validity.
  - `when` checks further conditions.
  - Statements after `when-any` are executed when one or more of the conditions are met.
  - Statements after `when-all` are executed when all of the conditions are met.
  - Statements after `when-none` are executed when none of the conditions are met.

  Example

```
a = 11

erg = 0

c = false

;

Check("every", a)

   value(1)

      erg = erg + 1 ; if a is equal to 1

   when(c = true)

      erg= erg + 1  ; if c is TRUE

   When-Any

      message = "any"  ; if a is equal to 1 OR c is TRUE

   When-All

      message = "all"  ; if a is equal to 1 AND c is TRUE

   When-None

      message = "none" ; if a is not equal to 1 AND c is not TRUE

end-check
```

- `CheckFirst`
  Checks validity of a variable. If `FIRST` is passed as the first parameter passed, only the statements after the first agreement are executed.
  - The variable to check is passed as the second parameter.
  - `value` checks the value passed for validity.
  - `when` checks further conditions.
  - Statements after `when-any` are executed when one or more of the conditions are met.

- Statements after `when-all` are executed when all of the conditions are met.
- The statements after `when-none` are executed when none of the conditions are met.

Example

```
a = 11

erg = 0

c = false

Check("FIRST", a)

   value(1)

      erg = 1    ; if a is equal to 1

   when(c = true)

      erg=2      ; if a is not equal to 1 AND c is TRUE

   When-Any

      Erg=3      ; if a is equal to 1 OR c is TRUE

   When-All

      Erg=4      ; if a is equal to 1 AND c is TRUE

   When-None

      Erg=5      ; if a is not equal to 1 AND c is not TRUE

end-check
```

- `If`

  Checks whether the condition in brackets is valid. If so, the following commands up to `End-if` are executed.

  If the condition in brackets is not valid, the script is continued after the `End-if` command. Enclose each individual condition in brackets.

  Enclose multiple conditions linked by `and` or `or` by additional brackets.

  Example

```
if (a < 10)

   b = 0

end-if

if ( (a = 10) and (b = 30) )

   c = 0

end-if
```

- `IfElse`

  Checks whether the condition in brackets is valid. If so, the following commands up to `Else` are executed. The script then continues with the statements following the respective `End-If` command.

If the condition in brackets is not valid, the commands between `Else` and the respective `End-If` are executed.

Example

```
if (a < 10)
   b = 0
else
   b = 1
end-if
```

- `While`
  Executes the statements in a loop as long as the condition in parentheses is `TRUE`. The condition is evaluated before the statements are executed.

```
while (a < 100)
   a = a + 1
end-while
```

- `Region`
  Defines a section of a script that you can display or hide using the plus or minus symbols. This makes large scripts more manageable.

  Example

```
#region InfoText
...
#endregion
```

Exception handling

You can execute the `TryFinally` control to catch exceptions that occur during script execution:

- `TryFinally`
  If an error occurs while executing a statement, a script is typically terminated with an exception.
  If the statement is inside a try-block, the statements in the finally-block are executed before the script is ended.

  Example

```
try
   InitDatabase("ODIN")
finally
   Protocol("Inside Finally-block", 9)
end-try
```

- `TryOnError` and `TryOnErrorFinally`
  If an error occurs while executing a statement in a try-block, use the command `Raise` to skip to the error label defined in `OnError`.

Only the first matching `OnError` block in a try-block is executed.

Choose the right order of `OnError` blocks so that specific `OnError` blocks are executed before general blocks.

The code in the `finally` block is executed after the `OnError` block is processed.

Example

```
;the special object "Error" contains information about the last error

;If no error occurred - all attributes contain an empty string

;

;Error.Category: the category identifies the sub-systems that threw the error (for
 example: "OdinBusinessFacade")

;Error.Token: the token is a one-word identifier for the error. Can be specified if you
 use the Raise function

;Error.Message: contains the error message

Protocol("before try: category '{0}'; error token: '{1}'; error message: '{2}'; ", 0,
 Error.Category, Error.Token, Error.Message)

;

;modify this value to play with the different OnError code-blocks

a = 10



try

   if(a < 10)

      ;This exception will be caught in the OnError("BUSINESS_ERR") block

      Raise("BUSINESS_ERR","a cannot be less than 10 sein")

   end-if

;

   ;This exception will be caught in the OnError() block

   ;because there is no OnError("Err") block

   Raise("Err", "Test-Raise")

;

;

OnError("BUSINESS_ERR")

   ;this code is executed if the exception token is "BUSINESS_ERR"

   ;the exception token is the first parameter of the Raise function

   Protocol("BUSINESS_ERR: {0}", 0, Error.Message)
```

```
;

   ;you can also rethrow the error

   ReRaise()

;

OnErrorCategory("OdinBusinessFacade")

   ;this code is executed if the category of the exception is "OdinBusinessFacade"

   Protocol("odin error caught", 0)

;

OnError()

   ;this code is executed if no matching OnError() was found up to this line

   ;information about the exception can be retrieved from the Error object

   Protocol("error category: '{0}'; error token: '{1}'; error message: '{2}'; ", 0,
 Error.Category, Error.Token, Error.Message)

;

Finally

;

   ;code in the finally block is ALWAYS executed

   ;this is the right place to free resources (database connections, file handles)

   Protocol("Inside Finally-block", 9)

End-try

;

Protocol("after finally: error category: {0}; error token: '{1}'; error message: '{2}';
 ", 0, Error.Category, Error.Token, Error.Message)
```

Standard functions

The following sections describe the functions available in the CCM Batch & Output Management scripting language.

Beep

Sends out the standard system signal tone.

SYNTAX	`Beep(frequency)`
ARGUMENTS	`Frequency` Signal frequency in hertz
EXAMPLE	`Beep(37)`

BigInteger

Converts a value to a BigInteger value. The variable content must be a number.

SYNTAX	`BigInteger(valueToConvert)`
ARGUMENTS	`valueToConvert` The variable to convert. If you define an empty string as the parameter value, it results in an exception at runtime.
RETURN	BigInteger value of the returned variable.

EXAMPLE	`Var = 100`
	
	`Var = BigInteger(Var)`

CallSystem

Calls a script from the same system.

Note Only the existing variables can be passed in `inputVariables` and `inputVariables`.

SYNTAX	`CallSystem(scriptName: String)`
	
	`CallSystem(scriptName: String ,inputVariables: String)`
	
	`CallSystem(scriptName: String ,inputVariables: String ,outputVariables: String)`

ARGUMENTS	`scriptName` — Name of the script to execute. `inputVariables` — Comma-separated list with the names of the variables to pass. `outputVariables` — Comma-separated list with the names of the variables to pass. These variables are also those returned.
RETURN	Comma-separated list with the names of variables to pass.

Examples

Call a script with no variable passing:

```
CallSystem("callTestScript")
```

Call a script passing input variables:

```
Amount1 = 1

Amount2 = 2

CallSystem("callTestScript", "Amount1,Amount2")
```

In this example, `callTestScript` is called from the system and the variables `Amount1` and `Amount2` are available in the called script.

Call a script passing input variables and defining target variables:

```
Amount1 = 1

Amount2 = 2

CallSystem("callTestScript", "Amount1=Number1,Amount2=Number2")
```

In this example, the script `callTestScript` is called from the system and the variables `Amount1` and `Amount2` are available as the variables `Number1` and `Number2` in the called script.

Call a script passing input variables and output variables

```
Amount1 = 1

Amount2 = 2

CallSystem("callTestScript", "Amount1,Amount2", "Sum")

Protocol("Sum of Amount1 and Amount2: {0} ", 0, Sum)
```

In this example, the script `callTestScript` is called from the system and the variables `Amount1` and `Amount2` are available in the called script.

After the script is called, the variable `Sum` from the called script is passed back to the calling script and is available for further processing.

CloseDatabase

Closes the connection to a database.

SYNTAX	`CloseDatabase(aliasName: String)`
ARGUMENTS	`aliasName` Name of the database alias
EXAMPLE	`CloseDatabase("DBAliasName")`

Contains

Checks whether the source string contains a search string.

SYNTAX	`Contains(sourceString, searchString)`
ARGUMENTS	`sourceString` — The string to search through `searchString` — The string to search for
RETURN	`TRUE` = The source string contains the search string. `FALSE` = The source string does not contain the search string.

EXAMPLE	`source_string = "CCM Software"`
	`search_string = "Soft"`
	`ok=Contains(source_string ,search_string)`

Contains_Exactly

Checks whether the source string contains a search string. The check is case-sensitive.

SYNTAX	`Contains_Exactly(sourceString, searchString)`
ARGUMENTS	`sourceString` — The string to search through `searchString` — The string to search for
RETURN	`TRUE` = The source string contains the search string. `FALSE` = The source string does not contain the search string.

EXAMPLE	`source_string = "CCM software"`
	`search_string = "ware"`
	`ok=Contains_Exactly(source_string ,search_string)`

CreateGuid

This function creates a Globally Unique Identifier (GUID) and returns its string representation, according to the provided format specifier. A GUID is a global 128-bit number and that is an implementation of the Universally Unique Identifier Standards (UUID).

A GUID is typically represented in the format **XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX**, where **X** is a placeholder for a hexadecimal character (0-9 or A-F). For example, **936DA01F-9ABD-4D9D-80C7-02AF85C822A8**.

SYNTAX	`CreateGuid(format)`

ARGUMENTS	`format` A single-format specifier that indicates how to format the value of this GUID. The format parameter can contain the following specifiers: • N: 32 digits (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx) • D: 32 digits separated by hyphens (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx) • B: 32 digits separated by hyphens, enclosed in brackets {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx} • P: 32 digits separated by hyphens, enclosed in parentheses (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx) • If format is an empty string (""), "D" is used. All digits in the return value are hexadecimal. Each character **x** represents a hexadecimal digit. Each hyphen, bracket, and parenthesis appear as shown.
RETURN	The value of this GUID represented as a series of lowercase hexadecimal digits in the specified format.
EXAMPLE	`retString = CreateGuid("N")`

CreatePool

Creates a new variable pool.

SYNTAX	`CreatePool(poolName)`
ARGUMENTS	`poolName` Name of the pool
EXAMPLE	`CreatePool("MyPool")`

Decimal

Converts a value to a decimal (floating point value).

Notes

- If the parameter for `Decimal` is a string or a constant, the period is always the decimal separator.
- The thousands separator is always a comma.
- If a variable is passed to `Decimal`, the content is interpreted according to the regional Windows settings for DecimalSeparator and ThousandSeparator.

SYNTAX	`Decimal(valueToConvert)`
ARGUMENTS	`valueToConvert` The value to convert. If an empty string is passed as the parameter value, it results in an exception at runtime.
RETURN	Floating point value

EXAMPLE	`b = Decimal(a)`

DeletePool

Deletes the specified pool.

The following pools cannot be deleted:

- BAUSTEIN (not applicable to English version)
- BLOCKPOOL
- DOCUMENTPOOL
- DOKUMENT (not applicable to English version)
- MODUSUSER
- SYSTEM
- SYSTEMPOOL
- ZUSATZVARIABLEN (not applicable to English version)

SYNTAX	`DeletePool(poolName)`
ARGUMENTS	`poolName` Name of the pool
EXAMPLE	`DeletePool("MyPool")`

EncryptString

Returns a string containing the encrypted text.

SYNTAX	`EncryptString(ClearText)`
ARGUMENTS	`ClearText` The string to encrypt
RETURN	A string containing the encrypted text
EXAMPLE	`retString = EncryptString("abc")`

EndsWith

Determines if the end of the input string matches the specified string.

SYNTAX	`EndsWith(inString, value, ignoreCase)`

ARGUMENTS	`inString` — The input string to check `value` — The string value to compare with the end of the `inString` value `ignoreCase` — `TRUE` = Ignores case in the search; `FALSE` = Considers case in the search
RETURN	`TRUE` = A string match is found `FALSE` = No string match is found
EXAMPLE	`retPosition = EndsWith("ABCxyz","xyz",false)`

Eof

Determines whether the cursor is at the end of the dataset.

SYNTAX	`Eof()`
ARGUMENTS	`datasetName` Name of the dataset
RETURN	`TRUE` = End reached `FALSE` = End not reached
EXAMPLE	`End = Eof("DSName")`

Execute

Executes a program.

SYNTAX	`Execute(commandLine,commandShow)`
ARGUMENTS	`commandLine` — Path to the program `commandShow` — Win32 API parameter that defines how the window is opened
RETURN	`ProgramHandle`
EXAMPLE	`PrgHandle = Execute("C:\Tools\ProgramName.EXE",1)`

Note for 64-bit operating systems

CCM Studio is a 32-bit program. If you use the `ExecuteSynchron` command to start a program from the directory `%windir%\System32`, you need to define the directory `%windir%\Sysnative` instead.

If you use the `ExecuteSynchron` command to start a program from the directory `C:\windows\System32`, you need to define the directory `C:\windows\Sysnative` instead.

For more information on the File System Redirector, see the Microsoft Developer Network website: msdn.microsoft.com.

Float

Converts a value to a floating-point variable. The value used must be in a number format.

SYNTAX	`Float(valueToConvert)`
ARGUMENTS	`valueToConvert` The variable to convert **Note** • If the parameter is a string or a constant, the period is always the decimal separator. The thousands separator is always a comma. • If a variable is passed to `Float`, the content is interpreted according to the Windows regional settings for DecimalSeparator and ThousandSeparator. If an empty string is passed as the parameter value, it results in an exception at runtime.
RETURN	Floating-point variable

Example

```
FloatVar = Float("9.15")            ;FloatVar = 9.15

FloatVar = Float("9,000.15")        ;FloatVar = 9000.15

FloatVar = Float("11,0.15")         ;FloatVar = 110.15

;

Test = "9.15"

FloatVar = Float(Test)             ;result depends on the windows regional settings, in
 Germany FloatVar = 915

;

Test = "9,15"

FloatVar = Float(Test)            ;result depends on the windows regional settings, in
 Germany FloatVar = 9,15
```

Format

Combines strings and extracts a specific substring of a string. You can format the result to a specific length and fill it with appropriate characters.

SYNTAX	`Format(startPosition, numberCharacters, resultLength, formatRule, fillCharacter,formatParams)`

ARGUMENTS	`startPosition`	Start position beginning from 1.
	`numberCharacters`	Number of characters from the start position to cut off.
	`resultLength`	Maximum length of the result string. 0 means the result string has no length restriction.
	`formatRule`	L = left-aligned R = right-aligned Z = centered
	`fillCharacter`	Filling character if the result string does not have the required length.
	`formatParams`	Data strings to link together.
RETURN	Formatted text	

Example

```
Date = "1.7.2014"
Day = Format(3, 1, 2, "R", "0",Date) ;   Day   ==> "07"
Month = Format(1, 1, 2, "R", "0",Date) ; Month   ==> "01"
Year = Format(5, 4, 4, "R", "0",Date) ;  Year   ==> "2014"
;-----------------------------------------
Name = "John James Miller"
; subst without padding
Name1 = Format(6, 50, 0, "L", "_", Name); Name1 = "James Miller"
Name2 = Format( 1, 10, 0, "L", "_", Name); Name2 = "John James"
Name3 = Format(6, 5, 0, "L", "_", Name); Name3 = "James"
; subst with padding, text left aligned
Name4 = Format(6, 50, 20, "L", "_", Name); Name4 = "James Miller________"
; subst with padding, text right aligned
Name5 = Format(6, 50, 20, "R", "_", Name); Name5 = "________James Miller"
; subst with padding, text centered
Name6 = Format(6, 50, 20, "Z", "_", Name); Name6 = "____James Miller____"
```

FormatStr

Formats a string by concatenating the strings `variable1` to `variableN`, according to the defined format rule.

If the result contains the % character, define it as %% in the `formatRule` parameter.

SYNTAX	`FormatStr(formatRule,formatParams: Variable1, Variable2, ..., VariableN)`
ARGUMENTS	`formatRule` — Format rule `formatParams` — Strings to format
RETURN	Formatted text

EXAMPLE	`d = "01"`
	
	`m = "02"`
	
	`dm = FormatStr("Day: %s, Month: %s", d, m) ;Result: dm = "Day: 01, Month: 02"`
	
	`;`
	
	`y ="2017"`
	
	`Date = FormatStr("%s/%s/%s", m, d, y) ;Result Date = "02.01.2017"`
	
	`; Percent sign in the result string:`
	
	`Percentage = "5"`
	
	`Res = FormatStr("Percentage: %s %% p.a.", Percentage) ; Res = Percentage: 5 % p.a.`

GetLookupValue

SYNTAX	`GetLookupValue(objectName,key)`
ARGUMENTS	`objectName` — Name of the lookup table `key` — Lookup key
RETURN	Lookup value

EXAMPLE	`VarContent="001"`
	`res = GetLookupValue("LookupTab",`
	` VarContent)`
	
	Error handling in case lookup table or key does not exist
	
	`try`
	
	`    TestVariable =`
	` GetLookupValue("TestLookUp", "LookUpKey")`
	
	`OnError("LookupValueNotFound")`
	
	`    ;Either the key within the lookup`
	` table or the lookup table itself was not`
	` found, therefore the variable is set to a`
	` default value`
	
	`    TestVariable = "DefaultValue"`
	
	`End-try`

GetObject

Creates an instance of the specified object.

SYNTAX	`GetObject(varName, className)`
ARGUMENTS	`varName` Name of the instance `className` Name of the DLL or class

EXAMPLE	`GetObject("date","MLDate")`
	
	`GetObject("Reg",`
	` "MSLReg_Ini_Lib.MSLRegistry")`

GetVariable

Checks whether the specified variable exists and, if so, returns its value. If the variable does not exist, no further action is taken, and no error message is shown.

SYNTAX	`GetVariable(varName)`
ARGUMENTS	`columnName` Name of the variable
RETURN	Value of the variable
EXAMPLE	`Forename =` ` GetVariable("ModusUser.Forename")`

GetSystemOId

Returns the SystemOID of the system in which the script is executed.

SYNTAX	`GetSystemOId()`
RETURN	SystemOID
EXAMPLE	`MyOID = GetSystemOID()`

InitDatabase

Creates a connection to a database.

SYNTAX	`InitDatabase(aliasName)`
ARGUMENTS	`aliasName` Name of the database alias
RETURN	`TRUE` = Connection established `FALSE` = Connection not established
EXAMPLE	`OK = InitDatabase("odinTest")`

InitDataset

Creates a connection to a dataset.

SYNTAX	`InitDataset(datasetName, aliasName)` `InitDataset(datasetName, aliasName, tableName)`
ARGUMENTS	`datasetName` — Name of the dataset. `aliasName` — Name of the database alias. `tableName` — Name of the database table. If the parameter is set, the table is read using Select *.
EXAMPLE	`InitDataSet("modusorder", "ORDERDATA")` `InitDataSet("modusorder", "ORDERDATA", "modus_xml_job")`

Integer

Converts a string to an integer variable. Variable content must be a number.

SYNTAX	`Integer(valueToConvert)`
ARGUMENTS	`valueToConvert` The variable to convert. If an empty string is passed as the parameter value, it results in an exception at runtime.
RETURN	Integer value of the variable
EXAMPLE	`Var = "10" Var = Integer(Var)`

Max

Returns the highest value found in a string list or list of variables.

SYNTAX	`Max(stringList)` `Max(values)`
ARGUMENTS	`stringList` Name of the string list `values` Values to check
RETURN	Highest value

EXAMPLE	`MaxVarL=Max(StrListe)`
	`v1=10`
	`v2=30`
	`v3=50`
	`MaxVar=Max(v1,v2,v3)`

Min

Returns the lowest value found in a string list or list of variables.

SYNTAX	`Min(stringList)` `Min(values)`
ARGUMENTS	`stringList` Name of the string list `values` Values to check
RETURN	Lowest value

EXAMPLE	`MinVarL=Min(StrListe)`
	`v1=10`
	`v2=30`
	`v3=50`
	`MinVar=Min(v1,v2,v3)`

Parse

Enables you to process texts with special functions, such as date, time, calendar days, or hex values.

SYNTAX	`Parse(formattedContent)`
ARGUMENTS	`FormattedContent` The string to parse

RETURN	Formatted text

Use the `^D[mm,dd,yy,yyyy]` function to insert the date.

Arguments	Definition
`mm`	Current month
`dd`	Current day
`yy`	Current year (two figures)
`yyyy`	Current year (four figures)

Example

```
Test = Parse("^Ddd.mm.yyyy;")
;-->Test = "05.08.2016"
;
Test = Parse("Current date: ^Ddd.mm.yy;")
;-->Test = "Current date: 05.08.17 "
```

Use the `^U[hh,mm,ss]` function to insert current time.

Arguments	Definition
`HH`	Current hour, 24-hour format
`hh`	Current hour, 12-hour format
`mm`	Current minute
`ss`	Current second

Example

```
Test = Parse("^Uhh:mm:ss;")
;-->Test = "10:23:55"
;
Test = Parse("Date: ^Ddd.mm.yy; Time: ^UHH:mm:ss;")
;--Test = "Date: 21.04.17 Time: 19:19:48"
```

Use the `^K` function to insert a calendar week number.

Example

```
Test = Parse("^K;")
;-->Test = "32"
```

Use the `^H[Hexvalue]` function to insert hexadecimal values

It has a hexadecimal value as an argument.

Example

```
Test = Parse("^H3F;")

;-->Test = "?"
```

Protocol

Writes an entry into the ModusSuite_Log.txt trace file in the installation directory.

SYNTAX	`Protocol(traceMessage, traceLevel, formatParams)`
ARGUMENTS	`traceMessage` — Trace message. `traceLevel` — Trace level. Possible values: 0 = Error 1-4 = Warning 5-9 = Information 10 = Verbose `formatParams` — Parameters set for each placeholder. Placeholders are {0}, {1}, and so on.
EXAMPLE	`ErrorNumber = "001"` `Protocol("Error message {0} ", 0, ErrorNumber)`

Raise

Triggers an error and then skips to the error label defined in `OnError`.

SYNTAX	`Raise(errorTypeToken, errorMessage)`
ARGUMENTS	`errorTypeToken` — The value passed to the `LastError` variable when the `Raise` function is executed. `errorMessage` — Detailed error message.
EXAMPLE	`Raise("ERR1","Error 1 thrown")`

ReRaise

Use this function inside an `OnError` block to pass the exception back to the calling context.

SYNTAX	`ReRaise()`

Round

This function rounds a floating point number to an integer value. If the floating number belongs to the number range of 2147483647 to -2147483648, you need to type cast the return value to integer.

If the floating number belongs to the number range of 2147483648 to 9223372036854775807 and -2147483649 to -9223372036854775808, type cast the return value to a big integer value.

SYNTAX	`Round(double)`
ARGUMENTS	`double` Floating point number
RETURN	Rounded value
EXAMPLE	`result=Round(2.5) ; result = 3` `result=Round(-2.5) ; result = -3`

SetExecSQL

Executes an SQL command without a result.

If the SQL statement contains parameters (`%s` or `:Param`), it is only executed if these parameters can be resolved or are recognized. This is done by means of the `paramsDescls` list.

If the list is already passed with `SetExecSQL`, the function is executed. If not, the parameters must be resolved or made available by calling `PrepareSQL` and passing the list. Following that, you need to execute the `ExecSQL` function.

SYNTAX	`SetExecSQL(datasetName, sqlList)` `SetExecSQL(datasetName, sqlList, paramDescls)`

The following table lists and describes the attributes.

`datasetName`	Name of the dataset.
`sqlList`	Name of the string list containing the SQL statement.

`paramDescls`	Name of the string list containing the description of the SQL parameters. The parameters used in the SQL statement are described in this string list. You need to specify the parameter descriptions in the same order as in the SQL statement. Each parameter must exist in the form `ParameterName=ParameterType`. A corresponding local variable with the value of the parameter must exist for each `ParameterName` defined. For the parameter type `b` (binary), the variable must contain the path and file name of the dataset to insert. The parameter type consists of two characters. The first character determines the direction of the parameter. The following characters are supported.

Parameter	Definition
`b`	Bi-directional (input and output parameter)
`i`	Input parameter
`o`	Output parameter
`r`	Return value (stored procedure)

If an unsupported character is defined, the parameter `i` is used.
The second character determines the format of the parameter.
The following characters are supported.

Parameter	Definition
`b`	Binary (`BLOB` field)
`d`	Date
`f`	Float
`i`	Integer
`s`	String
`t`	Time
`@`	Date time

If an unsupported character is defined, the parameter `s` is used.

SetPoolVariable

Sets the value of a pool variable.

If the variable does not exist, it is created. If the variable already exists, it is overwritten.

SYNTAX	`SetPoolVariable(poolName, varName, value)`

ARGUMENTS	`poolName` Pool name `varName` Variable name `value` Value to set

EXAMPLE	`VarName = "Var_"`
	`i = 10`
	`while (i < 10)`
	`PoolVarName = VarName + i`
	`SetPoolVariable("System", PoolVarName, i)`
	`i = i + 1`
	`end-while`

Sleep

Pauses script execution for the specified period of time.

SYNTAX	`Sleep(sleepTime)`
ARGUMENTS	`sleepTime` Pause in milliseconds
EXAMPLE	`Sleep(1000)`

SplitStr

Obtains a specific partial expression from a string.

SYNTAX	`SetValue(columnName, newValue)`
ARGUMENTS	`data` String. `splitedStr` Number of the partial expression beginning from 1. `seperator` Delimiter for individual partial expressions. `errorText` Result string returned when a partial expression does not exist.
RETURN	Partial expression of the string

EXAMPLE	`dbvar = "T06M07J08"`
	`s1 = SplitStr(dbvar, 1, "TMJX", "")`
	`;s1 = "06"`
	`;`
	`s2 = SplitStr(dbvar, 2, "TMJX", "")`
	`;s2 = "07"`
	`;`
	`s3 = SplitStr(dbvar, 3, "TMJX", "")`
	`;s3 = "08"`
	`;`
	`s4 = SplitStr(dbvar, 4, "TMJX", "-")`
	`;s4 = "-"`

StartsWith

Determines if the beginning of an input string matches the specified string.

SYNTAX	`StartsWith(inString, value, ignoreCase)`
ARGUMENTS	`inString` — The input string to check `value` — The string value to compare with the beginning of the `inString` value `ignoreCase` — `TRUE` = Case is ignored during the search; `FALSE` = Case is included in the search
RETURN	`TRUE` = A string match is found `FALSE` = No string match found
EXAMPLE	`retPosition = StartsWith("ABCxyz","ABC",false)`

StrCompare

This function compares two specified string objects, ignoring or adhering to their case, and returns an integer that indicates their relative position in the sort order.

SYNTAX	`StrCompare(strA, strB, ignorecase)`

<table>
<tr><td>ARGUMENTS</td><td>
<table>
<tr><td>strA</td><td>The first string to compare</td></tr>
<tr><td>strB</td><td>The second string to compare</td></tr>
<tr><td>ignoreCase</td><td>TRUE = Case is ignored during the search
FALSE = Case is included in the search</td></tr>
</table>
</td></tr>
<tr><td>RETURN</td><td>A 32-bit signed integer that indicates the lexical relationship between the two strings compared.
< 0 = strA is less than strB
= 0 = strA is equal to strB
> 0 = strA is greater than strB</td></tr>
<tr><td>EXAMPLE</td><td>

```
;Compare

retvalue = StrCompare("String to compare.","String to
 compare.",true)

if(retvalue <> 0)

  Raise("Error", "StrCompare() The result should be
 0.")

end-if



;Case sensitive test

retvalue = StrCompare("Cat","cat",false)

if(retvalue = 0)

  Raise("Error", "StrCompare() The result should not
 be 0.")

end-if
```

</td></tr>
</table>

StrDelChar

Deletes a specific character from a string.

<table>
<tr><td>SYNTAX</td><td>

```
StrDelChar(inString,delChar)
```

</td></tr>
<tr><td>ARGUMENTS</td><td>
<table>
<tr><td>inString</td><td>String or variable</td></tr>
<tr><td>delChar</td><td>Character to delete</td></tr>
</table>
</td></tr>
<tr><td>RETURN</td><td>Result string</td></tr>
</table>

EXAMPLE	`var1="a b c d"`
	
	`var2=StrDelChar(var1, " ")`
	
	`;var2=abcd`

StrIndexOf

Reports the index of the first occurrence of a source string, beginning from 1.

SYNTAX	`StrIndexOf(searchString, sourceString, startIndex, ignoreCase)`
ARGUMENTS	`searchString` — The string to search for `sourceString` — The source string beginning from 1 `startIndex` — The search starting position `ignoreCase` — `TRUE` = Case is ignored during the search; `FALSE` = Case is included in the search
RETURN	The index position, beginning from 1, of the value parameter if that string exists, or -1 if it does not exist.

EXAMPLE	`retvalue = StrIndexOf("m","animal",2,true)`
	
	`if(retvalue <> 4)`
	
	`  Raise("Error", "StrCompare() The result should be 4 but it is " + retvalue)`
	
	`end-if`

StrLastIndexOf

This function reports the index, beginning from 1, of the last occurrence of a source string.

SYNTAX	`StrLastIndexOf(searchString, sourceString, ignoreCase)`
ARGUMENTS	`searchString` — The string to search for `sourceString` — The source string `ignoreCase` — `TRUE` = Case is ignored during the search; `FALSE` = Case is included in the search
RETURN	The index position, beginning from 1, of the value parameter if the string exists, or -1 if it does not exist.

EXAMPLE	`retvalue = StrLastIndexOf("N","Dot Net Perls",true)`
	
	`if(retvalue <> 5)`
	
	`  Raise("Error", "StrLastIndexOf() The result should be 5 but is " + retvalue)`
	
	`end-if`

String

Converts a variable to the string format.

You can also use this function to revert a string list back to a string. The elements of the list are separated by [CRLF].

SYNTAX	`String(valueToConvert)`
ARGUMENTS	`valueToConvert` The variable to convert (alphanumeric, numeric, or a string list)
RETURN	Converted string

EXAMPLE	`SYNTAX`
	` String(valueToConvert)`
	
	`ARGUMENTS`
	` valueToConvert`
	` The variable to convert (alphanumeric, numeric or a string list)`
	
	`RETURN`
	` Converted string`
	
	`EXAMPLE`
	` i=5`
	
	`s=String(i)`
	
	`;`
	
	`StringlistAdd("StrList","Kofax CCM","Kerkenbos 10-129","6546 BJ Nijmegen")`
	
	`Address=String("StrList")`

StringListAdd

Adds strings to a string list.

SYNTAX	`StringListAdd(stringList, stringParameters)`
ARGUMENTS	`stringList` — Name of the string list `stringParameters` — Strings to add

EXAMPLE	`Street = "City Street 9"`
	`Code = "74172"`
	`City = "London"`
	`;`
	`StringlistAdd(SListe, Street, Code, City)`
	`Address = String(SListe)`
	`;-->Address = "City Street 9, 74172 London"`

StringListClear

Removes all elements from a string list. If the specified string list does not exist, it is created.

SYNTAX	`StringListClear(stringList)`
ARGUMENTS	`stringList` Name of the string list
EXAMPLE	`StringListClear(SList)`

StringListCount

Counts the rows in a string list.

SYNTAX	`StringListCount(stringList)`
ARGUMENTS	`stringList` Name of the string list
RETURN	Number of rows
EXAMPLE	`i = StringListCount(StrList)`

StringListGetCommaText

Returns all rows of a string list in a comma-separated string.

SYNTAX	`StringListGetCommaText(stringList)`
ARGUMENTS	`stringList` Name of the string list
RETURN	Content of the string list
EXAMPLE	`Commatext =` ` StringListGetCommatext(StrList)`

StringListGetText

Returns the text of a string list. Each row ends with [CRLF].

SYNTAX	`StringListGetText(stringList)`
ARGUMENTS	`stringList` Name of the string list
RETURN	Content of the string list
EXAMPLE	`SLText = StringListGetText(StrList)`

StringListGetValue

Reads the value of a name within a string list in the form `Name=Value`.

SYNTAX	`StringListGetValue(stringList, aName)`
ARGUMENTS	`stringList` — Name of the string list `aName` — Identifier of the value to read
RETURN	The value assigned to the identifier
EXAMPLE	`SLValue = StringListGetValue(StrList,Name)`

StringListIndexOf

Returns the position of the first element in the line of the string list that contains the specified value.

SYNTAX	`StringListIndexOf(stringList,aLine)`
ARGUMENTS	`stringList` — Name of the string list `aLine` — Search value
RETURN	Position of the element

EXAMPLE	`StringlistAdd("Title", "Mr.","Mrs.","Company","Family")`
	`Pos = StringListIndexOf ("Title","Company")`
	`;-->Pos=2`

StringListItemValues

Reads the value after the specified index and writes a comma-separated text in the target string list.

SYNTAX	`StringListItemValues(stringList, lineIndex,destList)`

ARGUMENTS	`stringList` — Name of the string list `lineIndex` — Line index of the string list, beginning with 0 `destList` — Name of the target string list
RETURN	Value name

EXAMPLE	`StringlistAdd("test","Valuelist=1,2,3,4,5")`
	`name = StringListItemValues("test",0,"Values")`
	`;--> Name = "Valuelist"`
	`;`
	`value = StringListLine("Values",0)`
	`;--> value = "1 "`
	`;`
	`value = StringListLine("Values",1)`
	`;--> value = "2 "`

StringListLine

Reads the content of a line in a string list.

SYNTAX	`StringListLine(stringList,lineIndex)`
ARGUMENTS	`stringList` — Name of the string list `lineIndex` — Number of the line to read beginning from 0
RETURN	Line content
EXAMPLE	`line = StringListLine("SListe",0)`

StringListLoad

Loads a list of strings from a file.

SYNTAX	`StringListLoad(stringList,fileName)` `StringListLoad(stringList,fileName,sectionName)`

ARGUMENTS	`stringList` — Name of the string list. `fileName` — Name of the file that loads the list. `sectionName` — Name of the section that loads the data. This parameter is optional.
RETURN	Value name
EXAMPLE	`StringlistLoad("SListe", "c:\Temp\Test.lst")` `StringlistLoad("SListe", "c:\Temp\Test.lst","Section1")`

StringListParams

Configures the parameters of a string list.

SYNTAX	`StringListParams(stringList, sortList, duplicateState)`
ARGUMENTS	`stringList` — Name of the string list `sortList` — `TRUE` = The string list is sorted; `FALSE` = The string list is not sorted `duplicateState` — `0 = Ignore` ignores duplicates; `1 = Accept` allows duplicates; `2 = Error` Shows an error message when inserting duplicates. Setting parameters for duplicates has no effect on double strings already included in the list.

EXAMPLE	`StringlistAdd("SListe", "Mr.","Mrs.","Company","Family")`
	`StringListParams("SListe",true,2)`
	`StringlistAdd("SListe", "Company")`

StringListSave

Saves a list of strings in a file.

SYNTAX	`StringListSave(stringList, fileName)` `StringListSave(stringList, fileName, encoding)`

ARGUMENTS	`stringList` — Name of the string list. `fileName` — Name of the file to which the list is saved. `encoding` — You can use the following values as encoding parameters: • Empty string. In this case, the default encoding (ANSI code page of the operating system) is used. • Code page. For example, "1252" • Name. For example, "windows-1252" • User-friendly name. For example, "Western European" (Windows) For a list of possible encoding parameters, see Encoding parameters.
EXAMPLE	`StringlistSave("SListe", "c:\Temp\Test.lst")`

StringListSetCommaText

Writes a comma-separated value in a string list.

SYNTAX	`StringListSetCommaText(stringList, commaText)`
ARGUMENTS	`stringList` — Name of the string list `commaText` — Comma-separated values to save in the string list. For example, "A, B, C"
EXAMPLE	`StringListSetCommatext ("List1", "A,B,C")`

StringListSetValue

Sets the value of a name within a string list in the form `Name=Value`.

SYNTAX	`StringListSetValue(stringList, aName, aValue)`
ARGUMENTS	`stringList` — Name of the string list `aName` — Identifier of the value to set `aValue` — Value to assign to the identifier
EXAMPLE	`StringListSetValue ("List1", "Lastname", "Miller")`

StringListSort

Sorts the string list.

SYNTAX	`StringListSort(stringList)`
ARGUMENTS	`stringList` Name of the string list
EXAMPLE	`StringListSort("StrList")`

StrLeft

Adds a left-aligned formatting to a string.

SYNTAX	`StrLeft(inString)`
ARGUMENTS	`inString` String or variable
RETURN	Left-aligned string

EXAMPLE	`Var = " 1234"`
	`VarL = StrLeft(Var)`
	`;--> VarL = "1234 "`

StrLen

Calculates the length of a string.

SYNTAX	`StrLen(inString)`
ARGUMENTS	`inString` String or variable
RETURN	String length

EXAMPLE	`Var = "1234"`
	`Len = StrLen(Var)`
	`;-->Len = 4`

StrLower

Converts a string to lowercase.

SYNTAX	`StrLower(inString)`
ARGUMENTS	`inString` String or variable
RETURN	String in lowercase

EXAMPLE	`Var = "CCM SOFTWARE"`
	`VarLow = StrLower(Var)`
	`; VarLow = "ccm software"`

StrPos

Finds a string in a string.

SYNTAX	`StrPos(searchStr, sourceStr, ignoreCase)`
ARGUMENTS	`searchStr` — String to find `sourceStr` — String to search through `ignoreCase` — `TRUE` = Case-sensitive; `FALSE` = Not case-sensitive
RETURN	Position of the word in the string or 0 if the search string does not exist.

EXAMPLE	`SourceStr = "Monday Tuesday Wednesday"`
	`SuchStr = "Tuesday"`
	`res = StrPos(SuchStr, SourceStr, TRUE)`
	`;-->res=8`

StrReplace

Replaces characters in a string with other characters.

SYNTAX	`StrReplace(inString, oldValue, newValue)`
ARGUMENTS	`inString` — String or variable `oldValue` — Character to replace `newValue` — Character to insert
RETURN	Altered string

EXAMPLE	Replace characters:
	`var1 = "10/11/2017"`
	`Var2 = StrReplace(Var1, "/", ".")`
	`Var3 = StrReplace(Var1, "/", " - ")`
	`;--> Var2 = "10.11.2011"`
	`;--> Var3 = "10 - 11 - 2017"`
	Replace quotes with parse:
	`Var1 = """John"","Peter"""`
	`;--> Var1 = "John","Peter"`
	`qoute = Parse("^H22;")`
	`Var2 = StrReplace(Var1, quote, "")`
	`;--> Var2=John,Peter`

StrReverse

Reverses the string.

SYNTAX	`StrReverse(inString)`
ARGUMENTS	`inString` String or variable
RETURN	Reversed string

EXAMPLE	`Var = "CCM SOFTWARE"`
	`VarR = StrReverse(Var)`
	`;--> VarR = "ERAWTFOS MCC"`

StrRight

Makes a string right-aligned.

SYNTAX	`StrRight(inString)`
ARGUMENTS	`inString` String or variable
RETURN	Right-aligned string

EXAMPLE	`Var = "1234 "`
	`VarR = StrRight(Var)`
	`; VarR = " 1234"`

StrTok

Splits a string after a specified delimiter and returns the extracted string in front of the delimiter.

SYNTAX	`StrTok(varName,delimiter)`
ARGUMENTS	`varName` — String or variable `delimiter` — Separator character
RETURN	Extracted string

EXAMPLE	`Variable = "1;2;3;4;"`
	
	`V = StrTok("Variable", ";") ;--> V="1" Variable="2;3;4;"`
	
	`V = StrTok("Variable", ";") ;--> V="2" Variable="3;4;"`
	
	`V = StrTok("Variable", ";") ;--> V="3" Variable="4;"`
	
	`V = StrTok("Variable", ";") ;--> V="4" Variable=""`
	
	`V = StrTok("Variable", ";") ;--> V="" Variable=""`

StrToLen

Reduces a string to a specific length.

SYNTAX	`StrToLen(inString, maxLen, fillChar, doRight)`
ARGUMENTS	`inString` — String or variable. `maxLen` — Result length. If the value of this parameter is less than the length of the string, the string is truncated at this position. Otherwise, the string fills to the required length with the defined filling character. `fillChar` — Filling character. If no filling character is defined, the default character is a space. `doRight` — `TRUE` = Inserts filling characters at the beginning; `FALSE` = Insert filling characters at the end
RETURN	Altered string

EXAMPLE	`Var = "1234"`
	`Var10 = StrToLen(Var, 10, "0", TRUE)`
	`;--> Var10="0000001234"`
	`Var10 = StrToLen(Var, 10, "0", FALSE)`
	`;--> Var10="1234000000"`

StrUpper

Converts a string to uppercase.

SYNTAX	`StrUpper(inString)`
ARGUMENTS	`inString` String or variable
RETURN	String in uppercase

EXAMPLE	`Var = "ccm software"`
	`VarUp = StrUpper(Var)`
	`; VarUp = "CCM SOFTWARE"`

StrWord

Extracts the part of a string separated by the delimiter.

SYNTAX	`StrWord(inString, wordNumber, delimiter)`
ARGUMENTS	`inString` String or variable `wordNumber` Word number beginning with 1 `delimiter` Separator character
RETURN	Extracted substring
EXAMPLE	`Var = StrWord("W1.w2.W3",2,".")` `;--> Var = "w2"`

StrWordCount

Calculates the number of words according to the specified delimiter.

SYNTAX	`StrWordCount(inString, delimiter)`

ARGUMENTS	`inString` — String or variable `delimiter` — Separator character
RETURN	Number of words
EXAMPLE	`Var = StrWordCount("W1.w2.W3",".")` `;--> Var = 3`

SubString

This function retrieves a substring from the given input string. The substring begins at a specified position and has a specified length.

SYNTAX	`SubString(inString, startIndex, length)`
ARGUMENTS	`inString` — The string from which a part is extracted. `startIndex` — The starting position beginning from 1. `length` — The number of characters in the substring. If you pass the value zero to this parameter, all characters starting from the start Index to the end of the given input string return.
RETURN	A string of the given length that begins at the start index of the given input string.
EXAMPLE	`retString = SubString("abc",2,1)`

Trim

Removes all leading and trailing white space characters from the input string object.

SYNTAX	`Trim(inString)`
ARGUMENTS	`inString` The input string to trim
RETURN	The string that remains after all white space characters are removed from the start and end of the given input string.
EXAMPLE	`retString = Trim(" abc XYZ ")` `; Result: retString = "abc XYZ"`

TrimEnd

Removes all trailing white space characters from the input string.

SYNTAX	`TrimEnd(inString)`
ARGUMENTS	`inString` The input string to trim
RETURN	The string that remains after all white space characters are removed from the end of the given input string.
EXAMPLE	`retString = TrimEnd(" abc XYZ ")` `; Result: retString = " abc XYZ"`

TrimStart

Removes all leading white space characters from the input string.

SYNTAX	`TrimStart(inString)`
ARGUMENTS	`inString` The input string to trim
RETURN	The string that remains after all white space characters are removed from the start of the given input string.
EXAMPLE	`retString = TrimStart(" abc XYZ ")` `; Result: retString = "abc XYZ "`

Trunc

This function truncates the decimal position of a floating point number:

- If the floating number belongs to the number range of 2147483647 to -2147483648, you need to type cast the return value to integer.
- If the floating number belongs to the number range of 2147483648 to 9223372036854775807 and -2147483649 to -9223372036854775808, you need to type cast the return value to big integer.

SYNTAX	`Trunc(value)`
ARGUMENTS	`value` Floating point number
RETURN	Integer

EXAMPLE	`d = 1.9`
	`i = Integer(Trunc(d))`
	`;--> i = 1`
	`d = 9999999999.001`
	`i = BigInteger(Trunc(d))`
	`;--> i = 9999999999`

VerifyPool

Checks if the specified pool exists. If not, it throws an exception.

SYNTAX	`VerifyPool(poolName)`
ARGUMENTS	`dllFileName` Name of the pool to check
EXAMPLE	`VerifyPool("MyPool")`

Version

Determines the version of the B&OM scripting language interpreter.

SYNTAX	`Version()`
RETURN	Version number
EXAMPLE	`MLVersion=Version()`

Date functions

With the `MLDate` object, you can create a date. To load the object, use the `GetObject` function (see GetObject).

Example

```
GetObject("Date","MLDate")
```

Format specifiers

The format specifiers required as parameters by some functions must comply with the rules of the .NET Framework.

Examples

Format specifiers	Result
dd	Day, two numeric digits
ddd	Name of the weekday (abbreviated)
dddd	Name of the weekday (full)
MM	Month, two numeric digits
MMM	Name of the month (abbreviated)
MMMM	Name of the month (full)
yy	Year, two numeric digits
yyyy	Year, four numeric digits
hh	Hours
mm	Minutes
ss	Seconds

All .NET Framework date and time formatting strings are supported. For more information, see the Microsoft Developer Network website: msdn.microsoft.com.

Functions of the MLDate class

CurrentDate

Returns the current date as a serial number.

SYNTAX	`CurrentDate()`
RETURN	Date as a serial number
EXAMPLE	`GetObject("date","MLDate")` `CurrDateAsInt = date.CurrentDate()`

CurrentDateToString

Returns the current date as a date string.

SYNTAX	`CurrentDateToString(format)`
ARGUMENTS	`format` Formatting rules that define how the date string is shown. See Format specifiers.
RETURN	Current date as a formatted date string.

EXAMPLE	`GetObject("date","MLDate")`
	`CurrDateAsStr =`
	` date.CurrentDateToString("MM.dd.yyyy")`

DateStringToInt64

Converts a date string to a serial number date.

SYNTAX	`DateStringToInt64(format, date)`
ARGUMENTS	`format` — Formatting rules that define how the date string is displayed. See Format specifiers. `date` — Date string to convert.
RETURN	Date serial number

EXAMPLE	`GetObject("date","MLDate")`
	`DateStrToInt = date.DateStringToInt64("MM/dd/yyyy",`
	` "10/31/2017")`

DayOfWeek

Returns the week day for a date specified as a serial number.

SYNTAX	`DayOfWeek(date)`

ARGUMENTS	`date` Integer date
RETURN	Weekday number 0 = Sunday to 6 = Saturday

EXAMPLE	`GetObject("date","MLDate")`
	` CurrDateAsInt = date.CurrentDate()`
	` DayOfWeek = date.DayOfWeek(CurrDateAsInt)`

DaysInMonth

Calculates the number of days in the specified month for a specific year.

SYNTAX	`DaysInMonth(month, year)`
ARGUMENTS	`month` — Month as an integer `year` — Year as an integer
RETURN	Number of days
EXAMPLE	`GetObject("date","MLDate")` `DaysInMonth = date.DaysInMonth(2, 2017)`

FormatDateTimeString

Formats the specified date string according to the rule for a date string.

SYNTAX	`FormatDateTimeString(format, date)`
ARGUMENTS	`format` — Formatting rules that define how the date string is displayed. See Format specifiers. `date` — Date string.
RETURN	Formatted date string
EXAMPLE	`GetObject("date","MLDate")` ` FormatedDateTimeStr = date.FormatDateTimeString("MM/dd/yyyy", "31/10/2017")`

IncDate

Adds a specific number of days, months and years to a date. If a negative number is defined, the number is subtracted from the date.

SYNTAX	`IncDate(date, days, months, years)`

ARGUMENTS	`date` Date serial number `days` Number of days `months` Number of months `years` Number of years
RETURN	Date serial number

EXAMPLE	`GetObject("date","MLDate")`
	
	`CurrDateAsInt = date.CurrentDate()`
	
	`IncDateInt = date.IncDate(CurrDateAsInt, 10, 0, 0)`

Int64ToDateString

Converts the specified serial number date to a formatted date string.

SYNTAX	`Int64ToDateString(format, date)`
ARGUMENTS	`format` Formatting rules that define how the date string is displayed. See Format specifiers. `date` Date serial number.
RETURN	Date as a formatted date string.

EXAMPLE	`GetObject("date","MLDate")`
	
	`CurrDateAsInt = date.CurrentDate()`
	
	`DateIntToStr = date.Int64ToDateString("MM.dd.yyyy", CurrDateAsInt)`

IsLeapYear

Checks whether the year in question is a leap year.

SYNTAX	`IsLeapYear(year)`
ARGUMENTS	`year` Year as a number
RETURN	`TRUE` = The year is a leap year `FALSE` = The year is not a leap year
EXAMPLE	`GetObject("date","MLDate")` `IsLeapYear = date.IsLeapYear(2014)`

WeekOfYear

Returns the week number from the specified date serial number in accordance with ISO 8601.

SYNTAX	`WeekOfYear(date)`
ARGUMENTS	`date` Date serial number
RETURN	Week number

EXAMPLE	`GetObject("date","MLDate")`
	
	`CurrDateAsInt = date.CurrentDate()`
	
	`WeekOfYear =`
	` date.WeekOfYear(CurrDateAsInt)`

File processing functions

With the `MLDos` object, you can access files and perform general file-based operations. To load the object, use the `GetObject` function (see GetObject).

Example

```
GetObject("DOS", "MLDos")
```

Functions of the MLDos class

Properties of the MLDos class

Property	Description
`NewLine`	Gets the string for a line break defined for a specific environment.
`NewLineLF`	Returns `LineFeed` as a string `("\n" - $0A)`.
`NewLineCRLF`	Returns `CarriageReturn` / `LineFeed (CRLF)` as a string `("\r\n" - $0D $0A)`.

ChangeExtension

Changes the extension of a path string.

SYNTAX	`ChangeExtension(path, extension)`
ARGUMENTS	`path` — Path information to modify `extension` — New extension
RETURN	String containing the altered path information

Combine

Combines two path strings.

SYNTAX	`Combine(path1, path2)`

ARGUMENTS	`path1` First path `path2` Second path
RETURN	A string containing the combined path entries. If one of the specified paths is a zero-length string, this function returns the other path. If `path2` contains an absolute path, the function returns `path2`.

CopyFile

Copies an existing file to a new file.

SYNTAX	`CopyFile(sourceFileName, destFileName)` `CopyFile(sourceFileName, destFileName, overwrite)`
ARGUMENTS	`sourceFileName` Name of the existing file. `destFileName` Name of the target file. The name cannot be a directory. `overwrite` To use an existing file name, set this parameter to True. Otherwise, set to False.

CreateTextFile

Creates a text file to read, write, or append.

SYNTAX	`CreateTextFile(newLine)`

ARGUMENTS	`inputFileName`
	Defines the string for a line break. If the parameter is not specified, `CRLF ("\r\n" - $0D $0A)` is used as the default line break.
	Possible values:
	`DOS.NewLineLF = LineFeed ("\n" - $0A)`
	`DOS.NewLineCRLF = CarriageReturn/LineFeed ("\r\n" - $0D $0A)`

RETURN	`MonaLisaTextFile` object to read, write, or append to a file.

CreateDirectory

Creates all directories and subdirectories according to a specified path.

SYNTAX	`CreateDirectory(path)`
ARGUMENTS	`path` Directory path to create

DeleteDirectory

Deletes the specified directory and all its subdirectories, if configured.

SYNTAX	`DeleteDirectory(path)` `DeleteDirectory(path, recursive)`
ARGUMENTS	`path` — Name of the empty directory to delete. This directory must be writable or empty. `recursive` — `TRUE` = Deletes the directories, subdirectories, and files in path. `FALSE` = Does not delete.

DeleteFile

Deletes the specified file.

No exception is thrown, if the file does not exist.

SYNTAX	`DeleteFile(path)`
ARGUMENTS	`path` Name of the file to delete

DeleteFiles

Deletes all files in a directory and subdirectory that match a search pattern.

SYNTAX	`DeleteFiles(path, searchPattern, recursive)`
ARGUMENTS	`path` — Name of the directory `searchPattern` — The search string to match against the names of files in path `recursive` — `TRUE` = Searches subdirectories `FALSE` = Does not search subdirectories

DirectoryExists

Determines whether the specified path points to a directory on a drive.

SYNTAX	`DirectoryExists(path)`
ARGUMENTS	`path` Path to test
RETURN	`TRUE` = Path exists `FALSE` = Path does not exist

ExecuteSynchron

Starts the specified program and returns an exit code when the program is closed.

SYNTAX	`ExecuteSynchron(file, parameter, hide)`
ARGUMENTS	`file` — Program with path `parameter` — Program parameters `hide` — `TRUE` = Hides the window of the started program; `FALSE` = Displays the window of the started program
RETURN	Program exit code

Note for 64-bit operating systems

CCM Studio is a 32-bit program.

If you use the `ExecuteSynchron` command to start a program from directory `%windir%\System32`, you need to define the directory `%windir%\Sysnative` instead.

If you use the `ExecuteSynchron` command to start a program from directory `C:\windows\System32`, you need to define the directory `C:\windows\Sysnative` instead.

For more information on the File System Redirector, see to the Microsoft Developer Network website: msdn.microsoft.com.

FileAppendText

Opens a file, appends the specified string to the file, and then closes the file.

If the file does not already exist, the function creates it, writes the string in the file, and then closes it.

SYNTAX	`FileAppendText(fileName, content)` `FileAppendText(fileName, content, encoding)`

ARGUMENTS	`fileName` — The file to which the string is appended. `content` — The string to append to the file. `encoding` — You can use the following values as encoding parameters. • Empty string. In this case, the default encoding (ANSI code page of the operating system) is used. • Code page. For example, "1252" • Name. For example, "windows-1252" • User-friendly name. For example, "Western European" (Windows) For a list of possible encoding parameters, see Encoding parameters.

FileExists

Checks whether the specified file exists.

SYNTAX	`FileExists(path)`
ARGUMENTS	`path` The file to check
RETURN	`TRUE` = The caller has sufficient rights, and the path contains the name of an existing file. `FALSE` = Invalid path or string length 0 set for the path. An exception is thrown if the caller does not have sufficient rights to read the specified file. The function returns FALSE regardless of whether the path exists.

FileReadAllBytes

Opens a file, reads the file content into a byte array, and then closes the file.

SYNTAX	`FileReadAllBytes(fileName)`
ARGUMENTS	`fileName` The file to open
RETURN	The complete file content as byte array

FileReadAllText

Opens a text file, reads all lines of the file, and then closes the file.

SYNTAX	`FileReadAllText(fileName)` `FileReadAllText(fileName, encoding)`
ARGUMENTS	`fileName` — The file to open. `encoding` — You can use the following values as encoding parameters. • Empty string. In this case, the default encoding (ANSI code page of the operating system) is used. • Code page. For example, "1252" • Name. For example, "windows-1252" • User-friendly name. For example, "Western European" (Windows) For a list of possible encoding parameters, see Encoding parameters.
RETURN	A string containing all lines of the file

FileSize

Returns the size of a file.

SYNTAX	`FileSize(fileName)`
ARGUMENTS	`fileName` Name of the file
RETURN	Size of the file

GetCommandLine

Returns the command line for a process.

SYNTAX	`GetCommandLine()`
RETURN	String containing all command line arguments

GetCommandLineArgByName

Returns the specified command line argument.

SYNTAX	`GetCommandLineArgByName(name)`
ARGUMENTS	`name` Name of the command line argument
RETURN	Value of the command line argument, or an empty string if the parameter does not exist.

GetCommandLineArgs

Returns a comma-separated string of the command line arguments of the current process.

SYNTAX	`GetCommandLineArgs()`
RETURN	Comma-separated string of the command line arguments, or an empty string.

GetCurrentDirectory

Gets the current working directory of a program.

SYNTAX	`GetCurrentDirectory()`
RETURN	String containing the path of the current working directory

GetDirectories

Returns the names of subdirectories in a specified directory as a comma-separated string.

SYNTAX	`GetDirectories(path)`
	`GetDirectories(path, searchPattern)`
	`GetDirectories(path, searchPattern, recursive)`

ARGUMENTS	`path` — Path where the names of the subdirectories are obtained. `searchPattern` — Search string to match against the names of files in the path. `recursive` — `TRUE` = Searches subdirectories. `FALSE` = Does not search subdirectories.
RETURN	Comma-separated string containing the names of all subdirectories

GetDirectoryName

Returns the directory information for a specified path string.

SYNTAX	`GetDirectoryName(path)`
ARGUMENTS	`path` Path to a file or directory.
RETURN	String containing directory information for a path, or an empty string if the path has no directory information.

GetDirectoryRoot

Returns information about the volume and/or root for the specified path.

SYNTAX	`GetDirectoryRoot(path)`
ARGUMENTS	`path` Path to a file or directory
RETURN	String containing information about the volume and/or root

GetEnvironmentVariable

Returns the value of an environment variable.

SYNTAX	`GetEnvironmentVariable(variable)`
ARGUMENTS	`variable` Name of the environment variable.
RETURN	The value of the environment variable specified as `variable`, or an empty string if the environment variable does not exist.

GetExtension

Returns the file extension of the specified path string.

SYNTAX	`GetExtension(path)`
ARGUMENTS	`path` Path string from which the extension is read.
RETURN	String containing the extension of the specified path, including a dot.

GetFileName

Returns the file name and extension of the specified path.

SYNTAX	`GetFileName(path)`
ARGUMENTS	`path` Path where the file name and extension are obtained.
RETURN	String containing the characters after the last directory character in the path. If the last character of the path is a directory or volume separator character, this function returns an empty string.

GetFileNameWithoutExtension

Returns the name of a file from the specified path string without the extension.

SYNTAX	`GetFileNameWithoutExtension(path)`
ARGUMENTS	`path` Path of the file.
RETURN	String containing the result string from `GetFileName` without the last dot and all following characters.

GetFiles

Returns the names of files in a specified directory as a comma-separated string.

SYNTAX	`GetFiles(path)`
	`GetFiles(path, searchPattern)`
	`GetFiles(path, searchPattern, recursive)`

ARGUMENTS	`path` — The directory from which the files are obtained. `searchPattern` — The search string to match against the names of files in path. The `searchPattern` parameter cannot end in two periods or contain two periods. `recursive` — `TRUE` = Searches in the current directory and all subdirectories. **Note** If the directory structure contains a link that creates a loop back within the directory structure, the search results in an infinite loop. `FALSE` = Searches only in the current directory.
RETURN	A comma-separated string of file names in the specified directory. Defining the `searchPattern` parameter returns the names of files that match the specified search pattern. File names contain the full path.

GetFileSystemEntries

Returns the names of all files and subdirectories in a specified directory in a comma-separated string.

SYNTAX	`GetFileSystemEntries(path)` `GetFileSystemEntries(path, searchPattern)`
ARGUMENTS	`path` — The directory that the files are obtained from. `searchPattern` — The search string to match against the names of files in the path. The `searchPattern` parameter cannot end in two periods or contain two periods.
RETURN	A comma-separated string containing the names of the file system entries found in the specified directory and matching the search criteria specified in `searchPattern`.

GetFullPath

Returns the absolute path to a specified path string.

SYNTAX	`GetFullPath(path)`
ARGUMENTS	`path` File or directory for which the absolute path information is retrieved.
RETURN	String containing the absolute path of the variable path. **Example** `C:\MyFile.txt`

GetLogicalDrives

Returns the names of the logical drives of a computer in this format: `<DriveLetter>:\`

SYNTAX	`GetLogicalDrives()`
RETURN	Logical drives of the given computer.

GetParentDirectory

Retrieves the parent directory of the specified path, including both absolute and relative paths.

SYNTAX	`GetParentDirectory(path)`
ARGUMENTS	`path` Path from which the parent directory is retrieved.
RETURN	The parent directory or an empty string when the path is the root directory, including the root of a UNC server or shared name.

GetPathRoot

Returns the root directory information of the specified path.

SYNTAX	`GetPathRoot(path)`
ARGUMENTS	`path` Path from which the root directory information is retrieved.
RETURN	A string containing the root directory from the path, such as C:\, or an empty string when the path contains no information on the root directory.

GetRandomFileName

Returns a random file or directory name.

SYNTAX	`GetRandomFileName()`
RETURN	Random file or directory name

Note the following when using the function:

- This function returns a cryptographically strong, random string that you can use as either a directory name or a file name.
- Unlike `GetTempFileName`, the `GetRandomFileName` function does not create a file.
- When the security of your file system is paramount, use this function instead of `GetTempFileName`.

GetTempFileName

Creates a uniquely named temporary file, with the .tmp extension, of zero bytes on a data storage device and returns the full path to this file.

SYNTAX	`GetTempFileName()`
RETURN	String containing the full path of a temporary file

GetTempPath

Returns the path to the temporary directory of the current system.

SYNTAX	`GetTempPath()`
RETURN	String containing the path to a temporary directory

HasExtension

Determines whether a path contains a file name extension.

SYNTAX	`HasExtension(path)`
ARGUMENTS	`path` Path in which to search for an extension.
RETURN	`TRUE` = The characters that follow the last directory separator or volume separator character, containing a period and one or more trailing characters. `FALSE` = The path does not contain a file name extension.

IsPathRooted

Returns a value containing information on whether the specified path string contains a relative or absolute path.

SYNTAX	`IsPathRooted(path)`
ARGUMENTS	`path` Path to test
RETURN	`TRUE` = The string contains an absolute path `FALSE` = The string contains a relative path

MoveDirectory

Moves a file or directory and its content to a new storage location.

SYNTAX	`MoveDirectory(sourceDirName, destDirName)`
ARGUMENTS	`sourceDirName` — Path to the file or directory to move `destDirName` — Path to the new storage location for `sourceDirName`. If `sourceDirName` is a file, `destDirName` must also be a file name.

MoveFile

Moves the specified file to a new storage location and also enables you to modify the file name.

SYNTAX	`MoveFile(sourceFileName, destFileName)`
ARGUMENTS	`sourceFileName` — Name of the file to move `destFileName` — New path to the file

ReplaceFile

Replaces the specified file content with the content of another file, deletes the original file, and then creates a backup copy of the replaced file.

SYNTAX	`ReplaceFile(sourceFileName, destinationFileName, destinationBackupFileName)`
ARGUMENTS	`sourceFileName` — The name of the file that `destinationFileName` is replaced with. `destinationFileName` — The name of the file to replace. `destinationBackupFileName` — The name of the backup file of the file replaced, or an empty string if no backup file is required.

SetCurrentDirectory

Sets the current working directory of the program to the specified directory.

SYNTAX	`SetCurrentDirectory(path)`
ARGUMENTS	`path` Path of the current working directory

Functions of the MonaLisaTextFile class

Properties of the MonaLisaTextFile class

Property	Description
`FileName`	Returns the file name of the opened file or an empty string if no file is opened.

Append

Opens a file to append text.

SYNTAX	`Append(string fileName)` `Append(string fileName, string encoding)`
ARGUMENTS	`fileName` — The path to the file to append. `encoding` — You can use the following values as encoding parameters. • Empty string. In this case, the default encoding (ANSI code page of the operating system) is used. • Code page. For example, "1252." • Name. For example, "windows-1252." • User-friendly name. For example, "Western European" (Windows). For a list of possible encoding parameters, see Encoding parameters.
EXAMPLE	`File.Append("c:\temp\test.txt")`

Close

Closes an open file.

Syntax

```
Close()
```

Eof

Determines whether the current position is the end of a file.

SYNTAX	`Eof()`
RETURN	`TRUE` = End of file reached `FALSE` = End of file not reached

Flush

Clears all buffers and writes them to a file.

Syntax

```
Flush()
```

Open

Opens a file to read a text.

SYNTAX	`Open(fileName)` `Open(fileName, encoding)`
ARGUMENTS	`fileName` — The file to open. `encoding` — You can use the following values as encoding parameters. • Empty string. In this case, the default encoding (ANSI codepage of the operating system) is used. • Code page. For example, "1252." • Name. For example, "windows-1252." • User-friendly name. For example, "Western European" (Windows). For a list of possible encoding parameters, see Encoding parameters.

ReadLine

Reads a line of characters from the current file and returns the data as a string.

SYNTAX	`ReadLine()`
RETURN	The next line of the file or an empty string if the end of the file is reached.

ReadToEnd

Reads the file from the current position to the end.

SYNTAX	`ReadToEnd()`

RETURN	The rest of the file from the current position to the end as a string. If the current position is the end of the file, an empty string ("") is returned.

Write

Writes a string to a file.

SYNTAX	`Write(text)`
ARGUMENTS	`text` The string to write

WriteFormat

Writes a formatted string using the same semantics as `FormatStr` (see FormatStr).

SYNTAX	`WriteFormat(formatRule, formatParams)`
ARGUMENTS	`formatRule` Formatting rule `formatParams` Objects to write to the formatted string

WriteLine

Writes a string followed by a character to a file for line termination.

SYNTAX	`WriteLine(text)`
ARGUMENTS	`text` The string to write

WriteLineFormat

Writes a formatted string and a new line using the same semantics as `FormatStr` (see FormatStr).

SYNTAX	`WriteLineFormat(formatRule, formatParams)`
ARGUMENTS	`formatRule` Formatting rule `formatParams` Objects to write to the formatted string

XML processing functions

With the `MLXmlDocument` object, you can access the `XmlDocument` class from the .NET Framework. This class helps you read and write XML files. To load the object, use the `GetObject` function (see GetObject).

For more information on the `XmlDocument` class, see the Microsoft Developer Network website: http://msdn.microsoft.com.

Example

```
GetObject("XML", "MLXmlDocument")
```

Additional functions of the MLXmlDocument class

GetNamespaceManager

Returns an `XmlNamespaceManager` object initialized with `NameTable` of given `XmlDocument`. This namespace manager can be used to run XPath queries with qualified namespaces. For this purpose, add the custom namespaces to the manager first.

SYNTAX	`GetNamespaceManager()`
RETURN	XML namespace manager object for this `MLXmlDocument` instance

Load

Loads XML data from a URL.

SYNTAX	Without namespaces preservation option:
	`Load(string filename)`
	With namespaces preservation option:
	`Load(string filename, bool stripOutNamespaces)`	

ARGUMENTS	`filename`	URL of the file to load, which contains an XML document. The URL may be a local file path or a HTTP URL (web address).
	`stripOutNamespaces`	`TRUE` = Removes namespace prefixes and definitions. `FALSE` = Preserves namespace prefixes and definitions.

LoadXml

Loads an XML document from a string.

SYNTAX	Without namespaces preservation option:
	`LoadXML(string xml)`
	With namespaces preservation option:
	`LoadXML(string xml, bool stripOutNamespaces)`

ARGUMENTS	`xml`	String containing the XML document
	`stripOutNamespaces`	`TRUE` = Removes namespace prefixes and definitions `FALSE` = Preserves namespace prefixes and definitions

Array functions

With the `MLArrays` object, you can work with object arrays. To load the object, use the `GetObject` function (see GetObject).

Example

```
GetObject("array","MLArrays")
```

Properties of the MLArrays class

Property	Description
`Count`	Number of elements in the array

Functions of the MLArrays class

AddToArray

Appends an element as a specific type, if specified, at the end of the array.

SYNTAX	`AddToArray(obj, addAsType)`	
ARGUMENTS	`obj`	Object to add
	`addAsType`	Optional. Element type

Clear

Clears all elements in an array.

SYNTAX	`Clear()`

GetArray

Returns a typed .NET array.

SYNTAX	`GetArray(type)`
ARGUMENTS	`type` Type of the array

RETURN	Typed .NET array

GetAtIndex

Retrieves the element in an array at the specified position.

SYNTAX	`GetAtIndex(index)`
ARGUMENTS	`index` Zero-based position from which to retrieve the element
RETURN	Object contained in the array

InitFromArrayObject

Initializes a class as an array with all elements from the provided `arrayObject`.

SYNTAX	`InitFromArrayObject(arrayObject)`
ARGUMENTS	`arrayObject` Object that implements the `IList` interface

InsertIntoArray

Inserts an element as a specific type into an array at the specified position.

SYNTAX	`InsertIntoArray(index, obj, addAsType)`
ARGUMENTS	`index` — Zero-based index at which the value is inserted `obj` — Object to insert `addAsType` — Optional. Element type

Object functions

With the `MLObjects` object, you can work with types and create dynamic object instances. To load the object, use the `GetObject` function (see GetObject).

Example

```
GetObject("object","MLObjects")
```

Functions of the MLObjects class

CreateNewInstanceOfType

Creates a new object instance of the specified type.

SYNTAX	`CreateNewInstanceOfType(type, args)`

ARGUMENTS	`type` — Type for which to create an instance. `args` — Arguments for the instance. This parameter is optional.
RETURN	New object instance of the type

GetEnumValue

Returns the enumeration value of a specific type by a specific untyped value.

SYNTAX	`GetEnumValue(type, value)`
ARGUMENTS	`type` — Enumeration type `value` — Untyped value of the enumeration value
RETURN	Typed enumeration value

GetModule

Returns the module in which the specified type is defined.

SYNTAX	`GetModule(type)`
ARGUMENTS	`type` Type for which to get the module
RETURN	Module in which the specified type is defined

GetTypeInModule

Resolves a typestring to a real type within a specific module.

SYNTAX	`GetTypeInModule(type, module)`
ARGUMENTS	`type` — Typestring used to retrieve the type `module` — Module in which the type is specified
RETURN	Real type

GetTypeOfObject

Returns the type of the specified object instance.

SYNTAX	`GetTypeOfObject(obj)`

ARGUMENTS	`obj` Object instance
RETURN	Type of the specified object instance

GetTypeOfObjectAsString

Returns the typestring of the specified object instance.

SYNTAX	`GetTypeOfObjectAsString(obj)`
ARGUMENTS	`obj` Object instance
RETURN	Typestring of the specified object instance

Scripting contexts

The context determines which objects can be assigned to a script and which properties and functions are available.

Process

Use this context for scripts in processes.

Properties of the Process context

Property	Description

`ModusProcess`	Provides the following properties and functions.
	Properties:
	`Objectname`: Name of the process.
	`SystemOId`: ID of the system where the process runs.
	Functions:
	• `Cancel()` Cancels the process.
	`ModusProcess.Cancel()`
	• `GetComponent()` Enables access to the properties and methods of components. `DocumentCollection = ModusProcess.GetComponent ("DocumentCollectionManager")`
	• `SendInfoMessage()` Sends an information message to logged on remote control clients. A time stamp is inserted before each message.
	`ModusProcess.SendInfoMessage("This is the message")`
	• `Stop()` Ends running processes.
	`ModusProcess.Stop()`

OnProcessError

Use this context for the OnError script in processes.

Properties of the OnProcessError context

Property	Description
`LastErrorMessage`	Last error message

`ModusProcess`	**Properties:**
	`Objectname`: Name of the process
	`SystemOId`: ID of the system where the process runs
	Functions:
	• `Cancel()` Cancels the process.
	`ModusProcess.Cancel()`
	• `GetComponent()` Enables access to the properties and methods of components. `DocumentCollection = ModusProcess GetComponent ("DocumentCollection Manager")`
	• `SendInfoMessage()` Sends an information message to logged-on remote control clients. A time stamp is inserted before each message.
	`ModusProcess.SendInfoMessage("This is the message")`
	• `Stop()` Ends running processes.
	`ModusProcess.Stop()`

`ResetError`	Resets an error state in an `OnError` script. `TRUE` = Prevents cancelling a process in the event of an error. `FALSE` = Allows cancelling a process in the event of an error. If the activities are run on a timer, the current process is rerun at the next scheduled interval. Otherwise, the process is terminated.

Property	Description
`ResumeOptions`	**Possible values:** • `ProcessResumeOptions.NextActivity`: The process is resumed with the next activity. • `ProcessResumeOptions.FromStart`: The process is started from the beginning. • `ProcessResumeOptions.FromStartWithIntervall`: The process is started from the beginning at the next scheduled interval. **Note** • Variable pools cannot be reset. • Setting `ResumeOptions` only applies when the process is run on a timer.

StreamingUnit

Use this context for scripts that run during the streaming of stacks, envelopes, and documents (see Add and modify document pages in streaming).

Properties of the StreamingUnit context

Property	Description
`Stream`	Returns the IOdinStream interface. Script alias: stream. The interface has the following functions: • AppendDocument • GetProperty • LoadDocument • SetProperty
`HasStream`	`TRUE` = A stream object exists in the `Stream` context property. `FALSE` = No stream object exists in the `Stream` context property.
`GetOrganisationalMetadata (key)`	Function that produces the value of a key in the organisational metadata. If a key is not present, an exception is thrown.
`HasRecipient`	Flag that indicates whether the recipient information is available for the distributed communication. Always set to false for the OnStackStart and OnStackEnd exit points.
`PrinterName`	Name of the printer used to distribute the communication. Unavailable for the OnStackStart and OnStackEnd exit points.

Property	Description
`RecipientType`	Recipient type of the distributed communication. Unavailable for the OnStackStart and OnStackEnd exit points.
`GetRecipientData (key)`	Function that produces the value of a contact field in the recipient information of the distributed communication. Throws an exception if the contact field does not exist.
`HasSender`	Flag that indicates whether the sender information is available for the distributed communication. Always set to false for the OnStackStart and OnStackEnd exit points.
`GetSenderData (key)`	Function that produces the value of a contact field in the sender information of the distributed communication. Throws an exception if no sender information is available, or if the contact field does not exist.
`GetScriptData (level, key)`	Retrieves a key/value pair. Throws an exception if the script tries to access an unavailable level, or if the key is not present for the level (see the table about levels and scopes later in this section).
`SetScriptData (level, key, value)`	Function that writes a key/value pair at the specified level. Allows the streaming scripts to transfer data between the exit points. Throws an exception if the script tries to access an unavailable level, or if the key is not present for the level.
`int LogicalPageNumber`	Contains the logical page number within the document.
`IsFirstPage`	True if this is the first page in a document.
`IsLastPage`	True if this is the last page in a document.
`PageCount`	Returns the number of pages in the current document.
`IsPortrait`	Returns the portrait mode.
`IsDuplex`	Returns the duplex mode.
`InputBin`	Combines `GetInputBin()` and `SetInputBin()`.

Level	Scope	Availability
0	Stack	Available in all exit points.
1	Envelope	Available in the OnEnvelope, OnDocument, and OnPage exit points.
2	Document	Available in the OnDocument and OnPage exit points.

Functions of the IOdinStream interface

AppendDocument

Adds a document loaded with `LoadDocument` to a stream.

SYNTAX	`AppendDocument(document, insertOMR, insertDVFrei)`
ARGUMENTS	`document` — Document to add. `insertOMR` — Defines whether OMR codes are printed on the document. OMR codes are not printed on a stack cover page. `insertDVFrei` — *Reserved for future use.*

EXAMPLE	`document = stream.LoadDocument("d:\import\Empty_Page.xps", OdinDuplexSetting.Simplex, "firstbin","nextbin")`
	
	`stream.AppendDocument(document, true, false)`

GetProperty

Retrieves streaming values such as a page counter.

SYNTAX	`GetProperty(propertyName)`
ARGUMENTS	`propertyName` Name of the property
RETURN	Property value. You can query the following properties. The identifiers in brackets are those defined in modus four that remain valid to ensure backward compatibility. They are automatically converted to the corresponding new names.

General

Property	Description
`EnvelopeId`	Envelope ID of the currently streamed envelope.
`JobId`	Job ID of the job or currently streamed document.
`OMRCode`	Read-only. Current value of the OMR code.
`OMRReset`	`TRUE` = A reset flag (all OMR marks set) is printed on all subsequent documents instead of the OMR code. The reset flag is printed until the property is set to `FALSE`. The `OnStackEnd` event inserts an additional document or an additional page after the stack that contains a reset flag for the production inserter.
`StackId`	Stack ID of the stack to stream.
`StreamFileName`	Name of the stream file.
`Sysdate`	Current date.
`Systime`	Current time.

Property	Description
VirtualPageDescription (VPageDesc)	For a stack cover page = StackCoverPage. For an envelope cover page = EnvelopeCoverPage. Otherwise, this is an empty string.

Stack

Property	Description
StackEnvelope	Envelope counter for the whole stack.
StackJob	Job or document counter for the whole stack.
StackPage	Page counter for the whole stack.
StackPaper	Sheet counter for the whole stack.
TotalStackEnvelopes (EnvCount)	Number of envelopes in a stack.
TotalStackJobs	Number of jobs or documents in a stack.
TotalStackPages (StackPage)	Number of pages in a stack.
TotalStackPapers	Number of sheets in a stack.

Envelope

Property	Description
EnvelopeEnd	TRUE = The current document is the last document of the envelope. FALSE = The current document is not the last document of the envelope.
EnvelopeJob	Job or document counter for the whole envelope.
EnvelopePage	Page counter for the whole envelope.
EnvelopePaper	Sheet counter for the whole envelope.
EnvelopeStart (EnvStart)	TRUE = The current document is the first document of the envelope. FALSE = The current document is not the first document of the envelope.
TotalEnvelopeJobs (JobCount)	Number of jobs or documents in an envelope.
TotalEnvelopePages	Number of pages in an envelope.
TotalEnvelopePapers	Number of sheets in an envelope.

Job / Document

Property	Definition
JobPage (PageCount)	Page counter for a document.
JobPaper (PageCountExtended)	Sheet counter for a document.
TotalJobPages	Number of pages in a document.

Property	Definition
TotalJobPapers	Number of sheets in a document.

Example

```
; Stack was processed by streaming...

; As an example, an additional page is inserted containing details of the number of
 pages/sheets, envelopes/jobs

;

TotalPages = stream.GetProperty("TotalStackPages")

TotalPapers = stream.GetProperty("TotalStackPapers")

TotalEnvelopes = stream.GetProperty("TotalStackEnvelopes")

TotalJobs = stream.GetProperty("TotalStackJobs")

;

; Load cover page for tray "coverpage"

document = stream.LoadDocument("c:\PerceptiveSoftware\modusSuite\data\EmptyPage.xps",
 OdinDuplexSetting.Simplex, "coverpage", "coverpage")

;

; print text as a test

document.SetFont("Tahoma", 16)

document.EditPage(0)

document.TextOut(20.5, 10.5, FormatStr("TotalPages/TotalPapers: %d/%d", TotalPages,
 TotalPapers), false, false)

document.TextOut(20.5, 17, "Number Envelopes/Jobs: %d/%d", TotalEnvelopes, TotalJobs),
 false, false)

document.Post()

;

; Print OMR-Reset character

Stream.SetProperty("OMRReset", true)

;

; Add document to stream

stream.AppendDocument(document, true, false)

;

; Free up the loaded document

FreeObject("document")
```

LoadDocument

Loads an XPS document. You can add this document as an additional stack or envelope cover page or as a virtual document by using the `AppendDocument` function.

SYNTAX	`LoadDocument(fileName, duplex, firstBin, nextBin)`
ARGUMENTS	`fileName` — Name of the document to load. `duplex` — **Possible values:** `OdinDuplexSetting.Simplex`: Simplex mode - no duplex; `OdinDuplexSetting.Horizontal`: Horizontal duplex (long edge); `OdinDuplexSetting.Vertical`: Vertical duplex (short edge) `firstBin` — Tray for the first page. `nextBin` — Tray for the following page.
RETURN	Returns the IOdinStreamDocument interface with the functions required to perform drawing operations on pages.
EXAMPLE	`document = stream.LoadDocument("d:\import\Empty_Page.xps", OdinDuplexSetting.Simplex, "firstbin","nextbin")`

SetProperty

Specifies new settings for properties.

Note Altering values affects counters. Use the function with caution.

SYNTAX	`SetProperty(propertyName, value)`
ARGUMENTS	`propertyName` — Property name `value` — Property value

StreamingPage

Use this context in scripts that run during the streaming of a document page (see Modify a document page).

Properties of the StreamingPage context

Property	Description
`HasStream`	`TRUE` = A stream object exists in the `Stream` context property. `FALSE` = No stream object exists in the `Stream` context property.
`GetOrganisationalMetadata (key)`	Function that produces the value of a key in the organisational metadata. If a key is not present, an exception is thrown.
`HasRecipient`	Flag that indicates whether the recipient information is available for the distributed communication.
`PrinterName`	Name of the printer used to distribute the communication.
`RecipientType`	Recipient type of the distributed communication.
`GetRecipientData (key)`	Function that produces the value of a contact field in the recipient information of the distributed communication. Throws an exception if the contact field does not exist.
`HasSender`	Flag that indicates whether the sender information is available for the distributed communication.
`GetSenderData (key)`	Function that produces the value of a contact field in the sender information of the distributed communication. Throws an exception if no sender information is available, or the contact field does not exist.
`GetScriptData (level, key)`	Retrieves a key/value pair. Throws an exception if the script tries to access an unavailable level, or if the key is not present for the level (see the table about levels and scopes later in this section).
`SetScriptData (level, key, value)`	Function that writes a key/value pair at the specified level. Allows the streaming scripts to transfer data between the exit points. Throws an exception if the script tries to access an unavailable level, or if the key is not present for the level.
`int LogicalPageNumber`	Contains the logical page number within the document.
`IsFirstPage`	True if this is the first page in a document.
`IsLastPage`	True if this is the last page in a document.
`PageCount`	Returns the number of pages in the current document.
`IsPortrait`	Returns the portrait mode.
`IsDuplex`	Returns the duplex mode.
`InputBin`	Combines `GetInputBin()` and `SetInputBin()`.

Level	Scope	Availability
0	Stack	Available in all exit points.

Level	Scope	Availability
1	Envelope	Available in the OnEnvelope, OnDocument, and OnPage exit points.
2	Document	Available in the OnDocument and OnPage exit points.

StackDistribution

Use this context for scripts that run during the distribution of output files.

Properties of the StackDistribution context

Property	Description
`Stack.Id`	Read-only. Returns the stack Identifier of the distributed stack.
`Stack.Channel`	Read-only. Returns the channel name of the channel for which this stack was created.
`Stack.Printer`	Read-only. Returns the printer name of the printer for which this stack was created.
`Stack.OutputFormat`	Read-only. Returns the output format as configured on the Conversion component. For information on the list of output formats, see the next table.
`Stack.OutputFolder`	Returns the location of the output folder where the print file is saved. To change the location, set `Stack.OutputFolder` to a new value.
`Stack.OutputPrintFile`	Returns the print file name. By default, this name consists of the stack ID and the `Stack.OutputFormat` as extension. For example, 42.pdf. To change the name, set `Stack.OutputPrintFile` to a new value. **Note** `Stack.OutputPrintFile` does not return the full path. To get the full path, use `Stack.OutputFolder` with `Stack.OutputPrintFile`.
`Stack.OutputPrintFileContent`	Returns the print file content as a byte array.
`Stack.OutputPrintFileContentBase64String`	Returns the print file content as a Base64 encoded string.

Property	Description
`Stack.OutputDescriptionFile`	Returns the name of the XML description file. By default, the name consists of the stack ID with extension .xml. For example, 42.xml. To change the name, set a new value to `Stack.OutputDescriptionFile`. **Note** This variable does not return the full path. To get the full path, use `Stack.OutputFolder` with `Stack.OutputDescriptionFile`.
`Stack.OutputDescriptionFileContent`	Returns the content of the generated description file stored in the storage folder as a byte array.
`Stack.OutputDescriptionFileContentAsString`	Returns the content of the XML description file as an XML string.
`Stack.OutputDescriptionFileContentBase64String`	Returns the content of the XML description file as a Base64 encoded string.
`Stack.GetOrganisationalMetadata (key)`	Function that produces the value of a key in the organisational metadata. It only returns values for those keys that are identical for all documents within the stack. If a key is not present, an exception is thrown.
`Stack.Save()`	Function that saves the print file and the XML description file. It copies the print file from the storage folder to the output folder as set in `Stack.OutputFolder`, under the name as set in `Stack.OutputPrintFile`. Also, it copies the XML description file from the storage folder to the output folder as set in `Stack.OutputFolder` under the name as set in `Stack.OutputDescriptionFile`.

Output format	Description
AFP	Advanced Function Presentation
PCL	Printer Command Language
PDF	Portable Document Format
POS	PostScript
PXL	PCL 6 Enhanced
TIFF	Tagged Image Format

Appendix A

Encoding parameters

Code page	Name	Displayed name
37	IBM037	IBM EBCDIC (USA-Canada)
437	IBM437	OEM USA
500	IBM500	IBM EBCDIC (International)
708	ASMO-708	Arabic (ASMO 708)
720	DOS-720	Arabic (DOS)
737	ibm737	Greek (DOS)
775	ibm775	Baltic (DOS)
850	ibm850	Western European (DOS)
852	ibm852	Eastern European (DOS)
855	IBM855	OEM Cyrillic
857	ibm857	Turkish (DOS)
858	IBM00858	OEM Multilingual Latin I
860	IBM860	Portuguese (DOS)
861	ibm861	Icelandic (DOS)
862	DOS-862	Hebrew (DOS)
863	IBM863	French, Canada (DOS)
864	IBM864	Arabic (864)
865	IBM865	Nordic (DOS)
866	cp866	Cyrillic (DOS)
869	ibm869	Greek, modern (DOS)
870	IBM870	IBM EBCDIC (Multilingual Latin-2)
874	windows-874	Thai (Windows)
875	cp875	IBM EBCDIC (Greek, modern)
932	shift_jis	Japanese (Shift-JIS)
936	gb2312	Simplified Chinese (GB2312)
949	ks_c_5601-1987	Korean
950	big5	Chinese (Traditional) (Big5)
1026	IBM1026	IBM EBCDIC (Turkish, Latin-5)

Code page	Name	Displayed name
1047	IBM01047	IBM Latin-1
1140	IBM01140	IBM EBCDIC (USA-Canada-European)
1141	IBM01141	IBM EBCDIC (Germany-European)
1142	IBM01142	IBM EBCDIC (Denmark-Norway-European)
1143	IBM01143	IBM EBCDIC (Finland-Sweden-European)
1144	IBM01144	IBM EBCDIC (Italian-European)
1145	IBM01145	IBM EBCDIC (Spain-European)
1146	IBM01146	IBM EBCDIC (Great Britain-European)
1147	IBM01147	IBM EBCDIC (France-European)
1148	IBM01148	IBM EBCDIC (International-European)
1149	IBM01149	IBM EBCDIC (Icelandic-European)
1200	utf-16	Unicode
1201	unicodeFFFE	Unicode (Big Endian)
1250	windows-1250	Central European (Windows)
1251	windows-1251	Cyrillic (Windows)
1252	Windows-1252	Western European (Windows)
1253	windows-1253	Greek (Windows)
1254	windows-1254	Turkish (Windows)
1255	windows-1255	Hebrew (Windows)
1256	windows-1256	Arabic (Windows)
1257	windows-1257	Baltic (Windows)
1258	windows-1258	Vietnamese (Windows)
1361	Johab	Korean (Johab)
10000	macintosh	Western European (Mac)
10001	x-mac-japanese	Japanese (Mac)
10002	x-mac-chinesetrad	Chinese traditional (Mac)
10003	x-mac-korean	Korean (Mac)
10004	x mac-arabic	Arabic (Mac)
10005	x-mac-hebrew	Hebrew (Mac)
10006	x-mac-greek	Greek (Mac)
10007	x-mac-cyrillic	Cyrillic (Mac)

Code page	Name	Displayed name
10008	x-mac-chinesesimp	Chinese simplified (Mac)
10010	x-mac-romanian	Romanian (Mac)
10017	x-mac-ukrainian	Ukrainian (Mac)
10021	x-mac-thai	Thai (Mac)
10029	x-mac-ce	Central European (Mac)
10079	x-mac-icelandic	Icelandic (Mac)
10081	x-mac-turkish	Turkish (Mac)
10082	x-mac-croatian	Croatian (Mac)
12000	utf-32	Unicode (UTF-32)
12001	utf-32BE	Unicode (UTF-32-Big Endian)
20000	x-Chinese-CNS	Chinese traditional (CNS)
20001	x-cp20001	TCA Taiwan
20002	x-Chinese-Eten	Chinese traditional (Eten)
20003	x-cp20003	IBM5550 Taiwan
20004	x-cp20004	TeleText Taiwan
20005	x-cp20005	Wang Taiwan
20105	x-IA5	Western European (IA5)
20106	x-IA5-German	German (IA5)
20107	x-IA5-Swedish	Swedish (IA5)
20108	x-IA5-Norwegian	Norwegian (IA5)
20127	us-ascii	US-ASCII
20261	x-cp20261	T.61
20269	x-cp20269	ISO-6937
20273	IBM273	IBM EBCDIC (Germany)
20277	IBM277	IBM EBCDIC (Denmark-Norway)
20278	IBM278	IBM EBCDIC (Finland-Sweden)
20280	IBM280	IBM EBCDIC (Italy)
20284	IBM284	IBM EBCDIC (Spain)
20285	IBM285	IBM EBCDIC (Great Britain)
20290	IBM290	IBM EBCDIC (Japanese Katakana)
20297	IBM297	IBM EBCDIC (France)
20420	IBM420	IBM EBCDIC (Arabic)
20423	IBM423	IBM EBCDIC (Greek)
20424	IBM424	IBM EBCDIC (Hebrew)

Code page	Name	Displayed name
20833	x-EBCDIC-KoreanExtended	IBM EBCDIC (Korean, extended)
20838	IBM-Thai	IBM EBCDIC (Thai)
20866	koi8-r	Cyrillic (KOI8-R)
20871	IBM871	IBM EBCDIC (Icelandic)
20880	IBM880	IBM EBCDIC (Cyrillic, Russian)
20905	IBM905	IBM EBCDIC (Turkish)
20924	IBM00924	IBM Latin-1
20932	EUC-JP	Japanese (JIS 0208-1990 and 0212-1990)
20936	x-cp20936	GB2312-80 Chinese (simple)
20949	x-cp20949	Korean Wansung
21025	cp1025	IBM EBCDIC (Cyrillic, Serbian-Bulgarian)
21866	koi8-u	Cyrillic (KOI8-U)
28591	iso-8859-1	Western European (ISO)
28592	iso-8859-2	Central European (ISO)
28593	iso-8859-3	Latin 3 (ISO)
28594	iso-8859-4	Baltic (ISO)
28595	iso-8859-5	Cyrillic (ISO)
28596	iso-8859-6	Arabic (ISO)
28597	iso-8859-7	Greek (ISO)
28598	iso-8859-8	Hebrew (ISO-Visual)
28599	iso-8859-9	Turkish (ISO)
28603	iso-8859-13	Estonian (ISO)
28605	iso-8859-15	Latin 9 (ISO)
29001	x-Europa	Europe
38598	iso-8859-8-i	Hebrew (ISO-Logical)
50220	iso-2022-jp	Japanese (JIS)
50221	csISO2022JP	Japanese (JIS, 1 Byte Kana allowed)
50222	iso-2022-jp	Japanese (JIS, 1 Byte Kana allowed -SO/SI)
50225	iso-2022-kr	Korean (ISO)
50227	x-cp50227	ISO-2022 Chinese (simplified)
51932	euc-jp	Japanese (EUC)
51936	EUC-CN	Chinese simplified (EUC)

Code page	Name	Displayed name
51949	euc-kr	Korean (EUC)
52936	hz-gb-2312	Chinese simplified (HZ)
54936	GB18030	GB18030 Chines simplified
57002	x-iscii-de	ISCII Devanagari
57003	x-iscii-be	ISCII Bengali
57004	x-iscii-ta	ISCII Tamil
57005	x-iscii-te	ISCII Telugu
57006	x-iscii as	ISCII Assamese
57007	x-iscii-or	ISCII Oriya
57008	x-iscii-ka	ISCII Kannada
57009	x-iscii-ma	ISCII Malayalam
57010	x-iscii-gu	ISCII Gujarati
57011	x-iscii-pa	ISCII Punjabi
65000	utf-7	Unicode (UTF-7)
65001	utf-8	Unicode (UTF-8)

Appendix B

HTML color names

This table lists supported HTML color names and their ARGB values given in the format #AARRGGBB (AA=Alpha-Transparency, RR=Red, GG=Green, BB=Blue).

HTML color name	ARGB value
AliceBlue	#FFF0F8FF
AntiqueWhite	#FFFAEBD7
Aqua	#FF00FFFF
Aquamarine	#FF7FFFD4
Azure	#FFF0FFFF
Beige	#FFF5F5DC
Bisque	#FFFFE4C4
Black	#FF000000
BlanchedAlmond	#FFFFEBCD
Blue	#FF0000FF
BlueViolet	#FF8A2BE2
Brown	#FFA52A2A
BurlyWood	#FFDEB887
CadetBlue	#FF5F9EA0
Chartreuse	#FF7FFF00
Chocolate	#FFD2691E
Coral	#FFFF7F50
CornflowerBlue	#FF6495ED
Cornsilk	#FFFFF8DC
Crimson	#FFDC143C
Cyan	#FF00FFFF
DarkBlue	#FF00008B
DarkCyan	#FF008B8B
DarkGoldenrod	#FFB8860B
DarkGray	#FFA9A9A9
DarkGreen	#FF006400

HTML color name	ARGB value
DarkKhaki	#FFBDB76B
DarkMagenta	#FF8B008B
DarkOliveGreen	#FF556B2F
DarkOrange	#FFFF8C00
DarkOrchid	#FF9932CC
DarkRed	#FF8B0000
DarkSalmon	#FFE9967A
DarkSeaGreen	#FF8FBC8B
DarkSlateBlue	#FF483D8B
DarkSlateGray	#FF2F4F4F
DarkTurquoise	#FF00CED1
DarkViolet	#FF9400D3
DeepPink	#FFFF1493
DeepSkyBlue	#FF00BFFF
DimGray	#FF696969
DodgerBlue	#FF1E90FF
Firebrick	#FFB22222
FloralWhite	#FFFFFAF0
ForestGreen	#FF228B22
Fuchsia	#FFFF00FF
Gainsboro	#FFDCDCDC
GhostWhite	#FFF8F8FF
Gold	#FFFFD700
Goldenrod	#FFDAA520
Gray	#FF808080
Green	#FF008000
GreenYellow	#FFADFF2F
Honeydew	#FFF0FFF0
HotPink	#FFFF69B4
IndianRed	#FFCD5C5C
Indigo	#FF4B0082
Ivory	#FFFFFFF0
Khaki	#FFF0E68C
Lavender	#FFE6E6FA

HTML color name	ARGB value
LavenderBlush	#FFFFF0F5
LawnGreen	#FF7CFC00
LemonChiffon	#FFFFFACD
LightBlue	#FFADD8E6
LightCoral	#FFF08080
LightCyan	#FFE0FFFF
LightGoldenrodYellow	#FFFAFAD2
LightGray	#FFD3D3D3
LightGreen	#FF90EE90
LightPink	#FFFFB6C1
LightSalmon	#FFFFA07A
LightSeaGreen	#FF20B2AA
LightSkyBlue	#FF87CEFA
LightSlateGray	#FF778899
LightSteelBlue	#FFB0C4DE
LightYellow	#FFFFFFE0
Lime	#FF00FF00
LimeGreen	#FF32CD32
Linen	#FFFAF0E6
Magenta	#FFFF00FF
Maroon	#FF800000
MediumAquamarine	#FF66CDAA
MediumBlue	#FF0000CD
MediumOrchid	#FFBA55D3
MediumPurple	#FF9370DB
MediumSeaGreen	#FF3CB371
MediumSlateBlue	#FF7B68EE
MediumSpringGreen	#FF00FA9A
MediumTurquoise	#FF48D1CC
MediumVioletRed	#FFC71585
MidnightBlue	#FF191970
MintCream	#FFF5FFFA
MistyRose	#FFFFE4E1
Moccasin	#FFFFE4B5

HTML color name	ARGB value
NavajoWhite	#FFFFDEAD
Navy	#FF000080
OldLace	#FFFDF5E6
Olive	#FF808000
OliveDrab	#FF6B8E23
Orange	#FFFFA500
OrangeRed	#FFFF4500
Orchid	#FFDA70D6
PaleGoldenrod	#FFEEE8AA
PaleGreen	#FF98FB98
PaleTurquoise	#FFAFEEEE
PaleVioletRed	#FFDB7093
PapayaWhip	#FFFFEFD5
PeachPuff	#FFFFDAB9
Peru	#FFCD853F
Pink	#FFFFC0CB
Plum	#FFB0E0E6
Purple	#FF800080
Red	#FFFF0000
RosyBrown	#FFBC8F8F
RoyalBlue	#FF4169E1
SaddleBrown	#FF8B4513
Salmon	#FFFA8072
SandyBrown	#FFF4A460
SeaGreen	#FF2E8B57
SeaShell	#FFFFF5EE
Sienna	#FFA0522D
Silver	#FFC0C0C0
SkyBlue	#FF87CEEB
SlateBlue	#FF6A5ACD
SlateGray	#FF708090
Snow	#FFFFFAFA
SpringGreen	#FF00FF7F
SteelBlue	#FF4682B4

HTML color name	ARGB value
Tan	#FFD2B48C
Teal	#FF008080
Thistle	#FFD8BFD8
Tomato	#FFFF6347
Turquoise	#FF40E0D0
Violet	#FFEE82EE
Wheat	#FFF5DEB3
White	#FFFFFFFF
WhiteSmoke	#FFF5F5F5
Yellow	#FFFFFF00
YellowGreen	#FF9ACD32